HW3
Discussion and Program Logs Write-up
Yixin Sun (Venus)

Part 1:

I've implemented my "value_play" to reference the values from the mediocre table when deciding on which moves from the legal moves is the best to make. "neuro_play" uses a simple 3-layer network created using the neural net framework defined in sctrach_nn and query the net each time to get the values for deciding the best move. And "expectimax" is implemented with the minimax algorithm with 2 ply exploration minimizing the maximum "harm", which in other words, is to choose the move that minimizes the maximum score the opponent could get from all the possible next steps.

Here is a log showing the performance of these new players against random.

```
Part 1:
value_play vs rand_play
0.624
neuro_play vs rand_play
0.557
expectimax vs rand_play
0.632
expectimax vs value_play
0.5075
```

As observed, "expectimax" with 2 ply shows some improvements from "value_play".

I also ran a round robin with all the current players and here is the result:

```
Round robin of current players: rand_play, first_play, last_play, score_play, va
lue_play, neuro_play, expectimax
[[0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [1 1 0 0 0 1 0]
 [1 1 1 0 0 1 0]
 [1 1 1 1 0 1 0]
 [1 1 0 0 0 0 0]
 [1 1 1 1 1 1 0]]
```

Part 2:

Hill-Climbing:

I implemented my hill-climbing neural network based on the framework set up in "scratch_nn". At first it starts with a network with all zero weights. Then it trains by playing a game with a version of itself plus random noises on the weights, and if the new version wins, the original net updates its weights according to a certain learning rate:

$$weights_{updated} = (1 - learningRate) * weights_{self} + learningRate * weights_{new}$$

And here are some results regards different training cycles (each cycle is one game play and potential update on weights) and different learning rates.

Training cycles	Learning rate	vs rand_play	vs value_play	Time used in training (s)
500	0.05	0.6125	0.4925	1204.08
200	0.05	0.597	0.4815	489.59
200	0.1	0.623	0.5145	455.84
500	0.1	0.635	0.506	1108.65
1000	0.1	0.63	0.495	2061.16
500	0.15	0.6355	0.504	1105.39
500	0.2	0.623	0.4875	1081.06

After a few tests I've chosen the combination of 500 training cycles and a learning rate of 0.15 to be my optimized yet time-saving hill-climbing training setting.

Bellman:

I implemented my bellman algorithm driven player with a start on a value table created using the minimax logics. That is, for each possible position, which is conveniently generated by "explore()", I swap the position as in opponent perspective and explore all the possible moves (dies from 1 to 6 and legal moves from that) upon that. Then I find the maximum score the opponent could get and assign that value to the original position I was exploring on. This procedure produces a minimax value table for me to start training with the bellman algorithm.

During one cycle of training using the bellman algorithm, for each position listed in the table, it explores all the possible moves (dies from 1 to 6 and all legal moves from that) upon that position and look up the value from the table of that move in an opponent perspective. Then it take the average from those values while they are "inversed", in other words, we are taking the average of all the (1-value)s. From that average then it updates the value for the starting position with a certain learning rate:

$$value_{updated} = (1 - learningRate) * value_{original} + learningRate * average$$

And here are some results regards different training cycles (each cycle is one iteration of the whole table) and different learning rates.

Training cycles	Learning rate	vs rand_play	vs value_play	Time used in training (s)
200	0.05	0.6365	0.5125	242.60
100	0.05	0.623	0.525	116.80
50	0.05	0.643	0.5295	60.82
40	0.05	0.652	0.51	47.82
30	0.05	0.642	0.5125	34.32
50	0.1	0.639	0.517	54.21
40	0.1	0.641	0.51	42.89

After a few tests I've chosen the combination of 40 training cycles and a learning rate of 0.05 to be my optimized bellman algorithm training setting. It's interesting that actually reducing the training cycles would improve the training result. I think it makes sense considering the way bellman algorithm works. To me bellman algorithm feels like it's "flattening" the landscape if we consider the table geographically. Then if we have "flattened" too much, the power the value table holds to distinguish different positions is diminished.

Matchbox:

I implemented my matchbox algorithm with an initial probability value table where each position is valued 1. And then each round of training is consisted of one game play with itself using the current value table, while tracing both paths, and depending on the result, award the positions on the path or punish the positions on the path. As for the awarding process, I did it so that the value for the position is added a factored copy of its original value. On the other hand, the punishing process will be subtracting that factored copy. And I've tried different factors like 1, which is the simplest awarding/punishing just like taking and putting beads in the box; and also function-based factors that are scaled bigger for later positions in the game and smaller for earlier positions in the game. To make it more clear, the code looks like this:

table[pos] += table[pos]*factor (this is the awarding process)

where factor could be 1 or function like ((20-t.index(pos))/10)
"t" being the traced list of positions, note that "t" is reversed before this calculation so that bigger factor could be applied to later positions in the game

And here are some results regards different combination of training cycles (each cycle is one game play and updating table values of the positions traced) and different "learning factors".

Training cycles	Learning factor	vs rand_play	vs value_play	Time used in training (s)
10000	1	0.5625	0.4305	13.20
50000	1	0.5665	0.474	51.07
100000	1	0.577	0.462	74.67
500000	1	0.5615	0.4455	374.57
10000	(10/(t.index(pos)+1))	0.547	0.4675	9.96
50000	(10/(t.index(pos)+1))	0.526	0.413	37.48
1000	(10/(t.index(pos)+1))	0.561	0.451	4.11
10000	((20-t.index(pos))/10)	0.5795	0.448	11.08
50000	((20-t.index(pos))/10)	0.581	0.469	49.42
100000	((20-t.index(pos))/10)	0.585	0.4635	77.07
500000	((20-t.index(pos))/10)	0.584	0.451	368.79

After a few tests I've chosen the combination of 100000 training cycles and a learning factor with the function of ((20-t.index(pos))/10) to be my optimized matchbox algorithm training setting. The fact that this one is running a lot of cycles with little time actually doesn't say much, since for this implementation, each cycle is just one game play while for the other ones like hill-climbing, each cycle is thousands of game plays. I would say that trying to feature engineer the better learning factor function for this one so that it becomes more powerful is not easy, but it is fun to see how these results turn out, since some of them becomes better as training cycle grows while other becomes worse. And for the ones that becomes better as training cycles, it seemed that they have reached plateau entering 100-thousand-levels.

Round-robin & final thoughts:

Here is the result for a round-robin of all the players:

```
Hill climbing...
Bell ringing...
Box matching...
Round robin of all players: rand_play, first_play, last_play, score_play, value_
play, neuro_play, expectimax, hc_play, bellman_play, matchbox_play
[[0 1 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0]
 [1 1 0 0 0 1 0 0 0 1]
 [1 1 1 0 0 1 0 0 0 1]
 [1 1 1 1 0 1 0 1 0 1]
 [0 0 0 0 0 0 0 0 0 0]
 [1 1 1 1 1 1 0 1 0 1]
 [1 1 1 1 0 1 0 0 0 1]
 [1 1 1 1 1 1 1 1 0 1]
 [1 1 0 0 0 1 0 0 0 0]]
```

Considering the three algorithms I have implemented, I think the one runs fast and performs best is Bellman algorithm. Even though matchbox runs pretty fast, it's not of

the same scale comparing to hill-climbing and bellman so I think the comparison wouldn't make sense. Also I think the reason why bellman performs best for me probably has to do with the fact that it is the easiest to engineer considering tuning the parameters. I do doubt that if I have more time and resources to experiment, I could potentially achieve better performance with hill-climbing and matchbox. Though at the same time I feel that for matchbox it might be hard to improve, considering the algorithm itself, could “nannon” be a problem too complex for matchbox?

p.s. For my codes, all of the detailed implementation are within the nannon folder, while run-able codes for each part are outside and can be run on the terminal with “python x.py”