

# Playing Draughts using Neural Networks and Genetic Algorithms

Thien Nguyen

Department of Computer Science  
Durham University

January 22, 2018

# Outline

Problem Description

Motivation

Related Work

Approach

Curent Progress

Conclusion

# Why Draughts?



Figure: Checkers Is Solved in 2007 thanks to this guy. [4]

# Problem Description

A problem in Computer Science (..before AlphaZero existed)

Presently, A variety of Draughts AI players tend to be designed to play at a fixed ability.

While it has produced very competitive and intelligent players, they require human intervention in order to improve their performance.

By combining Neural Networks and Genetic Algorithms, this issue could possibly be solved by creating a player that can grow in ability over time, without the dependency on move-banks.

# Motivation

Why have I chosen to tackle this?

- ▶ Enjoyed AI Search Submodule
- ▶ Interested in seeing whether genetic algorithms are still relevant
- ▶ Interested in Machine Learning (unfortunately not an option this year!)
- ▶ ..I like board games

# Related Work

Similar works of art but no cigar

## Samuel (59')

Uses Genetic Algorithms to improve coefficients of a set of heuristics to evaluate Draughts games. [3]

## Blondie24 (97')

Uses an Evolutionary Algorithm and Neural Networks to evaluate Draughts games. (Quite similar!) [1]

## Giraffe (15')

Uses contemporary machine learning techniques to train a Neural Network to evaluate Chess games. [2]

## Uber (A few days ago)

Uses genetic algorithms to train convolutional neural networks to play Atari Games. [2]

# Current Approach

How will I tackle this; in a nutshell

- ▶ Evaluate a checkerboard state
- ▶ Choose the best move for a given state
- ▶ Generate a population of agents
- ▶ Determine good agents from the population
- ▶ Make better agents from the good ones

# Evaluating Checkerboards

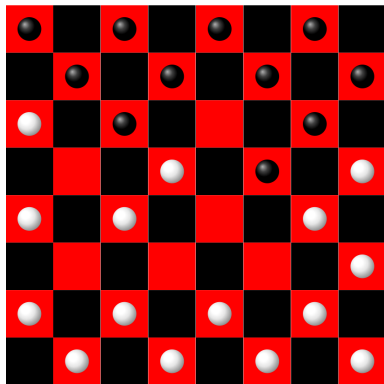
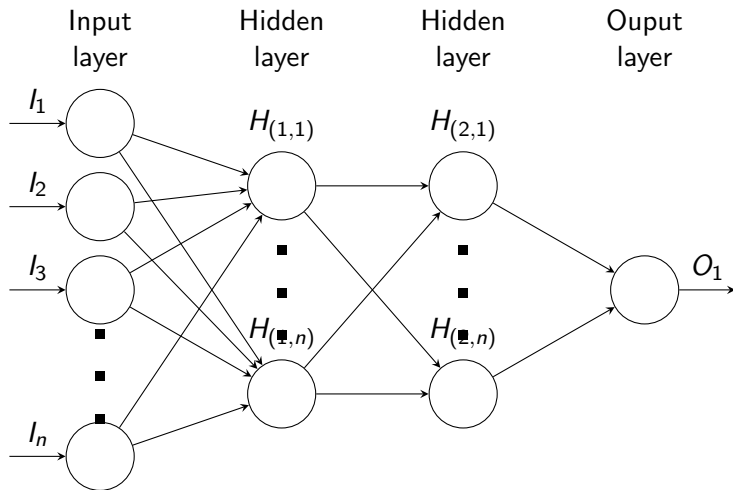


Figure: An example state of a checkerboard.

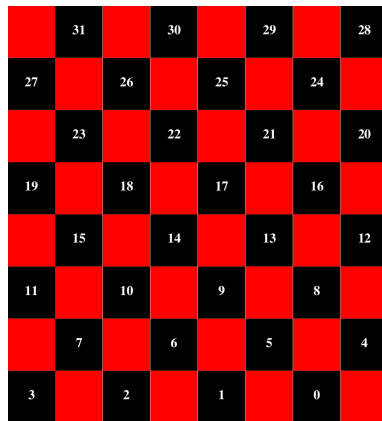


# Neural Networks



$$Output = ActivationFunction((Input * weight) + Bias)$$

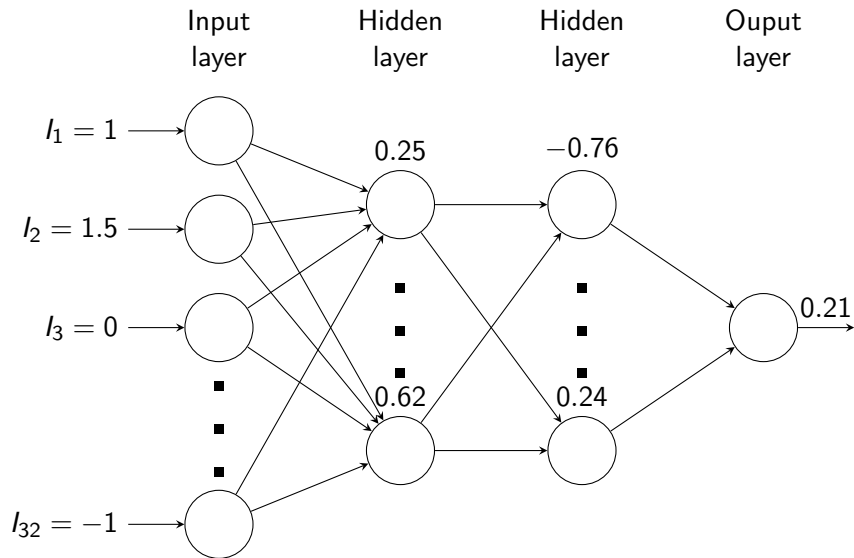
# Checkerboard



	31		30		29		28
27		26		25		24	
	23		22		21		20
19		18		17		16	
	15		14		13		12
11		10		9		8	
	7		6		5		4
3		2		1		0	

**Figure:** The indexes of the 32 pieces of the input layer are the immediate values of the positions on the board.

# Neural Networks



# Activation Function

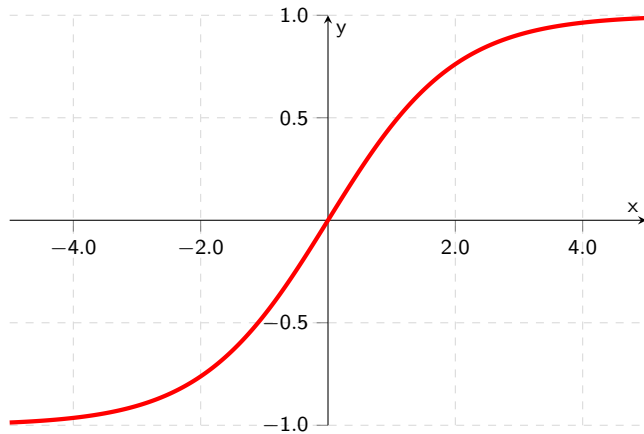
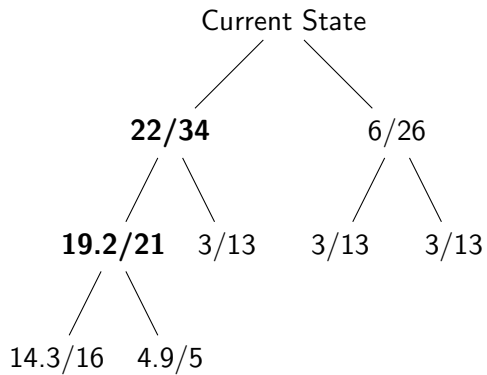


Figure: graph of tanh function  $f(x) = \frac{2}{1+e^{-x}} - 1$

## Choosing moves



# Monte Carlo Tree Search

A grossly simplified algorithm:

- ▶ Generate a set of possible moves  $m$  that can be made from some state
- ▶ Choose random move  $k \in m$ .
- ▶ Simulate a random, theoretical game from this;
- ▶ Keep choosing random moves  $n$  amount of times
- ▶ after  $n$  moves, evaluate using our evaluator to make  $x$ !
- ▶ Increment the number of simulations taken from  $k = i$
- ▶ Increment the chances of a good move from  $k$  by  $x$ .
- ▶ When satisfied, choose  $k$  with the biggest  $x/i$ !

The intuition behind MCTS is that it leans towards better moves probablistically.

# Generating Agents

An agent is a generated set of weights for the neural network.



# Tournament

1. Generate a population of random agents
2. Make agents play each other
3. Order agents by the amount of points scored
4. The best few agents are chosen to stay on for the next tournament
5. make new agents from those best few
6. the losers are destroyed
7. repeat step 2-6 with the new agents until satisfied

# Crossover Mechanism

Two offsprings would be created from a pair of parents, with each offspring being the reciprocal crossover of each other. The weights of both parents (now each treated as a 1D array of coefficients), are divided contingent on the number of weights and biases for a given layer. Each layer should be treated separately to reduce the potential dependency on a purely randomly generated neural network. For each set of weights in a given layer, the following algorithm represents the crossover process:

# Mutation

Weight and biases of an agent's neural network will increment by a random value that is created using the following formula, where  $WeightP$  is the current weight,  $K$  represents the number of weights and biases in the neural network, and  $m$  representing a random floating point in the range of  $[-1,1]$ :

$$WeightN = WeightP + \frac{m}{\sqrt{2 * \sqrt{K}}}$$

The weights, as explained earlier will have a soft maximum of  $[-1, 1]$ . This would consequently mean that the mutation is not controlled, and dependent on the number of weights in the system; The more weights in the network implies a less significant mutation.

# Evaluation

How will I judge my outcome?

- ▶ Will be used to play against human players on popular checkers websites
- ▶ Measurements held to measure evolution progress

# Current Progress

What have I done already?

- ▶ The initial set up is ready
- ▶ It plays relatively well (anecdotaly)
- ▶ Not necessarily effective at end game performances...

# Remaining Work

What do I still need to do?





- ▶ Squash Bugs that could impede performance/provide false positives
- ▶ Measure the effectiveness of genetic algorithms
- ▶ Optimise training (enforce threefold repetition)
- ▶ Train the system for a bit longer
- ▶ Measure the system's performance against other players

Fin.

Any Questions?

# References

Nanos gigantum humeris insidentes

-  Kumar Chellapilla and David Fogel.  
Evolving Neural Networks to Play Checkers without Expert Knowledge.  
*IEEE Transactions on Neural Networks*, 10(6):1382–1391, 1999.
-  Matthew Lai.  
Giraffe: Using Deep Reinforcement Learning to Play Chess.  
(September), 2015.
-  A L Samuel.  
Some studies in machine learning using the game of checkers.  
*IBM Journal of Research and Development*, 44(1.2):206–226, 2000.
-  Jonathan Schaeffer and Robert Lake.  
Solving the Game of Checkers.  
*Games of No Chance*, 29:119–133, 1996.