Evaluating Approaches to Evolving Neural Reversi Players

Anna Porter and Rob LeGrand, Ph.D.

Abstract

Reversi is a two-player, zero-sum, strategy board game whose complexity lies between that of checkers and chess. Our research seeks to create Reversi-playing agents by using a genetic algorithm to evolve weights of a neural network. Agents rely on their neural network to make their decisions for moves of a game. In this paper we compare different styles of evolution and various settings for the neural network.

*****

Introduction

Reversi Introduction

The game Reversi, also commonly known as Othello, is a two-player, zero-sum game. It is played on an 8×8 board with 64 identical game pieces. The pieces are double-sided with a white side and black side. The game begins with the center four positions filled with white and black tiles in a diagonal fashion. The black player moves first. New game pieces may only be placed on an empty space.

A newly placed game piece will flip enemy tiles if there is an unbroken sequence of enemy tiles between the newly placed piece and an existing friendly piece. Tiles may be flipped in any of eight directions; up, down, left, right and diagonals. Tiles flip in multiple directions if there are unbroken chains in multiple directions. A player may not pick and choose which directions or which tiles to flip. At any point in the game, a move is only valid if it flips enemy tiles. This means that turns may be passed back to the opposing player if no moves are available. However, voluntary passing of turns is not allowed.

The game ends when either the board is full or both players have no more legal moves to make. The final score is calculated by counting the number of tiles controlled for each player. The player with the most tiles wins the game. A tie results if the game ends with both players controlling the same number of game pieces.

Neural Networks

Artificial Neural Networks (ANNs) are mathematical representations of the human brain. They consist of neurons and weights. Each neuron takes real-valued inputs from the previous layer and multiplies these inputs by given weights. It then sums those products and adds a bias weight. Next, the neuron applies an activation function to the final sum. If this calculation results from a hidden neuron, it will be passed to the next layer as a new input. If the output neuron performed the calculation, the result will be the final value calculated from the neural network.

This structure is capable of approximating nonlinear, multivariate functions, making it useful for a variety of problems.

Genetic Algorithm

A genetic algorithm is a computational representation of the evolution that occurs in nature. An initial population will be generated using a set of guidelines. Organisms are represented by collections of genes. Genes may correspond to weights of a neural network. Each organism in the population will be evaluated by a fitness function. The ability of an organism to accomplish an assigned task is directly proportional to its fitness value. Based on these fitness values, parents will be selected and crossover between their genes will occur. Neural networks with more effective weight combinations will tend to pass more of their weights to the next generation. The resulting children from the crossover will be used as the next generation and the process will repeat itself for a certain number of generations. According to the "survival of the fittest" principle, the skill of organisms should improve in subsequent generations if conditions are appropriate.

Related Works

The complexity of Reversi is higher than checkers, but lower than chess (Chong et al., 2005). Due to this fact, Reversi is the topic of many research endeavors. We will be using an approach similar to the configuration used by Chong et al. Their most interesting variation was the addition of a spatial preprocessing layer. They were only able to achieve master level play using a spatial preprocessing layer; the agents lacking this aspect were much more difficult to train and did not achieve levels of play comparable to the spatial neural networks (2005). Our approach chooses to exclude this type of layer, to determine how much spatial information can be learned by a basic neural network.

Our strategy is also like Tuponja and Šuković's, however they used the popular puzzle game 2048. The primary difference in their approach was the evolution style. These networks were also allowed to change in shape during the evolutionary process (2016). This approach would be an interesting technique to explore in future work.

Shahzad et al. compared the level of play achieved by different evaluation functions. They included a standard Weight Piece Counter (WPC), Multilayer Perceptron Networks (MLP), Temporal Difference Learning (TDL), and a Monte Carlo algorithm using Tournament Play Technique. Of the evaluation functions examined, MLP, the strategy used in our work, was found to achieve the highest level of play (2012). Their results were encouraging for the purposes of our project, as we will be applying MLPs within our agents.

Festa and Davino proved fairly strong play can be achieved by using a minimax algorithm with a strong evaluation function (2013). Adding a minimax algorithm to our existing techniques could lead to further improvement of the skill of our organisms, but we have chosen to exclude it.

A large source of inspiration for our research project was work by David Fogel (2002). In his book, he discusses using genetic algorithms to train a neural network with a spatial preprocessing layer for the game of checkers. Their agents also used a minimax algorithm to search for optimal board states. Their agent Blondie24 achieved master levels of play, defeating Chinook, the contemporary computer checkers champion. Fogel is careful to create checkers players without using any predefined human knowledge, which is a principle our work tries to follow as well.

Methodology

The goal of our experimentation is to create a strong Reversi player who utilizes a neural network to make its decisions throughout the game. The player will analyze each available move according to a heuristic value calculated by its feedforward neural network. The inputs for the initial layer of the neural network are generated from the Reversi game board. Inputs will be either −1, 0 or 1 representing an enemy tile, an empty tile, and a friendly tile respectively. The highest heuristic value corresponds to the best move to choose. Similarly, poor move choices will have low heuristic values. Neural networks differ by their assigned or evolved weights. Finding a good combination of weight values is critical to creating a good player.

To increase the skill of our players, we utilize a genetic algorithm to generate a population of organisms to evolve these weights. Organisms are assigned a fitness score based on the outcomes of games played against their neighbors. Based on the fitness score, parents are selected from the existing organisms to crossover (mix) their genes and create a new generation of players. In our case, we consider the weights of the neural network to be the genes. Every 1,000 generations, our generated players' skill level is analyzed by playing every organism against every organism from another population. The opposing population is either the first generation or a subsequent increment of 1,000 generations. Pseudocode representing this process is shown in Figure 1.

```
currentPopulation ← Generate initial population
For whichGeneration ← 1 to numGenerations:
    For each organism in currentPopulation:
        Play Reversi against 6 neighbors as Black
    For each organism in currentPopulation:
        Select parents using fitness score based on Reversi results
        Crossover parent genes
        Mutate genes
        Create a new organism in newPopulation using genes
    currentPopulation ← newPopulation
    If  whichGeneration mod 1000 = 0:
        Save population for later evaluation
```

Fig. 1: Evolutionary Process Pseudocode

In these experiments, the initial generation is produced with no prior knowledge of the game. Established human strategies will not be given to the organisms in any way. Any appearance of intelligence they display is exclusively from the results of the genetic algorithm. A popular approach to increase skill level is to include a spatial preprocessing layer to provide organisms with more knowledge of the spatial relations of the board (Chong et al., 2005). However, our experiments will exclude this layer to see if our agents can learn spatial information on their own.

Vanilla Configuration

To begin our project, we settled on a basic set of parameters: a "vanilla" configuration. Features of the vanilla configuration include the following.

Games will be played on a Reversi board with size 8×8, with the center 4 tiles occupied by 2 black and 2 white tiles in a diagonal fashion. Neural networks will have 64 inputs, a single hidden layer of 8 neurons, and a single output neuron. Weights are generated at the start of an evolution by using a normal distribution with a mean of 0 and standard deviation of 1. The activation function used by the neural network will be softplus: $f(x) = \ln(1 + e^{x})$.

The final output neuron will not apply the activation function to its calculated sum. These networks will be exclusively feedforward ANNs, meaning there will be no cycles within the neural network. To generate the inputs given to our neural networks, if a space is occupied by a friendly tile, the value of the corresponding input is 1. For enemy tiles we use -1 and for empty tiles 0.

The size of a population is 10×10 arranged in a hexagonal pattern. Each organism plays 12 games, 6 as white and 6 as black, against its 6 neighbors. Organisms along the edge play organisms along the opposite edge. This creates a torus-shaped population. Organisms will only be allowed to search one move ahead of the current board state, thus no minimax algorithm is implemented within their gameplay. To rate our organisms, each is assigned a fitness score during this stage. To calculate the value, we use a combination of wins, losses, ties, and the number of pieces controlled by a player at the end of a match. Wins are awarded 64 points, losses earn 0 points, ties are awarded 32 points, and the number of pieces controlled at the end of the game is added to this sum. In this scheme, game result is most important, but emphatic wins count more than close ones.

Each organism breeds with another organism to produce a child organism. In the vanilla configuration, parents are selected from the 6 surrounding neighbors using assigned probabilities based on their fitness scores. The lowest-scoring neighbor is thrown out and its fitness score is subtracted from the remaining neighbors' fitness scores. This new fitness value is directly proportional to the probability it will be selected as a parent.

New organisms are created by crossing over the genes of the two selected parents. For each gene, there is an equal probability it will come from the mother or the father. Mutation occurs for every gene and will add a random number generated using a normal distribution with a mean of 0 and a standard deviation of 1. These organisms will go on to play games, restarting the process and creating another new generation.

Variations on Vanilla

Of these properties, several settings will be varied for each future configuration. Each experiment will only have a single change from the vanilla configuration, so as to isolate the variable and make direct comparisons to vanilla.

Activation Functions:

1. Rectifier: $f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$
2. Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$
3. Softsign: $f(x) = \frac{x}{1+|x|}$
4. Threshold: $f(x) = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$

Many activation functions can be used with neural networks. For our neural networks, in addition to the vanilla softplus, we have chosen to try using rectifier, softsign, sigmoid, or threshold activation functions. Changing the activation function results in a slight change to the output summations of each neuron. Different activation functions generate different levels of success and some activation functions are more appropriate for some problems than they are others. Notice that rectifier and softplus have an infinite range and sigmoid, softsign and threshold have a finite range.

Mutation Types:

5. Uniform Shake

6. Cauchy Shake
7. Normal Shake and 1% Replace

We will evaluate several types of mutation for the breeding stage of evolution. Instead of adding a number to each weight from a normal distribution, Variation 5 will use a uniform distribution between $-\sqrt{3}$ and $\sqrt{3}$. These values were chosen to get a similar mean and standard deviation to the other distributions. Variation 6 also adds a random number, but it is generated using a Cauchy distribution with an x_0 of 0 and a γ scale parameter of 1. A 1% replace mutation may be applied as well. This mutation re-initializes a gene with a 1% probability, replacing it with a randomly generated number using the same uniform distribution mentioned previously.

Network Shape:

8. 1 Neuron
9. 8-3-1 Neurons
10. 16-4-1 Neurons

Neural networks may vary by shape. The most basic shape is a single output neuron, which will be referred to as 1N. In this case, 64 weights approximate the value of owning the tile corresponding to the weight. The 65th weight is a bias weight. Recall the default shape is 8 neurons in a hidden layer, with one neuron at the output layer. This shape will be referred to as 8-1N. Once neural networks begin to have hidden layers, relationships between weights and the board are less intuitive since a single weight could be used for more than one calculation. It is also possible to have multiple hidden layers, and we have chosen two shapes from this category to test. 8 neurons in the first hidden layer, 3 neurons in the second hidden layer, with a single output neuron will be referred to as "8-3-1N". "16-4-1N" similarly represents 16 neurons in the first layer, 4 neurons in the next layer, and a single output neuron. It is significant to note that larger neural networks will have more weights, be capable of learning more interesting features, but they will also take more generations to arrive at these conclusions.

Parent Selection:

11. Father 7 Fitness Probability
12. Both 7 Fitness Probability
13. Best Father 6
14. Best Father 7
15. Best Both 7

Parent selection has also been chosen as a variable to consider. Figure 2a shows the surrounding bachelors with their raw fitness scores. Adjusted fitness scores are shown in figure 2b. We use those adjusted scores proportionally to generate probabilities that the corresponding bachelor will be selected. Some of our parent selection variations below will choose from 6 neighbors, as does vanilla. Some consider 7, including the 6 neighbors and the center organism.

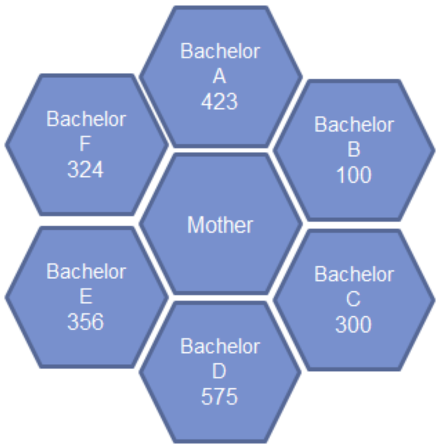


Fig. 2a: Raw Fitness Scores

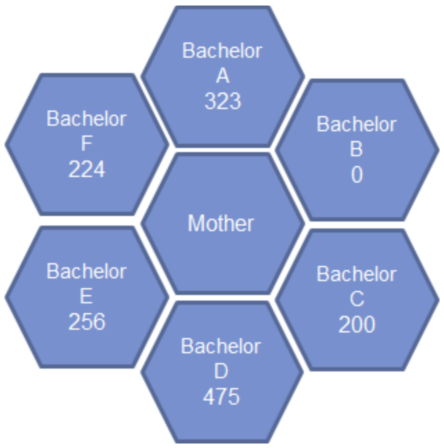


Fig. 2b: Scaled Fitness Scores

Fig. 2: Parent Selection Example

Variation 11 will select the father from all 7 possibilities using a probability directly proportional to their fitness scores. If the center organism is chosen as the parent, the organism will have two identical parents. Variation 12 will choose both parents according to the fitness probabilities; however, the parents chosen will not be allowed to be the same organism.

Variations 13, 14, and 15 are deterministic parent selections. Variation 13 chooses the best father, based on fitness score, from the 6 neighboring organisms, with the mother in the center. Variation 14 chooses the best father from all 7 options, with the mother in the center. Variation 15 takes the top two organisms from the 7 possibilities.

Results

Each variation was evolved for 10,000 generations and populations were saved and evaluated. Every 1,000 generations, each organism in the population plays each organism from the first generation. Winning percentages were calculated by adding the number of wins to half the number of ties and dividing by the total number of games played. Our goal was to measure the *progress* made by an evolution from its first, random generation.

The evaluation approach previously described was found to be a bit unfair. The skill of one of our initial populations was especially poor and resulted in the Best Both 7 parent selection becoming an outlier for the first run. To attempt to achieve a fairer comparison, we found an improved evaluation method. First, we created a much larger population of opponents and play each 10,000th generation against this *baseline* population of size 50×50. Each organism within the baseline population uses the vanilla configuration and is newly initialized without experiencing evolution. Winning percentages were calculated using the same strategy described above. We used this new evaluation method for all graphs below.

These experiments were repeated for a second run from a newly randomly initialized population for another 10,000 generations and evaluated using the baseline method described above.

Vanilla, 6 Times

We chose to run the vanilla configuration 6 separate times. The initial populations are randomly generated, so each of these evolutions will begin at a different starting point. The mutation along the way is also random, so one thread of evolution will never be the same as another. Results are very closely related. At the final generation, all populations achieved winning percentages against the baseline within an approximate range of about 3%. Each run experiences a large amount of progress from 0 to 1k, and subsequent generations cluster around similar winning percentages. The amount of variation we see among the 6 vanillas is due entirely to randomness. We will need to see larger differences in comparisons of the variants to be convinced that one is in fact better than another.

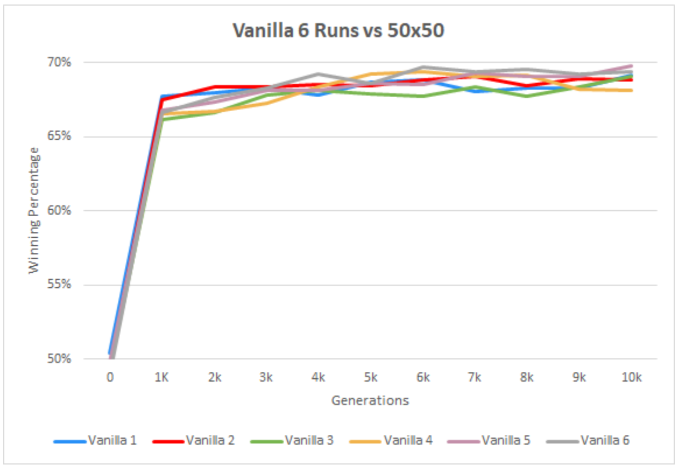


Fig. 3: 6 Vanilla Runs Compared to Baseline Population

Activation Functions

Activation functions with similar shapes performed relatively similarly throughout their evolutions. It seems that activation functions which can output arbitrarily large numbers have done better than functions which restrict their output values to a small range. The grouping of similarly shaped activation functions holds true for all graphs and the direct comparison of activation functions. Sigmoid appears to perform the worst. This was a surprising result, as sigmoid has been typically the standard activation function used with neural networks.

At each generation, softplus had the highest or second highest winning percentage, even when compared to the baseline population. When each 10,000th generation of each activation function were played against each other, the softplus population beat each other population one on one. Due to these properties, our decision to make softplus the vanilla setting was reaffirmed.

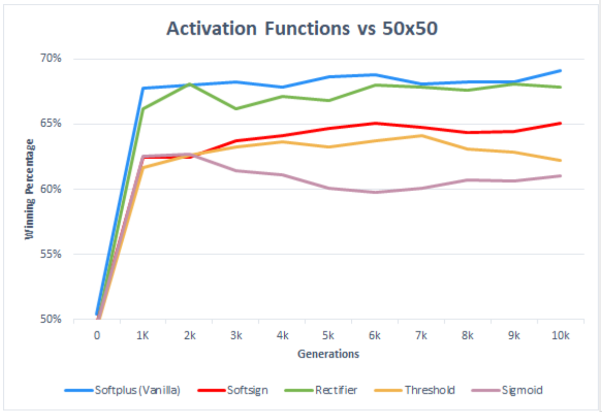


Fig. 4a: Activations Run 1 vs. Baseline

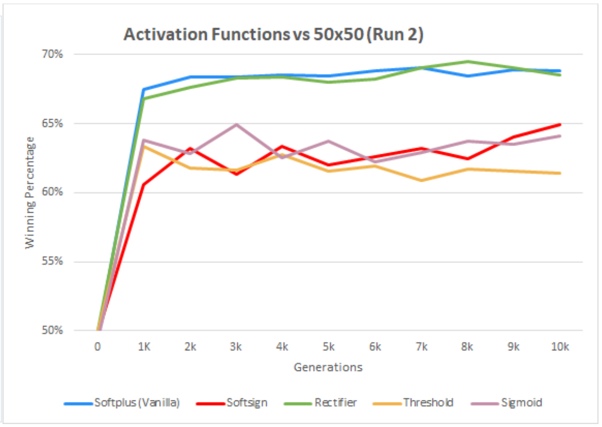


Fig. 4b: Activations Run 2 vs. Baseline

Mutation Types

At first glance, it is clear the Cauchy shake is the most volatile mutation type. This could mean that the distribution's probability of producing exceptionally large or small mutations is too great. When the final generations were played against each other, Cauchy's performance was remarkably poor. It did not come close to beating any other population and lost horribly against the baseline population. However, Cauchy's final population was the worst generation across the board. There could be some interesting ways to harness Cauchy's abilities. The 3 peaks in its performance might mean it can reach more local optima throughout its evolution.

Uniform and normal had similar progress and performance overall. The only difference between variant 7 and the vanilla normal shake is the replacement mutation occurring with a 1% probability. It seems to make the generations perform more poorly.

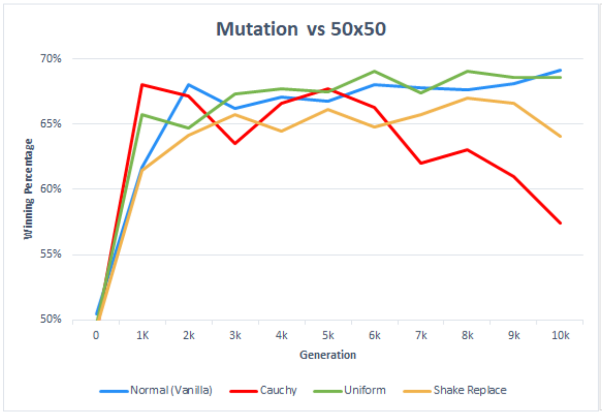


Fig. 5a: Mutation Run 1 vs. Baseline

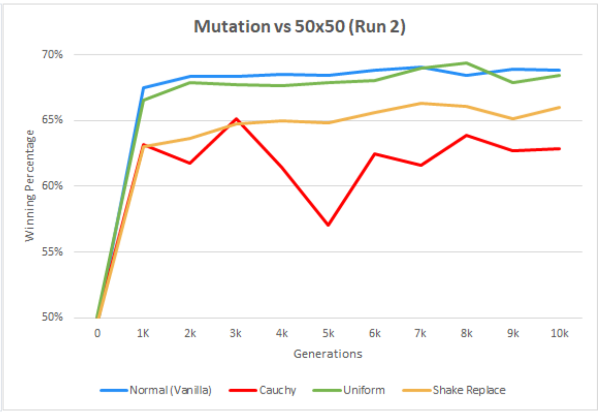
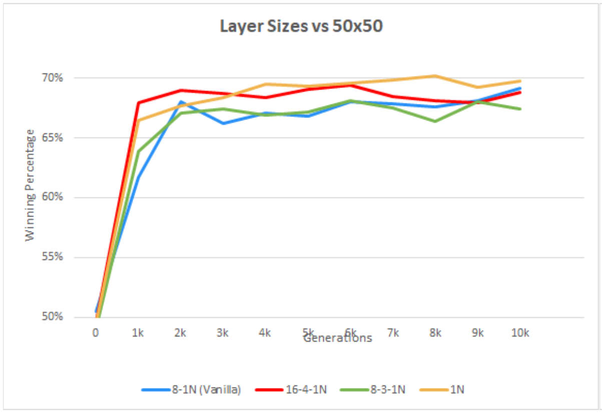


Fig. 5b: Mutation Run 2 vs. Baseline

Neural Network Shapes

Despite having varying numbers of layers and neurons, most of these variations made similar progress, with each of their generations falling within an 8% winning percentage range. When the 10,000th generations were played against each other, differences were more evident. 1 Neuron performed the best, and except for variant 10, larger neural networks did poorer than small neural networks. This could mean larger neural networks need more time to be trained, since they have more weights that need to be optimized. The relationship between weights is also more complex, another reason more generations may be required.

Within the baseline comparison for run 2, variant 9 did remarkably poorer than run 1, which could indicate that the initial population of the evolution was a poor starting position. Another explanation could be that it was unable to find any local maximum given the mutations that occurred along the way. Running these experiment parameters for additional runs could help to explain the sharp drop in performance.



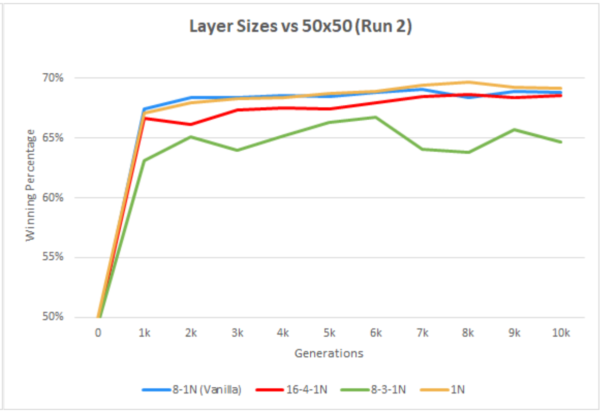
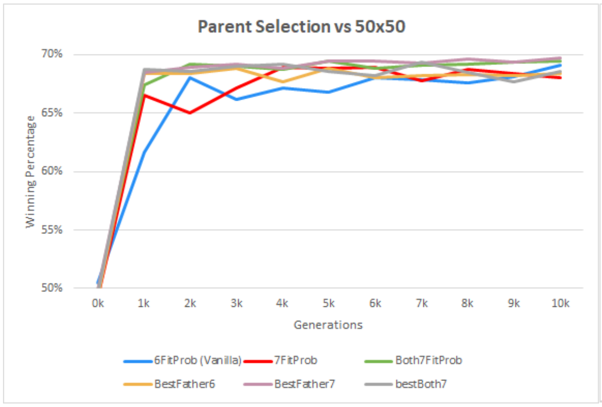


Fig. 6a: Layer Sizes Run 1 vs. Baseline Fig. 6b: Layer Sizes Run 2 vs. Baseline

Parent Selection

Overall, most of the variations of parent selection seem to perform similarly. However, in the *progress* graph for the second run, Best Both 7 was an outlier. Using the baseline population indicated the population was not as skilled as the first run graph suggests. According to the *baseline* results shown below, parent selection does not seem to have a large impact on evolution performance.



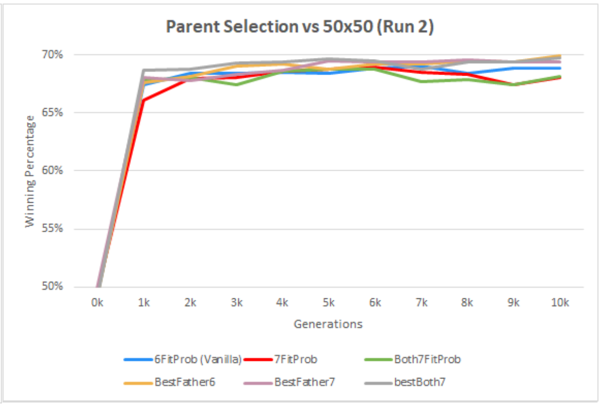


Fig. 7a: Parent Selection Run 1 vs. Baseline Fig. 7b: Parent Selection Run 2 vs. Baseline

Conclusions

- Given our style of input selection, the best activation function to use for this situation was softplus, with rectifier a close second. With our settings, sigmoid had the poorest performance, despite being one of the most commonly used activation functions. Our findings suggest it is better for activation functions not to restrict their output values within a finite range.
- According to our findings, additional layers do not increase performance in the first 10,000 generations. It could indicate that additional generations are needed for larger networks to reach their optimum levels of play. Of course, it may be that some untried combination of settings would be better at finding effective combinations of genes.
- It seems that selecting a distribution with skinnier tails rather than fat tails performs better. We also discovered that a 1% replacement mutation hurts performance overall.
- Parent selection does not seem to be a major factor for performance within the evolutions.

The agents resulting from our generations can beat completely random agents. However, when played against a novice human player or basic computer programs on easy settings, they do not perform well. The amount of information the agents have learned does not amount to intelligence. Since we started with absolutely no hard-coded human knowledge, our results are still significant.

Future Work

There are obviously many combinations of settings which were not analyzed by this research project and there are many other settings which could be varied. Any combination not analyzed by this project could produce interesting results. Settings which were not mentioned could include weight initialization for the first generation of organisms. Other styles of crossover could be produced, such as swapping sections of genes rather than individual genes. Another interesting type of mutation could be swapping two weights within a neural network. Perhaps another type of fitness function would be more successful. Possibilities are potentially effectively endless in this regard.

Since the weights of a single neuron network are quite intuitive, it could be beneficial to apply some human intuition to these weights. A basic principle of Reversi is the fact that corners are extremely valuable and tiles adjacent to the corners are detrimental to own. Using simple principles such as this, weights could be hard coded for an agent and tested against an evolved agent. We could also use similar intuition to add some sort of bias to an initially generated population and begin an evolution using this starting point.

It is possible for the value of a tile to have a different importance for the black player or white player. In our current configuration, a neural network considers the value of a tile for black to be the same value, but negative, for white. It could be beneficial to allow 128 inputs rather than 64 inputs for our neural network. Each tile would have an input corresponding to the value for black and the other would represent the value for white.

Since larger numbers of weights take more evolutionary time, narrowing down the number of inputs could have interesting results. The 8 lines of symmetry of the board could be used to produce 10 weights, rather than 64 weights. Perhaps these networks would be capable of reaching their optimum levels of play more quickly.

One flaw of our fitness function is that it does not take the strength of the surrounding neighbors into account. An organism could have neighbors that play poorly, and its fitness score could be quite high. However, an objectively stronger organism could have lost many games due to the strength of his neighbors, and its fitness score could be quite low. We have not yet tried to balance results based on the skill of the neighbors. To overcome this problem, varying subdivisions of the population could help. Using a strategy like the “critters” cellular automaton, at each new generation, the subdivisions would change. Each organism would play each other organism within its subdivision. If subdivisions from one generation to another were observed at the same time, the subdivisions would overlap. This could help equalize the playing field over time. Alternatively, Colley’s matrix method (2002) or a similar approach could take strength of opponents into account when estimating organism strength.

Currently, the organisms do not seem to be capable of learning large amounts of spatial information of the board. As Chong et al. discovered, it is quite difficult to achieve master levels of play without a spatial preprocessing layer. Adding this layer could strengthen the skill of our players by a large margin (2005).

Works Cited

Chong, Siang, Mei Tan & Jonathan White (2005, June). Observing the evolution of neural networks learning to play the game of Othello. *IEEE Transactions on Evolutionary Computation, Volume 9* (Issue 3), pp. 240-251.

Colley, Wesley N. (2002). Colley's bias free college football ranking method: The Colley matrix explained. Available at `www.colleyrankings.com/method.html`

Festa, Jacopo, & Stanislao Davino (2013, December 5). “IAgo vs Othello”: An artificial intelligence agent playing Reversi. *Proceedings of the Workshop Popularize Artificial Intelligence*, pp. 43-50. Turin, Italy.

Fogel, David B. (2002). *Blondie24: Playing at the Edge of AI.* Morgan Kaufmann, San Francisco, California.

Shahzad, Basit, Lolowah R. Alssum & Yousef Al-Ohali (2012). Selection of Suitable Evaluation Function Based on Win/Draw Parameter in Othello. *Ninth International Conference on Information Technology - New Generations*, pp. 802-806. Las Vegas, Nevada.

Toffoli, Tommaso, & Norman Margolus (1987). §12.8.2. Critters. *Cellular Automata Machines: A New Environment for Modeling*, pp. 132-134. MIT Press.

Tuponja, Boris, & Goran Šuković (2016, November 22-23). Evolving neural network to play game 2048. *24th Telecommunications Forum TELFOR 2016*. Belgrade, Serbia.