# Linear Versus Non-Linear Learning in the Context of Othello

**Ali Alanjawi**
**Department of Computer Science, University of Pittsburgh**
**alanjawi@cs.pitt.edu**

## Abstract

Acquiring more knowledge renders the need for deep searching. However, knowledge is hard to measure. In this paper, Linearity is proposed to measure the quality of knowledge. A comparison has been made among three learning techniques, namely: linear regression, neural network and support vector machine. These techniques were applied in the domain of Othello.

## 1. Introduction

Many game-playing programs depend on brute-forced search to play in a master level. However, search alone is not enough, since most games have exponentially growing search space. Incorporating knowledge with search is shown to be effective by many researchers [Berliner 1984; Schaeffer 2000]. One direction of the use of knowledge in search is to predict the evaluation function by using machine learning methods. One traditional problem, may arise in this context, is the search-knowledge tradeoff. That is whether to spend more efforts, and time, in computing the evaluation function, based on prior knowledge, or in searching deeper in the game-tree. As quality of knowledge increases, the need of deeper search to achieve a given performance decreases. Therefore, knowing how much knowledge needed to search up to depth 9 for instance would have a great value. The problem is, we do not know how to measure knowledge.

One possible way to estimate quality of knowledge is to use an expert; an expert could be a human or a strong program as proposed by Junghanns and Schaeffer (1997). This solution is not feasible, since an expert is not always available even if it is a program.

Other potential solution to this problem is using the linearity of the evaluation function to quantify the quality of knowledge. Linear functions are easy to learn and evaluate, giving more time for the program to gain deep search, but may not have a great value of knowledge. Whereas, nonlinear function need more time to learn and to evaluate, however, may captures more important knowledge, and hence gives more accurate evaluation, by their complexity.

The problem of linearity is not a trivial one though. Researchers have different opinions. Samuel (1967) suggested the use of nonlinear evaluation function in his checker program. Lee and Mahajan (1988) interpreted the successes of their Othello program (BILL) as indication that a nonlinear evaluation function is better than linear evaluation function.

On the other hand, Buro (1995) has showed in his Othello program (LOGISTELLO) that logistic regression obtains better results than a quadratic discriminant function. Tesauro (1989) noted, in experimentation on his backgammon program, that a single-layer neural network, which is linear, could outperform a nonlinear multi-layer neural network.

In this work we investigated the effect of linearity on the learning process; and examined whether linearity can capture quality of knowledge. The next section describes the game of Othello. Section 3 gives an overview of the learning techniques used. The main experimental results are presented in section 4 and discussed in section 5.

## 2. The Game of Othello

Othello (also called Reversi) is an 8-by-8 board game. It is played by two players, and each has either a black or white pieces to place on the board. Each of the two players tries to flip his opponent's pieces by placing them between two of his pieces in any line. The player who cannot flip any of his opponents' pieces must pass. The game end when both players pass successively. The player who has more pieces at the end wins the game. Figure 1 shows the Othello initial position.

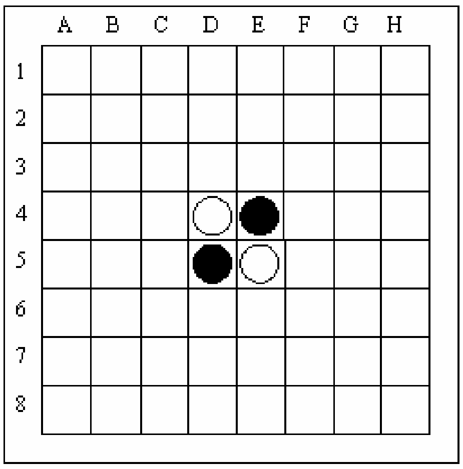

**Figure 1 Othello initial board position.**

Othello is considered a tractable domain in AI by many researchers [Rosenbloom 1982, Lee and Mahajan 1990, Moriarty and Miikkulainen 1994, Buro 1995]. Many programs were developed to play the game; some of them were considered beyond the human master level. The first noticeable program was Rosenbloom's Iago (1982). Iago achieved a world-championship level given the early-eighties hardware. Less than a decade later, Lee and Mahajan revealed the curtain of their impressive program BILL [Lee and Mahajan 1990]. It achieved a straight win over Iago on a set of test games, and won a single game against Brian Rose, the highest rated American Othello player at that time. In 1997, LOGISTELLO [Buro 97] played an exhibition match against world champion Takeshi Murakami. LOGISTELLO won all six games against Murakami.

The key of success of the programs mentioned above, and any other strong programs, is the use of deep search and extensive knowledge. Deep search can be attained by developing better search algorithms and by using heuristics to limit the scope of the search. On the other hand, Knowledge is achieved by machine learning techniques. Some of these techniques are summarized in the next section.

## 3. Learning Techniques

Human and machines are different in acquiring knowledge, however, they both obtain knowledge by learning. Learning is the ability of using previous knowledge to adapt future behavior. The problem with machine learning is that representation of knowledge must be easy for machines to understand and to learn afterward. One way to overcome this problem is to quantify the knowledge; that is to assign numerical values for attributes that describe the domain of knowledge.

In this context learning can be achieved by finding an approximation function $f$ that best maps the quantified knowledge to numerical values that can facilitate feature behavior. Therefore, the problem of machine learning boils down to an optimization problem. Many techniques for finding function $f$ do exist; some are linear such as linear regression, others are nonlinear such as neural network[1]. These techniques, along with the support victor machine, are discussed in the following sections.

### 3.1 Linear regression

Assume we have a training data $D$ represented by pairs of inputs $\boldsymbol{x_i}$ and outputs $y_i$, $<\boldsymbol{x_i},y_i>$. In the linear regression model we try to approximate the behavior of the training data by a linear function $f$, i.e. a straight line. That is

$$f = \boldsymbol{w_i} \cdot \boldsymbol{x_i} + b \qquad (1)$$

[1] In this article we use "neural network" to implicitly mean multi-layer neural network, and use linear regression to indicate a one layer neural network model, or perceptron.

Where $\boldsymbol{x_i}$ is the input vector of the training data, $\boldsymbol{w_i}$ is vector of the weights of $\boldsymbol{x_i}$, and b is a bias. Learning in linear regression involves choosing values for the weights $\boldsymbol{w_i}$. Moreover, we try to choose values of $\boldsymbol{w_i}$ that minimize the deviation of $f$ from $y_i$. The mean-square error (*MErr*) is usually used to measure such deviation. *MErr* is given by:

$$MErr = \Sigma_i \, (y_i - f(\boldsymbol{x_i}))^2 \qquad (2)$$

One way for solving this optimization problem is to change the weights after each training example according to the gradient descent rule[2]. The gradient descent rule is given as follows:

$$\boldsymbol{w}=\boldsymbol{w} - \alpha \nabla_w \, MErr_i(\boldsymbol{w}) \qquad (3)$$

Where α is the learning rate, and $\nabla_w$ *MErr* is the change of error from previous example. To obtain $\nabla_w$ *MErr*, we compute the derivative of *MErr*; and the problem is reduced to updating the weight vector according to the following equation:

$$\boldsymbol{w}=\boldsymbol{w} - \alpha \, . \, (y_i - f(\boldsymbol{x}_i, \boldsymbol{w}) \, . \, \boldsymbol{x}_i) \qquad (4)$$

## 3.2 Neural network

The idea of neural networks is firstly introduced by McCulloch and Pitts (1943). They were inspired by the model of the human brain; and attempted to duplicate the behavior of the brain on an abstract level.

A neural network consists of a set of units, links that connect one unit to another, and weights associated with each link. Usually, neural networks are layered, *i.e.* units are organized into groups of layers, and links go from units to units in adjacent layers. There are three types of layers: input, hidden and output layer. A neural network may consist of one input layers, one output layers, and one or more hidden layers. Input layer is where the units in the layer receive input from the training data. The units in the hidden layers receive inputs via links from units in previous layer and send it to the units in next layer; whereas the units, in the output layers, receive input from the units in last hidden layer and send outputs out of the network. Figure 2 illustrate an example of a neural network used in our experiments. The network consists of an input layer with 64 units, an output layer with one unit, and two hidden layers that contain 50 and 10 units respectively.

[2] Sometimes called the delta rule

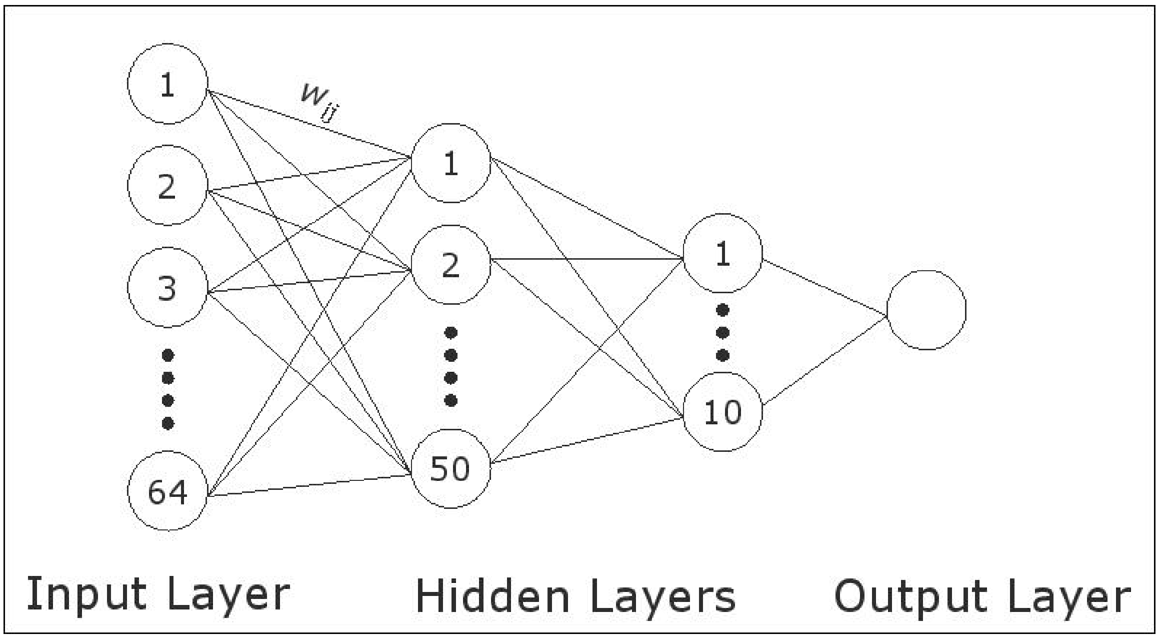

**Figure 2 A neural network with two hidden layers.**

Learning usually occurs by adjusting the weights on the links. Each unit has a set of weighted inputs, an activation level, and a way of computing its activation level on the next step. This is done by applying an activation function to the weighted sum of the unit's inputs. Generally, the weighted sum (also called the input function) is strictly linear sum, while the activation function may be nonlinear. If the value of the activation function is above threshold, the node “fires,” i.e. sends an output to the linked unit.

The standard method for learning in neural networks is back-propagation [Bryson and Ho 1969]. The back-propagation algorithm employs the gradient decent rule, which is described in section 3.1, to attempt to minimize the squared error between the network output values. The weights are propagated through the network and adjusted according to error difference. A detailed description of the algorithm can be found in [Mitchell1997].

## 3.3 Support vector machine

The Support Vector Machine (SVM) [Vapnik 1995] is a technique that allows us to approximate training data into a linear function within a given error$\varepsilon$. That is, to find a function *f* that has at most $\varepsilon$ deviation from the actually obtained targets $y_i$ for all the training data. Errors are negligible as long as they are less than$\varepsilon$, but will not accept any deviation larger than this. Furthermore, we might relax this constrain by adding penalties for deviations larger than$\varepsilon$. Hence, the problem reduced to optimizing the weights and the penalties associated with errors larger than$\varepsilon$.

Recall equation 1 in section 3.1; in SVM model we try to minimize *w*s as much as possible. One way to insure this is to minimize the Euclidean norm, $||w||^2$.
Thus the optimization problem became as follows:

$$\text{Minimize} \quad \tfrac{1}{2}||w||^2 + C\,\Sigma_i\,\xi_i$$

$$\text{With constraint} \quad \begin{cases} y_i - \boldsymbol{w}\boldsymbol{x}_i - b \leq \varepsilon + \xi_i \\ \boldsymbol{w}\boldsymbol{x}_i + b - y_i \leq \varepsilon + \xi_i \\ \xi_i \geq 0 \end{cases}$$

Where $\xi_i$ refers to errors with deviation larger than $\varepsilon$, and C is a constant that determines the cost, penalty, of having such errors. Figure 3 illustrate the idea graphically. Only the points outside the shaded region contribute to the cost C, as deviations are penalized in a linear fashion.

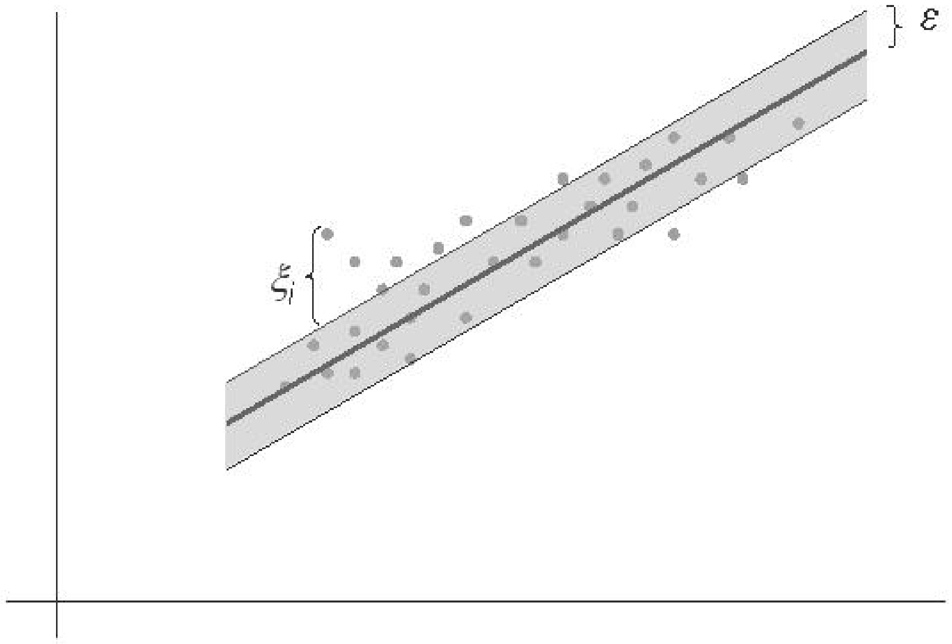

**Figure 3 SVM model. $\varepsilon$ is the width of allowed deviation, and $\xi_i$ is the penalized deviation. The points on the gray area boundaries are called support vector points; the line inside the gray area represents the model output.**

To make SVM algorithm nonlinear, we simply preprocess the training examples, $x_i$ , by mapping $\Phi: X \rightarrow F$ into some feature space $F$, and then apply the standard SVM algorithm. $\Phi$ is usually called Kernel function. In our experiments we use the Gaussian radial basis kernel, which is defined as

$$e^{-\delta\|a-b\|^2}$$

## 4. Experimentation and Results

The previous section presented a brief description of three learning models. We use these models to compare between linear and nonlinear models. In this section a description of the experiment design and the results will be presented.

To apply machine learning techniques we need a large dataset on Othello domain. Fortunately, many Othello games database do exist, and available on the internet. We choose to use the well-known WTHOR database provided by the France Federation of Othello (FFO). It is a huge data of Othello games, played in tournaments by master-level players since 1980. More than 69,000 games were recorded, which means more than 4,000,000 of training instances. This database records the games in special binary files; thus it needs to be translated to sets of attributes, or features.

We choose to translate the binary files into sets of primitive features, which represent the pieces in the board. Each feature represents one of the 64 tiles of the Othello board. Then each feature will be assigned a value according to the piece on it. For instance, black is -1, white is 1 and empty is 0. Structural features are chosen because they are linear, and since we want to measure linearity, we do not want to be biased by nonlinear complex features, Another reason is that we do not want to push the nonlinearity to the features construction, as in kernel functions discussed in section 3.3; instead we want to measure the linearity of the learning algorithm. The output of the game is represented by the disc difference at the end of the game; therefore, each training example will have a value that signifies the output of the game.

The three learning models described in section 3 has been implemented. For the linear regression, we implemented the gradient descent algorithm for updating the weights. The learning rate $\alpha$ has been adjusted by running the program many times and taking the best results. In the neural network, back-propagation has been implemented. We implemented two networks. Network 1 has one hidden layer with 50 units, and network 2 has two hidden layers with 50 and 10 units respectively.

The problem in building neural networks is deciding on the initial topology, e.g., how many units there are and how they are connected. Genetic algorithms have been used to explore this problem [Moriarty and Miikkulainen 1994], but it is a large search space and this is a computationally intensive approach. We configure the topology of our networks by simple trail and error approach. For instance, we found that a 50 units' layer is ideal for one hidden layer for our training examples.

In general, if the number of hidden nodes is too large, the network may learn only the training examples, while if the number is too small it may never converge on a set of weights consistent with the training examples.

For the SVM model we used the SVM light [Joachims 1999]. The SVM light is a package implements the Vapnik SVM algorithm, with some optimization that makes it possible to be applied in large-scale problems. We created two SVM models using the SVM light: a linear model, and an SVM model with a radial basis kernel.

A subset of 459,819 training example were drown from WTHOR. This subset was used to train the learning models described above. After training, each model produces an evaluation function. These evaluation functions were used, directly with no search, to play against a base player. In our experiment we choose the base player to be an alpha-beta search player with a depth of 6. The evaluation function of the alpha-beta search is simply the number of discs in the board. Each model played a number of games against the base player. To create different games, the first four moves were played, creating 244 different board positions. We ran the experiment on a 1.5 GHz Windows machine with 256 MB of RAM. The results of the games and the time spent in learning and evaluation is shown in Table 1. Note that in the SVM models, the time of learning and evaluation have a large magnitude in comparison with other models. That is due to the intensive IO operations needed to use SVM light package.

**Table 1. Learning models performance from 244 games against alpha-beta search at depth 6. Learning time is the whole time of training in seconds, where evaluation time is the time spent in each move measured in nanoseconds.**

| | No. of wins | No. of withdraws | Learning time(s) | Evaluation time(ns) |
|---|---|---|---|---|
| Linear regression | 4 | 1 | 10 | 3.3 |
| Neural Network 1 | 26 | 6 | 329 | 13.3 |
| Neural Network 2 | 28 | 5 | 355 | 18.3 |
| SVM –Linear | 21 | 3 | 969 | 426.7 |
| SVM –Radial | 51 | 1 | 9673 | 456.7 |

## 5. Discussion

Researchers have different point of view in linearity of the evaluation function. Some suggest the use of linear functions, others argue with nonlinear functions. From the results of this experiment, we would argue the use of nonlinear functions. Table 1 supports this argument, in which nonlinear models perform more than twice of the linear models. In particular, linear regression model perform poorly, and suggests not to be used in a rich domains, such as Othello. One reason of such poor performance, whereas others found it a good model, is the linearity of our features. Shifting the complexity from the learning model to the feature construction is another way of raising the linearity of the learning model. Figure 4 demonstrates a more clear view of the performance of the learning models.

The SVM model with radial kernel has the best performance among the other models. Although, the SVM model is a linear model, the kernel makes it nonlinear. Applying kernel functions to the features is another way of shifting the linearity to the feature construction. This suggests that the feature construction has a crucial role in the performance. Note that even without a kernel the SVM model outperform the linear regression one; signifying the idea of having flexible linear model helps much.

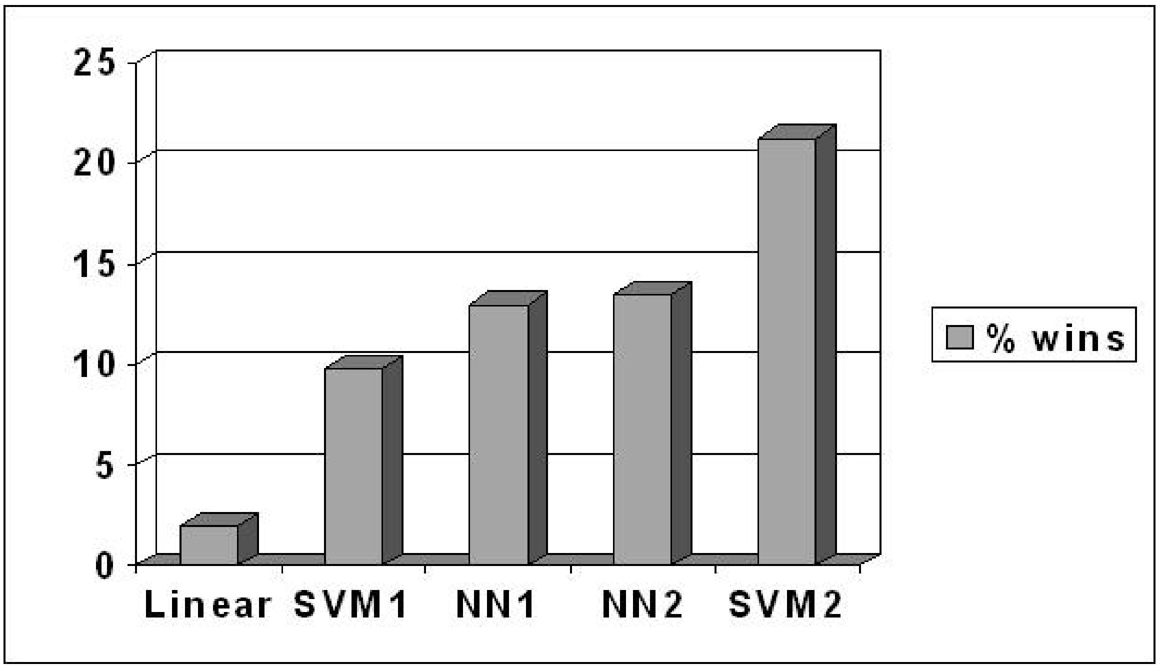

**Figure 4 Percentage of winning. The first column represents the linear regression model results. SVM1 is the SVM linear model, NN1 is the first neural network with one hidden layer, NN2 is the second neural network with two hidden layers, and SVM2 is the SVM model with the radial kernel.**

Other than the SVM models, the time needed for evaluation is relatively equivalent. The learning time is differs much though. It is considerably fast to learn in linear models, and thus suitable for real-time problems, where learning is done online. However, complex-feature construction needs to be considered to achieve nonlinearity and getting a reasonable performance.

In a previous experiment, a random player performance against the base player has been tested. The random player won 6% of the time. Note that the linear regression model has only 2% of wining against the base player, which give the intuition of avoiding the use of linear regression in Othello, especially with the set of primitive features. On the other hand, that suggests the other learning models were far away from random.

We constructed two neural networks; the results showed that the network with two hidden layers performs better than the one with one hidden layer. Thus, the more complex is the model, the better performance. Therefore, we construct a third network with three hidden layers to examine this argument. The performance of the third network remains the same as the one with the two hidden layers. The complex models have boundaries on performance, and may not capture further knowledge.

## 6. Conclusion

We compared the linear and nonlinear learning techniques. We found that the nonlinear learning models acquired more knowledge than the linear models in the domain of Othello. There is a remarkable difference between the two types of learning, suggesting the use of linearity to measure quality of knowledge. The more complex the model, the more knowledge it can extract from the domain, however, there is a bound that limit the performance of the complex models. Feature construction has a major factor in the learning process. Although, using complex hand-crafted features would shift the nonlinearity from the learning algorithm to the feature construction, the process of constructing features is a hard task, and requires involvement of other expertise of the domain.

## References

Berliner, H. (1984). Search vs. knowledge: an analysis from the domain of games. In Elithorn A., and Banerji R. (Eds.), Artificial and human Intelligence, New York, NY: Elsevier.

Bryson, A. and Ho, Y. (1969). Applied Optimal Control. Blaisdell, New York.

Buro, M. (1995). Statistical feature combination for the evaluation of game positions. Journal of Artificial Intelligence Research, 3:373-382.

Buro, M. (1997). The Othello match of the year: Takeshi Murakami vs. Logistello. Journal of the International Computer Chess Association, 20(3):189-193.

Joachims, T. (1999). Making large-scale SVM learning practical. Advances in Kernel Methods – Support Vector Machining.

Junghanns, A., Schaeffer, J. (1997). Search versus knowledge in game-playing programs revised. International Joint Conference on Artificial Intelligence. 692-697.

Lee, F, Mahajan, S. (1988). A pattern classification approach to evaluation function learning. Artificial Intelligence 36, 1-25.

Lee, F, Mahajan, S. (1990). The development of a world class Othello program. Artificial Intelligence 43(1), 21-36.

McCulloch, J. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics.

Moriarty D. and Miikkulainen, R. (1994). Evolving neural networks to focus minimax search. In Proceedings of 12th National Conference on Artificial Intelligence (AAAI-94), pages 1371-1377.

Mitchell, T. (1997). Machine Learning. McGraw Hill.

Rosenbloom, P. (1982). A world-championship-level Othello program. Artificial Intelligence, 19(3):279-320.

Samuel, A. (1959). Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3):211-229.

Samuel, A. (1967). Some studies in machine learning using the game of checkers. ii - recent progress. IBM Journal of Research and Development, 11(6):601-617.

Smola A. and Scholkpf, B. (1998). A tutorial on support vector regression. Neuro COLT Technical report NC-TR-98-03. Royal Holloway College, University of London, UK.

Schaeffer, J. (2000). The games computers (and people) play. In M. V. Zelkowitz (Ed.), Advances in Computers, Volume 50, pp. 189--266. Academic Press.

Tesauro, G. (1988). Connectionist learning of expert backgammon evaluations. In Proceedings of the 5th International Conference on Machine Learning, Ann Arbor, MI.. 200-206.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. Communications of the ACM, 38(3):58-68.

Vapnik, N. (1995). The Nature of Statistical Learning Theory. Springer Verlag.