CS440/ECE448 Spring 2022, Homework Five: Two-Player Games

Due: Monday, April 18, 11:59pm

Table of Contents

I. getting started

To get started on this assignment, download the template code. The template contains the following files and directories:

- `main.py` . This file plays the game (`python3 main.py`).
- `search.py` . This is the file that you will edit and submit.
- `grade.py` . In order to see your grade, you can type `python3 grade.py` .
- `grading_tests/test_visible.py` . These are the visible tests; the autograder will also include `test_hidden.py` , containing hidden tests.
- `grading_examples/` . This directory contains the JSON answer keys on which your grade is based (the visible ones). You are strongly encouraged to read these, to see what format your code should produce.
- `chess/` , `res/` , `tools/` . These directories contain code and resources from PyChess that are necessary to run the assignment.

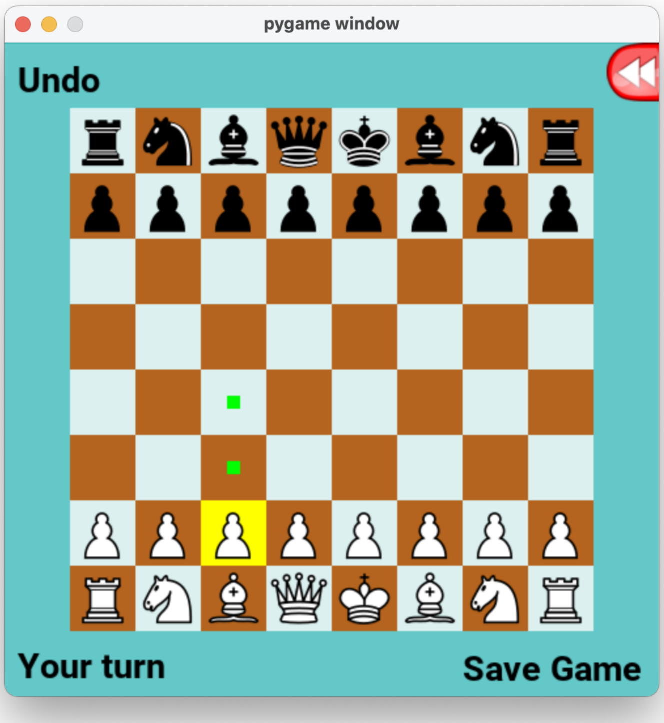
The `main.py` file will be the primary entry point for this assignment. Letâ€™s start by running it as follows:

```
python3 main.py --help
```

This will list the available options. You will see that both player0 and player1 can be human, or one of four types of AI: random, minimax, alphabeta, or stochastic. The default is `--player0 human --player1 random` , because the random player is the only one already implemented. In order to play against an "AI" that makes moves at random, type

```
python3 main.py
```

You should see a chess board pop up. When you click on any white piece (you may need to double-click), you should see bright neon green dots centered in all of the squares to which that piece can legally move, like this:



When you click (or double-click) on one of those green dots, your piece will move there. Then the computer will move one of the black pieces, and it will be your turn again.

If you have trouble using the mouse to play, you can debug your code by watching the computer play against itself. For example,

```
python3 main.py --player0 random --player1 random
```

If you want to start from one of the stored game positions, you can load them as, for example:

```
python3 main.py --loadgame game1.txt
```

We will grade your submissions using `grade.py`. This file is available to you, so that you can understand how this assignment will be graded.

Letâ€™s see what happens when we run this script:

```
python3 grade.py
```

```
EEEEEEEEEE
======================================================================
ERROR: test_alphabeta02 (test_visible.grading_tests)
----------------------------------------------------------------------
Traceback (most recent call last):
```

```
...
    raise NotImplementedError("you need to write this!")
NotImplementedError: you need to write this!

----------------------------------------------------------------------
Ran 10 tests in 0.008s
```

As you can see, all of the tests raise `NotImplementedError`, because we have not yet implemented the functions `minimax`, `alphabeta`, or `stochastic` in `search.py`. We will do this in the next few sections.

II. the PyChess API

The chess-playing interface that we're using is based on **PyChess**. All the components of PyChess that you need are included in the assignment5.zip file, but if you want to learn more about PyChess, you are welcome to download and install it. The standard distribution of PyChess includes a game-playing AI using the alphabeta search algorithm. You are welcome to read their implementation to get hints for how to write your own, but note that we have changed the function signature so that if you simply cut and paste their code into your own, it will not work.

You do not need to know how to play chess in order to do this assignment. You need to know that chess is a game between two players, one with white pieces, one with black pieces. White goes first. Players alternate making moves until white wins, black wins, or there is a tie. You don't need to know anything else about chess to do the assignment, though you may have more fun if you learn just a little (e.g., by playing against the computer).

Though you don't need to know anything about chess, you do need to understand a few key concepts, and a few key functions, from the PyChess API. The most important concepts are:

1. **player**. There are two players: Player 0, and Player 1.
   Player 0 plays white pieces, Player 1 black. Player 0 goes first.
   - **side**. PyChess keeps track of whose turn it is by using a boolean called **side**:
     `side==False` if Player 0 should play next.
2. **board**. A board is a 2-tuple of lists of pieces: `board==([white_piece0, white_piece1, ...], [black_piece0, black_piece1, ...])`.
   Each piece is a 3-list: `piece=[x,y,type]`.
   `x` is the x position of the piece (left-to-right, 1 to 8).
   `y` is the y position of the piece (top-to-bottom, 1 to 8).
   `type` is a letter indicating the type of piece.
3. **move**. A move is a 3-list: `move==[fro,to,promote]`.
   `fro` is a 2-list: `fro==[from_x,from_y]`. `to` is also a 2-list: `to==[to_x,to_y]`.
   `promote` is either `None` or `"q"`.

In addition to those key concepts, you also need to understand a few key functions, described in the following subsections.

II.A evaluate

The function `value=evaluate(board)` returns the heuristic value of the board for the white player (thus, in the textbook's terminology, the white player is Max, the black player is Min).

For example, you can find the numerical value of a board by typing:

```
from chess.lib.heuristics import evaluate

# If board is a valid PyChess game board, the following line will find its numerical value
value = evaluate(board)
```

II.B encode and decode

Lists cannot be used as keys in a dict, therefore, in order to give your `moveTree` to the autograder, you will need some way to encode the moves. `encoded=encode(*move)` converts a `move` into a string representing its standard chess encoding. The **decode** function reverses the processing of `encode`. For example:

```
from chess.lib.utils import encode, decode

# This statement evaluates to True
encode([7,2],[7,4],None)=="g7g5"

# This statement also evaluates to True
encode([5,7],[5,8],"q")=="e2e1q"

# This statement evaluates to True
decode("g7g5")==[[7,2],[7,4],None]
```

II.C generateMoves, convertMoves, makeMove

The function **generateMoves** is a generator that generates all moves that are legal on the current board. The function **convertMoves** generates a starting board. The function **makeMove** implements a move, and returns the resulting board (and side and flags). For example, the following code prints all of the moves that white can legally make, starting from the beginning board:

```
from chess.lib import convertMoves, makeMove
from search import generateMoves

# Create an initial board
side, board, flags = convertMoves("")

# Iterate over all moves that are legal from the current  board position.
```

```
    for move in generateMoves(side,board,flags):
        newside, newboard, newflags = makeMove(side, board, move[0], move[1], flags, move[2])
        print(move, newflags)
```

The **flags** and **newflags** variables specify whether or not it has become legal for black to make certain specialized types of moves. For more information, see `chess/docs.txt` .

II.D random

In order to help you understand the API, the file `search.py` contains a function from which you can copy any useful code. The function `moveList, moveTree, value = random(side, board, flags, chooser)` takes the same input as the functions you will write, and generates the same type of output, but instead of choosing a smart move, it chooses a move at random.

III. Assignment

For this assignment, you will need to write three functions: `minimax` , `alphabeta` , and `stochastic` . The content of these functions is described in the sections that follow.

III.A minimax search

For Part 1 of this assignment, you will implement minimax search. Specifically, you will implement a function `minimax(side, board, flags, depth)` in `search.py` with the following signature:

```
# search.py
def minimax(side, board, flags, depth):
    '''
    Return a minimax-optimal move sequence, tree of all boards evaluated, and value of best p
    Return: (value, moveList, moveTree)
      value (float): value of the final board in the minimax-optimal move sequence
      moveList (list): the minimax-optimal move sequence, as a list of moves
      moveTree (dict: encode(*move)->dict): a tree of moves that were evaluated in the searc
    Input:
      side (boolean): True if player1 (Min) plays next, otherwise False
      board (2-tuple of lists): current board layout, used by generateMoves and makeMove
      flags (list of flags): list of flags, used by generateMoves and makeMove
      depth (int >=0): depth of the search (number of moves)
    '''
```

As you can see, the function accepts `side` , `board` , and `flags` variables, and a non-negative integer, `depth` . It should perform minimax search over all possible move sequences of length `depth` , and return the complete tree of evaluated moves as `moveTree` . If `side==True` , you should choose a path through this tree that minimizes the heuristic value of the final board, knowing that your opponent will be trying to maximize value; conversely if `side==False` . Return the resulting optimal list of moves (including moves by both white and black) as `moveList`, and the numerical value of the final board as `value`.

A note about `depth`: The `depth` parameter specifies the total number of moves, including moves by both white and black. If `depth==1` and `side==False`, then you should just find one move, from the current board, that maximizes the value of the resulting board. If `depth==2` and `side==False`, then you should find a white move, and the immediate following black move. If `depth==3` and `side==False`, then you should find a white, black, white sequence of moves. For example, see wikipedia's page on minimax for examples and pseudo-code.

You are strongly encouraged to look at the grading examples in the `grading_examples` folder, to get a better understanding of what the `minimax` function outputs should look like. For example, to see the output of a depth-2 minimax applied to the board position in `res/savedGames/game0.txt`, you can type `more grading_examples/minimax_game0_depth2.json` which will give the following output, showing `value` on the first line, `moveList` on the second line, and `moveTree` on the third line:

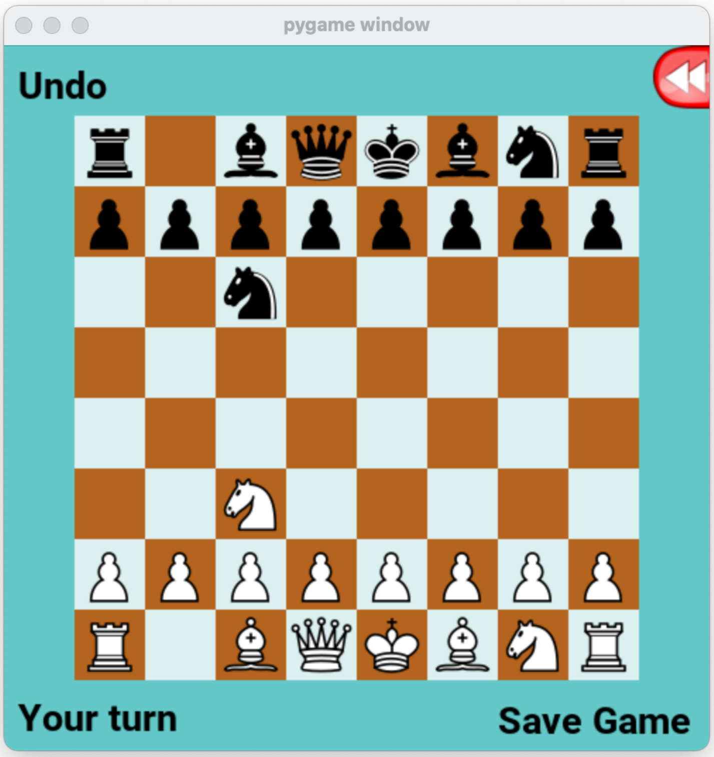
```
10.0
[[[2, 1], [3, 3], null], [[4, 7], [4, 5], null]]
{"a7a5": {"a2a4": {}, "a2a3": {}, "b2b4": {}, ...
```

You will certainly want to implement `minimax` as a recursive function. You will certainly want to use the function `generateMoves` to generate all moves that are legal in the current game state, and you will certainly want to use `makeMove` to find the newside, newboard, and newflags that result from making each move. When you get to `depth==0`, you will certainly want to use `evaluate(board)` in order to compute the heuristic value of the resulting board.

Once you have implemented minimax, you can test it by playing against it:

```
python3 main.py --player1 minimax
```

Test that it is working correctly by moving one of your knights forward. The computer should respond by moving one of its knights foward, as shown here:



If you want to watch a minimax agent win against a random-move agent, you can type

```
python3 main.py --player0 minimax --player1 random
```

III.B alphabeta search

For Part 2 of this assignment, you will implement alphabeta search. Specifically, you will implement a function `alphabeta(side, board, flags, depth)` in `search.py` with the following signature:

```
# search.py
def alphabeta(side, board, flags, depth):
    '''
    Return minimax-optimal move sequence, and a tree that exhibits alphabeta pruning.
    Return: (value, moveList, moveTree)
      value (float): value of the final board in the minimax-optimal move sequence
      moveList (list): the minimax-optimal move sequence, as a list of moves
      moveTree (dict: encode(*move)->dict): a tree of moves that were evaluated in the searc
    Input:
      side (boolean): True if player1 (Min) plays next, otherwise False
      board (2-tuple of lists): current board layout, used by generateMoves and makeMove
      flags (list of flags): list of flags, used by generateMoves and makeMove
      depth (int >=0): depth of the search (number of moves)
    '''
```

For any given input board, this function should return exactly the same value and moveList as `minimax`; the only difference between the two functions will be the returned `moveTree`. The tree returned by `alphabeta` should have fewer leaf nodes than the one returned by `minimax`, because alphabeta pruning should make it unnecessary to evaluate some of the leaf nodes.

You can test this using

```
python3 main.py --player1 alphabeta
```

or

```
python3 main.py --player0 random --player1 alphabeta
```

III.C stochastic search

For Part 3 of this assignment, you will implement stochastic search in order to find the best move. Specifically, you will implement a function `stochastic(side, board, flags, depth, breadth, chooser)` in `search.py` with the following signature:

```
#search.py
def stochastic(side, board, flags, depth, breadth, chooser):
    '''
    Choose the best move based on breadth randomly chosen paths per move, of length depth-1.
    Return: (value, moveList, moveTree)
      value (int or float): average board value of the paths for the best-scoring move
      moveList (list): a list of moves, with len(moveList)==depth
      moveTree (dict: encode(*move)->dict): a tree of moves that were evaluated in the searc
    Input:
      side (boolean): True if player1 (Min) plays next, otherwise False
      board (2-tuple of lists): current board layout, used by generateMoves and makeMove
      flags (list of flags): list of flags, used by generateMoves and makeMove
      depth (int >=0): depth of the search (number of moves)
      breadth: number of different paths
      chooser: a function similar to random.choice, but during autograding, might not be ran
    '''
```

This function has two new arguments:

- `breadth` specifies the number of different paths that should be evaluated for each initial move.
- `chooser` is a function that you should use to choose moves, randomly, at each level after the initial move. During normal game play, `chooser` will be the function `random.choice`, but during grading, it will be a non-random function, so that your answers can be compared to the answer key.

Stochastic search is different from minimax and alphabeta:
1. For every possible initial move, you should evaluate exactly `breadth` paths, chosen at random using the function `chooser`.
2. You should compute the values of the leaf nodes for each of these paths.
3. You should then average the path values in order to find the value of the

initial move.
4. Finally, among all possible initial moves, find the one that has the best average value ("best" means maximum value if `side==False`, otherwise it means minimum value). Return its value as `value`. As `moveList`, return any list of moves that starts with the optimal move.

You can test this using

```
python3 main.py --player1 stochastic
```

or

```
python3 main.py --player0 random --player1 stochastic
```

You will probably find that the stochastic player is not as good as the minimax or alphabeta players!

IV. Extra Credit

The heuristic we've been using, until now, is the default PyChess heuristic: it assigns a value to each piece, with extra points added or subtracted depending on the piece's location. For extra credit, if you wish, you can try to train a neural network to compute a better heuristic.

IV.A. The Game

The extra credit assignment will be graded based on how often your heuristic beats the default PhChess heuristic in a two-player game.

Ideally, the game would be chess. Unfortunately, grading your heuristic based on complete chess games would take too much time. Instead, the function `extracredit_grade.py` plays a very simple game:

1. Each of the two players is given the same chess board. Each of you assigns a numerical value to the board.
2. Then, the scoring program finds the depth-two minimax value of the same board. This value is provided in the lists called "values" in the data files `extracredit_train.txt` and `extracredit_validation.txt`; but if you have already completed the main assignment, it should be the same value that you'd get by running your `minimax` or `alphabeta` search with `depth=2`.
3. The winner of the game is the player whose computed value is closest to the reference value.

Notice that what we're asking you to do, basically, is to create a neural network that can guess the value of the PyChess heuristic two steps ahead.

Notice that, if you can design a funny neural net architecture that, instead of being trained to solve this problem, solves it without training by exactly computing a two-step minimax operation, then you're done. This is explicitly allowed, because we think it would be a very effective and very interesting solution.

Most of you, we guess, will choose a more general neural net architecture, and train it so that it imitates the results of two-step minimax.

IV.B. Distributed Code: Exactly Reproduce the PyChess Heuristic

Dowload the extra credit package, and unpack it. You will find the following files:
- extracredit.py: Trains the model. This is the code you will edit and submit.
- extracredit_embedding.py: Embeds a chess board into a (15x8x8) binary pytorch tensor.
- extracredit_train.txt: Training data: sequences of moves, and corresponding values.
- extracredit_validation.txt: Validation data: sequences of moves, and corresponding values.
- extracredit_grade.py: The grading script.

Try the following:

```
$ python3 extracredit.py
pygame 2.0.1 (SDL 2.0.14, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
$
```

What just happened? Well, if you open `extracredit.py` you will find these lines:

```
    # Well, you might want to create a model a little better than this...
    model = torch.nn.Sequential(torch.nn.Flatten(),torch.nn.Linear(in_features=8*8*15, out_f

    # ... and if you do, this initialization might not be relevant any more ...
    model[1].weight.data = initialize_weights()
    model[1].bias.data = torch.zeros(1)

    # ... and you might want to put some code here to train your model:
    trainset = ChessDataset(filename='extracredit_train.txt')
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=1000, shuffle=True)
    for epoch in range(2000):
        for x,y in trainloader:
            pass # Replace this line with some code that actually does the training

    # ... after which, you should save it as "model.pkl":
    torch.save(model, 'model.pkl')
```

The `torch.nn.Sequential` line has flattened the input board embedding, and then multiplied it by a matrix. The function `initialize_weights()` initializes that matrix to be exactly equal to the weights used in the PyChess linear heuristic. If you look closely at the training part of the code, you will see that it has done nothing at all; this part of the code is only here to show you how the ChessDataset and DataLoader can be used. After the `pass`, you see that the model, initialized but not trained, has been saved to `model.pkl`.

IV.C. Leaderboard

Now that you've saved `model.pkl`, you can score it using `extracredit_grade.py`:

```
$ python3 extracredit_grade.py
{
  "tests": [
    {
      "name": "train_winratio_or_eval_winratio_gt_0.5",
      "score": 0,
      "max_score": 10
    }
  ],
  "visibility": "visible",
  "execution_time": 0.7323920726776123,
  "score": 0,
  "leaderboard": [
    {
      "name": "winratio_evaluation",
      "value": 0
    },
    {
      "name": "winratio_validation",
      "value": 0.5
    },
    {
      "name": "winratio_train",
      "value": 0.5
    },
    {
      "name": "Time",
      "value": 0.7323920726776123
    }
  ]
}
```

The `leaderboard` section shows three separate scores:

- `winratio_train` is the fraction, of the 1000 boards in `extracredit_train.txt`, in which your neural net beat the PyChess heuristic. Notice that it's currently 0.5; that's because the untrained model is exactly equal to the PyChess heuristic, so that they tied (and each received 0.5 points) on all of the 1000 games.
- `winratio_development` is the same thing, but for the boards in `extracredit_validation.txt`.
- `winratio_evaluation` is the same thing, but for the boards in `extracredit_evaluation.txt`. Since you don't have that file, your score is set to 0. In order to see your actual score for this file, you'll need to submit your `extracredit.py` to the autograder.

IV.D. Extra Credit Grade

Your extra credit grade is given by the variable `score` in the output from `extracredit_grade.py`. It is calculated as

```
score = 10*min(1, sum([ max(0, min(1, 4*(winratio[x]-0.5))) for x in ['train','evaluatic
```

In other words, you get all 10 of the available points if your win ratio is better than 0.75 on either the training set (available to you) or the evaluation set (hidden). Notice: this means that you can get full credit if you get a win ratio better than 75% on the training data, even if you get a really bad win ratio on the evaluation data.

IV.E. Submission Instructions

Your file `extracredit.py` must create a pytorch `torch.nn.Module` object, and then save it in a file called `model.pkl`.

Pre-trained models (of any interestingly large size) cannot be uploaded to Gradescope, so your model will have to be created, trained, and saved by the file `extracredit.py`.

You are strongly encouraged to load `extracredit_train.txt` and/or `extracredit_validation.txt` in your `extracredit.py` function. You are not allowed to load `extracredit_evaluation.txt`. If we catch you loading that file, we'll take away your extra credit points.

In order to allow you to train interesting neural nets, we've set up Gradescope to allow you up to 40 minutes of CPU time (on one CPU). It is possible to get full points for this extra credit assignment in three or four minutes of training, but some of you may want to experiment with bigger models.

V. Submitting to Gradescope

Submit the main part of this assignment by uploading `search.py` to Gradescope. You can upload other files with it, but only `search.py` will be retained by the autograder.

Submit the extra credit part of this assignment by uploading `extracredit.py` to Gradescope. You can upload other files with it, but only `extracredit.py` will be retained by the autograder.