Combining Reinforcement Learning With Genetic Algorithms To Produce An Othello-Playing Program

Joshua A. Singer

September 1, 2000

Abstract

I describe a software experiment that combines evolutionary-computation methods with learning methods. The aim is to produce an Othello-playing computer-program superior to that which could have been achieved by either method alone. *Othello*, also known as *Reversi*, is a popular two-player board-game. After allowing a regime of alternating evolutionary and learning phases to run for three months through 900,000 individual matches of Othello, the regime yielded an Othello player that was competitive with my Linux installation's human-designed Othello player. The regime uses the genetic algorithm as its evolutionary method and reinforcement learning as its learning method.

1 Background

1.1 Overview

This paper describes a software experiment whose goal is to combine principles of evolutionary programming and computational learning in order to produce a program that can play Othello well. (*Othello* is a two-player board-game whose rules are described briefly below, and in detail at the end of this paper.)

My goal is to start out with a set of Othello-playing programs with random evaluation functions. They will naturally play a horrible game of Othello, but, by subjecting them to a regime whereby their evaluation functions adapt over time as they play against each other, I hope to ultimately end up with a set of programs that play a very good game of Othello.

In order to obtain a set of good players from a set of random players, I combine two different methods: the *genetic algorithm*, and *reinforcement learning*. My hope was that combining them into a single regime might produce an Othello-playing program better than either could have produced separately.

1.2 The Impetus For A Combined Approach

The goal of all learning methods can be characterized as the search for an optimal point within some fitness terrain. In the case of Othello, that fitness terrain is a parameterized space of static evaluation functions, and the point searched for, $\mathbf{u}^*$, is the parameter whose associated function $f_{\mathbf{u}}(\mathbf{x})$ most closely approximates the true value function of the game. In general, a globally optimal point cannot be found, and we content ourselves with methods that can find points that are highly fit (very close to optimal).

There are at least two reasonable ways to search for points with high fitness. One is to use gradient-descent to find a local optimum. This is the method used by reinforcement learning. The other is to exploit an assumed property of the fitness terrain: that points on the terrain can be characterized by more or less independent features whose presence or absence directly determines the fitness of the associated point. Points possessing many good features are highly fit, while points lacking many good features are not. This is the assumption made by the genetic algorithm and the justification for using crossover.

As an example of how crossover works to improve fitness, consider two parameters, $\mathbf{u} = (\mathbf{u}_1, \mathbf{u}_2)$, and $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2)$. If these are crossed to yield the two child parameters $\mathbf{c}_1 = (\mathbf{u}_1, \mathbf{z}_2)$ and $\mathbf{c}_2 = (\mathbf{z}_1, \mathbf{u}_2)$, and if $\mathbf{u}_1$ and $\mathbf{z}_2$ are features that confer high fitness on the parameters $\mathbf{u}$ and $\mathbf{z}$, then the assumption of the genetic algorithm is that the fitness terrain is so structured that $\mathbf{c}_1$ is likely to have even higher fitness than either $\mathbf{u}$ or $\mathbf{z}$.

Crossover has a weakness, though, in that it can't make something from nothing. Given a pool of points with low-fitness, possessed of no useful features, it is unlikely that crossing any two of them will result in a point with improved fitness. Crossover needs to be given a pool of points possessed of useful features, and then it can cross them to yield points with improved fitness.

gradient-descent does not possess this weakness. No matter how unfit a particular point is, it can be used as the starting point for gradient-descent, ultimately leading to a locally optimal point. Furthermore, it is a fairly safe guess that this point, being a local optimum, possesses at least one good feature!

Consider then the following scenario: we start with ten randomly chosen points in the fitness terrain. The fitness of each being poor, we don't expect crossover to yield any interesting results. So instead we perform ten gradient-descent searches, yielding ten different local optima. Presumably each of these local optima possesses at least one good feature, and, since they come from different portions of the fitness terrain, it may well be that they possess ten *different* good features. Now crossover has a lot to work with. If we cross pairs of points from this pool of local optima, we are very likely to produce new points of higher fitness than any in the pool.

We therefore have the following situation: given any point that is not already a local optimum, we can use gradient-descent to find a local optimum, yielding a new point with improved fitness. And, given any two points with relatively high fitness, we can use crossover to yield, with high probability, a new point with improved fitness. But if this new point truly has improved fitness over its parents, then it can't reside in the search space near either of its parents, since its parents are local optima. Therefore, we can use gradient-descent again on this new point to yield a new local optimum with still better fitness. By interleaving the two methods, we hope to produce a sequence of points with ever-increasing fitness.

The impetus then for the combined approach is to achieve a synergistic interplay between reinforcement learning, which finds local optima, and the genetic algorithm, which can combine local optima to produce new points with still higher fitness.

1.3 How Othello Is Played

Othello is played on an 8x8 board. One player plays as White, the other as Black. Players take turns placing a stone of their color on an unoccupied square until all the squares have a stone on them, or until neither player has a legal move. The player with the most stones on the board at the end of the game wins.

Every time a player places a stone, he causes some of the stones that his opponent has already played to "flip" colors. So, for example, every time White plays, some black stones on the board change to white stones, and every time Black plays, some white stones on the board change to black stones. The exact way in which this stone-flipping occurs is covered in the rules for Othello, which can be found at *http://www.othello.org.hk/tutorials/eng-tu1.html*.

The game's final score is $v = \frac{W}{W+B}$. Under this scoring method, .5 is a tie. Scores higher than .5 are wins for White, and scores lower than .5 are wins for Black. In typical game-theoretic fashion, White tries to maximize the score, while Black tries to minimize it.

1.4 Game-Playing And Evaluation Functions

There is one indispensible component to all game-playing programs: the *evaluation function*. The evaluation function is an approximation to its game-theoretic counterpart: the *value function*. To each possible game-state, the value function assigns a value that indicates the outcome of the game if play were to continue optimally, both players making their best possible moves from that state onward. In Othello, a game-state consists of two pieces of information: the board-position, and whose move it is.

At any point during a game, the player whose move it is has several possible legal moves, each leading to a different successor game-state. We denote the set of successor game states of a given state $\mathbf{x}$ by $Succ(\mathbf{x})$. If White knew the value function, he would select the move leading to the highest-valued successor state. Similarly, if Black knew the value function, he would select the move leading to the lowest-valued successor state.

The value function is not discoverable in practice, so instead we approximate it with what we call an *evaluation function*. The evaluation function is obtained by applying the minimax algorithm to what we call a *static evaluation function*: a parametrized function from some function space, $f_{\mathbf{u}}(\mathbf{x}) \in \mathcal{F}$. Here, $\mathbf{x}$ is a vector representing a game-state, and $\mathbf{u}$, called a *parameter*, is a vector of values that select a particular function from the family $\mathcal{F}$. For the experiment, I use neural-nets as static evaluation functions, which means that the parameter $\mathbf{u}$ consists of the weight-values and bias-values that uniquely determine the net.

The evaluation function, which we'll denote $h(\mathbf{x}; d, f)$, evaluates a game-state $\mathbf{x}$ by expanding the game tree beneath $\mathbf{x}$ to some specified search-depth, d, applying f to the terminal nodes of the expansion, and then using the minimax algorithm to trickle these values back up to the top to yield a derived value for $\mathbf{x}$. Specifically, for a given static evaluation function f and a fixed depth d, the evaluation of a game state $\mathbf{x}$ is

$$h(\mathbf{x}; d, f) \triangleq \begin{cases} f(\mathbf{x}), & d = 0 \\ \max\limits_{\mathbf{x}' \in Succ(\mathbf{x})} h(\mathbf{x}', d-1; f), & d \neq 0, \mathbf{x} \text{ is a White-to-play state} \\ \min\limits_{\mathbf{x}' \in Succ(\mathbf{x})} h(\mathbf{x}', d-1; f), & d \neq 0, \mathbf{x} \text{ is a Black-to–play state} \end{cases}$$

1.5 Choosing A Move

When it is a player's turn to move, it must decide which of the successors of the current state $\mathbf{x}$ it should move to. In this experiment, a player makes a weighted random choice, based on the relative values of each move. I use the following exponential weighting scheme. Let $\mathbf{x}'_1 \dots \mathbf{x}'_n$ be the successors $\mathbf{x}$. Then the probability of selecting the move that leads to successor $\mathbf{x}'_i$ is:

$$p_i = \begin{cases} \dfrac{e^{Th(\mathbf{x}'_i; d, f)}}{\sum_j e^{Th(\mathbf{x}'_j; d, f)}} & \mathbf{x} \text{ is a White-to-move state} \\ \\ \dfrac{e^{-Th(\mathbf{x}'_i; d, f)}}{\sum_j e^{-Th(\mathbf{x}'_j; d, f)}} & \mathbf{x} \text{ is a Black-to-move state} \end{cases}$$

where T is a tuning parameter that controls how tight or dispersed the probability distribution is. Higher values of T make it more likely that the best-rated move will actually be selected. The experiment uses a value of $T = 10$.

1.6 Neural-Networks As Evaluation Functions

In this project, I use neural-networks as evaluation functions. Each neural-network has a 192-node input layer, three hidden nodes, and a single output node. The input layer accepts a 192-bit encoding of the game state. When an Othello-player wants to evaluate a game-state, it passes the properly-encoded game-state to its neural-network and interprets the neural-network's output as the result of the evaluation.

Recall that the evaluation functions I start with are random, so initially these neural-networks will have random weights and biases. The goal is to have the neural networks adapt so that ultimately they represent accurate evaluation functions.

Neural-Net Learning Law

In this experiment I use backpropogation networks. When the network is in learning mode and is presented with a training pair $(\mathbf{x}, t)$, it uses the backpropogation learning law to alter its weights and biases. In practice , this means that its parameter $\mathbf{u}$ changes slightly to yield a new parameter $\mathbf{u}'$, with the property that $f_{\mathbf{u}'}(\mathbf{x})$ is closer to t than $f_{\mathbf{u}}(\mathbf{x})$.

Neural-Net Feed-Forward Processing

The activation of the i^{th} hidden node, α_i, is given by:

$$\begin{aligned}\alpha_i &= g(\sum_{j=1}^{192} w_{ij}a_j + b_i), \text{where} \\ g(z) &\triangleq \frac{1}{1+e^{-z}}\end{aligned}$$

and where a_j is the activation of the j^{th} input node, w_{ij} is the weight to the i^{th} hidden node from the j^{th} input node, and b_i is the bias of the i^{th} hidden node. The function g is called a *sigmoid function*. It normalizes the node activations so that they lie in $[0, 1]$.

The activation of the output node, y, is given by

$$y = g(\sum_{i=1}^{3} \omega_i \alpha_i + b)$$

where α_i, as above, is the activation of the i^{th} hidden node, ω_i is the weight from the i^{th} hidden node to the output node, and b is the bias of the output node.

Putting it all together yields the following equation for the value of the output node:

$$\begin{aligned}y &= g(\sum_{i=1}^{3} \omega_i \alpha_i + b) \\ &= g(\sum_{i=1}^{3} \omega_i (g(\sum_{j=1}^{192} w_{ij}a_j + b_i)) + b)\end{aligned}$$

Recall that we are using the neural-net as an evaluation function in this experiment. Therefore, we have:

$$\begin{aligned}y &= f_{\mathbf{u}}(\mathbf{x}) \\ &= g(\sum_{i=1}^{3} \omega_i (g(\sum_{j=1}^{192} w_{ij}a_j + b_i)) + b)\end{aligned}$$

But the second line above contains no references to $\mathbf{x}$ or to $\mathbf{u}$, so we must explain how we get from the first line to the second line. As mentioned earlier, the net's input layer activations, a_j, are an encoding of the game state, $\mathbf{x}$. Therefore, $\mathbf{a} = (a_1, \ldots, a_{192}) = \mathit{Encode}(\mathbf{x})$, and $\mathbf{a}$ is a function of $\mathbf{x}$. All the other values in the second line: w_{ij}, ω_i, b_i, b, are specified by the parameter $\mathbf{u}$. There are $192 \cdot 3 = 576$ weights w_{ij} incoming into the hidden layer, 3 weights ω_i incoming into the output node, 3 hidden-layer biases b_i, and 1 output bias b, for a grand total of 583 different values. Thus, $\mathbf{u} = (u_1, \ldots, u_{583}) = (w_{1,1}, \ldots, w_{1,192}, w_{2,1}, \ldots, w_{2,192}, w_{3,1}, \ldots, w_{3,192}, \omega_1, b_1, \omega_2, b_2, \omega_3, b_3, b)$.

$\mathit{Encode}(\mathbf{x})$ is the concatenation of three 64 bit vectors, $\mathbf{q} = q_1 \ldots q_{64}$, $\mathbf{r} = r_1 \ldots r_{64}$, and $\mathbf{s} = s_1 \ldots s_{64}$. That is, $\mathit{Encode}(\mathbf{x}) = \mathbf{qrs}$. These 64 bit vectors are determined as follows:

$$q_j = \begin{cases} 1 & \text{square } j \text{ of } \mathbf{x} \text{ has a white stone} \\ 0 & \text{otherwise} \end{cases}$$

$$r_j = \begin{cases} 1 & \text{square } j \text{ of } \mathbf{x} \text{ has a black stone} \\ 0 & \text{otherwise} \end{cases}$$

$$s_j = \begin{cases} 1 & \text{square } j \text{ of } \mathbf{x} \text{ is empty} \\ 0 & \text{otherwise} \end{cases}$$

For the above scheme, it is inconsequential how one numbers the squares of the Othello board, so long as they each have a unique number from 1 to 64.

Note that the encoding does not contain information about whose move it is. This was a technical oversight on my part. The input-layer should include an extra-bit specifying whose move it is. In practice, I doubt this oversight matters much for playing Othello, but I can't be certain.

Note also that, although the bit vectors $\mathbf{q}$, $\mathbf{r}$, and $\mathbf{s}$ are expressed above as having integer values 1 and 0 for their components, in practice these values must be converted to real values 1.0 and 0.0 when they are fed into a neural net.

2 Experiment Design

We begin with a pool of 10 players , each with its own randomly drawn neural-network. These players then undergo a regime of alternating evolutionary and learning phases.

In the evolutionary phase, the current generation of players gives rise to a next generation of players through the genetic algorithm; that is, through the mechanisms of mutation, crossover, and replication. Fitness values are assigned to the players based on their rankings in a single round-robin tournament.

In the learning phase, each player in the newly created generation augments the talent it has inherited from its parents by playing successive round-robin Othello tournaments, modifying its static evaluation function $f_{\mathbf{u}}(\mathbf{x})$ as it learns from experience, using the reinforcement learning algorithm.

A round-robin tournament is simply a tournament in which each player plays a game against every other player exactly once. In the evolutionary phase, the result of a single round-robin tournament determines the fitness of the players. In the learning phase, the results of the successive round-robin tournaments are irrelevant. The players are in effect just playing practice games.

In both phases, when it is a player's turn to move, the player determines which move to make as described in section 1.5. That is, it evaluates the successors of the current state and chooses one of the successors randomly, based on relative value. The evaluation is performed as described in section 1.4, with the player's neural-net acting as the static evaluation function f, and using a search-depth $d = 1$. Also, in both phases, the neural-nets are run in learning-mode. I kept learning-mode on and random move-selection on in both phases even though there are sound arguments that they should be turned off during the evolutionary phase. This was done for simplicity of coding, and because I felt it unlikely that these subtle differences would significantly impact the outcome.

At the end of each learning phase, the weights of the neural-nets may be substantially different than they were at the beginning of the learning phase. In the subsequent evolutionary phase it is these *new* weights that are carried over into the next generation via crossover and replication operations. Thus, this regime is Lamarckian: the experience gained by the nets during their "lifetimes" is passed on to their progeny.

2.1 Evolutionary Phase

The evolutionary method I use in this project is the genetic algorithm. During the evolutionary phase, we pit the players against each other in a round-robin tournament, assigning to each player as a fitness value the fraction of games that it won in the tournament. To be precise, the fitness measure we use is:

$$fitness = \frac{GamesWon}{TotalGames} + \frac{\frac{1}{2}GamesTied}{TotalGames}$$

which gives some credit for ties and always falls in $[0, 1]$. We then mate the players in this generation to produce the next generation: a new set of players, produced as the children of players in the current generation via the operations of crossover, mutation, and replication. In *crossover*, two players from the current generation are selected at random, but based on fitness, so that players with higher fitness values are more likely to be selected, and their neural-network evaluation functions are merged—in a way that I will describe in detail later—to produce two new child players in the new generation. In *replication*, a single player from the current generation is selected at random, but based on fitness, and copied verbatim into the new generation. Every child of the new generation thus created is subject to *mutation*: with a certain probability, one of the weights or biases on its neural-net is randomly perturbed.

Because of the selection pressure the genetic algorithm imposes, poor-performing players are likely to get culled from the population. More importantly, if two different players discover different features of the game-state that are useful for approximating the value function, these features can be combined via crossover into a child whose neural-net employs both features in its evaluation function.

Feature-Level Crossover

One way to view the structure of the neural-net evaluation functions is as follows:

$$\begin{aligned} f_{\mathbf{u}}(\mathbf{x}) &= g(\sum_{i=1}^{3} \omega_i (g(\sum_{j=1}^{192} w_{ij} a_j + b_i)) + b) \\ &= g(\sum_{i=1}^{3} \omega_i \phi_i(\mathbf{x}) + b) \end{aligned}$$

where

$$\phi_i(\mathbf{x}) \triangleq (g(\sum_{j=1}^{192} w_{ij} a_j + b_i))$$

Under this view of the neural-net's structure, the hidden nodes each encode a feature of the raw game-state. The normalized, weighted sum of these features (plus a bias term) then constitutes the evaluation function. Depending on the initial weight-settings of the net before learning begins, these hidden nodes may learn to represent quite different features of the raw game state, and it is these features which we would like crossover to intelligently combine for us through the genetic algorithm.

With this in mind, I structured the crossover operation so that it combines nets at the feature-level, rather than at the lowest-, or weight-level. For the purposes of crossover, a feature is considered to be the complex of incoming weights to a particular hidden node, coupled with that hidden-node's bias and outgoing weight. Since each net in this experiment has three hidden nodes, each net has three features.

When two nets are selected to mate in crossover, their features are pooled to create a set of 6 features, from which three are drawn at random to be the features of the first child. The remaining three features become the features of the second child. Thus, the complex of weights that determine a feature are never disturbed through crossover. All the crossover operation does is determine which combination of the parents' features the children will inherit.

Note that parents are chosen according to fitness, so that parents with good features are more likely to be chosen than parents with bad features. Thus, good features are more likely to be combined with other good features, and are more likely to survive from generation to generation.

I should point out a theoretical concern: if crossover only mixes existing features, but doesn't perturb their innards, how are new features ever to be discovered? There are two possible motive forces for feature-space exploration. First, when three features are combined into a new net in a way that they have not been combined before, the parameter $\mathbf{u}$ will be thrown into a new portion of parameter space, and when the learning phase is initiated, the learning algorithm will adjust all the weights and biases on the net to settle into a local minimum in that portion of parameter space. This will cause all the feature-functions to change, at least somewhat, though it isn't clear whether the change wouldn't just constitute fine-tuning of the existing features, rather than actual emergence of a new, qualitatively different feature.

The second possible motive force is mutation, which does operate at the weight level. However there is the concern that, unless the mutation rate is quite high, it will not be enough to bump a feature into a qualitatively different part of the feature space. In fact, the learning-phase is like a gravity well when it comes to feature-space exploration. Unless the mutation is sufficient to knock the feature out of the local bowl in which it resides, the next learning-phase will just pull it right back down to the bottom of the bowl.

Mutation

For every new child created, either through crossover or reproduction, there is a 1/10 probability that the child will have a mutation in precisely one of its weights or node-biases. When a mutation occurs, the current weight or bias is thrown away and replaced with a new value drawn uniformly on $[-.7, .7]$.

2.2 Learning Phase

The learning method I use in this project is reinforcement learning. During the learning phase, the Othello players modify the weights and biases of their neural-nets with every move of every game they play, in accordance with the reinforcement learning algorithm. The method is simple. At each game state, $\mathbf{x}$, $Player_1$ passes the training pair $(\mathbf{x}, h(\mathbf{x}; f_{\mathbf{u}_1}, d))$ to its neural-net, and $\mathbf{u}_1$ is updated accordingly. Similarly, $Player_2$ passes the training pair $(\mathbf{x}, h(\mathbf{x}; f_{\mathbf{u}_2}, d))$ to its neural-net. After both players have updated their nets based on game state $\mathbf{x}$, the player whose turn to move it is chooses a move using the fitness-based random-selection method described in section 1.5, and play continues.

A round-robin tournament pits every player against every other exactly once. So, with a pool of 10 players, a tournament consists of 45 games. In this experiment, I conduct 200 tournaments in each learning phase. The first 100 are conducted with the learning parameter of the nets set to .1 so that their weights will quickly hone-in on the bottom of the local error bowl. (Not to be confused with the weight vector parameter $\mathbf{u}$ of the net, the learning parameter, λ, is a scalar that determines how much the net alters $\mathbf{u}$ as the result of a single training pair. Higher values of λ result in greater changes to $\mathbf{u}$. Lower values are of λ are used for fine-tuning, when $\mathbf{u}$ is already close to optimal.) Then the next 100 tournaments are conducted with the learning parameter of the nets set to .01, for fine tuning. It must be said that I have no idea whether the nets come close to convergence after the first 100 tournaments. If they don't, then the next 100 tournaments, with a low learning parameter, are essentially wasted. I just take a gut-shot at something reasonable.

Endgame Calculation

Othello has a nice property: the maximum number of actual plays made in the game cannot exceed 60. Once the game reaches the point where there are only a handful of empty squares left on the board, the remaining game-tree can be completely calculated. During the learning phase, I had the players perform endgame calculation when there were 10 empty squares left. They then perform one final learning-step apiece, using as a target value $v(\mathbf{x})$, *the true value of the game-state*, obtained from the endgame calculation. That is, when $\mathbf{x}$ represents a game state that has only 10 empty squares left, then the final learning pair $(\mathbf{x}, v(\mathbf{x}))$ is passed to the neural-nets of both $Player_1$ and $Player_2$ and the game ends. (Note that which player wins the game is irrelevant to the learning phase.)

It is important to stress that whereas all previous learning-steps only use the approximation $h(\mathbf{x}; f_{\mathbf{u}}, d)$ for a target, the final learning-step uses the true game-state value, $v(\mathbf{x})$, obtained by performing a *complete* game-tree expansion from $\mathbf{x}$ and calculating $\frac{W}{W+B}$ at each of the terminal nodes of the expansion. Specifically, we have

$$v(\mathbf{x}) \triangleq \begin{cases} \frac{W}{W+B} & \mathbf{x} \text{ is a terminal game-state} \\ \max\limits_{\mathbf{x}' \in Succ(\mathbf{x})} v(\mathbf{x}') & \mathbf{x} \text{ is not a terminal game-state, and } \mathbf{x} \text{ is a White-to-play state} \\ \min\limits_{\mathbf{x}' \in Succ(\mathbf{x})} v(\mathbf{x}') & \mathbf{x} \text{ is not a terminal game-state, and } \mathbf{x} \text{ is a Black-to–play state} \end{cases}$$

3 Results And Analysis

I ran this experiment for 100 generations, saving to disk a copy of the best player from each generation, naming them $BestOfGen_1$, through $BestOfGen_{100}$. The experiment took three months to run, during which time 900,000 individual games were played at a rate of approximately one game every 9 seconds. At the end of the experiment, I invoked a final round-robin tournament among the 100 best-of-gen players, assigning to each player as a final fitness rating the fraction of games won, modified to account for ties:

$$fitness = \frac{GamesWon}{TotalGames} + \frac{\frac{1}{2}GamesTied}{TotalGames}$$

I should mention that, unlike while in learning mode, where I used a search-depth of 1, I ran the final tournament with a search-depth of 4, and with learning off (the nets did not alter their weights anymore). A search-depth of 4 is still very modest: the nets can calculate that in a fraction of a second, but I thought it would give a more robust appraisal of the evaluation functions.

Figure 4 at the end of this paper shows the fitness of these best-of-gen players as a function of generation number. One can clearly see that fitness steadily improves for the first thirty generations, after which it levels off at just a little above .5. The highest-fitness player is $BestOfGen_{29}$, with a rating of of .68.

I personally played about a dozen games against $BestOfGen_{29}$ and lost all but one of them. I am not a great Othello player, but it is some measure of success that $BestOfGen_{29}$ was able to beat me so handily.

I also played $BestOfGen_{29}$ against my Linux installation's computer Othello player, which incorporates human-designed features. The Linux Othello player has seven different skill settings. I played it on level five, which presumably uses a fairly deep search, since it takes a couple of seconds per move, and $BestOfGen_{29}$ lost as Black and won as White, indicating that it is competitive with a human-designed Othello program.

3.1 Diverse Features Failed To Develop

I am sorry to report that analysis of the best-of-gen nets reveals that no interesting crossover took place. In net after net, the three hidden nodes end up with the same general pattern of weight values. Figures 2 and 3 illustrate this similarity. I blame this failure on my choice of representation. Crossover can't create diversity where none exists. It seems likely that, given the topology I chose, there is only one useful feature to be discovered.

Despite this failure to obtain obviously different features, it is possible that the three hidden nodes are different in subtle ways not so easily detectable to the eye. They may in fact provide a useful redundancy, allowing the nets to be more robust. In this sense, the three hidden nodes are like three slightly different Othello-evaluators that are "bagged" together by the overall network. (*Bagging* is a method in statistics of averaging the results of different estimators in order to obtain a lower-variance estimate.)

Evaluation Functions Learned To Like Corners and Dislike Corner-Neighbors

Probably the single most important piece of strategic advice to give any beginning Othello player is: *get the corners.* An immediate corollary is to avoid placing a stone on any square adjacent to a corner, since this usually enables one's opponent to grab the corner. And a corollary to the corollary is that placing a piece adjacent to a square adjacent to a corner (i.e., two away) is probably good, since it may force one's opponent ultimately to place a piece on the adjacent square and thus cede the corner.

Reviewing the feature plots, one can see that, by generation 29, the nets have learned to value the corners. This is reflected by high-values (light shades of gray) in the corner squares of the white board and low-values (dark shades of gray) in the corner squares of the black board. Remember that the evaluation functions always assess the board-position from White's point of view, so a white-stone in the corner will be good (high value), whereas a black-stone in the corner will be bad (low value).

By generation 29, not only have the corners of the white-board become high values, but the adjacent squares have become low values, and the adjacent adjacent squares have become moderately high, though obviously not as high as the corners themselves. Thus, at the very least, we can conclude that the nets have learned the fundamentals.

The Fitness Plateau

I am somewhat surprised that the fitness levelled-off so quickly. I would have expected fitness to increase steadily throughout all 100 generations. But we must remember that each generation comprises not only a genetic-mixing step, but also 200 learning tournaments, consisting of 45 games each. Thus, by generation 30, 9000 games have already been played. It may be that this was enough for the reinforcement learning phases to reach convergence, given the net topology I chose. Had I chosen a more complex net topology, it is possible that the nets would have learned more and longer.

However, it is not surprising in itself that a plateau occurred. Indeed, this is almost inevitable. Once a fairly high-level of fitness is achieved, subsequent genetic pairings are likely to either leave fitness unchanged or reduce it, since there are many ways to go down, and few ways left to climb, the fitness terrain. And, given that a plateau occurs, the fitness level of those players in the plateau must perforce be around .5. Since, by assumption, they all have about the same fitness, the probability that one player from the plateau group beats another from the plateau group must be roughly .5.

4 Conclusions And Directions

The experiment was moderately successful. The results certainly indicate that steady learning occurred through 30 generations and that the features evolved by the nets incorporate a basic understanding of the game, such as the value of the corners.

The experiment failed to demonstrate any real utility for crossover, but this might be fixed in a future experiment that chooses a more complex topology. I believe the topology I chose simply did not allow the nets to discover more than one useful feature.

How To Read Figures 1—3

Figure 1 on the next page shows the values of the weights incoming to the hidden layer of neural-net $BestOfGen_1$. Each of the net's hidden nodes has 192 incoming weights. This complex of incoming weights is represented graphically as a column of Othello boards: a white board, a black board, and an empty board ($3 \cdot 64 = 192$). Since the net has three hidden nodes, the figure has three columns, one for each node. The squares of the boards are drawn with five different gray levels; white codes the highest value range, $[0.6, 1.0]$, and black codes the lowest value range, $[-1.0, -0.6]$ The color of the square represents the value of the associated incoming weight. Above each column of boards is printed the weight of the outgoing link on that hidden node. A hidden node with a relatively higher outgoing weight than the other two hidden nodes will participate more significantly in determining the final output value of the net.

Figures 2 and 3 are drawn in exactly the same format.

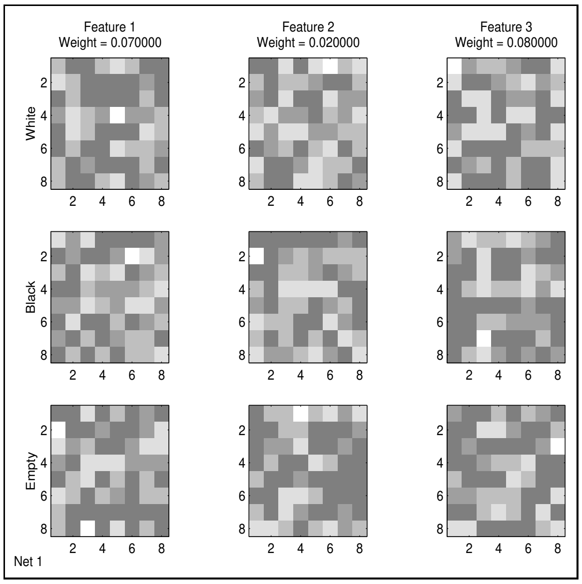

Figure 1: The best player of generation 1–a random neural-net

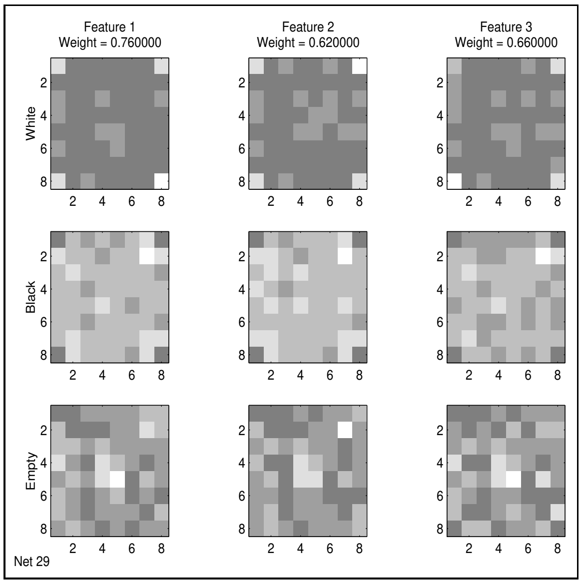

Figure 2: the best player of generation 29, which happens to be the best player overall. Note the high value placed on the corners.

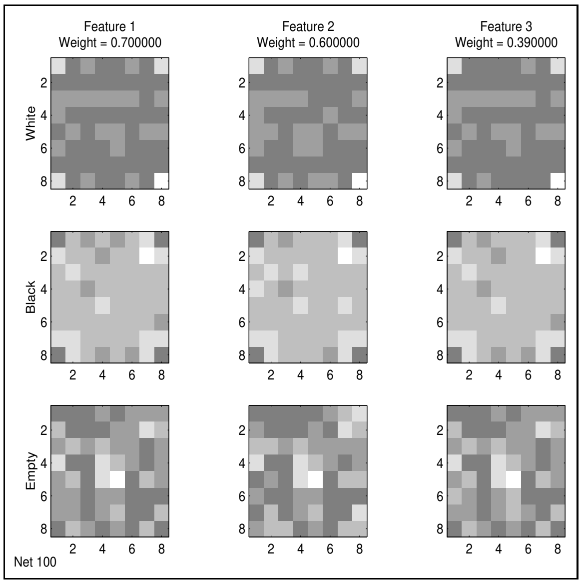

Figure 3: The best player of generation 100. This neural-net is nearly as good as net 29, winning 60 percent of its games, and its structure is very similar to that of net 29.

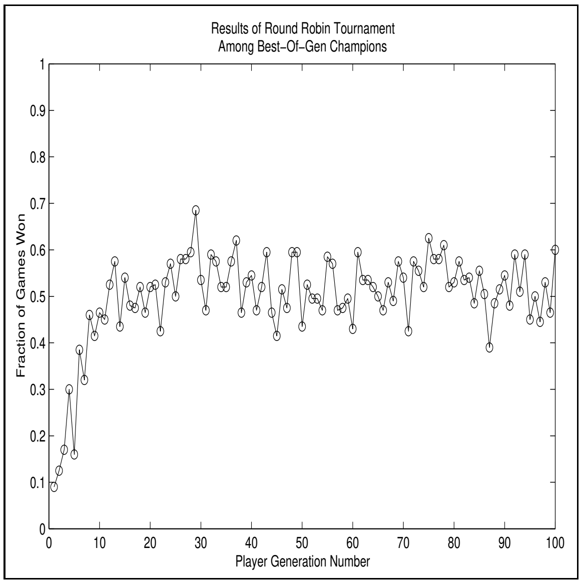

Figure 4: This graph shows the fraction of games won by each generation's best player in a round-robin tournament against the other generations' best players. The graph reveals a general trend of increasing fitness through generation 29, at which point fitness levels off.