Opponent Modeling in Machiavelli

M.H.J. Bergsma

19th June 2005

Abstract

Opponent modeling is a technique in computer game-playing which attempts to create a model of an opponent's strategy. This model can then be used to predict the opponent's future actions. This paper attempts to apply opponent modeling to the commercial card game Machiavelli[1], a game containing imperfect information. Neural networks are used to build the models. These neural networks are trained using both the backpropagation algorithm and genetic algorithms. To test this approach, neural network opponent models are trained on a number of bot strategies. Some promising results are obtained. However, both methods of training are currently unfit for use in practical applications.

1 Introduction

The aim of this paper is to study the use of opponent modeling in the card game Machiavelli. Unlike most games studied in computer game-playing, most notably computer Chess, Machiavelli is a game of imperfect information. Traditional approaches to computer game-playing, such as tree-search, are not applicable to such games. Because of this, progress on most games containing imperfect information has been stale. However, some succes has been achieved using the method of opponent modeling.

Opponent modeling attempts to create a model of an opponent's strengths and weaknesses by observing this player's actions and identifying his strategy. This approach has primarily been used in games with complete information, as an enhancement to existing tree-search algorithms [2, 7]. However, it also has great value in the domain of imperfect-information games. In these games, an opponent model can be used to predict the actions an opponent is going to take. Opponent modeling in this way has been succesfully implemented in the game of poker by Billings *et al.* [1, 4, 5].

[1] Machiavelli is a registered trademark of Hans im Glück Verlags GmbH, 80809 München.

In the game of Machiavelli, at each turn, players have to choose from a variety of different characters, each character possessing unique abilities. Choosing the right character, and predicting the hidden character choices of your opponents, is essential to winning the game. This paper will use opponent modeling to try to accurately predict the character choices of every opponent. Therefore the problem statement reads:

How can an opponent model be created so that it can be used to accurately predict an opponent's character choices in the game of Machiavelli?

A number of research questions can be derived from this statement:

- In which ways can an opponent model be created, and which way is best suitable for Machiavelli?
- What are the factors that contribute to a character choice in Machiavelli?
- Can opponent modeling be used to overcome the limitations caused by the presence of imperfect information in Machiavelli?

1.1 The Rules of Machiavelli

Machiavelli, also known as Citadels, is a card game for 2-7 players designed by Faidutti [8]. The game consists of 8 character cards, 64 district cards and 30 gold pieces. Every district card has one of 5 colors (green, yellow, red, blue or purple) and a value in gold pieces. A character card has a sequence number from 1 to 8, and may have a color. Every character card also has its own unique special abilities.

At the start of the game, a crown counter is assigned to one of the players, typically the oldest. Every player then receives 2 gold pieces and 4 (hidden) district cards. The game is played in rounds, and each round consists of two phases. In the first phase, the character cards are divided between the players. It starts by randomly putting a certain number of character cards, which can be calculated as $\max(0, (6 - n))$, where n is the number of players, face up on the table. Another character card is then randomly discarded face down. These cards are unavailable for choosing this round. The player with the crown counter then secretly selects a character from the remaining cards, and passes the rest on to the player at his left. That player also chooses a character, and so

on until every player has made a choice. The remaining card is also put face down on the table. For 7 players the procedure differs slightly, see [8] for details.

In the second phase players play their turns. During this phase the crown player calls the names of the characters in sequence. Whenever the character chosen by a certain player is called, that player announces his character to the others. He then plays his turn.

At the start of a turn, the player has to choose between receiving extra gold or district cards. He then has the option of building one of the district cards in his hand, by paying (to the bank) this district's value in gold. He may also use his character's special ability once at any time during his turn. The eight different characters in the game are:

1. *Assassin*: Can murder another character. A murdered character skips his turn in silence.
2. *Thief*: Can rob another character. A robbed character gives all his gold to the Thief, at the start of the robbed character's turn.
3. *Magician*: Can trade his entire hand with another players' hand, or trade in any number of cards.
4. *King*: Receives 1 gold for each yellow district. Receives the crown counter next turn.
5. *Bishop*: Receives 1 gold for each blue district. Has immunity against the Warlord's power (unless he is assassinated).
6. *Merchant*: Receives 1 gold for each green district. Receives 1 extra gold at start of turn.
7. *Architect*: Receives 2 extra district cards. May build up to 3 districts a turn.
8. *Warlord*: Receives 1 gold for each red district. May destroy one district, after paying this district's value−1 to the bank.

When every player has played his turn, the round is over. New rounds are played until one of the players builds his 8th district. At the end of this last round, scores are calculated as the sum of the values of each player's districts. Bonus points are given to the first player to build 8 districts (4 points), every subsequent player to build 8 districts (2 points), and any player who has built a district of each of the five colors (3 points). The player with the most points is then declared winner.

The remainder of this paper is organised as follows. Section 2 contains a short introduction to opponent modeling, and explains how it can be used in the game of Machiavelli. Section 3 describes how an opponent model can be created. The experimental setup is given in Section 4, and results are presented in Section 5. Finally, conclusions and ideas for future research are provided in Section 6.

2 Opponent Modeling

The purpose of this section is to explain the concept of opponent modeling, starting with a general introduction in Section 2.1. Then, Section 2.2 will show how opponent modeling can be used in Machiavelli. Finally, some requirements and limitations of opponent modeling in Machiavelli are listed in Section 2.3.

2.1 Introduction to opponent modeling

In any competitive game, the goal of every player is to defeat the other players. He will therefore need to find a better strategy than his opponents. Naturally, the moves opponents make affect the outcome of the game. Any good strategy should therefore take into account the strategies of other players. Usually, the assumption is made that opponents always play rationally, i.e., they play the move that seems the best under the circumstances. Obviously, this is not always the case. An opponent may not be able to find the 'best' move, or he may be biased towards certain strategies. An erroneous estimation of an opponent's strategy can lead to sub-optimal results.

By observing an opponent's moves, his strengths and weaknesses can be found and modeled. Instead of assuming the opponent plays optimally, a prediction for his next move can be made, based upon his previous moves. For example, in Machiavelli, a player can have a notable preference for a certain character, or may often choose the character corresponding with the color of which he owns the most districts. Naturally, most players will try to avoid repetition to make it harder for his opponents to guess their choices. The higher the variation in moves, the more difficult it is to learn a model of the opponent.

Perfect-information games can be succesfully played by performing searches of the game tree. To the best of our knowledge, as yet there is no method that enables standard tree-search algorithms to sufficiently deal with imperfect-information game trees. Therefore, opponent modeling could be a very useful alternative for achieving good results in such games.

2.2 Opponent modeling in Machiavelli

In the game of Machiavelli, choosing the right character is often the key to success. Characters provide the player with special abilities over his opponents, thereby creating opportunities to gain the leading position, and ultimately winning the game.

Some characters, like the Merchant and Architect, are designed solely to benefit their owners. Other characters, like the Assassin and the Thief, have abilities that benefit their owner by hindering his opponents. Most of these abilities target characters rather than players, although often one would want to hinder a certain player, like the player who is currently ahead in points. In

such cases knowledge of an opponent's choosing strategy would be very helpful.

In general, being able to accurately predict other players' characters can help a player to determine his opponents' strategies, and adapt his own strategy accordingly. For example, being fairly sure which character a certain player with a lot of gold will choose, makes choosing the Thief and robbing this character a valuable strategy. Also, one could infer from his predictions of his opponents' choices whether or not the Assassin has been available for selection. If not, it is safe to choose a valuable character like the Architect, but if the Assassin has likely been selected by another player, this choice would be unwise.

This paper attempts to design an opponent-modeling system that can learn to create opponent models that can accurately represent an opponent's choosing strategy for Machiavelli. Hereby we are concerned only with predicting this choice. How to use such a model during the players' turns is beyond the scope of this paper.

2.3 Requirements and Limitations

A good opponent model needs to be able to learn an opponent's strategy quickly and accurately. Here, quickly means using a small amount of training examples. This is particularly important for online learning, where the opponent is modeled during gameplay itself. For Machiavelli, it is unfeasable to start learning an opponent model from scratch during the game, as games usually last about 10-15 rounds, yielding only as many training samples as there are rounds. Aside from the fact that strategies for choosing are dependent on the phase of the game (e.g. the Warlord is much more valuable near the end of the game than at the beginning), no model can be expected to be useful, given so few examples of the opponent's strategy. Therefore, an opponent model has to be learned offline, before the start of the game. This restriction is in accordance with the real world, as human players often use knowledge of their opponent's strategies in previous games. When two players play against each other for the first time, they have no information about each other's playing styles, but if they play each other many times, they learn their opponent's strategy increasingly well. Also, many professional gamers study previous games of their opponent before a big match.

A more limiting aspect of opponent modeling in Machiavelli is the presence of imperfect-information. The ultimate goal of opponent modeling in this research is to try to eliminate the imperfect-information regarding the hidden choices of characters, but there are other aspects of unknown information making this goal more difficult to achieve. For instance, it is unknown which cards an opponent holds in his hands, and therefore unknown what actions this player will need to achieve a good result. To determine which character would give this player the best result an opponent model must go by parameters like the amount of gold and cards the player has. In many cases, the opponent will try to maximize his profits, for example choosing a character with which the player can gain more gold than needed to build a specific district. In those cases the opponent model will not be limited by a lack of knowledge about the opponent's cards.

The most problematic limitation is the fact that every player can only choose from a usually unknown subset of characters each time. One of the characters, known only by the current King player, is unavailable from the start, and characters previously selected by others are obviously unavailable for the remaining players. This means that in almost every case, the set of characters an opponent can choose from is unknown. Therefore, a prediction regarding this set has to be made based on the predicted choices of previous players, and the unavailable card. The King player has a major advantage in this case, as he is the only one to know with certainty which card is unavailable. As this character is selected completely at random, the other players have very few options to determine which character it is. As every player's set of available characters (and therefore their choice) is based on which characters are unavailable, an incorrect estimate of this character may lead to a chain of incorrect predictions. Still, every player knows the set of characters the player after him has to choose from, and knows which n characters will be distributed over the remaining $n-1$ players. The first player to choose knows which of these characters will not be used during this turn.

The fact that every player has to choose from a certain subset of characters also leads to another problem. An opponent model acts as a predictor, generating a probibility distribution over the possible actions the opponent can take. In this case, the set of possible actions is usually unknown, and is made up of different sizes and elements each time. Most opponent-modeling techniques, such as the one used in this research and described in Section 3.2, have difficulties adapting to a variable set of outputs. To this problem we propose the following solution: Have the opponent model give a probability distribution over *all* possible outputs, and normalise these probabilities over the expected set of outputs. This yields a new distribution from which the chosen character can be predicted. The assumption made here is that at any time, the player has a certain measure of preference for each character, and selects his character using these preferences. Some players will always select the character with the highest preference, while other player's choices can be predicted by drawing from the estimated distribution.

3 Approach

This section describes how the opponent models used in this research are created. Section 3.1 gives an overview of some of the techniques that could be used for this purpose. The architecture of the neural networks, which are used for building the models, is explained in Section 3.2. Before being able to use a neural network, its parameters have to be initialised by training. Two training algorithms have been researched for this paper, and both are briefly described in Section 3.3.

3.1 Opponent-modeling techniques

An opponent model can be implemented in several ways. One possibility is to collect statistical data from an opponent's play. This has previously been applied to the game of Poker by Billings *et al.* [1]. The results were favourable, though by the authors' own admission, the technique was limited in flexibility and accuracy. They went on to replace their approach by neural networks. In contrast to statistical opponent modeling, where relationships between statistics and actions have to be defined manually, a neural network is able to implicitly learn relationships between game state and actions automatically. In the Poker research, the use of neural networks has achieved significantly better results [3, 4].

3.2 Opponent model architecture

For this research, opponent models are created using neural networks. The main advantage of neural networks is that they are able to approximate almost any function, without requiring any domain-specific knowledge. Furthermore, they are flexible and can handle errors in training data very well. More information about neural networks can be found in [10, 11].

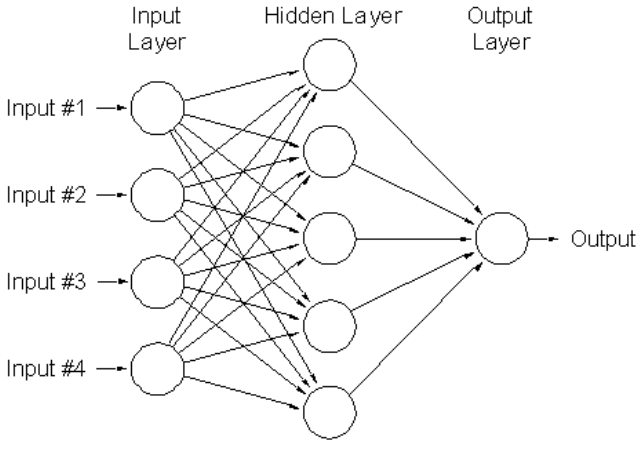

Figure 1: An example neural network.

The neural network architecture used for this research is a multi-layered network with one hidden layer, as just one hidden layer has been shown to be sufficient for most applications [11]. An example of this type of neural network is displayed in Figure 1, though this network differs from the ones used in this research by the number of nodes in each layer. There is no general rule to determine the number of neurons in the hidden layer. This number is therefore usually picked arbitrarily, by experimenting with different sizes. See the experiments in Section 4.3 for more information.

nr.	Range	descrption
1.	[0, ∞]	Current turn
2.	[0, 30]	Amount of gold
3.	[0, 65]	Score
4.	[0, 64]	Hand size
5.	[0, 10]	#Districts
6.	[0, 6]	#Green districts
7.	[0, 3]	#Yellow districts
8.	[0, 4]	#Red districts
9.	[0, 4]	#Blue districts
10.	[0, 10]	#Purple districts

Table 1: Overview of inputs for the neural network opponent model

The number of input neurons is partly dictated by the problem statement. In Machiavelli, any information about the current state of the game, such as amount of gold or cards held by certain players, can be used as input. The importance of each input will be determined during the training of the network, as will be shown in Section 3.3. Table 1 gives an overview of the inputs used in this research.

The goal of the neural network is to predict the character a certain opponent will choose. One possibility would be to have a single output neuron, which would produce the number of the character the opponent will choose, given the network's inputs. However, this information is meaningless. In this case, the character sequence numbers act as a label, instead of a numerical value, and no correlation between the different numbers exists. For example, if the network output is 4, this does not mean that characters 3 and 5 would also be likely choices. A more preferable output would be a probability distribution over the available characters. The assumption made here is that players always have a certain measure of preference for every character. A character prediction can then be made by either drawing from this distribution, or just picking the character with the highest preference. If a prediction turns out to be incorrect, indicating a flaw in the opponent model, the information gained from the distribution can be used to improve the model. A single output would not give any indication as to the cause of the error.

Because the set of available characters is variable and unknown, and it would be impossible to create a network for every subset of characters, the network requires an output for every character in the game. By using only the outputs of the (expected) available characters and normalising them, a probability distribution is created.

The transfer functions used in all layers is the log-sigmoid transfer function, given as:

$$x_i = \frac{1}{1 + e^{-a_i}}, \quad (1)$$

where a_i is the input, and x_i is the output for node i.

3.3 Training the opponent model

In some simple networks, weights can be determined manually using domain-specific knowledge. For most networks this is impossible however, so usually a training algorithm is used to calculate the weights automatically. The standard algorithm for training a multi-layered neural network is the backpropagation algorithm. Another popular technique is using genetic algorithms to train neural networks. Both of these techniques have been used for this research.

The backpropagation algorithm is the first and most widely used method for training multi-layered feed-forward neural networks. More information about the backpropagation algorithm, including the Levenberg-Marquardt variant used here, can be found in [10].

Neuro-evolution is the process of using genetic algorithms (GAs) to train a neural network. Rather than being a specific algorithm, neuro-evolution refers to an entire class of possible algorithms, based on the principles of GAs. An explanation of the method used in this research is given in [12]. This method has been designed and implemented by De Jong [6].

4 Experimental Setup

This section describes the experiments that have been done to test the performance of the opponent models as explained in the previous section. For these experiments, a computer simulation of the game has been created, as will be briefly shown in Section 4.1. Because it is difficult to test the models on human players, a number of bots have been created, each with its own character choosing strategy. These bots are listed in Section 4.2. Section 4.3 will then list some of the considerations that had to be made when training the neural networks. This section also shows exactly which experiments have been performed. The results of these experiments will be presented in Section 5.

4.1 Simulation environment

In order to test the methods described above, a computer simulation implementing the rules of Machiavelli has been built. This simulation was written in Java in cooperation with Groeneweg [9]. It implements all the rules of Machiavelli, with one exception. The purple district cards, which have special abilities in the regular game, are powerless in the simulation. These cards are not essential to the game, and therefore unnecessarily complicate the rules.

The rules of Machiavelli vary from country to country. The simulation exactly follows the version of the rules designed by creator Bruno Faidutti, as listed on his website [8].

4.2 Bot strategies

Several bots, or computer players, have been implemented for use with the simulation, to test the opponent models created for this research. These bots have in common that, every time they have to choose a character, they calculate a preference for each of the 8 characters, and subsequently normalise these to create a probability distribution. Finally, from the still available characters they select the one with the highest preference/probability. For the turn phase of the game, all of the bots use an unfinished version of Groeneweg's bot [9]. The different bots are:

- *ResourceBot*: This bot uses a few simple calculations to determine its preferences, such as the number of built districts of a certain color (affecting the character corresponding to this color), and the number of cards (affecting the Magician and the Architect).
- *Bot-KnowAll*: The character-choosing strategy used by Groeneweg's bot, as explained in [9].
- *CitadelsBot*: This bot is based on a Machiavelli bot created by W.L. Sims for an online Java implementation [13]. It calculates preferences mostly using basic inputs (amount of gold, hand size, number of districts by color, etc.) and the popularity of the characters, that is, the number of times they have been selected in the past.
- *SequenceBot*: A bot that simply selects the first available character.
- *RandomBot*: As this bot calculates all preferences at random, it should be impossible to model correctly. It can be useful to find a lower bound on the performance of an opponent model.

4.3 Opponent Models

As described in Section 3, opponents are modeled using neural networks, and trained in two different ways: using the backpropagation algorithm and using neuro-evolution. Both of these methods have been implemented in the computer simulation. The backpropa-

gation neural networks are trained in the MATLAB[2] software environment, which utilises the Levenberg-Marquardt variaton of the backpropagation algorithm. For neuro-evolution networks, De Jong's toolkit [6] is used.

Besides the two approaches mentioned above, an experiment has been performed using a random opponent model. This is not really an opponent model but, as the name suggests, it predicts characters randomly. Similar to the RandomBot, the random opponent modeler can be used to compare performances to the worst possible opponent model, namely a model that makes no assumptions about the opponent at all.

Determining the number of hidden nodes

Some experiments were performed to estimate the optimal number of hidden nodes to be used in the neural networks. This was done by training an opponent model for a certain bot (CitadelsBot) on a range of different numbers of hidden nodes, and comparing the performances. The range used here was the interval from 1 to 15, initializing every network 10 times.

Backpropagation training

To create the opponent models using backpropagation neural networks, a set of training data is required. This set should consist of a number of input vectors, together with their desired output vectors. In this case, the input vectors are combinations of input values as given in Table 1, and the outputs are vectors of 8 real-valued numbers from the [0, 1] interval, each one denoting the opponent's preference for a certain character. The trainig set can be obtained for a certain opponent by running the simulation a certain number of times, and extracting the desired inputs and outputs every time the opponent has to choose a character. No method to determine the required amount of training examples exists, but generally, the more the better. To create the training sets, data from 500 simulated games is used.

Another consideration when using the backpropagation algorithm is the number of training iterations. More iterations means the networks will be better fitted to the training data, but too many iterations can lead to overfitting. Empirically, a maximum of 50 iterations has been chosen for all data sets.

The backpropagation algorithm is sensitive to initialization. Network weights are initialized randomly, and the algorithm subsequently converges the performance to a local minimum. To improve the network performance, the algorithm should be performed a number of times, on different initializations. The resulting network with the best performance can then be selected for use. Here, we have chosen to use 10 network initializations.

[2]MATLAB is a registered trademark of The MathWorks, Inc.

Neuro-evolution training

For neuro-evolution training, an initial population of 50 individuals was created. Then, for every generation, for each individual a game of Machiavelli was played, whith the first bot using an opponent model based on the current individual. The fitness function used was the ratio of correct predictions made by this model:

$$p = \frac{\#\ correct\ predictions}{\#\ predictions}. \tag{2}$$

To evolve subsequential generations, a tournament selection size of 3 was used. The remaining individuals from the previous generation are copied to the next with a crossover probability of 0.65. Then, the whole generation is mutated with an average of 2 mutations per chromosome. Experiments have been conducted using a target fitness of 0.75, 0.9 and 1.

Testing the opponent models

Using the backpropagation algorithm, opponent models were created for all of the bots listed in Section 4.2. To test the performance of these models, an opponent modeler has been added to the computer simulation. Every player has his own modeler, which in turn has an opponent model for each opponent. Whenever a player has to make a character choice, the opponent modeler asks its opponent models to predict the character choices of the other players. At the end of a game, the modeler calculates the models' performances using the fitness function from Equation 2.

For this experiment, all players will use the backpropagation neural-network opponent models, except for the last player, who will use random opponent models. To achieve reliable results, the game will be simulated 1000 times.

5 Results and Discussion

This section will list the results of the experiments described in Section 4, starting with the results obtained by using the random opponent models in Section 5.1. These results will form a base for comparison of the results from the neural network opponent models. The results from the experiment to determine the optimal number of hidden nodes are presented in Section 5.2. These results are then followed by the performances of the trained backpropagation in Section 5.3, and neuro-evolution neural networks in Section 5.4.

5.1 Random opponent model results

Table 2 shows the performances of the random opponent modeler after playing 1000 games of Machiavelli for each bot type. For every bot, 5 players of this bot type were made to play against each other for 1000 games. One of these bots was given the opponent model, to predict

Bot	Performance
ResourceBot	29.69%
Bot-KnowAll	29.14%
CitadelsBot	30.09%
SequenceBot	29.07%
RandomBot	29.44%

Table 2: In-game performances for the random opponent model, by type of bot.

the choices of each of his opponents at every turn. Performance is measured using Equation 2. Unsurprisingly, performances between bot types are similar, averaging about 29.5%. From this relatively simple experiment we can conclude that any opponent model incorperating some knowledge should at least be able to achieve a performance better than this.

5.2 Determining the number of hidden nodes

The results of this experiment can be found in Figures 2 and 3. Figure 2 shows the performances of neural networks trained on data sets of 50, 100 and 500 games of Machiavelli, by number of hidden nodes. Note that the performance measure in this case is the mean square error of the neural networks on the training set.

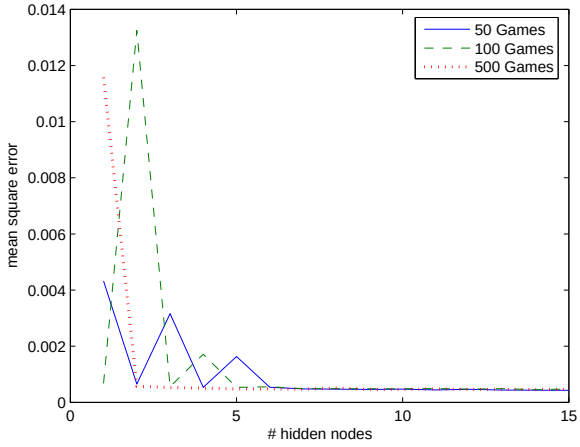

Figure 2: Performance of a neural network trained using the backpropagation algorithm, by the number of hidden nodes in the network.

Looking at these graphs, it appears that performances fluctuate for low numbers of hidden nodes, but stabilize once they reach a certain point. At 6 hidden nodes, the graphs displayed here all seem to have reached this point. Therefore, at least 6 hidden nodes should be used for the networks. When increasing the number of nodes beyond 6, performances keep improving, but only slightly. This would lead us to believe that larger numbers of hidden nodes would not have a noticable effect on game performance.

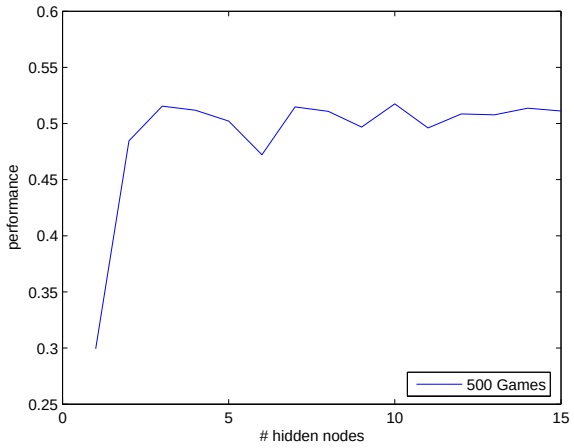

Figure 3: Game performance of a neural network trained using the backpropagation algorithm, by the number of hidden nodes in the network.

In Figure 3 the game performances of the same networks are displayed. Here, performance means the ratio of correct predictions as given in Equation 2. Unfortunately, the performances depicted here do not stabilize like the previous graphs, instead varying between a certain range. Apperently, small differences in mean square error do in fact notably impact game performance.

Based on this experiment, we have chosen to use 10 hidden nodes for the remaining experiments.

5.3 Backpropagation training

The performances of the neural network opponent models trained using backpropagation have been obtained by testing the models in the simulation. All experiments were conducted in the same way as for the random opponent model experiment. The performances listed in Table 3 are the ratios of correct predictions over all opponent character choices.

The results shown here differ considerably between bots. Performance against the RandomBot, which is impossible to model, is similar the performances achieved by the random opponent models. The best performance is that against the SequenceBot, which chooses its character based on a simple rule. Second best is against the ResourceBot. Using only inputs that are used as network inputs for character choices, this bot can be correctly predicted about 70% of the time.

Then there are the CitadelsBot and Bot-KnowAll. While the CitadelsBot uses a character-choosing algo-

Bot	5 Games	10 Games	25 Games
ResourceBot	53.26%	65.15%	70.33%
Bot-KnowAll	30.02%	43.97%	48.90%
CitadelsBot	45.49%	55.79%	58.11%
SequenceBot	79.26%	75.42%	75.48%
RandomBot	29.43%	29.13%	29.35%

Table 3: Performance of opponent models trained using the backpropagation algorithm against the various bots. The first column shows the bot's names, and the subsequent columns show the performances after training the models using different numbers of games of training data. All models use a neural network with 10 hidden nodes and 10 initializations.

rithm that is much more complicated than the ones discussed before, and uses some inputs not used for the opponent models, it still can be predicted with about 50-60% accuracy. When modeling Bot-KnowAll, another obstacle is encountered. This bot often gives its highest preference to multiple characters, and chooses from these characters at random. This makes it more difficult to predict the bot's choice, even when its preferences can be closely approximated. Still, performance against the bot approaches 50%.

An observation that can be made from these results is that, using backpropagation, the neural-network opponent models are relatively fast, that is, using only a small number of games as training data. The speed of convergence for the ResourceBot is displayed graphically in Figure 4. These results where obtained by creating a data set for every specified number of games, and training a neural network seperately on each of these data sets.

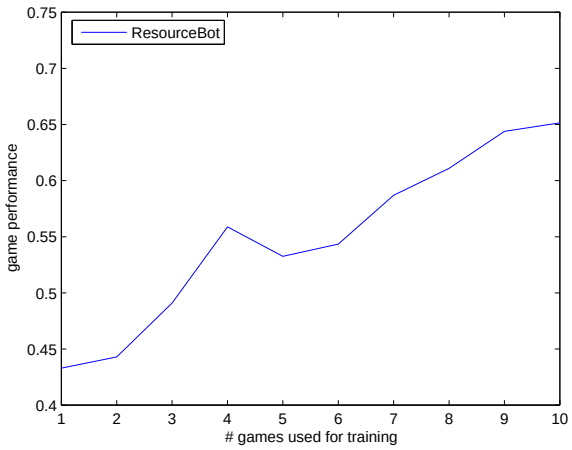

Figure 4: Game performance of a neural network trained using the specified number of games as training data.

5.4 Neuro-evolution training

Neuro-evolution has been used on all bot types, but unfortunately, without any success. All results have been comparable to those of the random opponent modeler shown in Table 2. The neuro-evolution algorithm was unable to improve on these performances in the slightest. The reason for this lack of performance has not been identified due to time constraints. Possibly, a single fitness for an entire game of Machiavelli is not representative for the quality of a neural network.

6 Conclusions and Future Research

The results that have been obtained using backpropagation are promising, though far from perfect. For a bot using no hidden information when choosing its characters, opponent models can reach a prediction performance of up to 70%. This is without having any knowledge about the face-down character card, and while dealing with variable, partially unknown outputs. Another bot, using a more complex algorithm for predicting characters, and making use of hidden information, can still be predicted with 50% accuracy. Unfortunately, the meaning of these percentages is still unclear. To find the true usefulness of the opponent models, they should be incorporated into Machiavelli turn strategies. In this way, the effect of using the models on game scores can be measured.

A major drawback of using backpropagation neural networks to create the opponent models, is that it requires precise actual outputs for training. This makes using backpropagation in real applications unpractical, as a player's detailed preference for each character at every choice is unavailable in competitive environments, and usually impossible for human players to express.

The solution to this problem might have been the use of neuro-evolution, which trains using a fitness function, that can easily be obtained from publicly available data. Neuro-evolution as currently used in this research was unable to model an opponent, but hopefully the cause of this can be found and corrected in the future.

The application of opponent modeling in Machiavelli remains a work in progress. Some promising results have been achieved using backpropagation neural networks, but this method is currently unsuited for use in practical applications. Still, it has been shown that trained neural networks, using only public knowledge about the state of the game, and having no knowledge about the face-down character, can correctly predict over half of his opponents' character choices for complex bots. For simpler bots, the percentage of correct predictions is up to 70-80%.

Future research

Further work could be done to increase the accuracy of the neural-network opponent models. For instance, using opponent´s character preference to make a calculated guess about the face-down character or adding knowledge about hidden information to the network inputs may be worthwile. The district cards in an opponents hand may be unknown, but some inferences could be drawn from, e.g., the size of the hand, combined with the amount of gold the opponent has.

Unfortunately, the neural networks trained using neuro-evolution have not produced any useful results. The ability of neuro-evolution to train using simple, publicly available performance measures is attractive however, so more research should be done to determine whether the method can be made to work for this application.

The real test of opponent modeling in Machiavelli is to measure its usefulness for enhancing players' strategies. How can knowledge about the opponents's likely choices benefit your own, and how does it ultimately affect your score? The logical step would be to combine the research of Groeneweg [9] with the research described in this paper.

References

[1] Billings, D., Papp, D., Schaeffer, J., and Szafron, D. (1998). Opponent modeling in Poker. *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pp. 493–498, AAAI Press, Madison, WI.

[2] Carmel, D. and Markovitch, S. (1993). Learning models of opponent's strategy in game playing. *Proceedings of The AAAI Fall Symposium on Games: Planing and Learning*, pp. 140–147, Raleigh, NC.

[3] Davidson, A. (1999). Using Artificial Neural Networks to Model Opponents in Texas Hold'em. http://spaz.ca/aaron/poker/poker.html.

[4] Davidson, A., Billings, D., Schaeffer, J., and Szafron, D. (2000). Improved opponent modeling in poker. *Proceedings of The 2000 International Conference on Artificial Intelligence (ICAI'2000)*, pp. 1467–1473.

[5] Davidson, A. (2002). *Opponent Modeling in Poker: Learning and Acting in a Hostile and Uncertain Environment.* M.Sc. thesis, University of Alberta.

[6] Jong, S. de (2005). Steven's personal page. `http://www.cs.unimaas.nl/steven.dejong/`.

[7] Donkers, H.H.L.M (2003). *Nosce Hostem - Searching with Opponent Models.* Ph.D. thesis, Universiteit Maastricht.

[8] Faidutti, B. (2005). Citadels. `http://www.faidutti.com/`.

[9] Groeneweg, G. (2005). Monte Carlo Machiavelli. *Submitted to BA-KECS 2005.*

[10] Hagan, M. T., Demuth, H. B., and Beale, M. H. (1995). *Neural Network Design.* PWS Publishing Company.

[11] Krose, B. and Smagt, P. Van der (1996). *An Introduction to Neural Networks.* University of Amsterdam.

[12] Urlings, M.C.M. (2005). Comparing Reinforcement Learning with Genetic Algorithms in RoboCode. *Submitted to BA-KECS 2005.*

[13] Weisshaar, C. (2005). Citadels. `http://wsims.com/cweissha/citadels/Citadels.html`.