# Playing Draughts using Neural Networks and Genetic Algorithms

Thien Nguyen

Department of Computer Science
Durham University

January 22, 2018

# Outline

# 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 coefficents 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
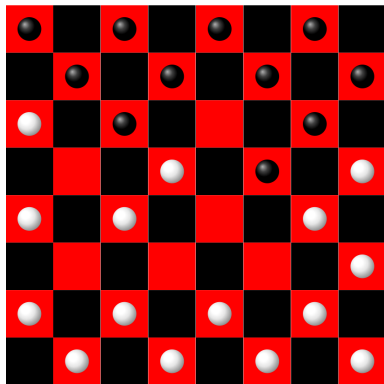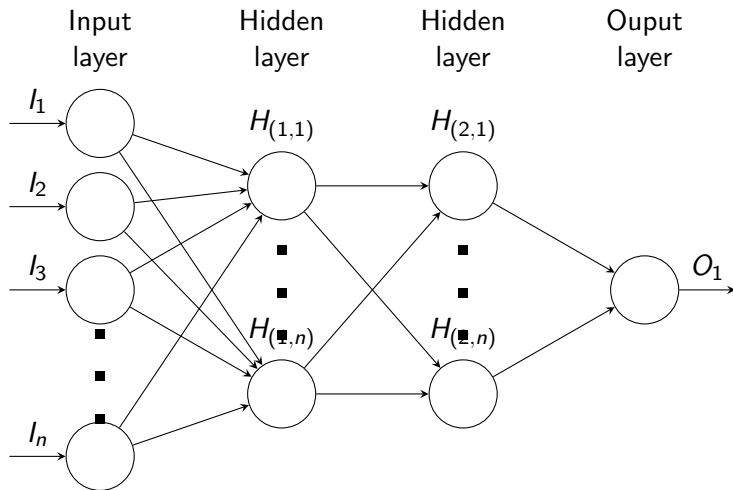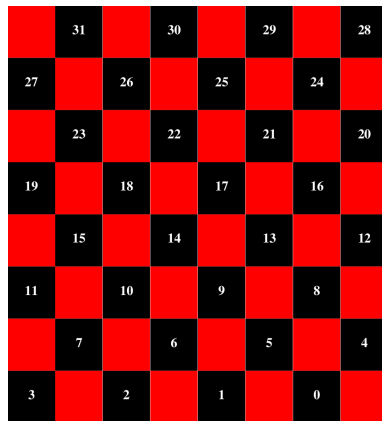- Make better agents from the good ones

# Evaluating Checkerboards



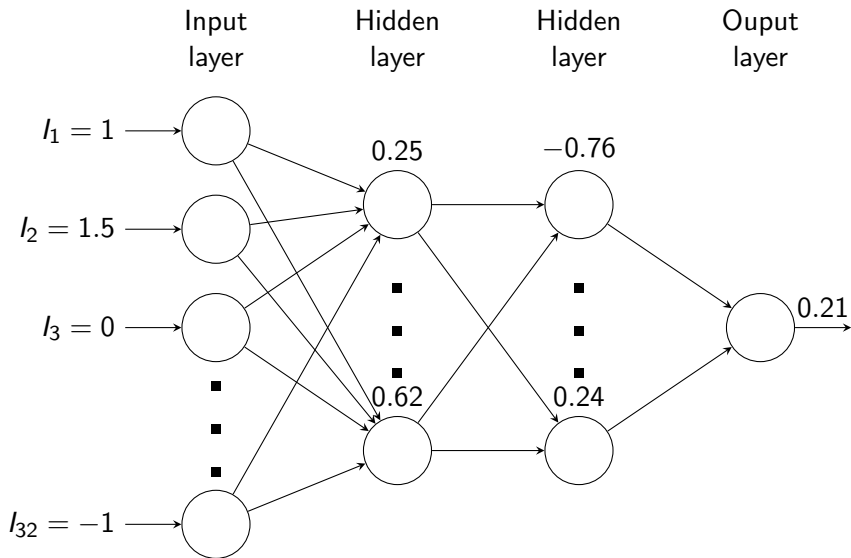Figure: An example state of a checkerboard.

# Neural Networks

# Checkerboard



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

Neural networks use activation functions to simulate an output of a node given a set of input(s).
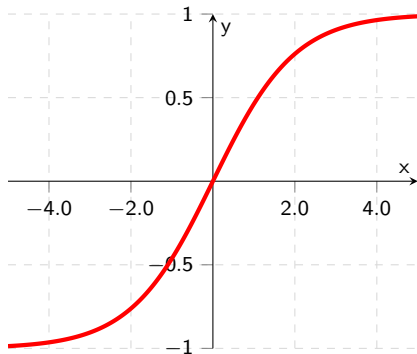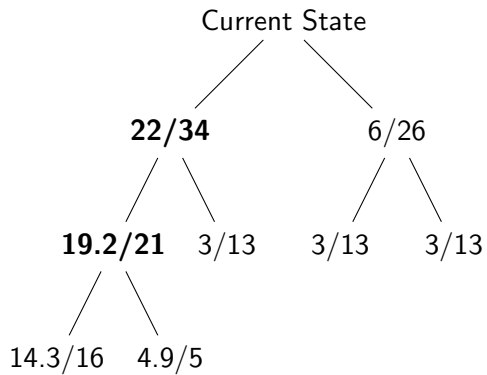
$$O = f((Input * weight) + Bias)$$



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

### Agent

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

- An initial agent would have a randomly generated set of weights and biases
- initial values range from -0.2 to 0.2.

# 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 them
6. The losers are destroyed
7. Repeat step 2-6 with the new agents until satisfied

# Crossover Mechanism

$$P_1 = [\ 1\ 2\ 3\ 4\ 5\ ] \quad P_2 = [\ 6\ 7\ 8\ 9\ 10\ ]$$

$$O_1 = [\ 1\ 7\ 3\ 9\ 5\ ],\ O_2 = [\ 6\ 2\ 8\ 4\ 10\ ]$$

Offspring agents are created from a pair of parent agents. Each offspring is a reciprocal crossover of each other. Weights here are treated as 1D coefficents.

# Mutation

*WeightP* is the current weight, *K* represents the number of weights and biases in the neural network, and *m* represents a random floating point in the range of [-1,1]:

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

Mutation influence is 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?

- Measurements held to measure evolution progress
- Play final agent against agent from prior stages to indicate whether it has been learning
- Play against a randomly playing bot to see it is actually thinking
- Play against different move making algorithms (Vanilla MCTS, $\alpha\beta$ MiniMax)
- Also play against human players on popular checkers websites

# Current Progress
What have I done already?

- The initial set up is ready*
- It has been trained*
- It plays relatively well (anecdotally)
- Relatively ineffective at end game performances...

# Remaining Work
What do I still need to do?

- ▶ * Squash potential bugs that could impede performance/provide false positives
- ▶ Experiment with different crossover and mutation mechanisms
- ▶ 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