Co-evolving a Neural-Net Evaluation Function for Othello by Combining Genetic Algorithms and Reinforcement Learning

Joshua A. Singer

Stanford University

Abstract. The neural network has been used extensively as a vehicle for both genetic algorithms and reinforcement learning. This paper shows a natural way to combine the two methods and suggests that reinforcement learning may be superior to random mutation as an engine for the discovery of useful substructures. The paper also describes a software experiment that applies this technique to produce an Othello-playing computer program. The experiment subjects a pool of Othello-playing programs to a regime of successive adaptation cycles, where each cycle consists of an evolutionary phase, based on the genetic algorithm, followed by a learning phase, based on reinforcement learning. A key idea of the genetic implementation is the concept of feature-level crossover. The regime was run for three months through 900,000 individual matches of Othello. It ultimately yielded a program that is competitive with a human-designed Othello-program that plays at roughly intermediate level.

1 Background

This paper describes a way to search the space of neural-net functions using an approach that combines the genetic algorithm with reinforcement learning. The hybrid approach is founded on the intuition that crossover and gradient descent exploit different properties of the fitness terrain and can thus be synergistically combined. A secondary idea is that of *feature-level crossover*: each hidden node of a three-layer neural-net may be thought of as extracting a feature from the net's input. It is these features that are mixed by the crossover operation.

The paper also describes a software experiment that employs this hybrid approach to evolve a pool of Othello-playing programs. The best evolved program played about as well as *kreversi*, the hand-crafted Othello program written by Mats Luthman and ported to KDE/LINUX by Mario Weilguni. (See www.kde.org for more information.) *Othello*, also known as *Reversi*, is a popular two-player board-game whose rules are described briefly in section 1.3.

1.1 Neural Nets as Game-Players

Many people have used neural-nets as the basis for playing games, notably Tesauro [1], whose TD Gammon program applied reinforcement learning to a neural-net in self-play and greatly surpassed all previous efforts at computer

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2074, pp. 377–389, 2001.

backgammon, and recently Chellapilla and Fogel [2], who applied a mutation-only genetic algorithm to a pool of neural-nets, producing a net that could play checkers at near-expert level. Neural-net approaches to Othello in particular include that of Moriarty and Miikkulainen [3], who applied the genetic algorithm to a flexible encoding representation that allowed them to evolve multi-layer, recurrent neural-net topologies.

1.2 The Impetus for a Combined Approach

The goal of all learning methods can be roughly characterized as the search for an optimal point within some space, where optimality usually means maximizing a fitness function or minimizing a cost function on the space. In the case of Othello, the search space 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 global optimum 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 use crossover, and in so doing to make what we might call the *decomposability assumption*: that the fitness of a point in the space can be approximated as the sum of fitness-values associated with substructures of the point. For example, suppose the search space is $\mathbf{R}^n$ with associated fitness function F, and that we can decompose $\mathbf{R}^n$ into the cross-product of subspaces $\mathbf{R}^m$ and $\mathbf{R}^k$, each with their own fitness-functions, F_1, and F_2, in such a way that when we decompose $\mathbf{u} \in \mathbf{R}^n$ as $\mathbf{u} = (\mathbf{u}_1, \mathbf{u}_2)$, $\mathbf{u}_1 \in \mathbf{R}^m$, $\mathbf{u}_2 \in \mathbf{R}^k$, we find that $F(\mathbf{u})$ is approximately equal to $F_1(\mathbf{u}_1) + F_2(\mathbf{u}_2)$. Then the original fitness terrain, $(\mathbf{R}^n, F)$, is a decomposable fitness terrain.

As an example of how crossover works to improve fitness in a decomposable fitness terrain, consider two parameters, $\mathbf{u} = (\mathbf{u}_1, \mathbf{u}_2)$ with fitness-values $(50, 1)$ for a total fitness of approximately 51, and $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2)$ with fitness-values $(2, 60)$, for a total fitness of approximately 62. If these are crossed to yield the two child parameters $\mathbf{c}_1 = (\mathbf{u}_1, \mathbf{z}_2)$ and $\mathbf{c}_2 = (\mathbf{z}_1, \mathbf{u}_2)$, then $\mathbf{c}_1$ will have a fitness of approximately 110.

Crossover has a weakness though: it can't make something from nothing. Given a pool of points with low-fitness, possessed of no useful substructures, crossover is no more likely than macromutation to yield a point with improved fitness [4].

Gradient-descent does not possess this weakness of crossover. While, in the standard genetic algorithm, mutation is the motive force for discovering useful substructures, in this combined approach, gradient-descent can be used to find useful substructures much more quickly. 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 substructure!

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 substructure, and, since they come from different portions of the fitness terrain, it may well be that they possess ten *different* good substructures. 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 original 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

Virtually all game-playing programs contain a subroutine called an *evaluation function*. The evaluation function is an approximation to its game-theoretic counterpart: the *value function*. To each possible game-state, the value function as-

signs 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 selects a particular function from the family $\mathcal{F}$. For the experiment, we 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

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

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 the experiment, a player makes a weighted random choice, based on the relative values of each move, using the following exponential weighting scheme: let $\mathbf{x}'_1 \ldots \mathbf{x}'_n$ be the successors $\mathbf{x}$. Then the probability of selecting the move that leads to successor $\mathbf{x}'_i$ is:

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

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 $T = 10$. At $T = 0$, all moves are equally likely to be chosen, regardless of their evaluation. At $T = 10$, if move A has an evaluation .1 greater than move B, then move A is $e^{10 \cdot .1} = e$ times more likely to be chosen than move B.

1.6 Neural-Networks as Evaluation Functions

This project uses neural-networks as static evaluation functions. Each neural-network has a 192-node input layer, three hidden nodes, and a single output node. A network in learning mode will use backpropagation to modify its weights. Network nodes have standard sigmoid functions $g(z) \stackrel{\triangle}{=} (1 + e^{-z})^{-1}$.

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.

A game state $\mathbf{x}$ is encoded as 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, $Encode(\mathbf{x}) = \mathbf{qrs}$. 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.

2 Experiment Design

We begin with a pool of 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 outcomes 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 $d = 1$.

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 used in this project is the genetic algorithm. The evolutionary phase pits 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 is:

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

which gives some credit for ties and always falls in $[0, 1]$. Players in this generation are then mated 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. The experiment used a generation size of 10.

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 to be described 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, some of the weights or biases on its neural-net are randomly perturbed.

Because of the selection pressure the genetic algorithm imposes, poorly-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

Given the topology of the neural nets in the experiment, the function they represent is always of the form:

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

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, b_i is the bias of the i^{th} hidden node, g is the sigmoid function, ω_i is the weight from the i^{th} hidden node to the output node, and b is the bias of the output node.

One way to view the above function is as follows:

$$f_{\mathbf{u}}(\mathbf{x}) = g(b + \sum_{i=1}^{3} \omega_i \phi_i(\mathbf{x}))$$

where

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

The ϕ_i can be thought of as features of the raw game state. Their normalized, weighted sum (plus a bias term) then constitutes the evaluation function. Under this view of the neural-net's structure, the hidden nodes, which implement the functions ϕ_i above, each extract a feature of the raw game-state. Depending on the initial weight-settings of the nets before learning begins, the hidden nodes of various nets may learn to represent quite different features of the raw game state, and some of these features may be good predictors of the value function.

If we assume that the value function can be approximated by a roughly linear combination of more or less independent features of the game state, and if a hidden node is capable of representing such features, then any net possessing even one such feature will very likely be more fit than a net possessing none, and a net possessing two different features will be more fit still.

Given the above assumptions, the fitness terrain is decomposable, with a point $\mathbf{u} \in \mathcal{F}$ decomposing as

$$\mathbf{u} = (\phi_1, \phi_2, \phi_3, \omega_1, \omega_2, \omega_3, b)$$

where the ϕ_i are the above feature functions, expressed as vectors whose components are the bias and all incoming weights of the i^{th} hidden node. That is, the components of ϕ_i are the elements $\{w_{ij}, b_i\}$, $j = 1 \ldots 192$.

The crossover operation is structured to take advantage of the presumed decomposability of the fitness terrain. It combines nets at the feature-level, rather than at the lowest-, or weight-level. When two nets are selected to mate in crossover, their ϕ_i 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 become the features of the second child. In addition, the ω_i of each child are

set to that of the corresponding inherited ϕ_i. For example, if a child inherits ϕ_2 from parent 1, then it also inherits ω_2 from parent 1. Finally, the b value of each child is randomly inherited from one of its parents. The important point is that the set of values 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.

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 $[-.7, .7]$. This is a very low mutation rate, and, unless the mutation is sufficient to knock the neural net's parameter out of the local bowl in which it resides into a qualitatively different part of the fitness terrain, the next learning-phase will probably just pull it right back to very nearly the same local optimum it occupied before the mutation. Mutation plays a minor role in the experiment because the gradient-descent behavior of reinforcement learning, rather than mutation, is supposed to provide the impetus for the discovery of useful substructures.

2.2 Learning Phase

The learning method used 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. This experiment conducts 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 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.

Endgame Calculation
Othello has a nice property: with each move made, the number of empty squares left on the board decreases by one. Therefore, 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, the players perform endgame calculation when there are 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

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

3 Results and Analysis

The experiment ran for 100 generations, taking three months to complete, 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, a final round-robin tournament was played among the 100 best players of each generation, assigning to each player as a final fitness rating the fraction of games won, modified to account for ties:

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

Unlike while in learning mode, when the experiment used a search-depth of 1, the final tournament used a search-depth of 4. A search-depth of 4 is still very modest—the nets can calculate that in a fraction of a second—and probably gives 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 twelve 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.

For some comparison with a human-written program, $BestOfGen_{29}$ was played against *kreversi*, a well-known program available with KDE for LINUX. kreversi plays a modified positional game. A purely positional game assigns different values to each square of the board and evaluates a game state as the

sum of the values of the squares occupied by white minus the the sum of the values of the squares occupied by black. kreversi uses a convex combination of position and piece-differential to evaluate its game states, with piece-differential receiving higher and higher weight as the game progresses. kreversi also uses a dynamic search-depth: states near the end of the game are searched more deeply.

$BestOfGen_{29}$ played a twenty game tournament against kreversi on its level three setting, which uses a minimum search-depth of 4, with the following results: as white, $BestOfGen_{29}$ won 7 games and lost 3. As black, $BestOfGen_{29}$ won 3 games, lost 6, and tied 1.

3.1 Diverse Features Failed to Develop

Casual analysis of the best-of-gen nets fails to reveal that any 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. It is possible that, given the fixed three-layer topology, there is only one useful feature to be discovered. On the other hand, 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.)

How to Read Figures 1—3
Figure 1 on the last 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 set 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. Note that the biases of the nets are not shown on any of the figures.

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.

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. Thus, at the very least, we can conclude that the nets have learned the fundamentals.

The Fitness Plateau

It is perhaps somewhat surprising that the fitness levelled-off so quickly. One might have expected fitness to increase steadily throughout all 100 generations. But one 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 are played. It may be that this was enough for the reinforcement learning phases to reach convergence. Had a more complex net topology been chosen, 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 Future Directions

The central idea of this paper is that, in domains where it can be applied, reinforcement learning may be better than random mutation as an engine for the discovery of useful substructures. A secondary idea is that of feature-level crossover: that the substructures we wish to discover may be identified with the hidden nodes of a neural net. Feature-level crossover is merely a way to do crossover on neural-nets. It needn't be coupled with reinforcement learning, nor restricted to fixed-topology nets.

As a side note, the distinction between the terms *substructure* and *feature* is actually one of genotype and phenotype. In the space of neural-net functions, the substructures that confer high fitness are simply those that lead to useful feature-extractors in the phenotype.

Whether or not the Othello fitness terrain is a decomposable terrain is an open question. If it is not, then the crossover operation should perform no better than macromutation in finding points of improved fitness [4]. It is however reasonable to suspect that the Othello fitness terrain can be characterized by more or less independent features, because this is how humans analyze board positions. It is also an open question whether reinforcement learning applied to Othello can lead to the discovery of features along the lines suggested by this paper. Although the experiment yielded a fairly good player, there is little indication that diverse features were discovered, or that crossover played a significant role.

Further, any hybrid strategy, being inherently more complex than a single method, requires justification based on its practical worth, the more so because it seems that mutation alone can still be very effective in finding points of high fitness [2,7]. While this paper does not investigate the efficacy of the hybrid method relative to other methods, that is an important direction for future research.

Acknowledgments. I would like to thank John Koza for his many helpful comments and suggestions.

References

1. G. Tesauro, "Temporal Difference Learning and TD-Gammon". *Communications of the ACM*, vol. 38, no. 3, pp. 58-68, 1995
2. K. Chellapilla and D. B. Fogel, "Evolution, Neural Networks, Games, And Intelligence". *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1471-96, 1999
3. D. E. Moriarty and R. Miikkulainen, "Discovering Complex Othello Strategies Through Evolutionary Neural Networks". *Connection Science*, vol. 7, no. 3-4, pp. 195-209, 1995
4. Terry Jones, "Crossover, Macromutation, and Population-Based Search", *Proceedings of the Sixth International Conference on Genetic Algorithms*
5. Dimitri Bertsekas and John Tsitsiklis, *Neuro-Dynamic Programming*. Belmont, Massachusetts: Athena Scientific, 1996.
6. John Koza, *Genetic Programming*. MIT, 1996
7. J. B. Pollack and A. D. Blair, "Co-Evolution in the Successful Learning of Backgammon Strategy", *Machine Learning*, vol. 32, no. 3, pp. 225-40, 1998
8. Brian D. Ripley, *Pattern Recognition And Neural Networks*. Cambridge University Press, 1996.
9. Richard Sutton and Andrew Bartow, *Reinforcement Learning*. Windfall Software, 1999.

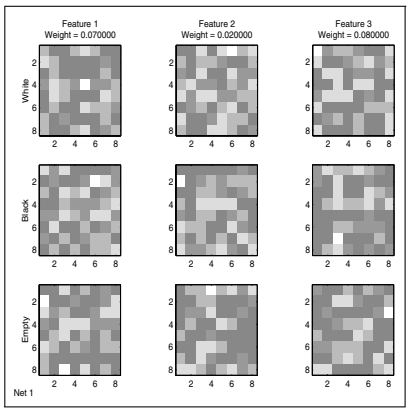

Fig. 1. The best player of generation 1–a random neural-net

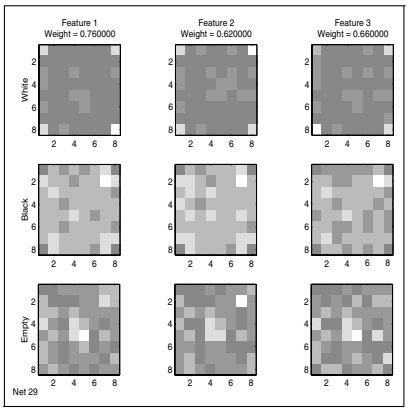

Fig. 2. the best-of-run player, from generation 29. Note the high value placed on the corners

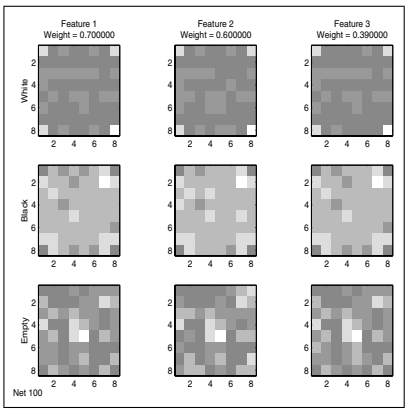

Fig. 3. The best player of generation 100. This neural-net is nearly as good as net 29, winning 60 percent of its games, and its structure is very similar to that of net 29

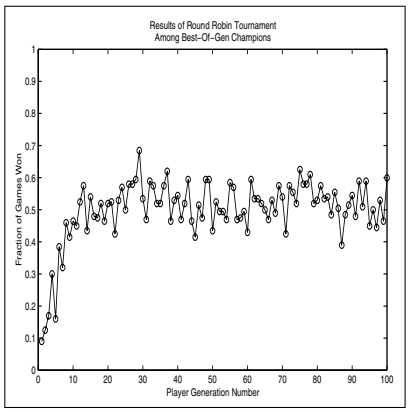

Fig. 4. This graph shows the fraction of games won by each generation's best player in a round-robin tournament against the other generations' best players. The graph reveals a general trend of increasing fitness through generation 29, at which point fitness levels off