CIS 667 Project Final Report

CIS 667 Project Final Report and Tree+NN-Based AI

Team member:

Jiaxin Song **NetID: jsong37**
Xiang Li **NetID: xli336**
Xiaohan Liu **NetID: xliu194**
Yiwen Cheng **NetID: ycheng36**

Github repository:

https://github.com/JiaxinSong/tictactoe

Implementations:

Milestone 1: Game Implementation:

Rule of the game:

This game is basically a tic-tac-toe game. Players alternate turns placing a stone of their color on an empty intersection. The winner is the first player to form an unbroken chain of a certain number which should be modified with the size of boards of stones horizontally, vertically, or diagonally. In this project, we modified the rule as at the end of each turn, there is a 25% chance that the other player's turn gets skipped and the current player gets to go again. We use numpy[1] to process our matrix and python[2] to coding.

Game implementations and modification:

To represent our game's state, we decided to use a 2xNxN array. The first dimension represents the player, 0 is the first player, 1 is the second. NxN is the size of a board, if there are no pieces on board, the corresponding position will be 0, otherwise it will be 1 according to the player. If a player chooses to place a piece at an empty position on the

board, and the coordinate is inside the board, it will be considered as a valid action.

At the beginning, the human can decide the board size N and how many consecutive pieces C can determine a win. If a player succeeds in placing C consecutive pieces in a row, whether horizontally, vertically or diagonally, then the player will win the game and the game is over. If there is no possible space for a player to place a piece, and no one has won the game, then it is a draw and the game will end.

To explain it more explicitly, here are some figures below. All figures are screenshot when the program is running. The following small example uses N = 5 and C = 3 as the board size and winning condition. The first player's pieces are "o" and the second player's pieces are "x". The empty positions are "+".

Figure 1. Board state at the beginning of the game

The figure 1 shows the board state at the beginning of the game, no one has placed the pieces yet.

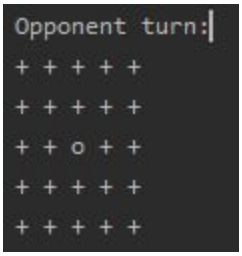


Figure 2. Board state after the first player's turn

The figure 2 shows the state of board after the first player chose to place a piece at position (2, 2) and now it's the second player's turn to place a piece.

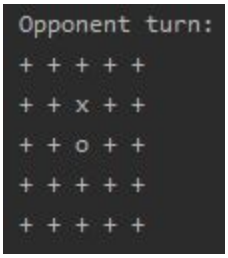


Figure 3. Board state after the second player's turn

The figure 3 shows the state of board after the second player chose to place a piece at position (1, 2) and the first player chose to skip the turn, now it's still the second player's turn.

Figure 4. First player won the game

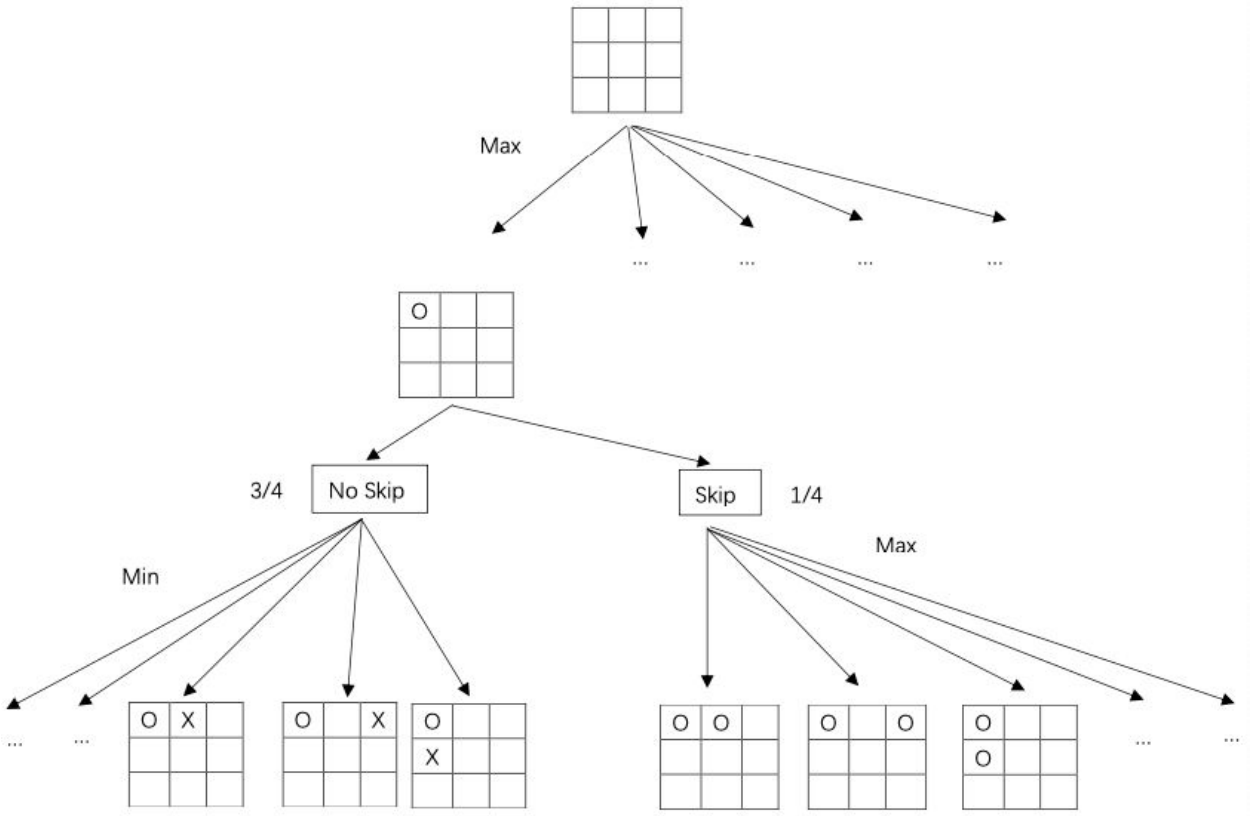
The figure 4 shows that after several steps, the first player gets 3 consecutive pieces "o" in a row and wins the game.

In this game, we can modify our board size and the number of consecutive pieces required to win, as long as the latter is not bigger than the former and all player's actions are valid, the game will work perfectly.

Milestone 2: Tree-Based AI Implementation:

Expectiminimax[3] tree design and scoring method:

Since this game has a 25% chance that the other player's turn gets skipped and the current player gets to go again, the number of nodes our tree searched each layer should be twice as many as the regular minimax tree. In a 3*3 chessboard, the structure of my tree is shown below and this idea came from our Instructor of this course Garrett Katz.



If the next turn is not skipped, x will make its action, otherwise, o can make its action again. And if x wins, the utility will be negative. If o wins, the utility will be positive. If it is a draw, the utility should be 0. The utility of the node on the highest layer should be the max(for o) or the min(for o) of the utility of its child nodes. For the lower layers, the utility should be 3/4*minimum utility of its child nodes which is x's turn and 1/4*maximum utility of its child nodes which let o play again. It is the way for x.

How to calculate the number of nodes AI searched each turn? The number of nodes searched each turn should be valid positions on the chessboard*the sum of the nodes on each layer of the search tree. We should double the nodes of each layer except the first layer. For example, if the board size is 5*5 and the search depth is 3, for the first turn, the node searched should be:

25*(1+1*2*24+1*2*24*2*23)=56425 nodes

For the scoring method, since we can limit the depth of our tree, there are two situations for leaf nodes: game over or not. How to calculate the score of them in a fair way is a problem. Fortunately, we worked this out.
For the nodes that end the games, we score them by 1 (or -1)*(depth-current depth+1). For example, if o wins on depth 2 and the depth set is 5, the score for it should be 4. This

method has a benefit that the action which can win the game early will have the larger score.
For the nodes that can not end the game and are on the last layer of the search tree, we have a special method to calculate its utility. For each row, column and 2 diagonals we find the maximum pieces in a row that have potential to win the game. For example, we have a row like this:

o o x o o o

If the goal to win the game is 4 pieces in a row, the maximum o pieces in this row that have potential to win should be 0 since o has no chance to achieve the goal in this row.
For the row below, the maximum o pieces in this row should be 2.

o o - - x o o

Then we compare the maximum pieces of o and x. If o is larger, the score should be 0.5. If it is a draw, the score should be 0, otherwise, it should be -0.5. By using this scoring method, the winning rate of our AI is excellent.

Experimental Results

To test the performance of our tree-based AI, we used 5 different sizes of chess boards to make experiments. For each size, we let two random AI battle 100 times first and let random AI versus tree-based AI 100 times. To show the performance better, we let random AI play first and tree-based AI go second. This is the result for 100 battles between two random AI players:

Size(goal) Result	7*7(5)	8*8(8)	9*9(5)	10*10(8)	13*13(5)
O win	50	3	56	22	53
X win	39	3	44	17	47
Draw	11	94	0	61	0

For 100 battles between random AI and tree-based AI, we got the results below.
In order to get shorter response time, we limited the depth of the search tree from the second layer(for the first layer we still searched every valid position), and we also limited

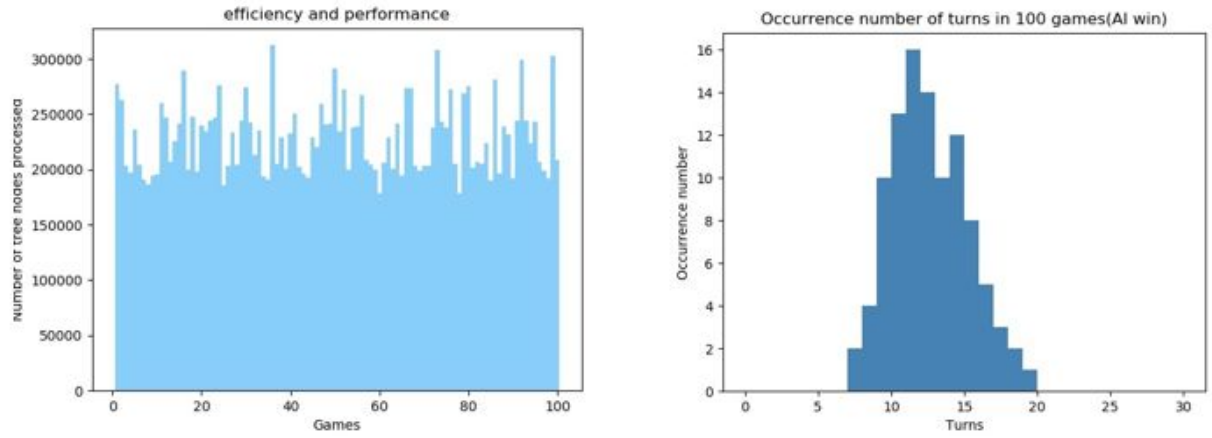
the number of nodes searched each layer, which means we randomly chose a certain number of nodes each layer.
Here is the parameter we set:
size: 7x7 winning goal: 5 in a row depth: 3 nodes of each layer: 15
size: 8x8 winning goal: 8 in a row depth: 3 nodes of each layer: 10
size: 9x9 winning goal: 5 in a row depth: 3 nodes of each layer: 12
size: 10x10 winning goal: 8 in a row depth: 3 nodes of each layer: 9
size: 13x13 winning goal: 5 in a row depth: 3 nodes of each layer: 8
And this is the result of the 100 games:

Size(goal) Result	7*7(5)	8*8(8)	9*9(5)	10*10(8)	13*13(5)
O win	0	0	0	2	0
X win	100	56	100	90	100
Draw	0	44	0	8	0

Performance and efficiency：
We also made two figures for each set of experiments to indicate the performance and efficiency of our tree based AI. We have a bar chart[3] for the result and number of nodes tree-based AI search each game. The orange bars mean the first player(random) wins the game, blue bars for the tree-based AI winning and green bars for draws. How many turns of the games which tree-based AI wins is shown in a histogram.

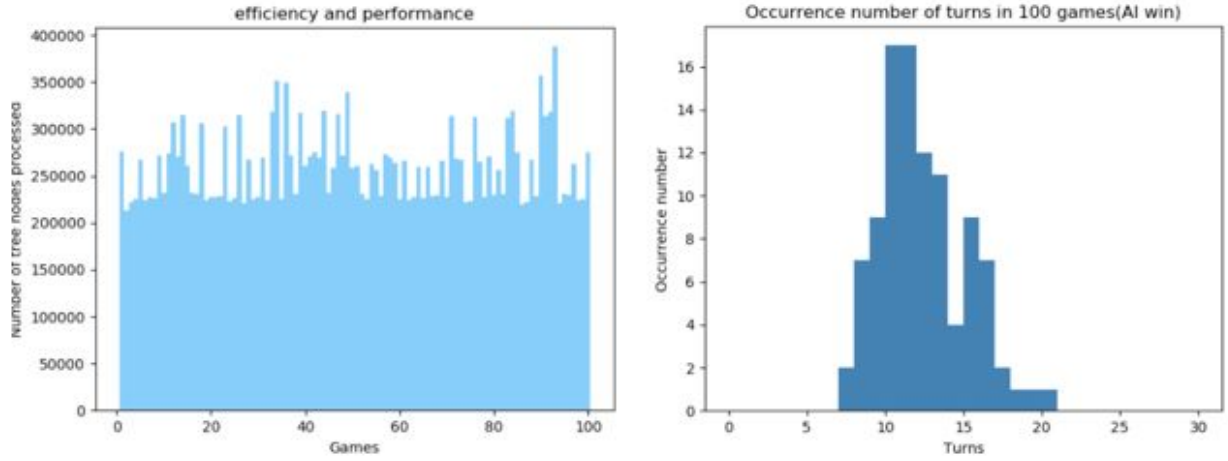
We started from different sizes but with the same winning rule: 5 pieces in a row.

Size 7*7(5 to win):



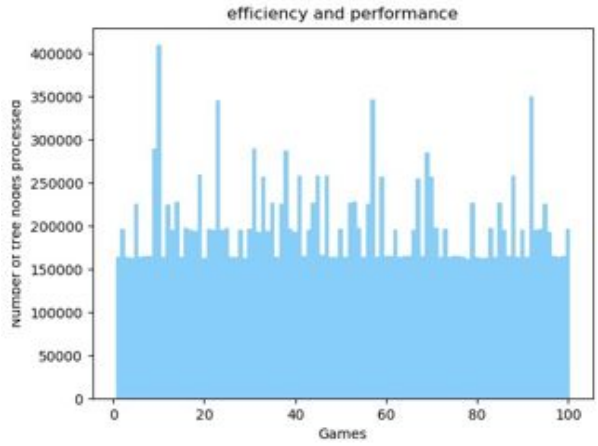
In the left figure, the blue bar represents the tree-basedAI win. For this size, the tree-based AI won all the games, and most of the games were ended in 16 turns.

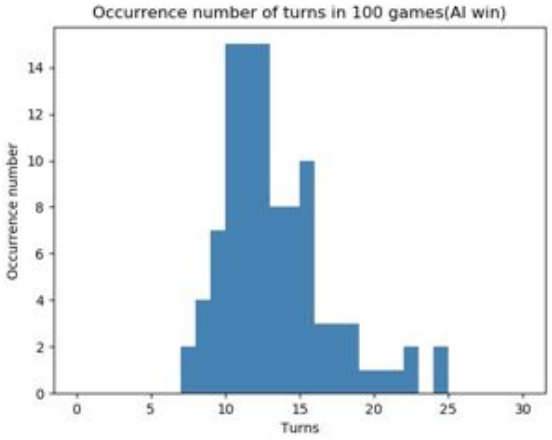
Size 9*9(5 to win):



In the left figure, the blue bar represents the tree-basedAI win. For this size, the result is very close to the previous one.

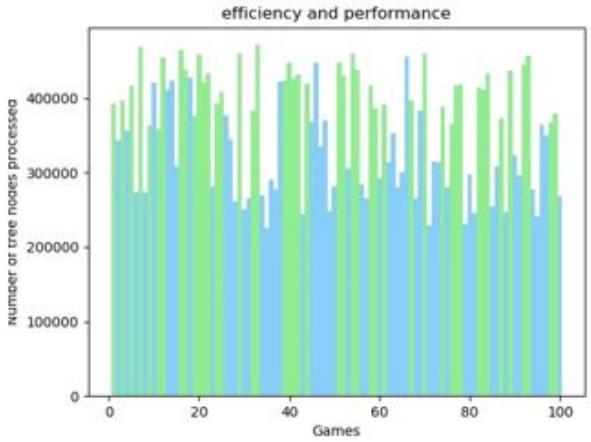
Size 13*13(5 to win):

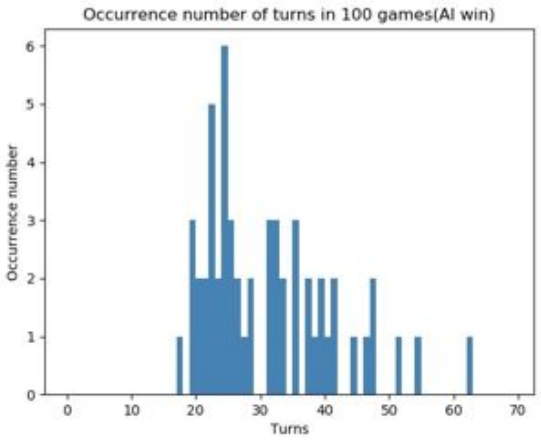




In the left figure, the blue bar represents the tree-basedAI win. The result is very close to the previous one again. So we can come up with a single conclusion that our tree-based AI has a very good performance on games that the board sizes are larger enough than the winning goals.
Then we made 2 experiments that the board sizes and the winning goals are relatively close. And many draw games came out.

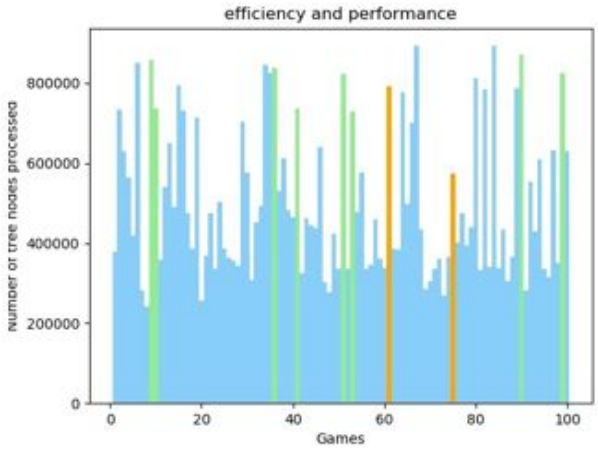
Size 8*8(8 to win):

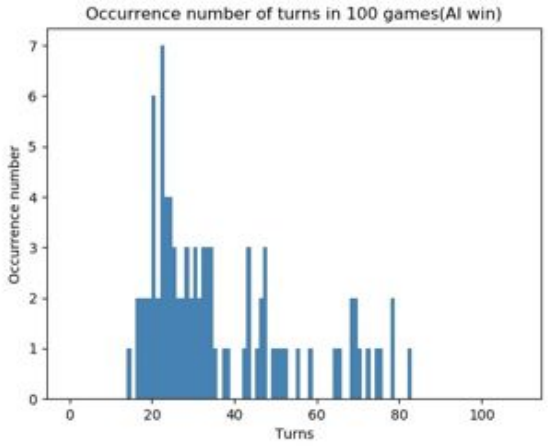




In the left figure, the blue bar represents the tree-based AI wins and the green bar represents the draw. We can see that AI needs more turns to win the game. And the number of draw games is very close to AI winning games.

Size 10*10(8 to win):





In the left figure, the blue bar represents the tree-based AI wins, the green bar represents the draw and the orange bar represents the random AI wins. Eventually, tree-based AI lost 2 games this time. The reason for this result is probably because we searched a relatively small number of nodes each layer, but it still took almost 10 hours to run.
In conclusion, our tree-based AI is very smart and has a wonderful winning rate for most cases.

Milestone 3: Tree+NN-Based AI Implementation:

Neural Network Approach

Training Data and State Representation

In the NN part, our training data is generated by two tree-based AI from 25 different complete game plays. Using the board's state encoded in one-hot as the input and the corresponding utility value as output. In other words, every input is a 3*N*N array where N is the size of the board, and output is the utility value. In this project, we use PyTorch[4, 5] to build our neural network model.

Error Function

For error function, we decided to use the sum of squared errors:

$$e = \sum_n (y^{(n)} - y_{\text{targ}}^{(n)})^2$$

Using Adam optimizer[6] to optimize the parameters. Here we set 0.01 as our learning rate.

Incorporation into Tree-Based AI

This neural network is given the state as input and trained to approximate the state's utility score as its output. Basically, we use the model to predict every valid child state's utility and at every step. To collect our data, we made 25 battles between two tree-based AI with the same parameters. On board size 3,4,5,6, we made our tree-based AI search three layers in the minimax tree and 15 nodes on each layer. On board size 7, we search

8 nodes on each layer to save time. To test performance, we made NN AI battle with tree-based AI with the same parameters. For more information, you can read the readme file on github.

Experimental Results:

Jiaxin Song's Results :

My neural network structure:

Input layer has 3*board size* board size units, the first hidden layer has 40 units. The second hidden layer has 20 units. Then I use the sigmoid[7] activation function. Output layer has 1 unit.

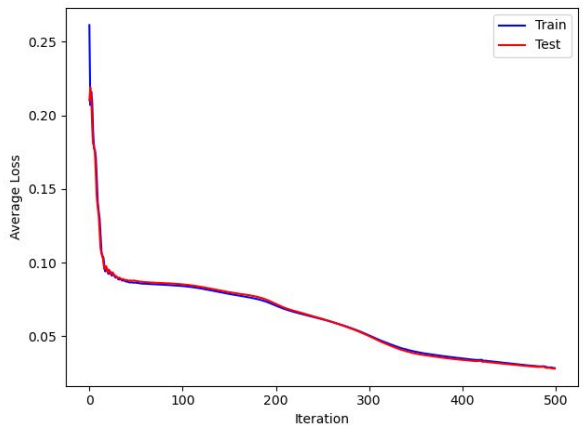
```
modules = Sequential(Flatten(),
                     Linear(3*board_size*board_size, 40),
                     Linear(40, 20),
                     Sigmoid(),
                     Linear(20, 1))
```

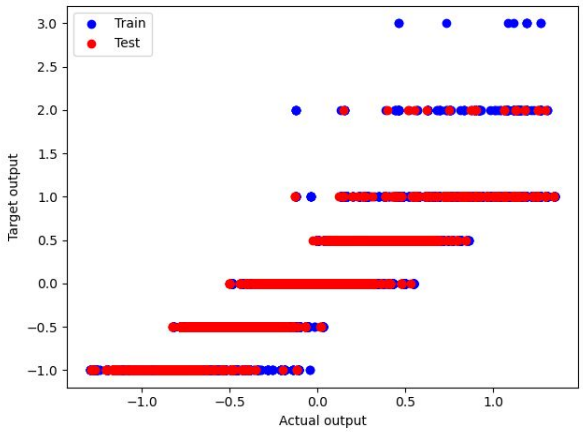
Learning Curve and Datapoint:

In all training processes, my proportion of train data and test data is 9:1.

board size 3*3, 3 pieces to win:

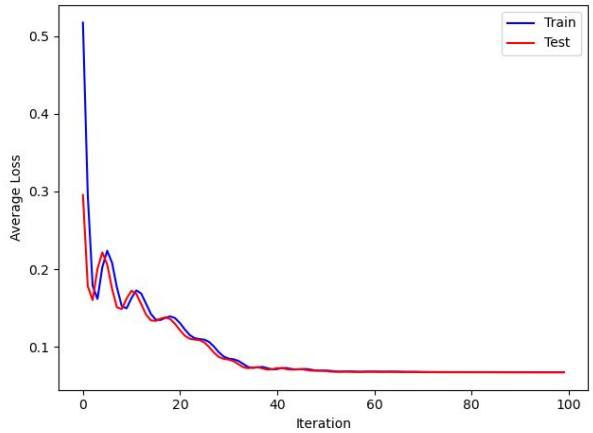
100 iterations:

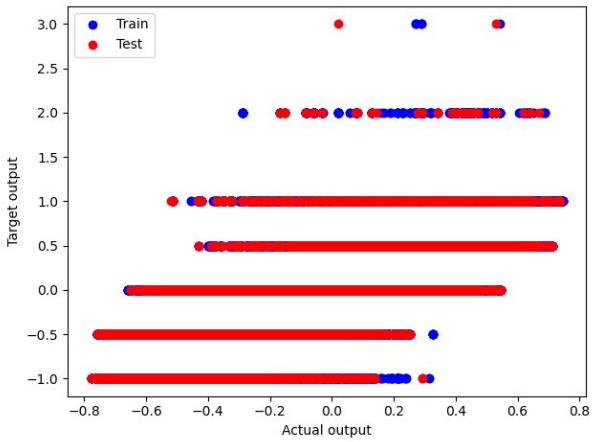




board size 4*4, 3 pieces to win:

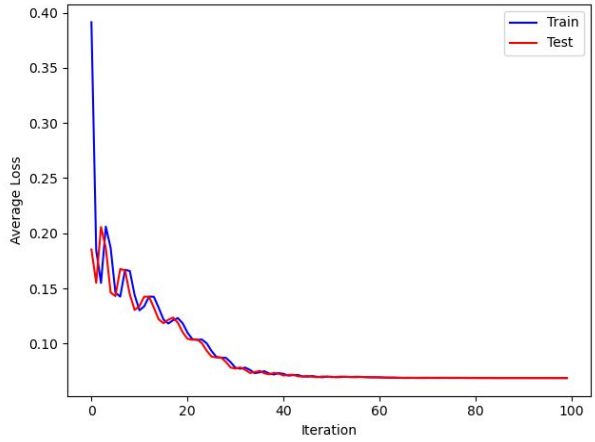
100 iterations:

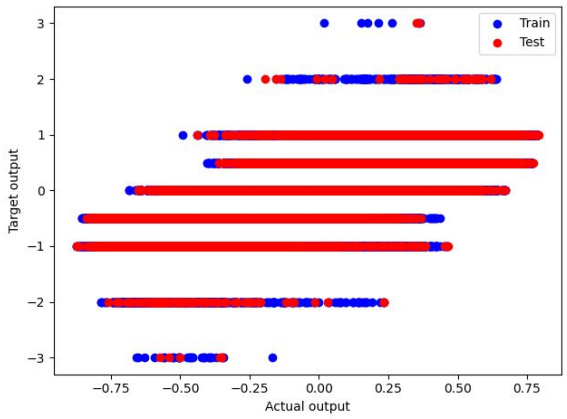




board size 5*5, 3 pieces to win:

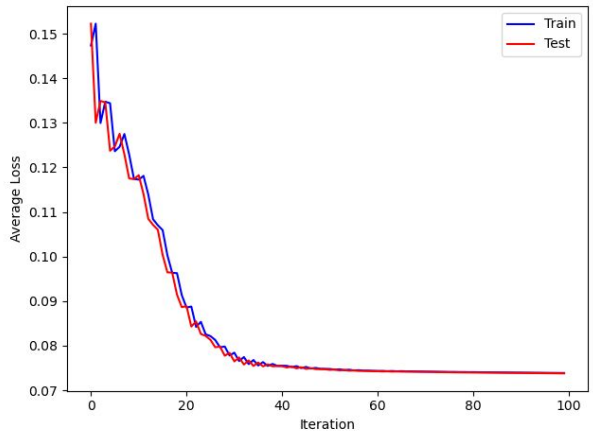
100 iterations:

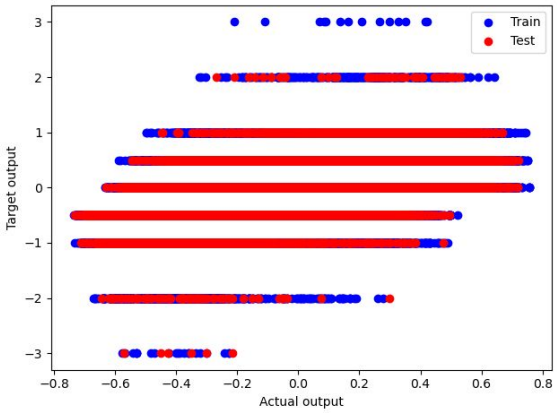




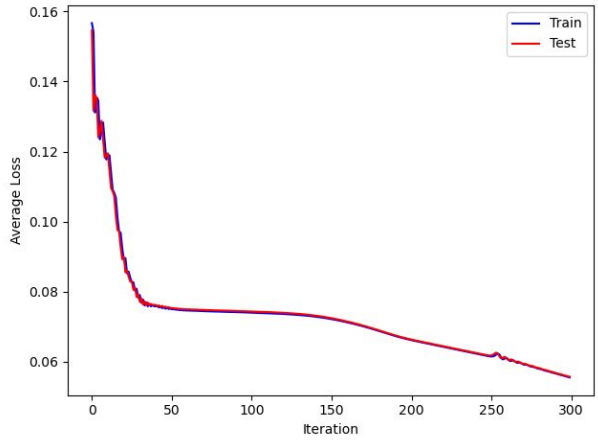
board size 6*6, 4 pieces to win:

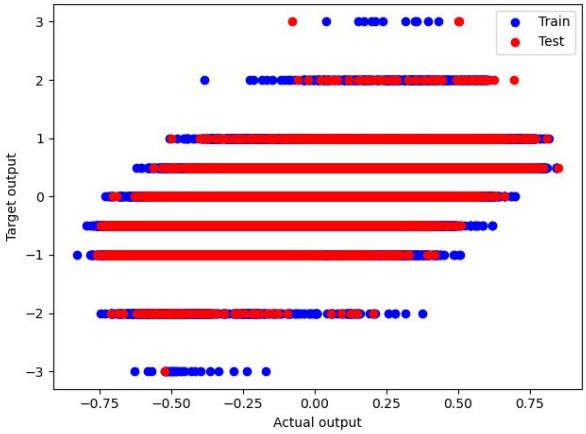
100 iterations:





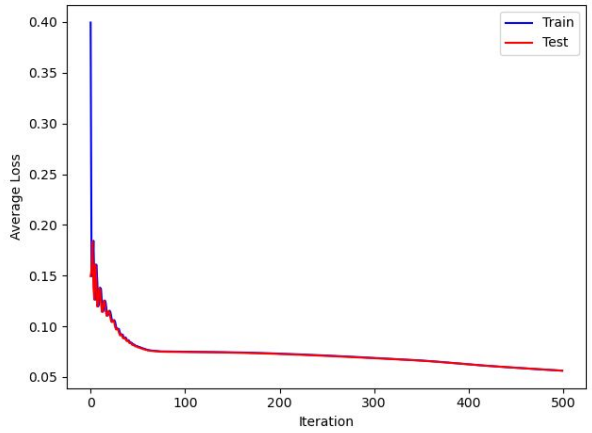
300 iterations:

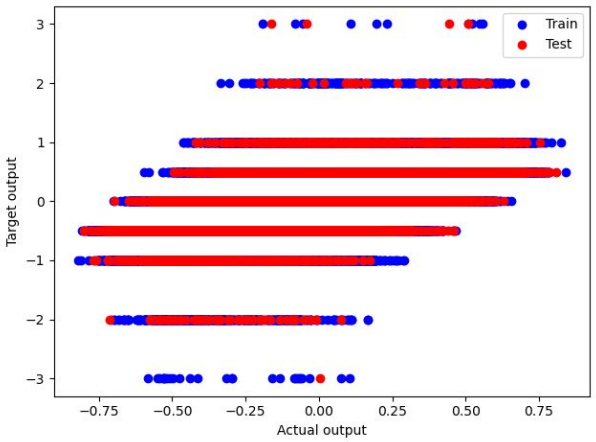




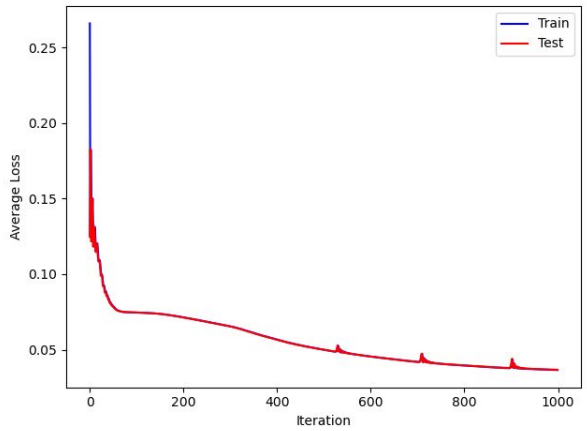
board size 7*7, 4 pieces to win:

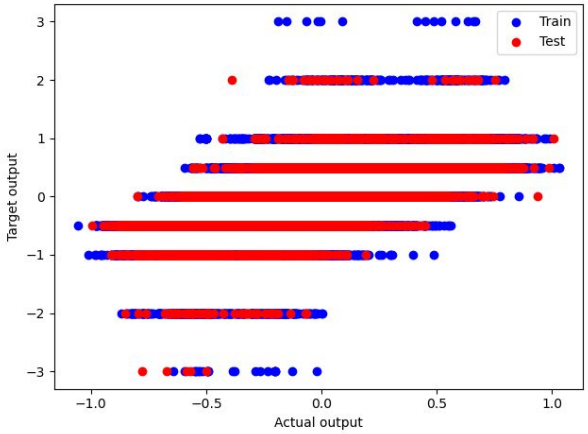
500 iterations:





1000 iterations:





Here are 100 battle results. This time I choose NN AI to play first, since the tree-based AI is hard to defeat.

Size(target) Iterations Result	3*3(3) 100	4*4(3) 100	5*5(3) 100	6*6(4) 100	6*6(4) 300	7*7(4) 500	7*7(4) 1000
NN AI win	58	56	78	47	66	57	54

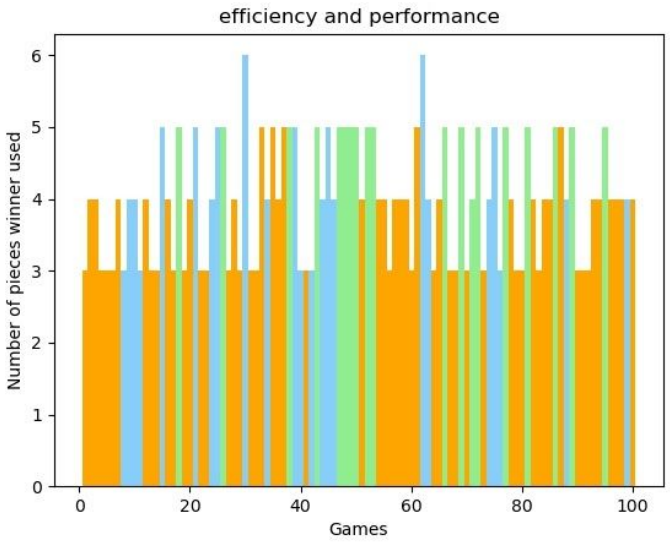
Tree-based AI win	23	44	22	53	34	43	46
Draw	19	0	0	0	0	0	0

Final score bar graph

In all graphs, orange means NN AI wins. Blue means tree-based AI wins. Green means draw.

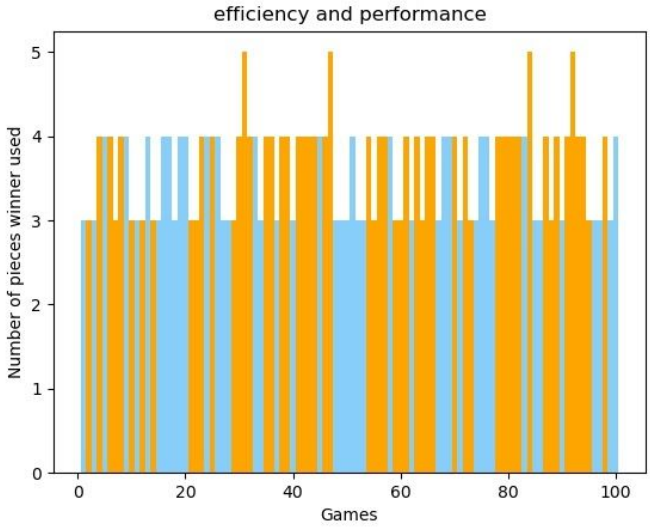
board size 3*3, 3 pieces to win:

After 100 iterations, the battles results show that NN AI has a relatively good performance. Since the board size is so small, draws cannot be avoid.



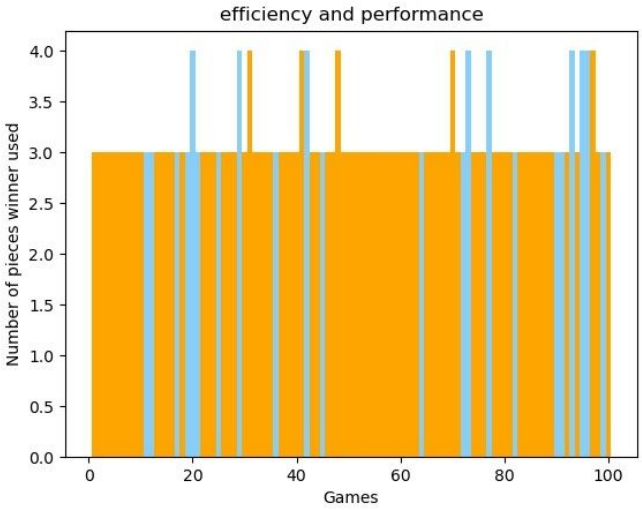
board size 4*4, 3 pieces to win:

On 4*4 board, NN AI seems a little weak, probably because iterations are not much enough.



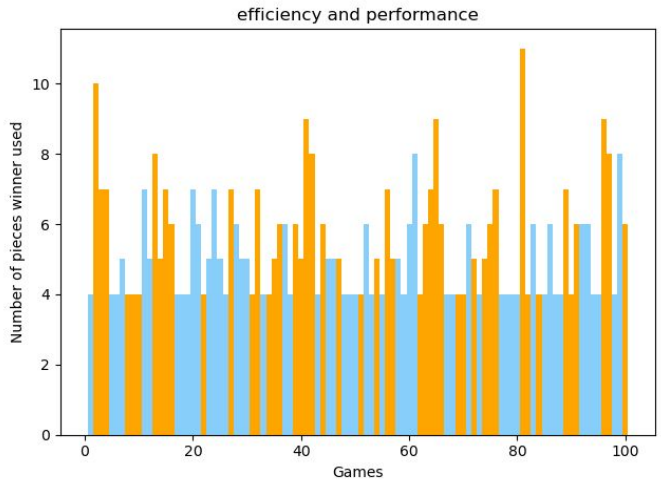
board size 5*5, 3 pieces to win:

On 5*5 board, NN AI has better performance than on 4*4 board: it wins 78 times.

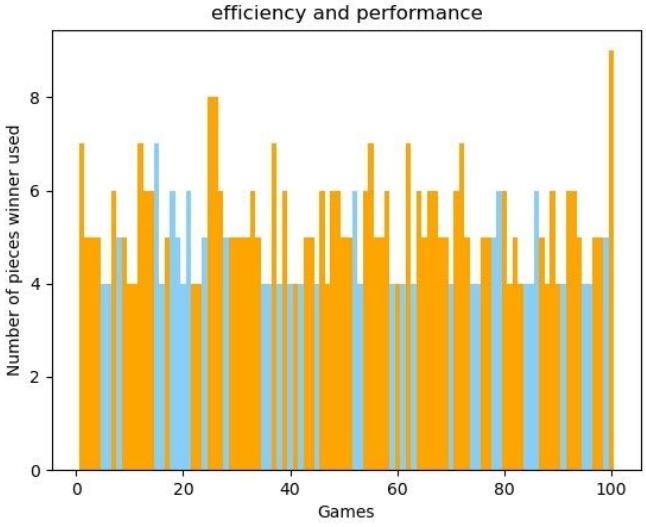


board size 6*6, 4 pieces to win:

After 100 iterations, NN AI even has less win games than tree-based AI.

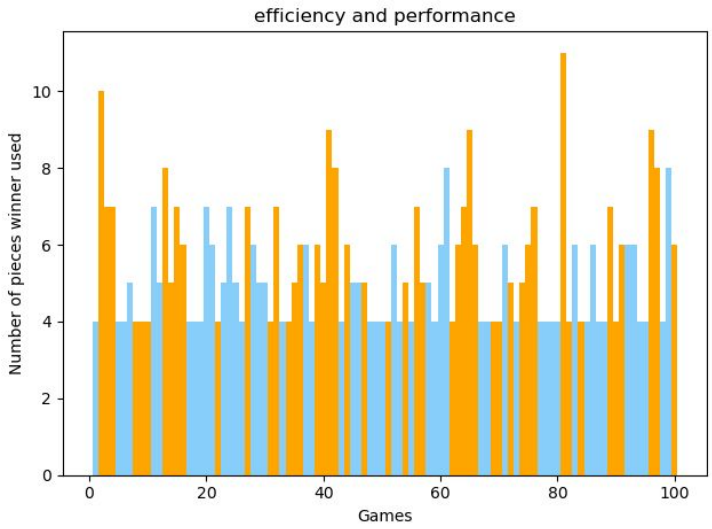


So I decided to change iterations to 300 to see what will happen. This time NN AI has a significant improvement: it wins almost 20 more games.

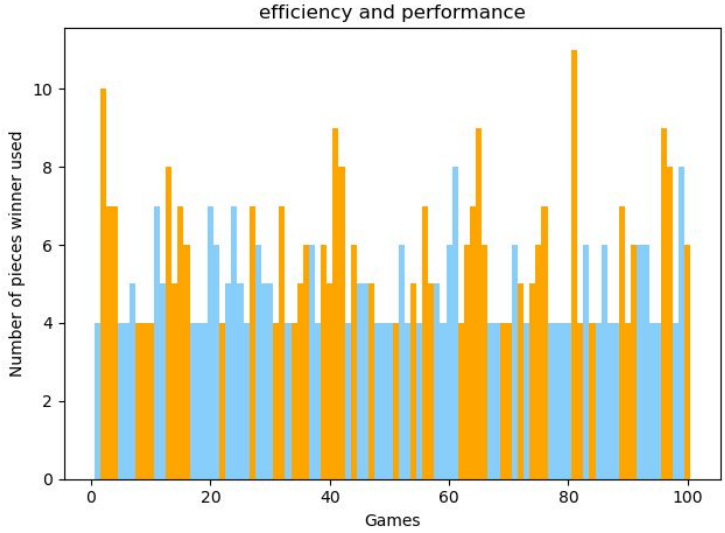


board size 7*7, 4 pieces to win:

This time I started with 500 iterations, however, NN AI is not impressive enough: just win 57 games.



So, I decided to change iterations to 1000 to see what will happen. However, it performs even a little worse. It is probably because of overfitting.



Conclusion

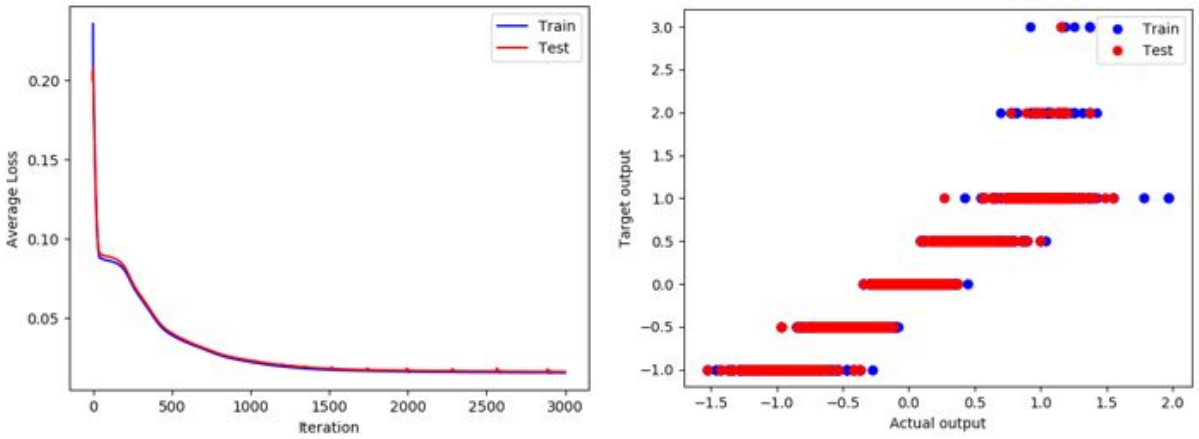
The performance of NN AI is not very stable but still acceptable. There are few reasons for these results. First, my model is not very well-designed, since I have little experience of neural network design and parameter choice. Then, the iteration times are hard to set, too small or too large will both cause performance decrement. Last but not least, due to

limited time, I cannot perform this game on a bigger board size, since data collection is very time-consuming. For 6*6 board size, we made 2 tree-based AI battles 25 times and each step they searched 3 layers of the tree and 15 nodes each layer. This takes almost 8 hours to write data6.pkl. So we search less nodes each layer on 7*7 boards. Anyway, I am still satisfied with this result, since NN AI has more winning rates on each board size.

Xiang Li's Results :

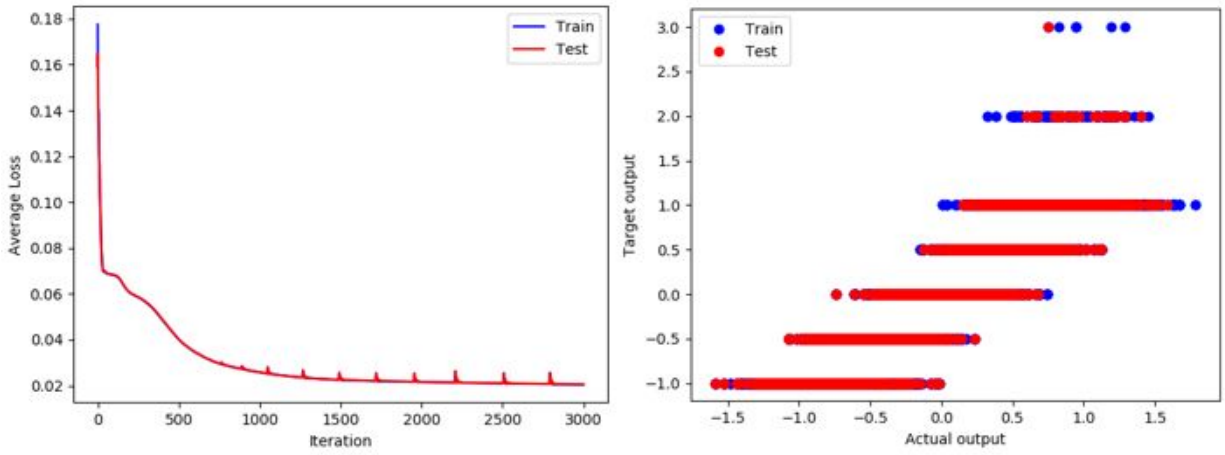
In the project, I used one input layer, one hidden layer and one output layer as my neural network. In the input layer, there are 3*N*N units where N is the board size. In the hidden layer, I set 20 as the number of hidden layer units and for the output layer there is 1 unit. Each layer is linear type and the activation function I used is sigmoid function.

Learning Curve and Datapoint:

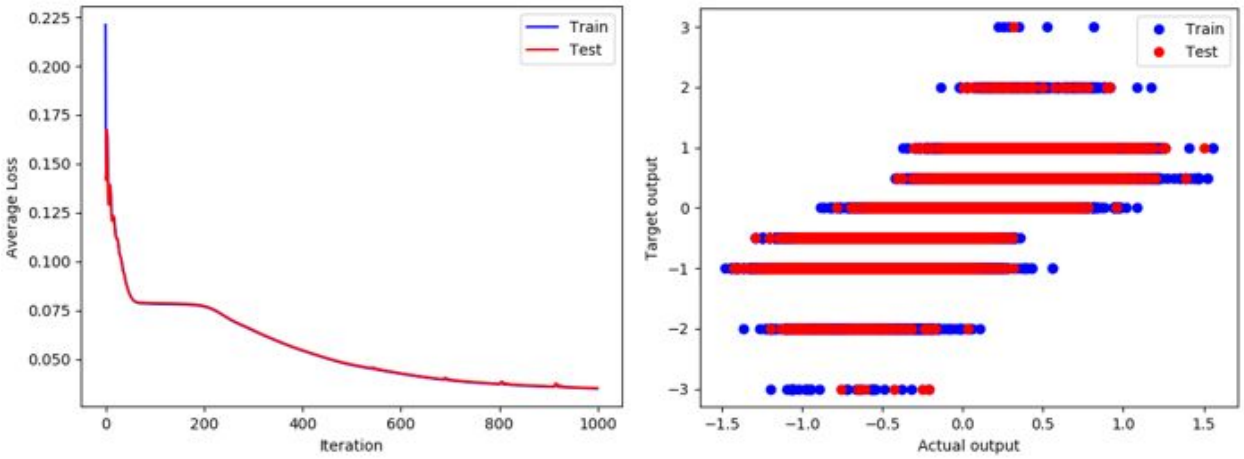


The result in the figure above uses a board size of 3. Winning condition is 3 and iteration

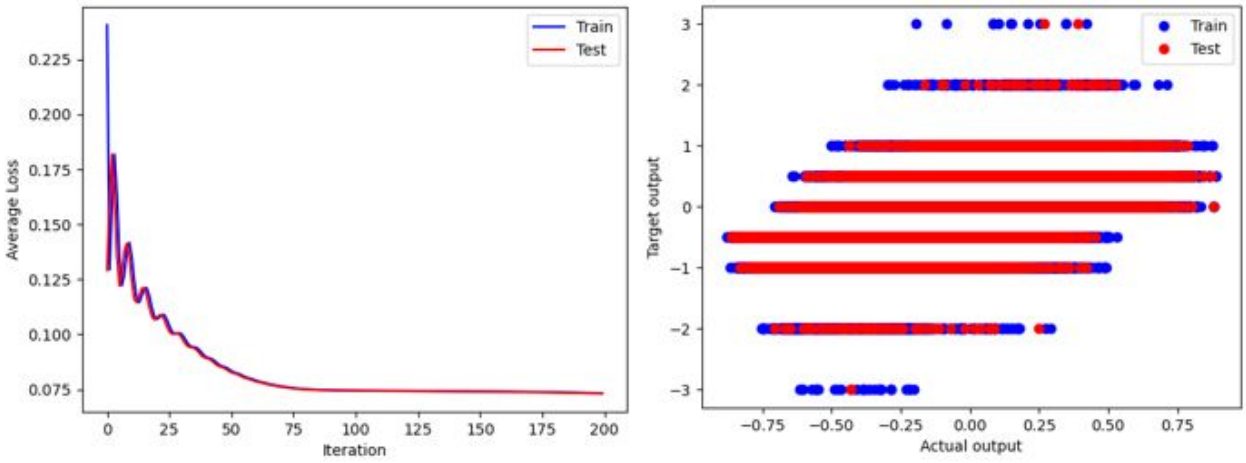
time is 3000.



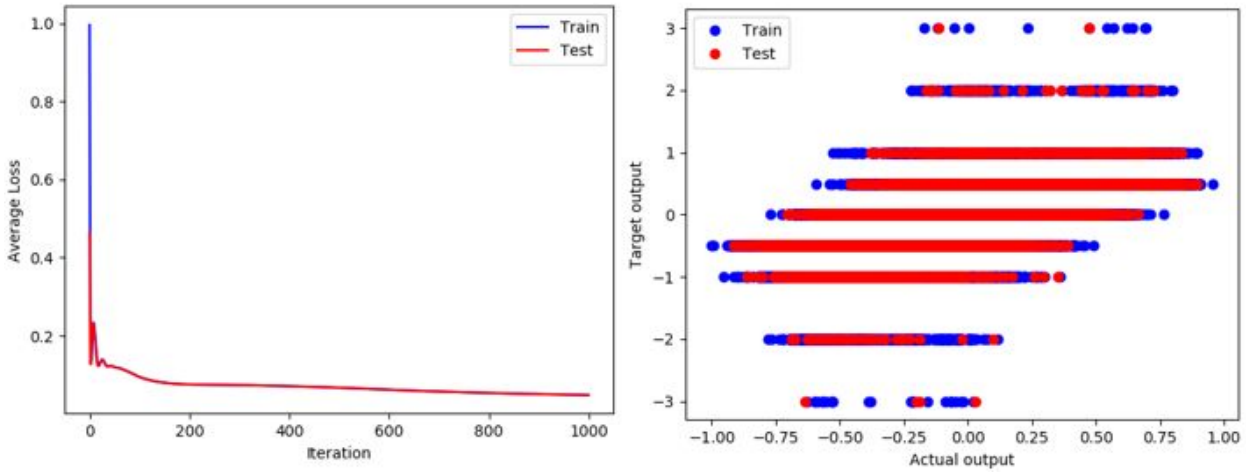
The result in the figure above uses a board size of 4. Winning condition is 3 and iteration time is 3000.



The result in the figure above uses a board size of 5. Winning condition is 3 and iteration time is 1000.



The result in the figure above uses a board size of 6. Winning condition is 4 and iteration time is 200.



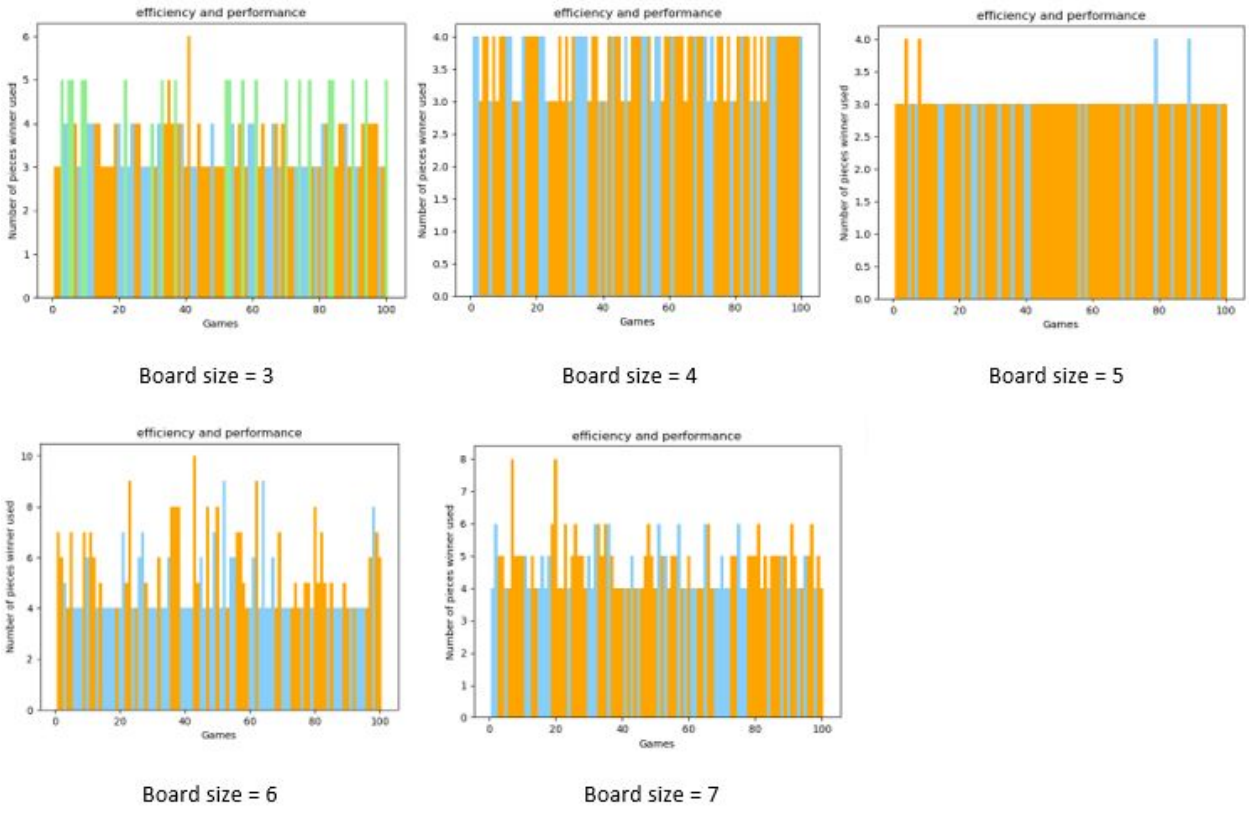
The result in the figure above uses a board size of 7. Winning condition is 4 and iteration time is 1000.

Final score bar graph

For each problem size we conduct 100 games between our tree-based AI and tree+NN-based AI. We let the tree+NN-based be the first player and have our result shown in the table below.

Size(goal)	3*3(3)	4*4(3)	5*5(3)	6*6(4)	7*7(4)
Result					

Tree-based AI win	31	36	21	46	62
Tree+NN-based AI win	48	64	79	54	38
Draw	21	0	0	0	0



In the figure above, the blue bar represents the tree-based AI wins, the green bar represents the draw and the orange bar represents the tree+NN-based AI wins. The x-axis is the game number and y-axis is the number of pieces the winner used.

We can see when board size is 3, most of the time tree+NN-based AI wins with less pieces. When the board size is 4, the number of chess pieces used to win is similar which means the performance is very close, but tree+NN-based AI wins more games. When the board size is 5, the number of chess pieces used to win is almost the same but again,

tree+NN-based AI wins way more than tree-based AI. When the board size is 6, tree-based AI wins with less pieces most of the time, since I change the iteration time to 200, the tree+NN-based AI model may not be very good. When board size is 7, the performance of each is similar but still, tree+NN-based AI wins more.

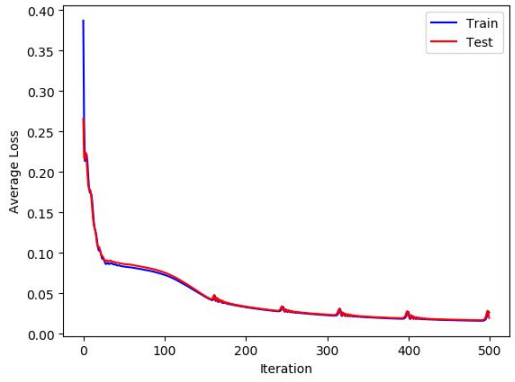
Xiaohan Liu's Results :

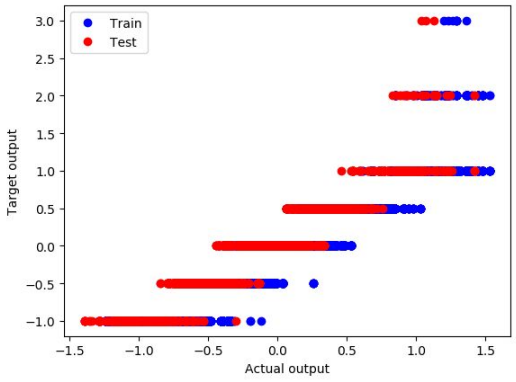
My neural network module has 3 linear type layers and I used Tanh() as the activation function: input layer with 3*board size*board size units, Tanh activation function, then a hidden layer with 30 units, and a Tanh[8] activation function, finally a output layer with 10 units and output size is 1.

For the iteration time of gradient descent to train this model: when the board size is 7, I run 300 iterations; when the board size is smaller than 7, I run 500 iterations.

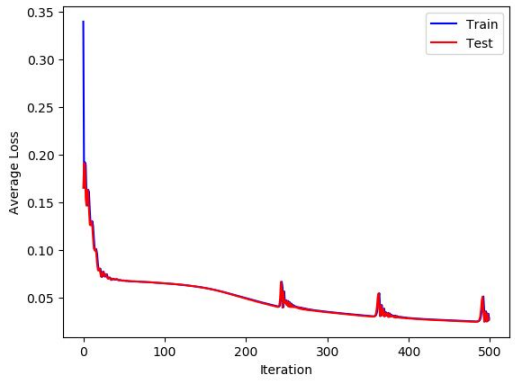
Learning Curve and Datapoint

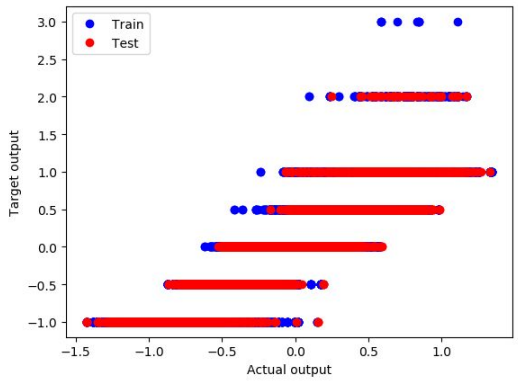
Board size 3, winning target 3 pieces, 500 iterations of gradient descent:



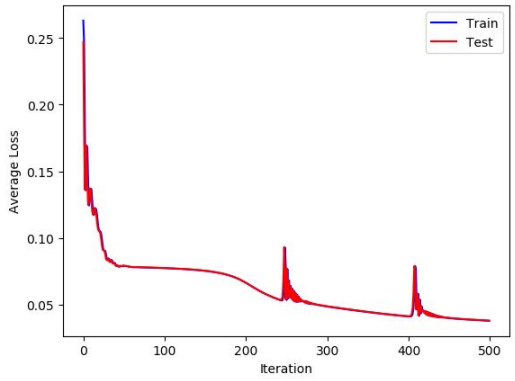


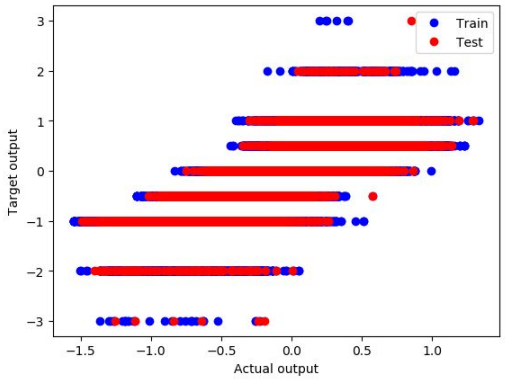
Board size 4, wining target 3 pieces, 500 iterations of gradient descent:



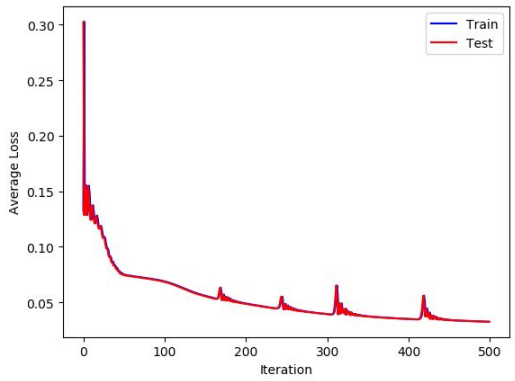


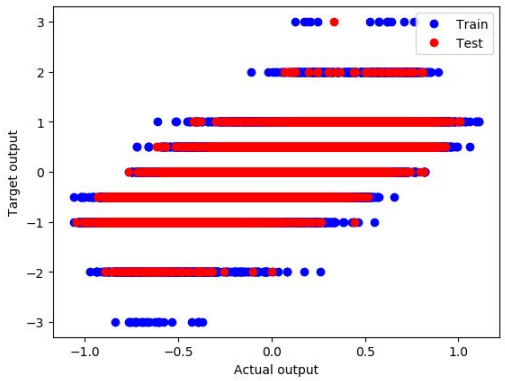
Board size 5, winning target 3 pieces, 500 iterations of gradient descent:



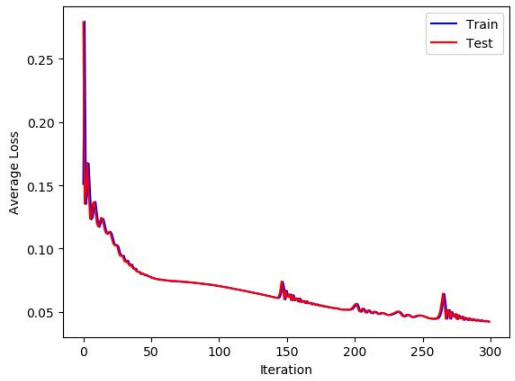


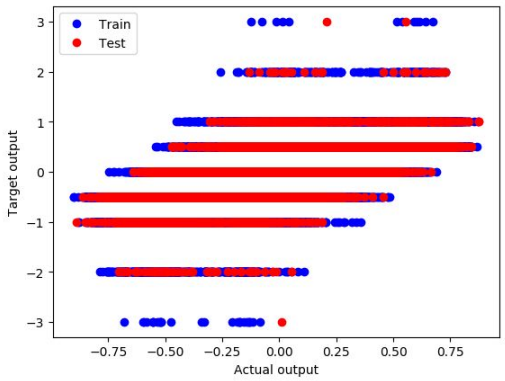
Board size 6, winning target 4 pieces, 500 iterations of gradient descent:





Board size 7, winning target 4 pieces, 300 iterations of gradient descent:





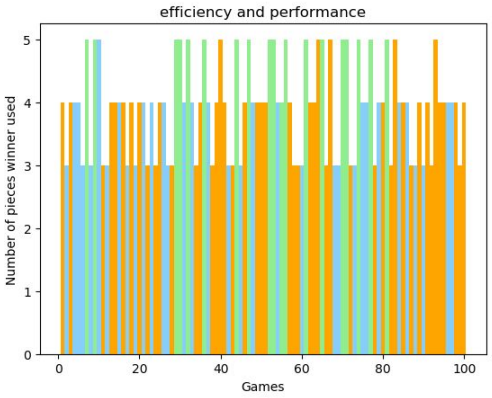
Final score bar graph

This form is the win times of each AI (tree+NN-based AI play against tree-based AI) in the 100 games

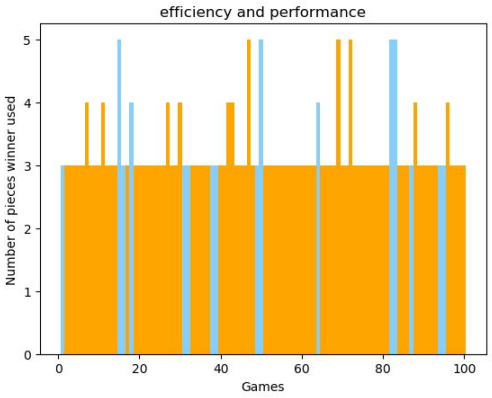
Size(goal) Result	3*3(3)	4*4(3)	5*5(3)	6*6(4)	7*7(4)
Tree-based AI win	34	16	25	48	38
Tree+NN-based AI win	47	84	75	52	62
Draw	19	0	0	0	0

For the final score histogram we use the number of winner used pieces to represent. When tree+NN based AI wins, the bar color is orange and when tree based AI wins the bar color is blue. We use color green for a draw.

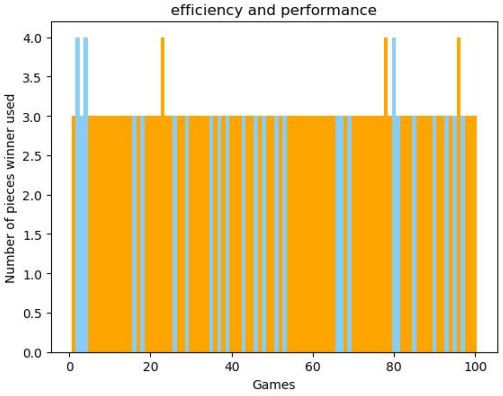
Comparing these five figures, it's obvious to see that this tree+NN AI works well when board size equals to 4, 5 and 7. In the board size 6 and 3, tree+NN AI wins a little bit more time. When the size board is 3, there even is 19 draws.



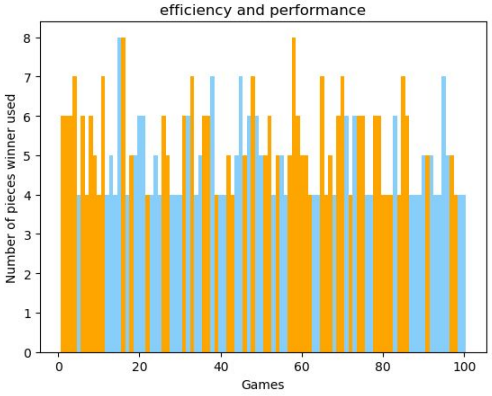
Board size = 3



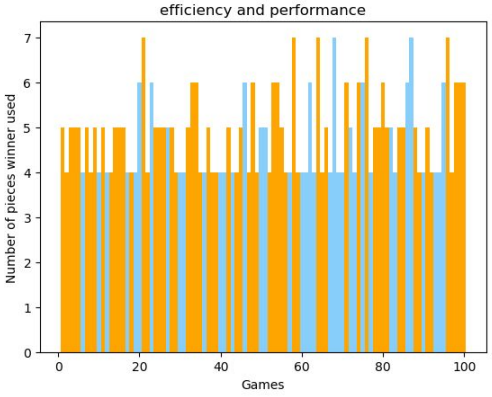
Board size = 4



Board size = 5



Board size = 6

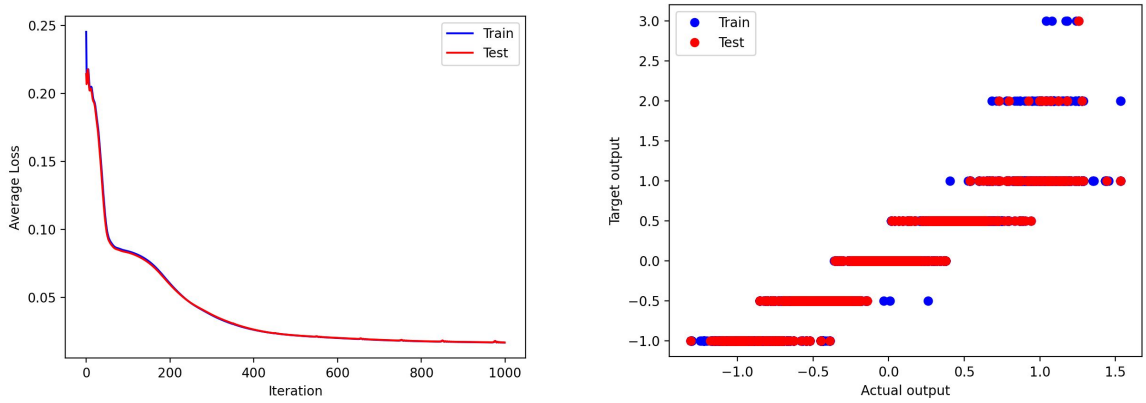


Board size = 7

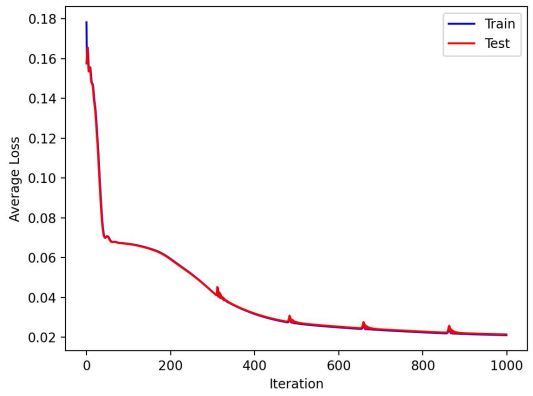
Yiwen Cheng's Results :

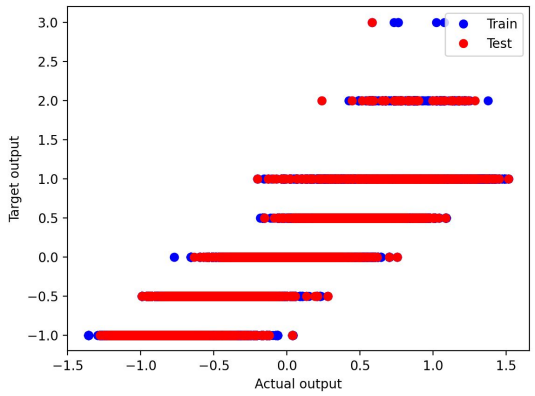
In my experiment, there is an input layer, two hidden layers and one output layer in my neural network. In the input layer, there are 3*N*N units where N is the board size. In the hidden layers, the number of layer units are 10 and 20 and for the output layer there is 1 unit. Each layer is linear type and the activation function I used is sigmoid function.

Learning Curve and Datapoint

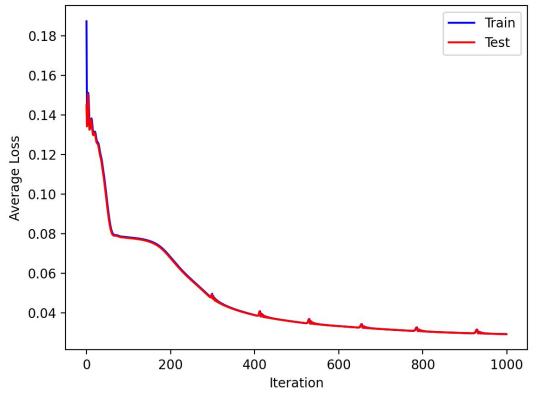


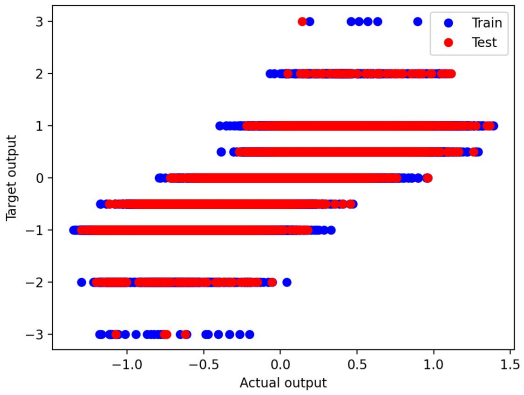
The result in the figure above uses a board size of 3. Winning condition is 3 and iteration time is 1000.



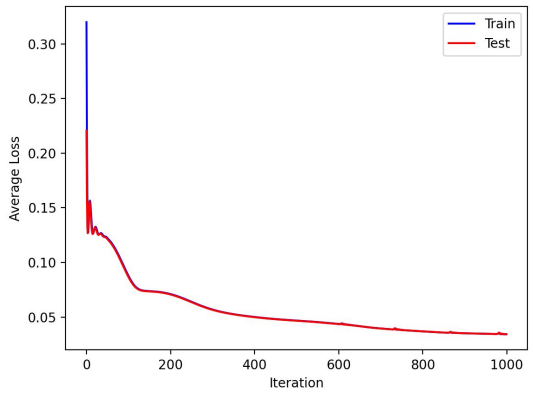


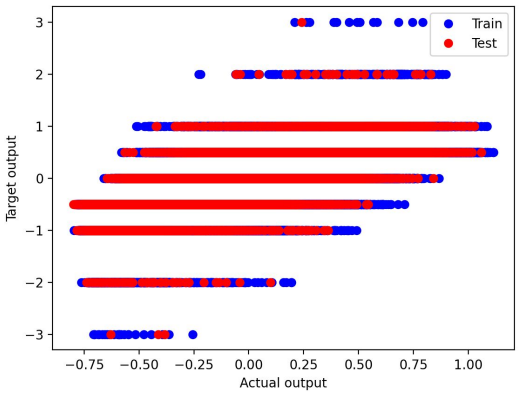
The result in the figure above uses a board size of 4. Winning condition is 3 and iteration time is 1000.



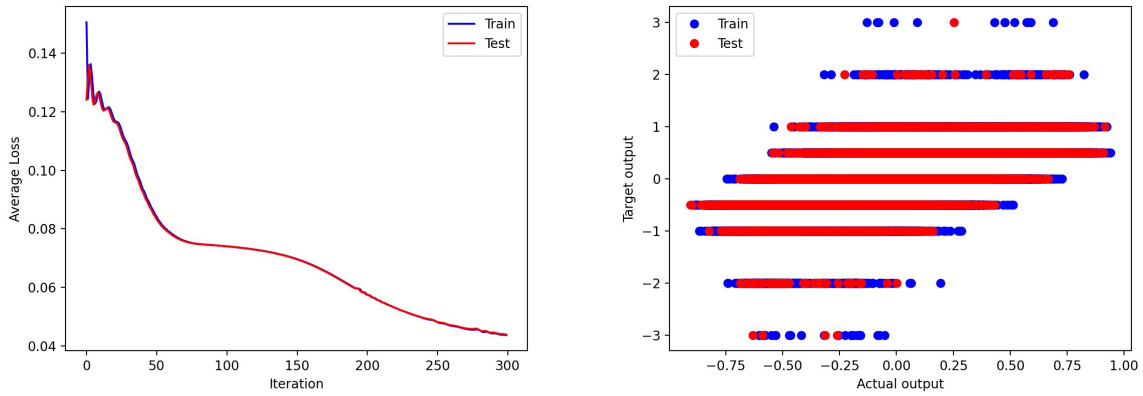


The result in the figure above uses a board size of 5. Winning condition is 3 and iteration time is 1000.





The result in the figure above uses a board size of 6. Winning condition is 4 and iteration time is 500.

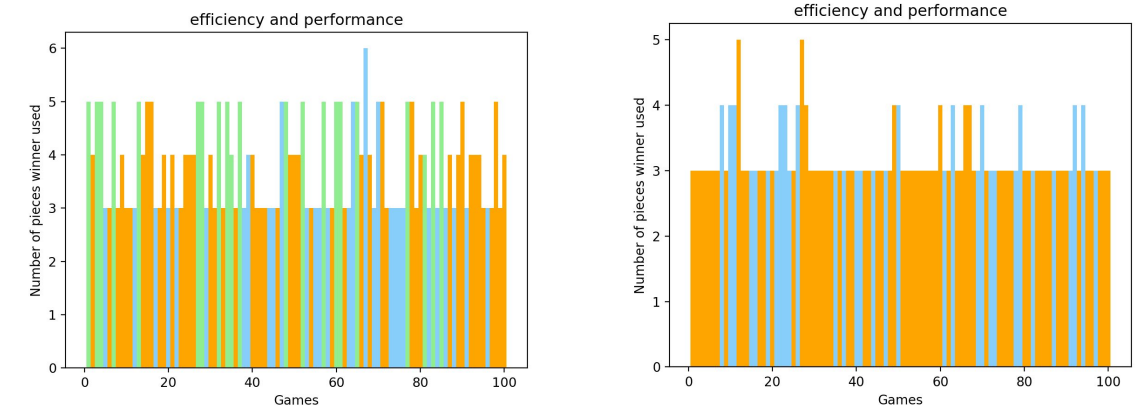


The result in the figure above uses a board size of 7. Winning condition is 4 and iteration time is 300.

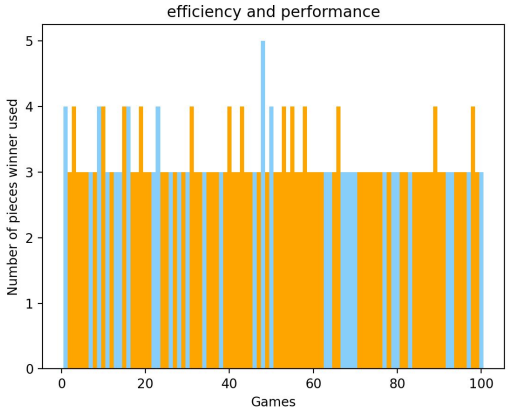
Final score bar graph

For each problem size we conduct 100 games between our tree-based AI and tree+NN-based AI. We let the tree+NN-based be the first player and have our result shown in the table below.

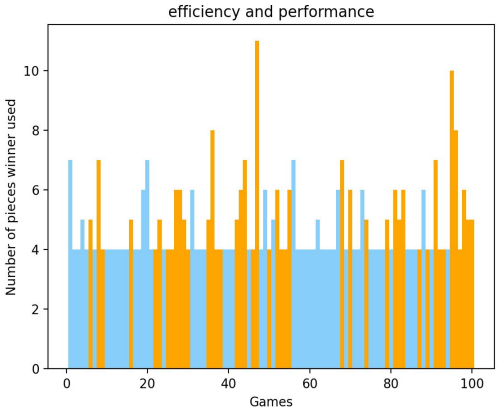
Size(goal) Result	3*3(3)	4*4(3)	5*5(3)	6*6(4)	7*7(4)
Tree-based AI win	30	32	31	51	45
Tree+NN-based AI win	49	68	69	49	55
Draw	21	0	0	0	0



Broad size = 3

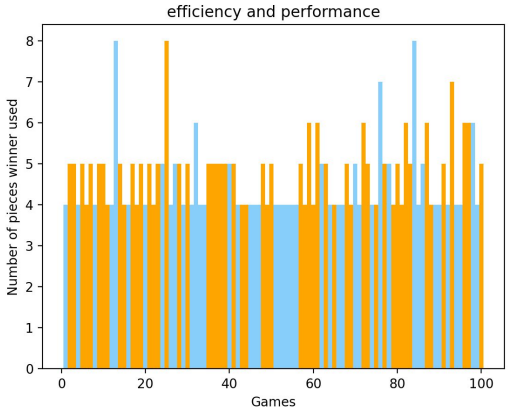


Broad size = 4



Broad size = 5

Broad size = 6



Broad size = 7

According to the figure we acquired, the blue bar means tree-based AI wins, the green bar means draw and orange bar means tree+NN-based AI wins. The x-axis is the number of the game and y-axis is how many pieces the winner used to win the game.

When broad size is 3, Tree+NN-based AI wins most of the games and it seems that it uses less pieces. Besides, there even exists draw when size is 3. When broad size is 4,5 and 7, Tree+NN-based AI still win most of the games. When the broad size is 6, I run 500 iteration times. Tree-based AI and Tree+NN-based AI win a similar number of games. They both use less pieces. Although the performance is not very stable, Ai still won more games. Maybe the reason is that it' s hard to determine proper interaction times.

References:

[1] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020)

[2] Van Rossum, G. (2020). The Python Library Reference, release 3.8.2. Python Software Foundation.

[3] Melkó, Ervin, and Benedek Nagy. "Optimal strategy in games with chance nodes." Acta Cybernetica 18.2 (2007): 171-192.

[4] Paszke, Adam, et al. "Automatic differentiation in pytorch." (2017).

[5] Paszke, A. et al., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach et al., eds. Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024–8035.

[6] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[7]Han, Jun; Morag, Claudio (1995). "The influence of the sigmoid function parameters on the speed of backpropagation learning". In Mira, José; Sandoval, Francisco (eds.). From Natural to Artificial Neural Computation. Lecture Notes in Computer Science. 930. pp. 195–201. doi:10.1007/3-540-59497-3_175. ISBN 978-3-540-59497-0.

[8]Weisstein, Eric W. https://mathworld.wolfram.com/HyperbolicTangent.html