

# Neural networks – simulation and application in strategy games

Tobias Schlatter      Daniel Meister

May 2007

## Contents

<b>0</b>	<b>Introduction</b>	<b>2</b>
0.1	Background . . . . .	2
0.2	Project aim . . . . .	2
<b>1</b>	<b>Theory</b>	<b>2</b>
1.1	Biological background . . . . .	2
1.2	Basic entities . . . . .	2
1.3	The network . . . . .	2
1.4	Mathematical description . . . . .	3
1.5	Learning algorithms . . . . .	3
<b>2</b>	<b>Simulation</b>	<b>4</b>
<b>3</b>	<b>Application</b>	<b>4</b>
3.1	Tic-tac-toe . . . . .	4
3.2	The Settlers of Catan . . . . .	5
<b>4</b>	<b>Outlook</b>	<b>8</b>
4.1	Multiple neural networks . . . . .	8
4.2	Expert systems . . . . .	8
4.3	Network architecture . . . . .	9
4.4	Parameters . . . . .	9

## 0 Introduction

### 0.1 Background

Since the construction of the first computers, it has been questioned whether machines can solve complex non-analytical problems. For a long time computers were unable to solve problems for which they had no specific algorithm. Therefore it was difficult to program a good game engine because programmers had to consider every possible situation and tell the program how to react. Since evaluation functions were difficult to write for certain complex games, the idea emerged of evaluating situations in the game instead of searching for the best move, which led to the development of artificial neural networks.

### 0.2 Project aim

The aim of our project was to understand more about artificial neural networks. We wanted to investigate the limits of these neural networks and various learning algorithms. We decided to do this by implementing a neural network for a complex strategy game.

## 1 Theory

### 1.1 Biological background

Artificial neural networks are modeled on the nervous systems of living organisms. These nervous systems are built of special cells so-called neurons. The nervous system of a human being consists of several million connected neurons. These neurons are capable of transmitting electrical pulses. Whenever a neuron receives a strong enough pulse, it transmits a pulse to all neurons it is connected to; usually this requires several incoming pulses. The connections between neurons can become weaker or stronger and can be created or disappear. Living organisms learn by altering their neural connections. Artificial neural networks also consist of connected nodes that can send pulses to each other, but an artificial neural network is substantially smaller and less complex than a real one. The other big difference is the time aspect: artificial neural networks work synchronously. A real neural network works asynchronously, i.e. it constantly processes incoming data.

### 1.2 Basic entities

Every neural network consists of individual data processing units or neurons. Each neuron also has incoming and outgoing connections (see Figure 1.a).

### 1.3 The network

The complete neural network is built of interconnected neurons. Each neuron receives signals through its incoming connections (*input* signals) and sends signals through its outgoing connections (*output* signals). To further specify the type of neural network, the following parameters have to be considered:

#### 1.3.1 Number of neurons / connection pattern

A very important property of a neural network is the number of neurons and the pattern in which they are connected. For this project we only studied so-called *multilayer perceptrons* (MLP), a network type in which neurons are arranged in multiple layers with each neuron of a layer connected with every neuron in the adjacent layers.

#### 1.3.2 Propagation rule

Secondly, a rule is needed to determine when a given neuron processes input signals and sends its output signal.

#### 1.3.3 Combination rule

The *combination rule* describes how all of the input signals of a neuron are combined. In most cases this rule is given by

$$net_j = \sum_i x_i w_{ij}$$

#### 1.3.4 Transfer function

After the input signals are combined, the transfer function calculates the output signal of the neuron. For MLPs the transfer function is usually (see Figure 1.b):

$$f(x) = \frac{1}{1 + e^{-x}}$$

a simple non-linear function which has an easily calculated derivative

$$f'(x) = f(x) (1 - f(x))$$

### 1.3.5 Learning rule

The *learning rule* specifies how the weights are updated after the neural network produces an output and the error of the output is calculated. This rule is very different for different types of neural networks. Sometimes it is just a simple function, but often it is an algorithm for calculating the weight updates.

### 1.4 Mathematical description

- $o_j$  output signal of neuron  $j$
- $t_j$  expected output signal of neuron  $j$
- $w_{ij}$  weight of the connection from neuron  $i$  to neuron  $j$
- $net_j$  combined input signals of neuron  $j$ , which is usually

$$net_j = \sum_i o_i w_{ij}$$

- $f_j$  transfer function of neuron  $j$ , if  $f_k = f_i \forall k, j$  then the index is omitted and  $f$  is defined as  $f := f_j$  and  $o_j = f(net_j)$
- $E_k$  error of the network for a given pattern  $k$
- $E$  total error of the network

$$E := \sum_k E_k$$

### 1.5 Learning algorithms

#### 1.5.1 Backpropagation

The error  $E_k$  for a pattern  $k$  is given by

$$E_k := \frac{1}{2} \sum_j (o_j - t_j)^2 \quad (1)$$

The backpropagation algorithm is a gradient algorithm, i.e. it searches the minimum of the error surface without knowing the whole surface (see Figure 2). The gradient of error is given by

$$\nabla E = \left( \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_n} \right)^T$$

or by

$$\nabla E = \frac{\partial E}{\partial w_{ij}} \quad (2)$$

Because of the chain rule

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} \quad (3)$$

Now  $\delta_j$  is defined as

$$\delta_j := -\frac{\partial E}{\partial net_j}$$

and therefore

$$\frac{\partial E}{\partial w_{ij}} = -\delta_j \frac{\partial net_j}{\partial w_{ij}}$$

From the combination rule of a MLP

$$net_j = \sum_i x_i w_{ij}$$

it is obvious that

$$\frac{\partial net_j}{\partial w_{ij}} = x_i$$

For the value of  $\delta_j$ : by definition

$$\delta_j = -\frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

from equation (1) it is clear that

$$\frac{\partial E}{\partial o_j} = -(t_j - o_j)$$

and because the output signal  $o_j$  is calculated by

$$o_j = f(net_j)$$

it is apparent that

$$\frac{\partial o_j}{\partial net_j} = f'(net_j)$$

and hence

$$\delta_j = (t_j - o_j) f'(net_j)$$

Equation (3) now implies that

$$\frac{\partial E}{\partial w_{ij}} = -(t_j - o_j) f'(net_j) x_i$$

The weight updates are therefore calculated by

$$\Delta w_{ij} = \eta \delta_j x_i$$

where  $\eta$  is the *learning rate*, a parameter to adapt the algorithm for different problems.

Unfortunately this type of adjustment is only possible for connections that are attached to neurons in the output layer, because  $\delta_j$  is only defined for these neurons. [1] proposes the *extended delta rule*: the error of each neuron in the other layers is calculated by

$$\delta_j = f'(net_j) \sum_k \delta_k w_{jk}$$

### 1.5.2 TD( $\lambda$ )

The application of the backpropagation algorithm is difficult with strategy games, because the  $t_j$  of the output neurons cannot be defined exactly. There are two different approaches to solving this problem. One possibility is to use a database of previously played games to train the network. This method has one big disadvantage: the potential performance of the neural network is limited by the quality of the sample cases. Otherwise it is possible to train the neural network *online*, i.e. it learns while playing. During the game there are different states  $(S_1, S_2, \dots, S_f)$ . One possibility would be to save the different states and then train the neural network at the end of the game with the training pairs  $(S_1, T_f), (S_2, T_f), \dots, (S_f, T_f)$ , where  $T_f$  is the result of the game. There are two problems with this option: it is very slow, and it needs large amounts of memory to save all the states of a game. The method of *temporal difference learning* (*TD-learning*) provides the neural network with the training pairs  $(S_1, P_2), (S_2, P_3), \dots, (S_f, P_{f+1})$  where  $P_t$  is the prediction of the result of the game at time  $t$ , and it is obvious that  $P_{f+1} = T_f$ . In order to train neural networks efficiently with this method, [2] proposed an incremental algorithm to update the weights:

$$\Delta w_t = \eta (P_{t+1} - P_t) \sum_{k=1}^t \lambda^{t-k} \nabla_w P_k \quad (4)$$

where  $\eta$  is the *learning rate* and  $\lambda$  is a second parameter which grades the importance of past moves. The weight updates can be calculated incrementally by defining

$$a_t := \sum_{k=1}^t \lambda^{t-k} \nabla_w P_k$$

$a_t$  can be calculated incrementally:

$$a_0 = 0$$

$$a_t = \lambda a_{t-1} + \nabla_w P_t \quad \forall t \geq 1$$

The weight updates are then given by

$$\Delta w_t = \eta (P_{t+1} - P_t) a_t$$

In order to avoid volatile updates, the weight updates are not applied until the end of the game.

## 2 Simulation

We chose the programming language *Java* for our simulations and implemented an object-oriented framework which can simulate different types of neural networks and different learning algorithms. We tested our simulations with a few simple applications. First of all, we implemented a neural network that was able to learn how to calculate the arithmetic average of two input numbers. Then we tested many other simulations including one for very simple handwriting recognition.

## 3 Application

### 3.1 Tic-tac-toe

#### 3.1.1 Motivation

As a first step, we decided to implement a small game with simple rules for which we could easily write a classic AI-player to train the neural network.

#### 3.1.2 The game

The Tic-tac-toe field consists of 9 small squares, which are ordered in a 3-by-3 arrangement (see Figure 3). The two players alternate in putting their marks (usually either a cross ( $\times$ ) or a circle ( $\circ$ )) in a free square. The player who succeeds in placing three marks in a vertical, horizontal, or diagonal row wins the game.

#### 3.1.3 Implementation

We implemented this game with different classes based on our framework (see Figure 4):

**Engine** This class regulates the flow of the game and is responsible for saving the field and the current status and for observing the rules. **Engine** has a constructor that has two players as pass parameters and a function `play()` that starts a new game and returns the winner of it.

**Player** This abstract class represents one player. It has two functions: `getNextTurn()` which asks the player for his next turn and `win()` which tells the player whether he has won the game or not.

We implemented several subclasses of the player class to represent the different player types.

**HumanPlayer** The function `getNextTurn()` passes the question to the human player actually playing the game.

**RandomPlayer** This is a player that chooses possible moves at random.

**TDPlayer** The most important player `TDPlayer` uses a TD-learning class from our framework to calculate its next move. The function `win()` is used to train the neural network.

**TrainerPlayer** This player is very similar to the `TDPlayer` but also has the possibility of performing random moves.

**IntelligentPlayer** There is only one strategy to win against this player; every other strategy leads to a tie or a loss. We used this player to determine whether a neural network is able to find this strategy.

### 3.1.4 Network architecture

The neural network for the `TDPlayer` consists of one input layer with 9 neurons, one hidden layer with 20 neurons, and one output layer with 1 neuron as well as one bias neuron (see Figure 5). Each neuron in the input layer is attached to one square on the field. It receives the value 1 if a mark from the `TDPlayer` is on this square,  $-1$  if the opponents mark is there and 0 if the square is empty. The output neuron produces the probability of the `TDPlayer` winning. The weights are updated according to formula (4), where

$$\eta = 0.0316$$

$$\lambda = 0.0837$$

### 3.1.5 Achievements

We managed to train the neural network for Tic-tac-toe so that it was able to play at almost a human level for the first few turns. However, its performance was not stable. Sometimes the neural network played very well for one or two turns but then lost the game because of an illogical move.

The neural network did learn how to beat the `IntelligentPlayer` very quickly. After just 100 learning cycles it was winning 5 games out of 10. We could also partially solve the problem of Tic-tac-toe having no random elements by training the `TDPlayer` against a `TrainerPlayer`, which introduces a random element.

### 3.1.6 Conclusion

We discovered that it is very difficult to train neural networks to play non-random games. Therefore, we decided to move on to *The Settlers of Catan*, which includes a lot of randomness.

## 3.2 The Settlers of Catan

### 3.2.1 Motivation

With the implementation of the board game *The Settlers of Catan*, we wanted to explore the limits of artificial neural networks. Thus we chose a game involving sophisticated strategies and complex rules.

### 3.2.2 The game

The field consists of 19 hexagons that are divided into 5 different resource types (see Figure 6). Around the resource fields are water fields, which offer the possibility of trading resources at certain points. On each resource field there is a small chip with a number from 2 to 12. At the beginning of each turn, two dice are thrown. The sum of the two dice determines which resource field issues resources on that turn.

**Goal** The goal of the game is to acquire victory points by building streets, settlements and cities or by buying development cards. The first player who has 10 victory points wins the game.

**Building** With the resources, a player can build new streets or settlements or expand an existing settlement to a city. There are certain rules about where a player can build new streets or settlements.

**Resources** Whenever a field gives out resources, every player that has a settlement or a city adjacent to this field receives the resources corresponding to the resource type of the field. Each player

receives one resource for every settlement he has built next to this field and two for every city.

**Robber** There is also a robber involved in the game. The robber is placed on one resource field and can be moved by any player that rolls a 7 with the dice. The resource field the robber is placed on does not give out resources until the robber moves away.

The 7 has another important effect: each player with more than 7 resources has to discard half of them.

**Development cards** Players also have the opportunity to buy development cards. They get one card of 5 different types at random. Development cards give players certain advantages, such as the possibility of moving the robber or an additional victory point.

**Trading** Players are allowed to trade resources with each other. Player can also exchange 4 resources of one type for one resource of another type. Every player that has settlements or cities at predefined locations is allowed to trade certain resource types at better conditions.

**Special cards** There are also two different special bonus cards, each worth two victory points. One special bonus gives two victory points to the player with the longest continuous street; the other involves development cards.

### 3.2.3 Implementation

We decided to divide our application in two parts: the *front-end* and the *back-end*. The *back-end* is responsible for handling the user inputs, saving the status of the game, training the network, and integrating the rules of the game. The *front-end* handles the communication between the *back-end* and the user by providing a graphical user interface. The user is able to see the current state of the game and to give his input through the *front-end*.

**Back-End** (see Figures 9-10)

**Programming language** We decided to write the *back-end* of *The Settlers of Catan* not in Java but in C++ because the better performance of C++

was crucial for the time-consuming learning process.

**Field** The most important issue in the implementation of this game was the representation of the field. We numbered the possible locations for streets and buildings and the hexagons (see Figures 7-8). Then we created multiple mapping arrays, e.g. `streets2Points[][]`, which give the numbers of two points connected by a given street.

**Basics** For the programming of the *back-end* we concentrated on the performance of the program. We tried to design the whole program in such a way that the learning process can be implemented very efficiently (see Figure 11).

`main()` The main function is responsible for setting up the game and regulating the flow of the game. It first determines how many players participate in the game and what player type they are. Then it initializes the field, asks the `Player` classes for various decisions, distributes all the cards according to the rules and determines whether a player has won the game.

`Player` Each instance of this class represents one player at the table. The `Player` class also saves the most important aspects of the current game state for the player in order to improve the performance of the learning process, for example:

- `number` saves the number of the player represented by this instance
- `res[]` saves the resources of the player
- `event[]` saves the development cards of the player
- `settlements[]` saves all settlements the player has built
- `cities[]` saves all cities the player has built
- `mstreets[]` saves all streets the player has built
- `siegePoints` saves the victory points of the player

The `Player` class also includes several important functions. Among others, these include:

- `doTurn()` asks the player for his next move

- `win()` tells the player he has won the game
- `lose()` tells the player he has lost the game
- `trade()` asks the player if he wants to accept a given trade
- `discard()` tells the player he has to discard resources
- `moveBandit()` tells the player he has to move the robber to a new location
- `streetBuilt()` tells the player a street has been built
- `settlementBuilt()` tells the player a settlement has been built
- `cityBuilt()` tells the player a city has been built
- `addSettlement()` decides whether the player is allowed to build a new settlement and builds this settlement if allowed
- `addStreet()` does the same as the function `addSettlement()` but for streets
- `convertCity()` upgrades a given settlement to a city
- `checkStreet()` checks whether a street can be built at a given point
- `checkSettlement()` checks whether a settlement can be built at a given point
- `checkCity()` checks whether a given settlement can be upgraded to a city

`HumanPlayer` is a subclass of `Player` which is connected to the *front-end* and enables a human player to play the game. `TDPlayer` is also a subclass of `Player`, but it is connected to the `TDNet` class which implements a MLP with TD-learning.

**Street finder** The `StreetFinder` class searches for the longest continuous street and determines which player it belongs to. This is important for the bonus card that is given to the player with the longest continuous street.

**Compiler options** The *back-end* can be compiled with different compiler options for different needs. There is a compiler option that generates one version of the *back-end* just for training the neural network, one for playing via the *front-end*, and one for training the neural networks on several computers in parallel.

### Front-end

**Text interface** It is possible to play the game directly with the *back-end* via a simple text-based interface. The use of this interface requires an original board game and a map with all the numbers of the points and the streets in order to actually play the game.

**Graphical user interface** We also produced a graphical user interface (see Figure 12). With this interface the user is able to play directly on the computer.

### 3.2.4 Network architecture

The `TDNet` class implements a MLP with TD-learning. The neural network has an input layer with 435 neurons, one hidden layer with 200 neurons and an output layer with 1 neuron. The output neuron produces the win probability for the `TDPlayer`. The weights are updated according to formula (4), where

$$\eta = 0.2$$

$$\lambda = 0.8$$

and  $P_{f+1}$  is set to 1 if `TDPlayer` wins the game and to  $\frac{\text{victory points}}{20}$  in all other cases.

### 3.2.5 Parallel training

To improve the efficiency of the learning process, we wrote a second application which enabled us to train the neural network on several computers in parallel. One computer in the network acts as a server, generates a new neural network with randomly distributed weights, and sends this neural network to the clients. The clients perform a pre-defined number of learning cycles (we chose 10) and then send the results back to the server process. The server calculates a new network from all

the networks it receives from the clients by averaging the weights. The generated network is then sent to the clients. This process is repeated numerous times.

We had the opportunity to train the neural networks on 20 computers in parallel for one weekend. After this extended training session, the neural networks played better but not that much better. We assume the computers might train the neural networks in different directions, i.e. one computer increases the weight of a certain connection and another decreases the same weight, and when the server averages the connection weights, the modifications are cancelled out. Therefore, a better combination rule for the different networks is needed, for example a (genetic) algorithm to decide which of the neural networks is the most intelligent.

### 3.2.6 Findings

When we saw the neural networks play against each other for the first time, we were quite surprised how many trades they made per turn. At times, these trades were very good, but there were also very stupid ones. The neural networks learned to keep the number of their resources low in order not to have to discard them, but they did so the wrong way. Therefore we decided to deny the neural networks the possibility of trading for the next few hundred learning cycles. After these cycles the “neuronal” players no longer traded among themselves, but they built settlements at special positions which allowed them to independently trade resources.

### 3.2.7 Achievements

We managed to implement the *back-end* as well as the graphical user interface and to train multiple networks in such a way that they are able to play against each other. Playing against human opponents turned out to be very difficult because the behavior of the “neuronal” player is incomprehensible to humans.

## 4 Outlook

(Many ideas described in this section are based directly on the board game *The Settlers of Catan*. Therefore basic knowledge of the rules of the game (see section 3.2.2 on page 5) is crucial.)

## 4.1 Multiple neural networks

One method that could improve the playing ability of the computer player is to use multiple neural networks at the same time. There are two different approaches for the distribution of the tasks.

### 4.1.1 Different networks for important situations

The first two moves are crucial to the whole game in *The Settlers of Catan*. However, these two steps are the most difficult to train for the TD( $\lambda$ )-algorithm because there are many moves between this first decision and the end of the game. If a player makes a bad choice in his first turn he has very little chance of winning the game, because he never gets the resources he needs to build new streets or new settlements. And because he cannot build, he has no possibility of reaching new resource fields. Even the relative positions of the first two settlements can be important because of the special bonus card which awards two victory points to the player with the longest continuous street. One potential solution is to distribute the decisions over two different neural networks. One network learns only the first two moves of the game, but gets the full reinforcement signal and is therefore able to learn much more efficiently. The other neural network is responsible for the remaining moves.

### 4.1.2 Different networks for different strategies

Another idea is to train different neural networks for different strategies. A central neural network decides which strategy it wants to follow and then requests the concrete moves from the corresponding neural network.

## 4.2 Expert systems

As previously mentioned, the first turns in *The Settlers of Catan* are extremely important for the result of the game. Therefore, one option is to write a decision function for an expert system to determine what the best locations for the first two settlements are. The probability of a resource field giving out resources can be calculated using simple stochastics. Based on this probability, the distribution of the resource types, and the relative positions, an expert system could work out the two

best locations. This method would allow the neural network to train more efficiently, because the network would no longer need to learn the initial moves.

[2] Sutton, R.S. (1988). *Learning to predict by the methods of temporal differences*. Machine Learning 3, 9-44.

### 4.3 Network architecture

A big disadvantage of *The Settlers of Catan* is its very complex field, which cannot be transferred to the input layer of a small or medium-sized neural network. The final neural network in the present project had 435 input neurons, but did not include a distinction between the different opponents.

An “intelligent” mapping function would be a very useful improvement because when the network gets smaller the learning process gets faster. A perfect mapping function would be able to map the whole field to a minimal number of inputs without losing any essential information.

### 4.4 Parameters

The choice of the two parameters of TD-learning,  $\eta$  and  $\lambda$ , is also very important but there is no theory to guide us in calculating these variables. Therefore further research is required to determine the optimal values of these parameters.

## Acknowledgments

We want to thank Dr. Nadine Tschichold for recommending our paper to *Schweizer Jugend forscht* and for very valuable suggestions to improve its quality. We also want to express our gratitude to Clemens Holenstein, who supervised our initial project at the *Kantonsschule Oerlikon*.

Special thanks go to Maureen Ehrensberger for helping us with the English version of our paper.

## References

[1] Rumelhart, D.E., Hinton, G.E. & Williams, R.J. (1986). “Learning Internal Representations by Error Propagation”, in Rumelhart, D.E. & McClelland, J.L. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Vol. 1. Cambridge, MA: MIT Press.