Using a Support Vector Machine to learn to play Othello

Daniël Karavolos

March 8, 2010

Abstract

Like chess and backgammon, the game Othello is a popular field of application for Machine Learning (ML) techniques. An especially effective technique for learning to play a game is a neural network with Temporal Difference Learning (TDL). Such a neural netwerk has achieved a worldclass level of play in backgammon. Another succesful, increasingly popular ML technique is the Support Vector Machine (SVM). However, this technique has not yet been applied to a game. This experiment compares the use of an SVM to learn to play Othello with the use of a TD neural network and a few other techniques for playing Othello. It appears that the player that is trained with an SVM performs better than a player with random moves. However, it is generally defeated by the heuristic positional strategy and loses almost every game versus the mobility strategy and a TD player.

1 Introduction

Everyone would agree that one needs some form intelligence to strategically play a board game. Therefore, it is not a miracle that board games are a popular field of application for ML research. Besides the learning component, it is an extra challenging application because of the competition with humans. Will we be able to a create computer progam that can defeat a human expert?

The Deep Blue program, made by IBM, is probably the most famous example of such a case. Even though it succeeded mainly by using brute force computations, it defeated the contemporary world champion Gary Kasparov. A program that got less attention, but which was actually far more sophisticated than Deep Blue, was Tesauro's (1995) TDGammon. TDGammon is a program that uses Temporal Difference Learning (Sutton, 1988) to train a neural network to learn a state evaluation function for the game backgammon, while only playing against itself. A technique that has also been applied to other games, like chess (Wiering, Patist, & Mannen, 2007) and Othello (Van den Dries & Wiering, 2007). Judging from abovementioned results with TD learning, it is probably the best learning method for games like checkers, backgammon and Othello so far.

But TDL is not the only Machine Learning technique that can be applied to games. Support Vector Machines could be used to evaluate game states as well. SVMs are generally used to classify certain data. When used to classify a set of game states as resulting in a win or a loss, a proper evaluation function - for choosing a move - may be found.

1.1 This research

The main question of this paper is: "Is Reinforcement learning as Supervised Learning using a Support Vector Machine a better way to learn Othello than Temporal Difference Learning?" The described experiment will compare the use of SVM classification with various parameter settings with two standard strategies for playing Othello and with a neural network trained with a TDL algorithm.

SVMs have been succesfully applied in several areas, like Optical Character Recognition (Cortes & Vapnik, 1995; Schölkopf, Burges, & Vapnik, 1995) for which it was initially developed, object recognition (Blanz et al., 1996) or speaker identification (Schmidt, 1996). In most of these cases, the generalization performance of the SVM is equal to or better than that of competing methods. If it would turn out that the SVM is a better Othello player than the TD neural network, it could indicate that research on creating computer players should be more focused on Support Vector Machines. One could think of applying it to other games or testing whether another type of SVM (e.g. regression) is even more succesful.

1.1.1 Outline of this paper

In the next section I will introduce the game Othello and its standard strategies resulting from the game mechanics. Section 3 will shortly explain Temporal Difference learning and the neural network that is used in the experiment. Section 4 will explain the mechanisms of Support Vector Machine classification. Section 5 describes the experimental setup and section 6 the results for the experiments. Section 7 will conclude this paper with a summary, a discussion of the results and notes on future research.

2 Learning to play Othello

The game Othello is also known as Reversi. It is a two-player game and is played on a board consisting of 8 x 8 squares. The game starts with the middle four squares filled with two pieces of each player, like in figure 1. Black has the first move. A legal move is adding a new piece to the game, which encloses one or more pieces of the opponent between the player's piece. Each of the opponent's pieces then become pieces of the player. A player is only allowed to pass if it is impossible for that player to enclose one of the opponent's pieces.

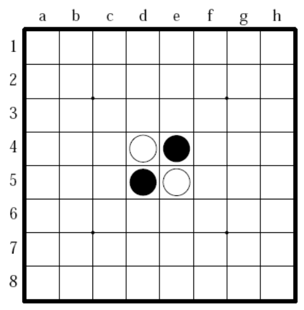

Figure 1: Starting position of Othello

However, not all positions are equally simple to take over. Pieces at the border of the field have only two possible ways to be taken over, instead of eight and the corners of the field cannot be lost at all. Therefore, a positional strategy seems simple, yet sensible. One would try to take control of borders and corners, but refrain from taking positions next to them so as not to give those positions away to the opponent. However, when one has already taken control of a corner or borderposition, it is not as bad to take a position next to it.

Another strategy resulting from these game mechanics is a mobility strategy. During the game one could try to keep as many potential moves as possible, while trying to constrain the potential moves of the opponent. This way, one could force the opponent to do a bad move.

The trick to play this game is to find the right value for each move. There should be an evaluation function which could generate the correct priority list of moves, resulting from the assignment of a value to each move in each state. If there is such an evaluation function it should be possible to approximate this function with a neural network. This will be adressed in the next section.

3 Temporal Difference Learning

Temporal Difference learning is a class of Reinforcement Learning techniques that uses differences between consecutive states to compute a state value function. It can be used to find the optimal weights of a neural network by temporal credit assignment, i.e. propagating the end-state value back to earlier states. Although the basic idea of this algorithm has been around since Samuel's (1959) checker player, the algorithm is usually ascribed to Sutton (1988), because he refined the technique and proved its convergence. The major advantage of this technique compared to standard backpropagation is that it does not need a large database of labeled game records. The network can be trained while just playing games versus itself.

The simplest form is TD(0), that estimates the expected reward of the current state by adding the value of the next state to the next (expected) reward. The update of the state value function after making a transition from state s_t to state s_{t+1} and receiving a reward of r_t on this transition looks like this:

$$\delta V(s_t) = \alpha(r_t + \gamma V(s_{t+1}) - V(s_t)) \tag{1}$$

Where $0 < \alpha \leq 1$ is the learning rate, which should decay over time for proof of convergence.

The opposite of TD(0) is a more obvious approach, called TD(1), which computes the difference between the current expected reward and the final reward. Although both systems will converge with infinitely many training samples, Sutton (1988) has proved that the TD(0) method using a network with linear activation functions converges to optimal estimates with a finite amount of training, while TD(1) produces suboptimal estimates, see (Olson, 1993) for a short, graphic explanation.

The general form is called TD(λ). The setting of lambda interpolates between the two abovementioned extremes. The update function could then be described as follows. Let us first define the TD(0)-error of $V(s_t)$ as

$$\delta_t = (r_t + \gamma V(s_{t+1}) - V(s_t)) \tag{2}$$

TD(λ) uses the factor λ to discount TD(0)-errors of future time steps:

$$\delta V(s_t) = \alpha \sum_{i=0}^{\infty} (\gamma\lambda)^i \delta_{t+i} \tag{3}$$

with $\lambda \in [0, 1]$.

The TD algorithm has been extended to be easier to combine with game-tree search, like the minimax algorithm. The resulting algorithm is called TDLeaf(λ) and is described in (Baxter, Tridgell, & Weaver, 1998). TDLeaf(λ) was the actual algorithm that was used to train the neural network of his experiment.

4 Support Vector Machines

The technique of Support Vector Machines is based on statistical learning theory, or VC theory, which has been developed in the seventies by Vapnik and Chervonenkis (1974) and Vapnik (1982;1995). The SVM got its current form in the nineties by virtue of Boser, Guyon, and Vapnik (1992), Vapnik (1995), (Schölkopf et al., 1995), among others.

Although initially developed as a pattern recognition algorithm, over the years SVMs have been adapted to cope with other types of datasets, which led to several different types of SVMs, like ν-SVM, one-class SVM and SVM regression (see (Chang & Lin, 2001) for a short overview). However, this experiment is based on the original C-Support Vector Classification algorithm, which will be explained in short below.

4.1 Classification

The basic principle of the C-Support Vector Classification algorithm is to construct a maximum margin separating plane between classes. Due to the relevance to this experiment, the explanation of the C-SVC algorithm will be limited to a binary classification task. For a more extensive explanation, please refer to (Burges, 1998; Boser et al., 1992; Cortes & Vapnik, 1995). Due to space limitations of this paper only the maximize margins approach will be explained, refer to (Bennett & Campbell, 2000) for an explanation of the convex hull method.

4.1.1 The linearly separable case

Let us first consider the simplest of cases, two classes that are linearly separable. Let us label the data $\{\mathbf{x}_i, y_i\}$, with $i = 1, ..., l, y_i \in \{-1, 1\}$ and $\mathbf{x}_i \in R^d$. The challenge is now to find a hyperplane that will separate the positive from the negative examples. The points $\mathbf{x}$ which lie on the hyperplane satisfy $\mathbf{w} \cdot \mathbf{x} + b = 0$, where $\mathbf{w}$ is normal to the hyperplane, $|b|/||\mathbf{w}||$ is the perpendicular distance from the hyperplane to the origin, and $||\mathbf{w}||$ is the Euclidean norm of $\mathbf{w}$. A hyperplane supports a class if all points in that class are on one side of that plane. For the points with class label +1 we would like there to exist $\mathbf{w}$ and b such that $\mathbf{w} \cdot \mathbf{x}_i + b > 0$. Let us suppose the smallest value of $|\mathbf{w} \cdot \mathbf{x}_i + b|$ is k, then $\mathbf{w} \cdot \mathbf{x}_i + b \geq k$. The argument inside the decision function is invariant under a positive rescaling so we will implicitly fix a scale by requiring:

$$\mathbf{w} \cdot \mathbf{x}_i + b \geq 1 \tag{4}$$

For the points with class label -1 we similarly require:

$$\mathbf{w} \cdot \mathbf{x}_i + b \leq -1 \tag{5}$$

These can be combined into one set of inequalities:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0 \qquad \forall i \tag{6}$$

To find the plane furthest from both sets, we can simply maximize the distance (or margin) between the support planes for each class. The support planes are "pushed" apart until they "bump" into a small number of data points from each class. These data points are the so-called support vectors. If they were removed, the found solution would change. The margin between these parallel supporting hyperplanes (4) and (5) is $\frac{2}{||\mathbf{w}||}$. We can find the pair of hyperplanes which gives the maximum margin by minimizing $\frac{||w||^2}{2}$, subject to constraints (6).

Thus a solution for a typical two dimensional case should have the form shown in Figure 2.

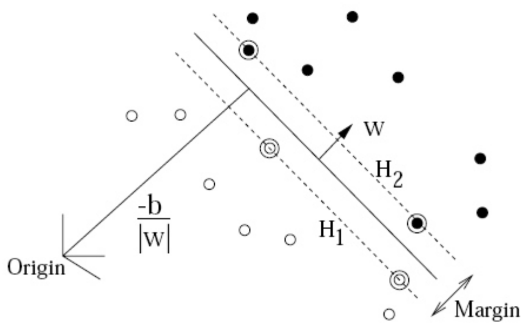

Figure 2: The linearly separable case, support vectors are circled. (Burges, 1998)

It is beneficial to consider the equivalent Lagrangian formulation of this problem. There are two reasons for this. The first is that the constraints (3) will be replaced by constraints on the Lagrange multipliers themselves, which will be much easier to handle. The second is that in this reformulation of the problem, the training data will only appear (in the actual training and test algorithms) in the form of dot products between vectors. This is a crucial property which will allow us to generalize the procedure to the nonlinear case. If we introduce the positive Lagrange multipliers $a_i, i = 1, ..., l$, the Lagrangian dual looks like this (see (Burges, 1998) for derivation):

$$L_D = \sum_i^l a_i - \frac{1}{2} \sum_{i,j}^l a_i a_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \tag{7}$$

with solution:

$$\mathbf{w} = \sum_i a_i y_i \mathbf{x}_i \tag{8}$$

and constraints:

$$\sum_i a_i y_i = 0 \tag{9}$$

$$a_i \geq 0 \qquad \forall i \tag{10}$$

Support vector training, in this case, thus amounts to maximizing L_D with respect to the a_i, subject to constraints (9, 10) and the positivity of a_i, with solution given by 8. Also notice that there is a Lagrange multiplier a_i for every training point. In the solution, those points for which $a_i > 0$ are called support vectors, and lie on one of the hyperplanes. All other training points have $a_i = 0$ and lie either on one of the hyperplanes (such that the equality in (6) holds), or on that side of the hyperplanes such that the strict inequality in (6) holds. For these machines, the support vectors are the critical elements of the training set. They lie closest to the decision boundary, thus if all other training points were removed (or moved around without crossing either hyperplane), and training was repeated, the same separating hyperplane would be found.

4.1.2 The linearly inseparable case

So what happens when this is applied to non-separable data? Even if the data is separable except for just one datapoint, the objective function (i.e. the dual Lagrangian) will grow arbitrarily large and the algorithm will find no feasible solution. Thus to make the algorithm work again, it is

necessary to reduce the influence of any single point. This could be accomplished by relaxing constraints (4) and (5), which demand that each point is on the appropriate side of its supporting hyperplane. Let us introduce nonnegative slack variables $\xi_i, i = 1, ..., l$ in the constraints, which will then look like this:
For $\xi_i \geq 0, \forall i$ and $y_i = +1$:

$$\mathbf{w} \cdot \mathbf{x}_i + b \geq 1 - \xi_i \tag{11}$$

For $\xi_i \geq 0, \forall i$ and $y_i = -1$:

$$\mathbf{w} \cdot \mathbf{x}_i + b \leq -1 + \xi_i \tag{12}$$

However, any point falling on the wrong side is considered to be an error. So the goal is now to simultaneously maximize the margin and minimize the error. The objective function should thus be changed from $\frac{||w||^2}{2}$ to $\frac{||w||^2}{2} + C \ \sum \xi_i$; where C is the weight of the penalty term, a parameter to be chosen by the user.
The Lagrangian dual of this problem then looks like this:

$$L_D = \sum_i^l a_i - \frac{1}{2} \sum_{i,j}^l a_i a_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \tag{13}$$

with contraints:

$$0 \leq a_i \leq C \tag{14}$$

and:

$$\sum_i a_i y_i = 0 \tag{15}$$

Again, the solution is given by

$$\mathbf{w} = \sum_i^{N_S} a_i y_i \mathbf{x}_i \tag{16}$$

where N_S is the number of support vectors. The only difference from the optimal hyperplane is that the a_i now have an upperbound of C. Figure 3 summarizes this situation.

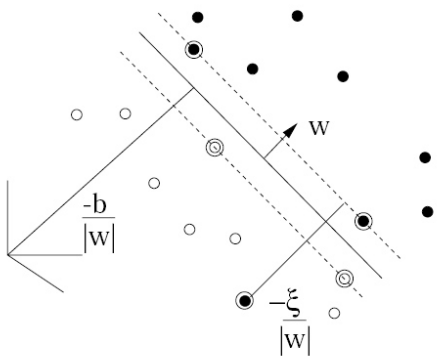

Figure 3: The linear inseparable case, support vectors are circled. (Burges,1998)

4.1.3 The nonlinear case

So now it is not necessary that the clusters of datapoints are neatly separated. But what happens when the data is not separable by a straight line, like in Figure 4?

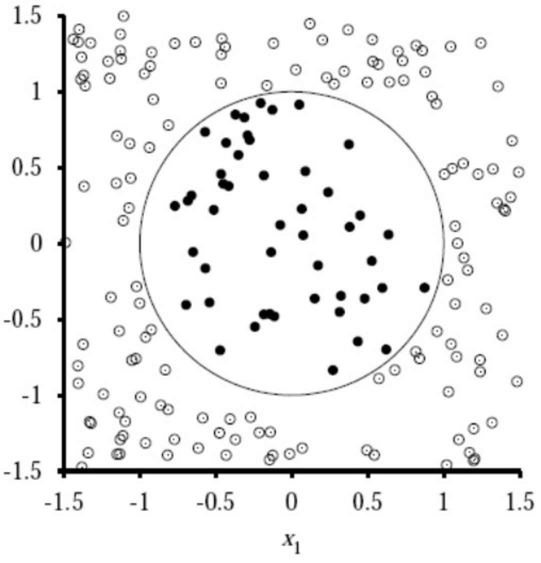

Figure 4: Data with a circle as classification function. (Russel & Norvig, 2002)

This specific example needs a quadratic function, i.e. a circle, as classification function. This could be achieved by adding additional attributes to the data that are nonlinear functions of the original data. When existing linear classification algorithms would then be applied to the new feature space, they would produce nonlinear functions in the original input space. Let's call this higher (possibly infinite) dimensional feature space H and the mapping from the input space to this space Φ, so that:

$$\Phi : \mathbf{R}^d \mapsto H \tag{17}$$

Notice now that the representation of the original problem is in the form of dot products, equations (13 - 16), and that with the abovementioned mapping, the training algorithm would still depend on the data through dot products. Yet now it would be in H and the dot products would of the form $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$. If there would be a "kernel function" K such that $K(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$ one would only need to use K in the training algorithm and never even have to know explicitly what Φ is.

So when computing the sign of

$$f(x) = \sum_{i=1}^{l} a_i y_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) + b \tag{18}$$

with constraints 14 and 15, we can substitute the dot product by a kernel function and instead calculate

$$f(x) = \sum_{i=1}^{l} a_i y_i K(\mathbf{x}_i, \mathbf{x}_j) + b \tag{19}$$

The advantage is that (19) needs less computation, because the function is already given. Actually it is not given, one is free to choose any kernel of their liking. However, it is preferential to choose a kernel that satisfies Mercer's Condition ((Vapnik, 1995); Courant and Hilbert, 1953). Then convergence, i.e. a solution, is guaranteed. Popular kernels are:

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^p \tag{20}$$

$$K(\mathbf{x}, \mathbf{y}) = e^{-\|\mathbf{x}-\mathbf{y}\|^2/2\sigma^2} \tag{21}$$

$$K(\mathbf{x}, \mathbf{y}) = \tanh(k\mathbf{x} \cdot \mathbf{y} - \delta) \tag{22}$$

The abovementioned functions respectively result in a polynomial classifier of degree p, a Gaussian radial basis function (RBF) classifier and a particular two-layer sigmoidal neural network. The experiment described in this paper uses the most common kernel, a Gaussian radial basis function.

5 Experimental setup

The experiment uses a 3 x 3 x 2 design. The first parameter is C, the punishment of the violation of constraints (4) and (5) in section 4.1. The three different values are 1, 16 and 256. When computing the number of errors the SVM is allowed to make in a set of games, a good value for C would be 16. However, some preliminary tests showed that the value of C doesn't really matter. The other two values are values on both sides of this default with a relatively large distance to enlarge possible effects of changes in C.

The second parameter is γ, a parameter for the gradient of the RBF. The three used values are 1/16, 1/32 and 1/64. The default value would be 1/64, which is one over the number of features in the dataset, each feature being the occupancy of a certain square in the 8 x 8 field.

The third parameter is the number of training games per batch and the two values are 25 and 50. One could hypothesize that the number of games trained on each time has an effect on the acquired knowlegde of the game. Twentyfive games per batch means that the SVM is trained quite often, but with few examples. Fifty games per batch means that the SVM is trained fewer times, but with more examples.

The neural network that is used as an opponent is a fully connected neural network with 64 inputs, 8 hidden neurons and 1 output (even though better performing neural networks can be constructed, see (Van den Dries & Wiering, 2007)). The network was trained on 10.000 games of self-play with the TDLeaf(0) algorithm. For efficiency reasons each state's mirrored input was also trained on, because of the symmetry of the board. For more details of this network, see (Van den Dries & Wiering, 2007).

5.1 Training

Each combination of parameters was trained on 1000 games of self-play and every 100 games a model of the contemporary settings was saved. Before the SVM player was trained, it would generate a random value between -0.1 and 0.1 for each possible next state. The number of times the player was trained depended on the 'number of games per batch' parameter. The package LIBSVM (Chang & Lin, 2001) was used for training the actual SVM. Each combination of parameters was trained five separate, complete runs, which resulted in five different players with the same SVM parameters. This was done to compensate for noise in the trainingset and to be able to quantify testresults.

One might wonder why the SVM is trained on only a thousand games, while the neural network it is compared with is trained on ten thousand games. One of the SVMs was trained on ten thousand games as well, but preliminary results showed that the win percentage versus a random opponent hardly changed after a few hundred games. To save time, the number of training games was then decreased to one thousand.

5.2 Testing

After training, each of the 18 parameter settings had five players. Each of those five players were then tested against four opponents with different strategies: random, positional strategy, mobility strategy and a TD neural network as described above. Each of the SVM trained players consisted of ten models and each model played 244 unique games against each opponent. In each of these games the first four moves (two of each player) were done randomly to create a wider variety of games than which would be the case with a normal start. To account for the unfairness of the random starting positions each player played each game twice, once as the black player and once as the white player. This resulted in a total of 5 * 2 * 244 = 2440 games per model per opponent for each parameter combination. Each win counted as a full point, each draw counted as half a point.

6 Experimental Results

The testresults will be presented in order of the 'games per batch' parameter.

6.1 Results of 25 games per batch

6.1.1 Results per model

Figure 5 shows the average win percentage of each model versus each opponent, summarized over all parameter settings. Note that each line is horizontal, which means that there is no learning curve. At first look there seems no significant difference between the models.

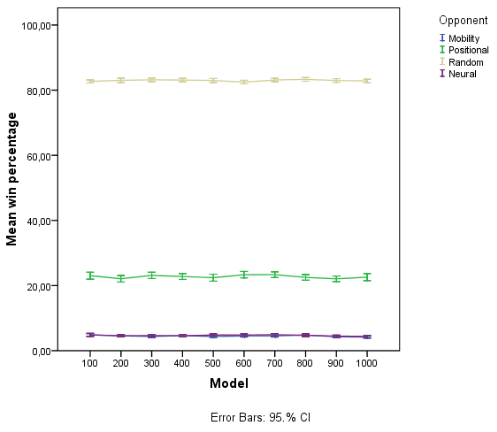

Figure 5: Average win percentage versus an opponent, per model (25 games per batch).

An overview of the mean win percentage and standard deviation(SD) versus each opponent is presented in Table 1. None of the means of each individual model deviates more than one SD from the overall mean, while the SD of the individual models is similar to the overall SD. Thus it seems very unlikely that any further analysis will show a significant difference between the models.

Table 1: Mean win percentage and standard deviation (SD) against each opponent

Opponent	Win percentage	
	Mean	SD
Random	0,8297	0,0187
Positional	0,2273	0,0321
Mobility	0,0455	0,0131
TD neural network	0,0463	0,0134

6.1.2 Results per parameter setting

Figure 6 shows the average win percentage of each SVM setting versus each opponent, summarized over all models. Again the results are horizontal lines. This would mean that there is no difference between the different parameter settings. Because of space limitations the settings are abbreviated in the graph. The abbreviation C1G16, per example, means $C = 1$ and $\gamma = 1/16$.

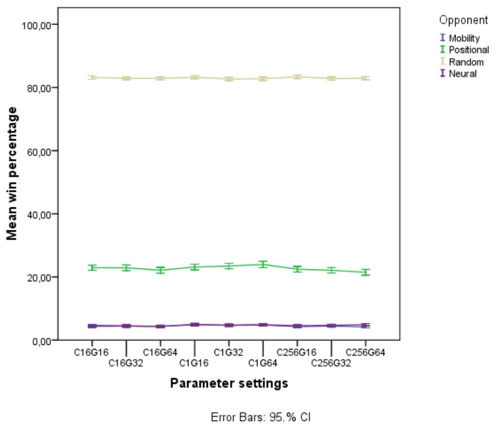

Figure 6: Average win percentage versus an opponent, per parameter setting (25 games per batch).

The average win percentages are of course the same as in Table 1. The means of the indiviual parameter settings also do not deviate more than one SD from the total average, while they also have a SD similar to the overall SD. Thus it seems very unlikely that any further analysis will show a significant difference between the various parameter settings.

6.2 Results of 50 games per batch

6.2.1 Results per model

Figure 7 shows the average win percentage of each model versus each opponent, summarized over all parameter settings. Each line is horizontal, which means that there also is no learning curve for the case of 50 games per batch.

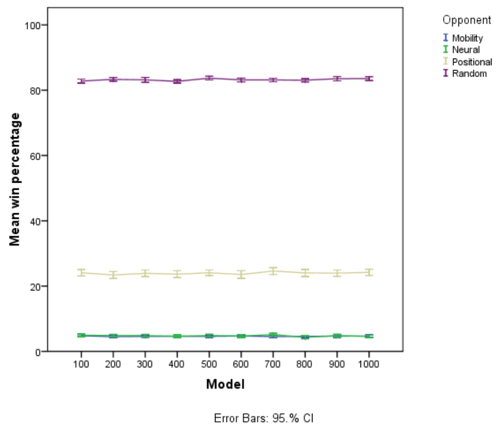

Figure 7: Average win percentage versus an opponent, per model (50 games per batch).

An overview of the mean win percentage and standard deviation versus each opponent is presented in Table 2. Like the results of 25 games per batch, none of the means of each individual model deviates more than one SD from the overall mean, while the SD of the individual models is similar to the overall SD. Thus again it seems very unlikely that any further analysis will show a significant difference between the models.

Table 2: Mean win percentage and standard deviation (SD) against each opponent

Opponent	Win percentage	
	Mean	SD
Random	0,8320	0,0199
Positional	0,2396	0,0339
Mobility	0,0474	0,0145
TD neural network	0,0464	0,0125

6.2.2 Results per parameter setting

Figure 8 shows the average win percentage of each parameter setting versus each opponent, summarized over all models. Again, the results are horizontal lines. This would mean that also for 50 games per batch, there is no difference between the different parameter settings. The settings are abbreviated in the same way as in Figure 6.

The average win percentages are the same as in Table 2. The means of the indiviual parameter settings also do not deviate more than one SD from the total average, while they also have a SD similar to the overall SD. Thus it seems very unlikely that any further analysis will show a significant difference between the various parameter settings.

7 Discussion

The results of this experiment are disappointing in three ways. Not only was the SVM not able to perform better than any player with a strategy, there also was no significant difference in

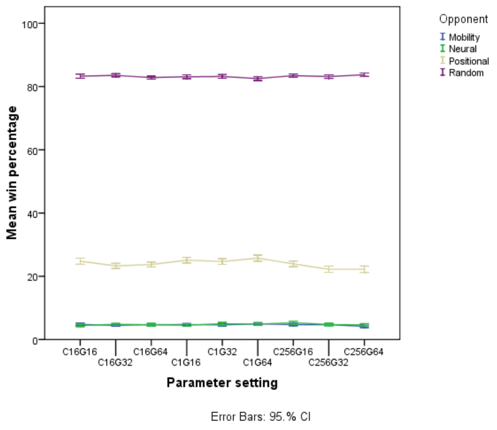

Figure 8: Average win percentage versus an opponent, per parameter setting (50 games per batch).

performance between the various parameter settings. Perhaps the most disappointing result is that the performance of the SVM did not increase over time, i.e. it didn't learn.

The fact that the parameter settings and the total number of played games did not influence the performance at all hints that there is something wrong with the SVM. After all, under normal circumstances all those variables should have some effect. Thus, there must be something wrong with the representation of the problem. There are two things that contribute to the representation of the problem, viz. the input of the SVM and the kernel function.

Let us first consider the input of the SVM. It is a vector of 64 numbers, each number describes the occupancy of one square of the board using 1 for a square occupied by itself, 0 for an unoccupied square and -1 for a square occupied by the opponent. It is quite a raw input, no knowledge of the game has been imposed on or added to it. This is of course what we want, because the SVM has to work everything out for itself. Removing some of the input would lead to an incomplete image of the game and adding more input would violate our goal of autonomy. So changing the input does not seem to be a viable option.

However, there is no such argument for the kernel function. Although the SVM was equiped with the most common kernel in literature, the Gaussian radial basis function, it could easily be replaced by another kernel. Perhaps the implicit mapping of the RBF is not the right mapping for this data. Surely, it gives a solution, but perhaps this somehow captures the wrong features. Since the game Othello is obviously not a linear problem, the polynomial and the sigmoidal kernel remain as possible options.

Future research should be directed towards using these two other kernels. Given the comparison between a high dimensional polynomial and a neural network with 64 inputs, the sigmoid seems more likely to provide interesting results. Another possible direction would be to try the use of SVM regression instead of classification. For this problem the regression algorithm would try to approximate the optimal evaluation function of the board states, instead of classifying the moves that lead to a win or a loss (for more on SVM regression, see (Smola & Schölkopf, 2003)).

References

Baxter, J., Tridgell, A., & Weaver, L. (1998). Tdleaf(λ): Combining temporal difference learning with game-tree search. In *Proceedings of the 9th australian conference on neural networks (acnn-98)* (pp. 39–43).

Bennett, K. P., & Campbell, C. (2000). Support vector machines: Hype or hallelujah? *SIGKDD Explorations*, *2*(2), 1-13.

Blanz, V., Scholkopf, B., Bultho, H., Burges, C., Vapnik, V., & Vetter, T. (1996). Comparison of view-based object recognition algorithms using realistic 3d models. In *Artificial neural networks icann96* (pp. 251–256). Springer.

Boser, B., Guyon, I., & Vapnik, V. A. (1992). A training algorithm for optimal margin classifiers. In *In fifth annual workshop on computational learning theory.* ACM.

Burges, C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, *2*, 121–167.

Chang, C.-C., & Lin, C.-J. (2001). LIBSVM: a library for support vector machines [Computer software manual]. (Software available at `http://www.csie.ntu.edu.tw/ cjlin/libsvm`)

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Leaming*, *20*, 273-297.

Dries, S. Van den, & Wiering, M. (2007). *Het verkrijgen van evaluatiefuncties voor othello met behulp van temporal difference leren en co-evolutie.* (Unpublished)

Olson, D. (1993). *Learning to play games from experience: An application of artificial neural networks and temporal difference learning.*

Russel, S., & Norvig, P. (2002). *Artificial intelligence: A modern approach (second edition).* Prentice Hall, New York.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, *3*, 210-229. (Reprinted in E. A. Feigenbaum & J. Feldman (Eds.), Computers and thought. New York: McGraw-Hill.)

Schmidt, M. (1996). Identifying speaker with support vector networks. In *Interface 96 proceedings.*

Schölkopf, B., Burges, C., & Vapnik, V. (1995). Extracting support data for a given task. In U. Fayyad & R. Uthurusamy (Eds.), *Proceedings, first international conference on knowledge discovery & data mining.* AAAI Press.

Smola, A., & Schölkopf, B. (2003). A short introduction to learning with kernels. In *Advanced lectures on machine learning* (p. 41 - 64). Springer Berlin / Heidelberg.

Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*, 9–44.

Tesauro, G. (1995). Temporal difference learning and tdgammon. *Communications of the ACM*, *38*(2), 58–68.

Vapnik, V. N. (1982). *Estimation of dependences based on empirical data.* Springer, Berlin.

Vapnik, V. N. (1995). *The nature of statistical learning theory.* Springer-Verlag, New York.

Vapnik, V. N., & Chervonenkis, A. (1974). *Theory of pattern recognition (in russian).* Nauka, Moscow. (German Translation: W. Wapnik & A. Tscherwonenkis, Theorie der Zeichenerkennung, Akademie-Verlag, Berlin, 1979)

Wiering, M., Patist, J., & Mannen, H. (2007). *Learning to play board games using temporal difference methods* (Tech. Rep.). University of Utrecht.