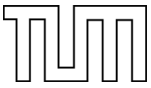


Technische Universität München

Department of Mathematics

Master's Thesis

Dynamical Learning: Case Study on Tic-Tac-Toe

Michael Heining

Supervisor: Prof. Dr. Massimo Fornasier

Advisor: Prof. Dr. Massimo Fornasier

Submission Date: 30.04.2017

I assure the single handed composition of this master's thesis only supported by declared resources.

Garching,

Zusammenfassung

Diese Arbeit behandelt am Beispiel Tic-Tac-Toe den Einsatz Neuronaler Netze zum Erlernen gewinnmaximierender Strategien gegen Minimax-Agenten von verschiedener Suchtiefe. Kapitel 1 beinhaltet eine Einführung ins Themengebiet Neuronale Netze. Im zweiten Kapitel wird mit dem Backpropagation-Algorithmus ein Verfahren zum Training Neuronaler Netze besprochen, während in Kapitel 3 der Minimax-Algorithmus dargestellt wird. Im zweiten, experimentellen Teil der Arbeit werden zwei verschiedene Ansätze untersucht, um die Spielstrategie eines unerfahrenen Agenten durch dynamisches Lernen zu optimieren. Der erste Ansatz vefolgt das Ziel, statistische State Value Functions durch Neuronale Netze zu approximieren. Dabei zeigt sich, dass Agenten von intelligenten Gegnern schneller lernen als von Gegnern mit geringer Suchtiefe. Im zweiten Ansatz wird versucht, die Lösung einer bellman-ähnlichen Funktionalgleichung durch ein Neuronales Netz anzunähern. In beiden Fällen wird anhand der Funktionswerte der Neuronalen Netze eine Spielstrategie formuliert.

Contents

1 Neural Networks

In this introductory chapter we will explain the fundamentals of neural networks. In chapter 1.3, 1.4 different network topologies will be introduced, whereas in chapter 2 we will learn about training algorithms. Some nice introductions can be found for example in books by Bishop [Bis95] or Nielsen [Nie15]. Chapter 1.1 to 1.3 in this thesis is based on Kröse's and Van Der Smagt's *An Introduction To Neural Networks, chapter 1* [KvdS96] with a slight difference in notation. Other than [Bis95, KvdS96] we will introduce and use compact matrix notations. After reading the first two chapters of this thesis, readers should be able to implement their own neural networks.

1.1 Introduction

The simplest form of a neural network is a single-layer network with one output unit. An output unit is defined by its weight vector $w = (w_1, \dots, w_n)$, offset value θ and activation function $F : \mathbb{R} \mapsto \mathbb{R}$. The output unit takes a n-dimensional input $x \in \mathbb{R}^n$ and produces some output $y \in \mathbb{R}$. First, the input is summed up with respect to the weight vector,

$$s = \sum_{i=0}^{n} x_i w_i + \theta.$$

This weighted sum is then presented to the activation function and for the output y we get

$$y = F(s). \tag{D1}$$

In a more compact way we can also write

$$y = F(\langle x, w \rangle + \theta), \tag{D2}$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product.

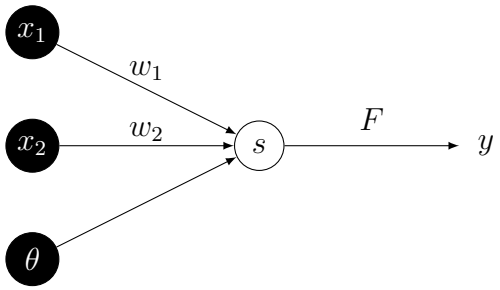


Figure 1: Output unit with two-dimensional input and one-dimensional output.

In most applications the activation function is a sigmoidal function like the logistic function or *tangens hyperbolicus*. For this introductory example we consider the *sign* function:

$$F(s) = \begin{cases} 1 & \text{if } s \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

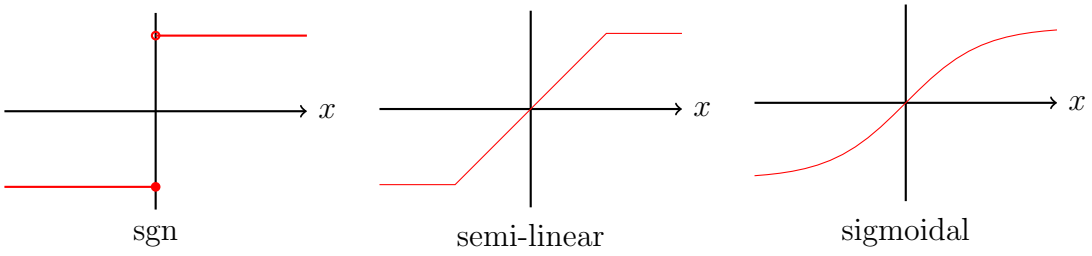


Figure 2: Examples of common activation functions. Figure from [KvdS96, p. 17].

The output y of this function is either +1 or -1, hence it can be used to classify data into two different classes C_+ and C_-. In case of input data $x \in \mathbb{R}^n$ and an output unit with weight vector w, the linear equation

$$\langle x, w \rangle + \theta = 0$$

represents a $(n-1)$-dimensional hyperplane which separates the two classes C_+ and C_-.

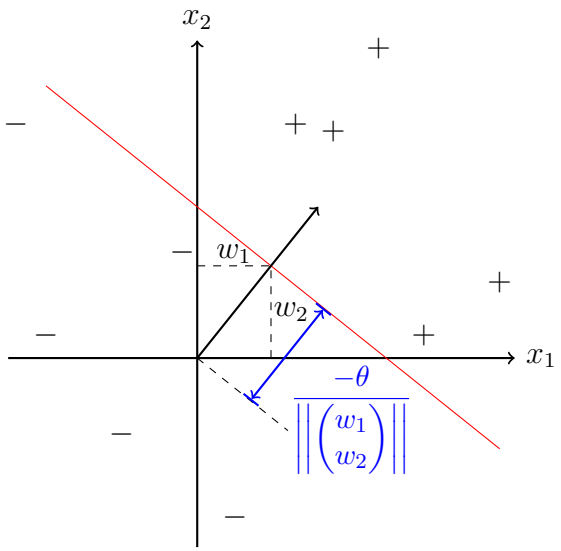


Figure 3: The weights of the output unit define a hyperplane that seperates data into two different classes. Figure from [KvdS96, p. 24].

Hence, given some linearly separable data set, there exists a set of network weights such that the network classifies the data correctly into two classes. If a set of data is not linearly separable, then there exist no such weights.

Example 1.

Consider the following: There are two classes C_{+1} and C_{-1}. Two input vectors $p_1 = (-1, -1)$, $p_2 = (1, 1)$ belong to class C_{-1} and $p_3 = (-1, 1)$, $p_4 = (1, -1)$ belong to C_{+1}.

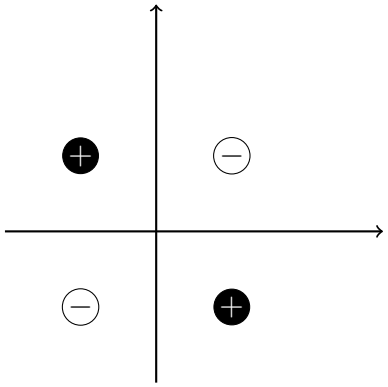


Figure 4: In case two groups of data are not linearly separable, a single layer neural network fails to classify the data correctly.

It is impossible to draw a line which separates these two classes. Therefore, single-layer neural networks with *sign* activation function are insufficient whenever we need to classify data that is not linearly separable. Nevertheless, by adding an additional "hidden" output unit to the neural network we are able to solve this classification problem.

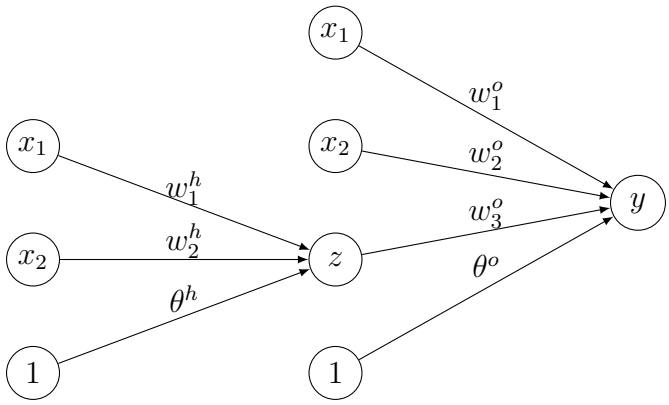


Figure 5: By adding an additional input element, the output unit is able to solve a bigger class of classification tasks.

Consider an additional input element z that is also the output element of another output unit. The output y is then defined by

$$y = sgn\left(\left\langle \begin{pmatrix} x_1 \\ x_2 \\ z \end{pmatrix}, w^o \right\rangle + \theta^o\right), \text{where } z = sgn\left(\left\langle \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, w^h \right\rangle + \theta^h\right). \tag{1.1}$$

By setting proper weights we can solve the classification problem we could not solve with a single layer network.
If

$$w^h = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \theta^h = -0.5 \qquad \text{and} \qquad w^o = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}, \theta^o = -0.5, \tag{1.2}$$

then the neural network will perform correctly on the classification task.

input	z	y
$(-1,-1)$	$(-1,-1,-1)$	(-1)
$(1,1)$	$(1,1,1)$	(-1)
$(-1,1)$	$(-1,1,-1)$	$(+1)$
$(1,-1)$	$(1,-1,-1)$	$(+1)$

Figure 6: By adding an additional hidden layer, the classification task in Example 1 can be solved. This table shows the hidden output z and network output y if parameters given in (1.2) are applied to (1.1).

1.2 Single-layer networks

Of course, neural networks are not restricted to one-dimensional outputs. We can design a neural network with several output units. But before we do that, let us introduce some notation.

Let $x \in \mathbb{R}^n$ be an input vector, define

$$\hat{x} := \begin{pmatrix} x \\ 1 \end{pmatrix} \in \mathbb{R}^{n+1}.$$

Instead of using the offset value θ, from now on we notate the offset of an output unit as an additional element to the weight vector. This means an output unit with n-dimensional input values has n+1-dimensional weight vector $w = (w_1, w_2, \ldots, w_n, w_{n+1})$, where $w_{n+1} := \theta$. With this new notation equation (D2) now reads

$$y = F(\langle \hat{x}, w \rangle).$$

In the first part of this chapter we used some neural networks with one-dimensional output. If we add several output units we are able to design a neural network which produces multi-dimensional output $y = (y_1, \ldots, y_m)$.

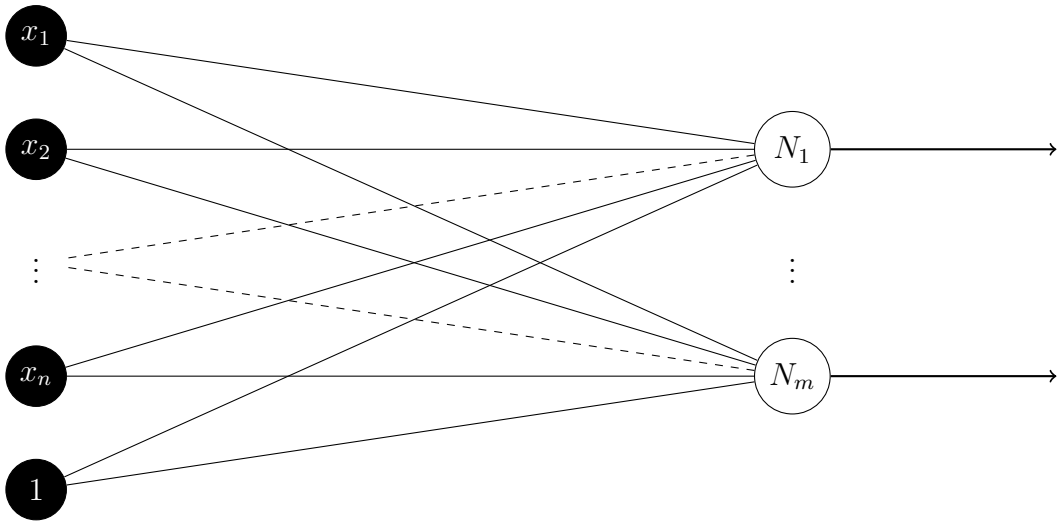


Figure 7: Network with m output units.

Given a network that takes n-dimensional input, every output unit N_i, $i = 1, \ldots, m$ has a weight vector $w_i \in \mathbb{R}^{n+1}$ and an output function F_i. Let

- $w_{i,j}$ be the j-th element of output unit i's weight vector,
- s_i be the dot product of the input and output unit i's weight vector, i.e.

$$s_i = \langle \hat{x}, w_i \rangle,$$

- $F_i : \mathbb{R}^m \to \mathbb{R}$ be output unit i's activation function and
- y_i output of output unit i.

For reasons of compact notation we summarize the weight vectors of all output units in a $m \times (n+1)$ matrix

$$W = \begin{pmatrix} w_{1,1} & \ldots & w_{1,n+1} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \ldots & w_{m,n+1} \end{pmatrix}.$$

The weighted sums are placed in a m-dimensional vector

$$s = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_m \end{pmatrix} = \begin{pmatrix} \langle \hat{x}, w_1 \rangle \\ \langle \hat{x}, w_2 \rangle \\ \vdots \\ \langle \hat{x}, w_m \rangle \end{pmatrix} = W\hat{x}.$$

The weighted sums are presented to the activation Functions $F_1, \ldots, F_m$ and we get the m-dimensional output y, i.e.

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} F_1(s) \\ F_2(s) \\ \vdots \\ F_m(s) \end{pmatrix}$$

or just

$$y = F(s) = F(W\hat{x}).$$

This architecture of a neural network is also called layer and leads to following definition.

Definition 1. *A Layer with weight matrix $W \in \mathbb{R}^{m \times n+1}$ and partial differentiable function $F : \mathbb{R}^m \to \mathbb{R}^m$ is defined as*

$$\begin{aligned} L : \mathbb{R}^{m \times (n+1)} \times \mathbb{R}^n &\to \mathbb{R}^m \\ (W, x) &\mapsto F(W\hat{x}) \end{aligned}$$

where $\hat{x} = \begin{pmatrix} x \\ 1 \end{pmatrix}$.
The output dimension m is also called size of the layer.

Note that we require the activation function to be partial differentiable with respect to s. The reason for this will be clear in later chapters when we use gradient descent for learning tasks. The choice of the activation function and the number of output units depends on the specific problem we want to solve. Remind example 1 on page 2 where we used the *sign* function to categorize data into two different groups. In the next example we introduce an activation function that is very useful when we need to categorize data into more than two different classes. For now, please do not think about how the network training is done. For the moment we want to introduce some possible choices for architectures of a neural network and its layers and activation functions.

1.2.1 Softmax-output

Consider a problem of classifying data into m different classes. For example, think of using a neural network for recognizing images of handwritten digits $0, \ldots, 9$. Seeing the output of the neural network we should be able to say what class the image is categorized to. For these problems we can use a layer of size m with softmax activation function, introduced in [Bis95, ch. 6.1] .
Given the weighted sum $s = W\hat{x}$ of an input to a neural net. The *softmax-output* of a layer is defined by

$$y = F(s) = \frac{1}{e^{s_1} + e^{s_2} + \cdots + e^{s_m}} \begin{pmatrix} e^{s_1} \\ e^{s_2} \\ \vdots \\ e^{s_m} \end{pmatrix}.$$

Note that the output values y_i are positive numbers between 0 and 1. Because of the normalization of the output vector we get

$$\sum_{i=1}^{m} F_i(s) = \sum_{i=1}^{m} \frac{e^{s_i}}{e^{s_1} + e^{s_2} + \cdots + e^{s_m}} = 1.$$

Therefore we can interpret the output y as a discrete probability distribution over m elements. The input is then classified to class C_{i^*} with highest probability, i.e.

$$i^* = \underset{i \in 1,\ldots,m}{\operatorname{argmax}} y_i.$$

1.2.2 Neural network derivatives

In the following we will make extensive use of partial derivatives and calculating these partial derivatives by applying chain rule. The notation with all its indices will get a little bit messy and demands the reader's concentration. Unfortunately, this cannot be avoided, especially when we are dealing with multi-layered neural networks. In reward we will get a compact and easy to implement learning algorithm in matrix notation. This will be a great benefit when implementing the so called back-propagation algorithm. So here we give some clarifications on the notation:

- If $y \in \mathbb{R}$ is a scalar and $x \in \mathbb{R}^m$ is a vector, then the partial derivative of y with respect to x is given by

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_m} \end{bmatrix}.$$

- If $y \in \mathbb{R}^n$ is a vector and $x \in \mathbb{R}$ is a scalar, then the partial derivative of y with respect to x is given by

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \frac{\partial y_2}{\partial x} \\ \vdots \\ \frac{\partial y_n}{\partial x} \end{bmatrix}.$$

- If $y \in \mathbb{R}^n$ is a vector and $x \in \mathbb{R}^m$ is a vector, then the partial derivative of y with respect to x is given by

$$\frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{pmatrix}.$$

- If $y \in \mathbb{R}$ is a scalar and $A \in \mathbb{R}^{n \times m}$ is a matrix, then the partial derivative of y with respect to A is given by

$$\frac{\partial y}{\partial A} = \begin{pmatrix} \frac{\partial y}{\partial a_{1,1}} & \cdots & \frac{\partial y}{\partial a_{1,n+1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial a_{1,n+1}} & \cdots & \frac{\partial y}{\partial a_{m,n+1}} \end{pmatrix}$$

Now we show how to calculate the derivative of the layer's output with respect to its input as well as the derivative with respect to its weight matrix.

Theorem 1 (Derivative of a Layer)**.** *Given a Layer $L : \mathbb{R}^{m \times (n+1)} \times \mathbb{R}^n \to \mathbb{R}^m$ with partial differentiable activation function $F : \mathbb{R}^m \to \mathbb{R}^m$ and weight matrix $W \in \mathbb{R}^m \times \mathbb{R}^{n+1}$. Then for $i = 1, \ldots, 9$ the output $y = L(W, x)$ satisfies*

$$\frac{\partial y_i}{\partial W} = \left[\frac{\partial F_i(s)}{\partial s}\right]^T x^T \qquad \text{and} \tag{T2}$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(s)}{\partial s} \tilde{W} \tag{T2}$$

where $\tilde{W} := \begin{pmatrix} w_{1,1} & \cdots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \cdots & w_{m,n} \end{pmatrix}$ *is equivalent to W after removing its last row.*

Proof. First, we write down the three equations that determine the output of the layer given some input x, namely

$$\hat{x} = \begin{pmatrix} x \\ 1 \end{pmatrix} \tag{1.3}$$

$$s = W\hat{x} \tag{1.4}$$

$$y = F(s). \tag{1.5}$$

By applying chain rule to equation (1.3) and (1.4) we get

$$\frac{\partial y}{\partial x} = \frac{\partial F(s)}{\partial s}\frac{\partial s}{\partial x} = \frac{\partial F(s)}{\partial s}\frac{\partial s}{\partial \hat{x}}\frac{\partial \hat{x}}{\partial x}. \tag{1.6}$$

Calculating the derivatives of $\frac{\partial s}{\partial \hat{x}}$ and $\frac{\partial \hat{x}}{\partial x}$ we obtain

$$\frac{\partial \hat{x}}{\partial x} = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ 0 & 0 & \ldots & 0 \end{pmatrix} \qquad \text{and} \tag{1.7}$$

$$\frac{\partial s}{\partial \hat{x}} = \frac{\partial W x}{\partial \hat{x}} = W. \tag{1.8}$$

By putting (1.7) and (1.8) back in (1.6) we proved (T2) since

$$\frac{\partial y}{\partial x} = \frac{\partial F(s)}{\partial s} W \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \\ 0 & 0 & \dots & 0 \end{pmatrix} = \frac{\partial F(s)}{\partial s} \tilde{W}.$$

In order to show (T2) we are using chain rule and obtain

$$\begin{aligned} \frac{y_i}{w_{k,l}} &= \frac{\partial y_i}{\partial s} \frac{\partial s}{\partial w_{k,l}} \\ &\overset{\text{(D1)}}{=} \begin{bmatrix} \frac{\partial F_i(s)}{\partial s_1} \cdots \frac{\partial F_i(s)}{\partial s_m} \end{bmatrix} \begin{bmatrix} \frac{\partial s_1}{\partial w_{k,l}} \\ \vdots \\ \frac{\partial s_m}{\partial w_{k,l}} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial F_i(s)}{\partial s_1} \cdots \frac{\partial F_i(s)}{\partial s_m} \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \hat{x}_l \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow k\text{-th element} \\ &= \frac{\partial F_i(s)}{\partial s_k} \hat{x}_l \end{aligned} \tag{1.9}$$

Writing (1.9) in matrix notation we get (T2). □

Example 2.

Consider a single-layer neural network with activation function $F_i(s) = tanh(s_i), i = 1, 2$ and weight matrix W. First, we calculate the derivative of the activation function

$$\frac{\partial F(s)}{\partial s} = \begin{pmatrix} \frac{\partial F_1(s)}{\partial s_1} & \frac{\partial F_1(s)}{\partial s_2} \\ \frac{\partial F_2(s)}{\partial s_1} & \frac{\partial F_2(s)}{\partial s_2} \end{pmatrix} = \begin{pmatrix} 1 - tanh^2(s_1) & 0 \\ 0 & 1 - tanh^2(s_2) \end{pmatrix}.$$

In order to save computational time when implementing neural networks, we can express these derivatives also in terms of the output y. Hence, we get

$$\frac{\partial F(s)}{\partial s} = \begin{pmatrix} 1 - y_1^2 & 0 \\ 0 & 1 - y_2^2 \end{pmatrix}.$$

Now we can apply Theorem 1 and get the derivatives of the network by

$$\frac{\partial y_1}{\partial W} = \begin{pmatrix} 1 - y_1^2 \\ 0 \end{pmatrix} x^T$$
$$\frac{\partial y_2}{\partial W} = \begin{pmatrix} 0 \\ 1 - y_1^2 \end{pmatrix} x^T$$
$$\frac{\partial y}{\partial x} = \begin{pmatrix} 1 - y_1^2 & 0 \\ 0 & 1 - y_2^2 \end{pmatrix} \tilde{W}.$$

1.3 Multi-layer feed-forward neural network

As we saw in example 1 on page 2, there are many classification tasks which cannot be done by single-layer neural networks. Adding one or more hidden layers to the network can overcome this problem. At the same time these multi-layer networks are more complex and more difficult to analyze.

Definition 2. *A Multi-layer Feed-forward Neural Network is a composition of layers.*

A multi-layer feed-forward neural network (MLFFNN) has a layered structure. It is a composition of $k \in \mathbb{N}$ hidden layers $L^{(k)}, L^{(k-1)}, \ldots, L^{(1)}$ and one output layer $L^{(0)}$. The input vector x is presented to the first hidden layer $L^{(k)}$ resulting in hidden output $y^{(k)}$. The hidden output $y^{(k)}$ is then presented to layer $L^{(k-1)}$ producing output $y^{(k-1)}$ and so forth. The input is forward-propagated through the network layer by layer.

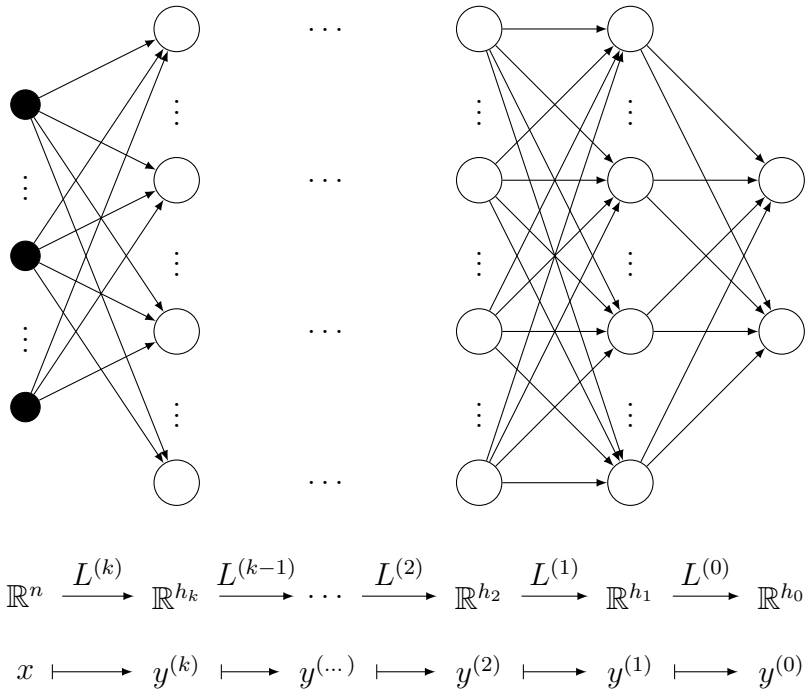


Figure 8: Network with k hidden layers

Notation. Let $\mathcal{N} = (L^{(k)} \circ L^{(k-1)} \circ \cdots \circ L^{(1)} \circ L^{(0)})$ be a MLFFNN. We define that

- $h^{(l)}$ denotes the size of layer $L^{(l)}$,
- x is the n-dimensional input vector to the network,
- $s^{(l)}$ is the $h^{(l)}$-dimensional weighted sum of layer $L^{(l)}$,
- $y^{(l)}$ is the $h^{(l)}$-dimensional output vector of layer $L^{(l)}$ and also the input to layer $L^{(l-1)}$,
- $F^{(l)} : \mathbb{R}^{h^{(l)}} \to \mathbb{R}^{h^{(l)}}$ is the activation function of layer $L^{(l)}$ and
- $W^{(l)}$ is the $h^{(l)} \times (h^{(l+1)} + 1)$-dimensional weight matrix of layer $L^{(l)}$. If $l = k$ then $W^{(k)} \in \mathbb{R}^{h^{(k)} \times (n+1)}$.

Example 3 (Forward-propagation through a MLFFNN)**.**

Consider a neural network processing two-dimensional input data and producing one-dimensional output. Consider furthermore that the network is a composition of two hidden layers with activation functions *tangens hyperbolicus* and an output layer with linear activation function. Both hidden layers are of size 5. Given this architecture the network has weight matrices

$$W^{(2)} \in \mathbb{R}^{5\times 3}, \quad W^{(1)} \in \mathbb{R}^{5\times 6}, \quad W^{(0)} \in \mathbb{R}^{1\times 6},$$

and activation functions

$$\begin{aligned} F_i^{(2)} &: \mathbb{R}^5 \to \mathbb{R}^5, s \mapsto tanh(s_i), i = 1, \ldots, 5 \\ F_i^{(1)} &: \mathbb{R}^3 \to \mathbb{R}^3, s \mapsto tanh(s_i), i = 1, \ldots, 5 \\ F^{(0)} &: \mathbb{R} \to \mathbb{R}, s \mapsto s. \end{aligned}$$

By forward-propagating the input x layer by layer we obtain both hidden outputs

$$\begin{aligned} y^{(2)} &= F^{(2)}(W^{(2)}\hat{x}), \\ y^{(1)} &= F^{(1)}(W^{(1)}\hat{y}^{(2)}) \end{aligned}$$

and the network output

$$y^{(0)} = F^{out}(W^{(0)}\hat{y}^{(1)}).$$

1.4 Back-propagation

When using multi-layer neural networks for approximating functions, we have to adapt the network weights in order to minimize some error (we will see this in the next chapter). Therefore we have to be able to determine the derivatives of the network outputs with respect to the network's weight parameters. Our goal now is to derive an algorithm for calculating these derivatives. This algorithm is known as *Back-propagation* and can be found in many teaching books about neural networks (e.g [KvdS96, Bis95]).

Similar to proof of Theorem 1 we first write down the equations that, given some input vector x, determine the output of the neural net:

$$\begin{aligned} s^{(k)} &= W^{(k)}\hat{x} \\ s^{(k-1)} &= W^{(k-1)}\hat{y}^{(k)} \\ &\vdots \\ s^{(l)} &= W^{(l)}\hat{y}^{(l+1)} \\ &\vdots \\ s^{(1)} &= W^{(1)}\hat{y}^{(2)} \\ s^{(0)} &= W^{(0)}\hat{y}^{(1)} \end{aligned} \tag{1.10}$$

$$\begin{aligned} y^{(k)} &= F^{(k)}(s^{(k)}) \\ y^{(k-1)} &= F^{(k-1)}(s^{(k-1)}) \\ &\vdots \\ y^{(l)} &= F^{(l)}(s^{(l)}) \\ &\vdots \\ y^{(1)} &= F^{(1)}(s^{(1)}) \\ y^{(0)} &= F^{(0)}(s^{(0)}) \end{aligned} \tag{1.11}$$

Before stating the next two lemmas, we introduce some useful notation and define the *error signal* of layer $L^{(l)}$ by

$$\delta_i^{(l)} := \frac{\partial y_i^{(0)}}{\partial s^{(l)}}. \tag{D3}$$

Lemma 1. *Given a multi-layered neural network with k hidden layers, one output layer and weight matrices $W^{(0)}, W^{(1)}, \ldots, W^{(k)}$. The derivative of the output with respect to the weight matrix $W^{(i)}, i = 0, \ldots, k$ is given by*

$$\frac{\partial y_i^{(0)}}{\partial W^{(l)}} = \delta_i^{(l)^T} \hat{y}^{(l+1)^T}. \tag{L1}$$

Proof. By applying chain rule to $\frac{\partial y_i^{(0)}}{\partial w_{i,j}^{(l)}}$, we get

$$\frac{\partial y_i^{(0)}}{\partial w_{p,q}^{(l)}} = \sum_{j=1}^{h^{(l)}} \frac{\partial y^{(0)}}{\partial s_j^{(l)}} \frac{\partial s_j^{(l)}}{\partial w_{p,q}^{(l)}}. \tag{1.12}$$

$$\tag{1.13}$$

In the sum on the right hand side, all but one summands can be omitted since

$$\frac{\partial s_j^{(l)}}{\partial w_{p,q}^{(l)}} \overset{(1.10)}{=} \frac{\partial \langle w_j^{(l)}, \hat{y}^{(l+1)} \rangle}{\partial w_{p,q}^{(l)}} = \begin{cases} \hat{y}_q^{(l+1)} & \text{if } j = p \\ 0 & \text{else.} \end{cases}$$

Therefore equation (1.12) simplifies to

$$\frac{\partial y_i^{(0)}}{\partial w_{p,q}^{(l)}} = \frac{\partial y_i^{(0)}}{\partial s_p^{(l)}} \hat{y}_q^{(l+1)}. \tag{1.14}$$

By writing this equation in matrix notation we proved (L1), because

$$\frac{\partial y_i^{(0)}}{\partial W^{(l)}} = \begin{pmatrix} \frac{\partial y_i^{(0)}}{\partial w_{1,1}^{(l)}} & \cdots & \frac{\partial y_i^{(0)}}{\partial w_{1,h(l+1)}^{(l)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_i^{(0)}}{\partial w_{1,h(l)}^{(l)}} & \cdots & \frac{\partial y_i^{(0)}}{\partial w_{h(l),h(l+1)}^{(l)}} \end{pmatrix}$$

$$\overset{(1.14)}{=} \begin{bmatrix} \frac{\partial y_i^{(0)}}{\partial s_1^{(l)}} \\ \frac{\partial y_i^{(0)}}{\partial s_2^{(l)}} \\ \vdots \\ \frac{\partial y_i^{(0)}}{\partial s_{h(l)}^{(l)}} \end{bmatrix} \begin{bmatrix} \hat{y}_1^{(l+1)} & \hat{y}_2^{(l+1)} & \cdots & \hat{y}_{h(l)+1}^{(l+1)} \end{bmatrix}$$

$$= \left(\frac{\partial y_i^{(l)}}{\partial s^{(l)}} \right)^T \hat{y}^{(l+1)^T}$$

$$\overset{(D3)}{=} \delta_i^{(l)^T} \hat{y}^{(l+1)^T}. \quad \square$$

Lemma 2. *Given a multi-layered neural network with k hidden layers, one output layer and weight matrices $W^{(0)}, W^{(1)}, \ldots, W^{(k)}$. The error signal $\delta_i^{(l)}$ fulfills the recursive equations*

$$\delta_i^{(l)} = \delta_i^{(l-1)} \tilde{W}^{(l-1)} \frac{\partial F^{(l)}(s^{(l)})}{\partial s^{(l)}} \quad \textit{for } i = 1, \ldots, k \qquad \textit{and} \tag{L2}$$

$$\delta_i^{(0)} = \frac{\partial F^{(0)}(s^{(0)})}{\partial s^{(0)}}. \tag{L3}$$

Proof. Equation (L3) is just the definition of $\delta_i^{(l)}$. For $l = 1, \ldots, k$, by applying chain rule twice we get

$$\begin{aligned} \delta_i^{(l)} &\overset{(D3)}{=} \frac{\partial y_i^{(0)}}{\partial s^{(l)}} \\ &= \frac{\partial y_i^{(0)}}{\partial s^{(l-1)}} \frac{\partial s^{(l-1)}}{\partial y^{(l)}} \frac{\partial y^{(l)}}{\partial s^{(l)}} \\ &\overset{(1.8)}{=} \frac{\partial y_i^{(0)}}{\partial s^{(l-1)}} \tilde{W}^{(l-1)} \frac{\partial y^{(l)}}{\partial s^{(l)}}. \\ &\overset{(D3),(1.11)}{=} \delta_i^{(l-1)} \tilde{W}^{(l-1)} \frac{\partial F^{(l)}(s^{(l)})}{\partial s^{(l)}}. \quad \square \end{aligned}$$

With help of Lemma 1 and Lemma 2 we can state the so called back-propagation algorithm.

Back-propagation algorithm. *Given a multi-layered neural network with k hidden layers, one output layer of size $h^{(0)}$ and weight matrices $W^{(0)}, W^{(1)}, \dots, W^{(k)}$. In order to calculate $\frac{\partial y_i^{(0)}}{\partial W^{(l)}}$, for $l = 0, 1, \dots, k$ do the following steps.*

1. *Forward-propagate x through the neural network and obtain the weighted sums $s^{(k)}, s^{(k-1)}, \dots, s^{(0)}$ and outputs $y^{(k)}, y^{(k-1)}, \dots, y^{(0)}$.*
2. *Calculate the derivatives $\frac{\partial F^{(0)}(s^{(0)})}{\partial s^{(0)}}, \frac{\partial F^{(1)}(s^{(1)})}{\partial s^{(0)}}, \dots, \frac{\partial F^{(k)}(s^{(k)})}{\partial s^{(k)}}$.*
3. *For $i = 1, \dots, h^{(0)}$,*
   (a) *use* (L2) *and* (L3) *to evaluate $\delta_i^{(0)}, \delta_i^{(1)}, \dots, \delta_i^{(k)}$ and*
   (b) *use* (L1) *to obtain the sought derivatives $\frac{\partial y_i^{(0)}}{\partial W^{(l)}}, \dots, \frac{\partial y_i^{(0)}}{\partial W^{(k)}}$.*

Example 4.

We now show in an example how to use the back-propagation algorithm. Given a neural network with two hidden layers, one output layer of size 1 and weight matrices $W^{(2)}, W^{(1)}, W^{(0)}$. Their activation functions are given by

$$F_i^{(2)}(s^{(2)}) = tanh(s^{(2)}_{h^{(1)}}) \; i = 1, \dots, h^{(2)};$$
$$F_i^{(1)}(s^{(1)}) = tanh(s^{(2)}_{h^{(1)}}) \; i = 1, \dots, h^{(1)} \quad \text{and} \qquad F^{(0)}(s^{(0)}) = tanh(s^{(0)}).$$

Given some input vector x, we first forward-propagate x through the network according to equations (1.10) and (1.11) and obtain $s^{(2)}, s^{(1)}, s^{(0)}$ and $y^{(2)}, y^{(1)}, y^{(0)}$. Next, we have to determine the derivatives of the activation functions. In order to save computational time it is more convenient to express the derivatives $\frac{\partial F^{(0)}(s^{(0)})}{\partial s^{(0)}}, \frac{\partial F^{(1)}(s^{(1)})}{\partial s^{(0)}}$ and $\frac{\partial F^{(2)}(s^{(2)})}{\partial s^{(2)}}$ in terms of $y^{(0)}, y^{(1)}$ and $y^{(2)}$. So we get

$$\frac{\partial F^{(2)}}{\partial s^{(2)}} = \begin{pmatrix} 1 - tanh^2(s^{(2)}_{h^{(2)}}) & & \\ & \ddots & \\ & & 1 - tanh^2(s^{(2)}_{h^{(2)}}) \end{pmatrix} = \begin{pmatrix} 1 - (y_1^{(2)})^2 & & \\ & \ddots & \\ & & 1 - (y^{(2)}_{h^{(2)}})^2 \end{pmatrix},$$
$$\frac{\partial F^{(1)}}{\partial s^{(1)}} = \begin{pmatrix} 1 - tanh^2(s^{(1)}_{h^{(1)}}) & & \\ & \ddots & \\ & & 1 - tanh^2(s^{(1)}_{h^{(1)}}) \end{pmatrix} = \begin{pmatrix} 1 - (y_1^{(1)})^2 & & \\ & \ddots & \\ & & 1 - (y^{(1)}_{h^{(1)}})^2 \end{pmatrix},$$
$$\frac{\partial F^{(0)}}{\partial s^{(0)}} = 1 - tanh^2(s^{(0)}) = 1 - (y^{(0)})^2.$$

Now we use Lemma 2 to evaluate the error signals

$$\delta^{(0)} = \frac{\partial F^{(0)}}{\partial s^{(0)}}, \qquad \delta^{(1)} = \delta^{(0)} \tilde{W}^{(0)} \frac{\partial F^{(1)}}{\partial s^{(1)}} \qquad \text{and} \qquad \delta^{(2)} = \delta^{(1)} \tilde{W}^{(1)} \frac{\partial F^{(2)}}{\partial s^{(2)}}.$$

Furthermore, by Lemma 1 we get the desired derivatives

$$\frac{\partial y^{(0)}}{\partial W^{(0)}} = \delta^{(0)^T} \hat{y}^{(1)^T}, \qquad \frac{\partial y^{(0)}}{\partial W^{(1)}} = \delta^{(1)^T} \hat{y}^{(2)^T} \qquad \text{and} \qquad \frac{\partial y^{(0)}}{\partial W^{(2)}} = \delta^{(2)^T} \hat{x}^T.$$

2 Error Functions and Gradient Descent

Assume we want to use a multi-layer feed forward neural network $\mathcal{N}$ with weights w to approximate some function $V : X \to \mathbb{R}^m$. Based on N input data points $x^{(1)}, \dots, x^{(N)} \in X$ and their respective target values $t^{(j)} = V(x^{(j)}), j = 1, \dots, N$ we want to adapt the weights of the network such that the difference between the network outputs and the target values becomes small. A simple and often used measure for how good a neural network approximates a function is the mean squared error.
We measure the error of a single data point x by

$$E(w, x) = \frac{1}{2}\|\mathcal{N}(w, x) - V(x)\|^2 = \frac{1}{2}\sum_{i=1}^{m}(y_i^{(0)} - t_i)^2.$$

The mean error over the whole training set $\{x^{(i)} |\, t^{(i)}\}$ where $i = 1, \dots, N$ is then given by

$$E(w) = \frac{1}{N}\sum_{i=1}^{N} E(w, x^{(i)}).$$

Our goal is to minimize the mean error on the training set with respect to the weights of the network, i.e. the optimization problem

$$\underset{w}{minimize}\ E(w).$$

2.1 Gradient descent

In order to find a minimum on the error surface we update the set of weights w by gradient descent algorithm which has update rule

$$w(t+1) = w(t) + \Delta w(t) \qquad \text{and}$$
$$\Delta w(t) = -\gamma \frac{\partial E(w(t))}{\partial w(t)} = -\gamma \sum_x \frac{\partial E(w(t), x)}{\partial w(t)},$$

where $\gamma > 0$ is called *learning rate*.

Regularization. Training large neural networks on a small training set can lead to overfitting. One common technique to reduce overfitting is known as L2 regularization (see [Nie15]). The idea is to add an extra term to the error function which makes sure that the network weights tend to be small. We can write the regularized error function as

$$E_{\mathcal{R}}(w) = E(w) + \frac{\lambda}{2N}\sum_w w^2,$$

where $\lambda \in [0, 1)$ is called *regularization parameter*.

To obtain the regularized update rule for gradient descent we just take the derivative of $E_{\mathcal{R}}(w)$ with respect to the weights:

$$\frac{\partial E_{\mathcal{R}}(w)}{\partial w_{i,j}} = \frac{\partial E(w)}{\partial w_{i,j}} + \frac{\lambda}{N} w_{i,j},$$

respectively

$$\frac{\partial E_{\mathcal{R}}(w)}{\partial w} = \frac{\partial E(w)}{\partial w} + \frac{\lambda}{N} w.$$

Thus, the update rule for gradient descent with L2 regularization reads

$$\begin{aligned} w(t+1) &= w(t) - \gamma \frac{\partial E_{\mathcal{R}}(w(t))}{\partial w(t)} \\ &= w(t) - \gamma \left(\frac{\partial E(w(t))}{\partial w(t)} + \frac{\lambda}{N} w(t) \right) \\ &= (1 - \frac{\gamma\lambda}{N}) w(t) - \frac{\partial E(w(t))}{\partial w(t)}. \end{aligned}$$

Momentum. As described in [Bis95, p. 267] it may be useful to add a so called momentum term to the gradient descent formula, i.e.

$$\Delta w(t) = -\gamma \frac{\partial E(w(t))}{\partial w(t)} + \mu \Delta w(t-1),$$

where $\mu \in [0, 1)$ is called *momentum parameter*.
When we interpret gradient descent as a motion on the error surface, we can say that the momentum term adds inertia to this motion. Assume gradient descent is located at a point on the error surface with small curvature. Then the changes in $\Delta w(t)$ will be very small and therefore the iteration of weight updates give

$$\begin{aligned} \Delta w(t+1) &\approx -\gamma \frac{\partial E(w(t))}{\partial w(t)} + \mu \left(\frac{\partial E(w(t))}{\partial w(t)} + \mu \left(\frac{\partial E(w(t))}{\partial w(t)} + \dots \right) \right) \\ &= -\gamma \frac{\partial E(w(t))}{\partial w(t)} \left(\mu + \mu^2 + \mu^2 + \dots \right) \\ &= -\frac{\gamma}{1-\mu} \frac{\partial E(w(t))}{\partial w(t)}. \end{aligned}$$

This means in very flat parts of the error surface the momentum term acts like an increased learning rate, whereas in parts with oscillating weight updates, the momentum term cancels out the preceding weight update and therefore has the effect of decreasing the learning rate. Thus, the momentum term can lead to faster convergence.

2.2 Stochastic gradient descent

Calculating the derivatives $\frac{\partial E(w,x)}{\partial W}$ is computationally expensive. When applying gradient descent algorithm, for every weight update we have to calculate $\frac{\partial E(w,x)}{\partial W}$ for every single data point in the training set. As described in [Bot12], instead of summing up $\frac{\partial E(w,x)}{\partial W}$ over the whole training data, it is also possible to use stochastic gradient descent method (SGD). SGD suggests to update the weight parameters w not by

$$w(t+1) = w(t) - \gamma \sum_{i=1}^{N} \frac{\partial E(w, x^{(i)})}{\partial w(t)}$$

but choosing some $x^{(i)}$ randomly from the training set and update the weights by

$$w(t+1) = w(t) - \gamma \frac{\partial E(w, x^{(i)})}{\partial w(t)}.$$

When using SGD we hope to have higher chances to escape local minima and that the randomness caused by this procedure is weaker than the averaging effect of the algorithm. Not only empirical results assure this hope, but also theoretical results from the paper *Stochastic Gradient Descent, Weighted Sampling, and the Randomized Kaczmarz algorithm* of Needell et al. [NWS14]. Essentially, their paper says that if $E(w, x)$ is strongly convex with respect to W and x, and if $E(w, x)$ is smooth in w then

$$\mathbb{E}(w(t)) \to w^*,$$

for $t \to \infty$, where w^* are the optimal weights.

2.3 Error back-propagation

Consider a neural network with weight matrices $w = \{W^{(0)}, \ldots, W^{(k)}\}$. In the following we will see how to evaluate the derivative $\frac{\partial E(w,x)}{\partial W^{(i)}}$ for $i = 0, \ldots, k$. Most of the work is already done because in chapter 1.4 we showed how to calculate the derivative $\frac{\partial y^{(0)}}{\partial W^{(i)}}$. Recall the definition of the error term $\delta_i^{(l)}$ at page 12 and see that for the derivatives of the error function by chain rule we get

$$\begin{aligned} \frac{\partial E(w,x)}{\partial W^{(l)}} &= \sum_{i=1}^{m} \frac{\partial E(w,x)}{\partial y_i^{(0)}} \frac{\partial y_i^{(0)}}{\partial W^{(l)}} \\ &= \sum_{i=1}^{m} \frac{\partial E(w,x)}{\partial y_i^{(0)}} \delta_i^{(l)} y^{(l-1)}. \end{aligned} \tag{2.1}$$

Similar to $\delta_i^{(l)}$ we define $\delta_{error}^{(l)}$ as

$$\delta^{(l)}_{error} := \sum_{i=1}^{m} \frac{\partial E(w,x)}{\partial y^{(0)}_i} \delta^{(l)}_i. \tag{D3}$$

Equation (2.1) then simplifies to

$$\frac{\partial E(w,x)}{\partial W^{(l)}} = \delta^{(l)}_{error} y^{(l-1)}. \tag{2.2}$$

In order to calculate $\delta^{(l)}_{error}$ for $l = 0, \ldots, k$ we derive a recursive formula.

Lemma 3. *Given a multi-layered neural network with k hidden layers, one output layer and weight matrices $W^{(0)}, W^{(1)}, \ldots, W^{(k)}$. The error signal $\delta^{(l)}_{error}$ fulfills the recursive equations*

$$\delta^{(l)}_{error} = \delta^{(l-1)}_{error} \tilde{W}^{(l-1)} \frac{\partial F^{(l)}(s^{(l)})}{\partial s^{(l)}} \quad \text{for } l = 1, \ldots, k \qquad \text{and} \tag{L3.1}$$

$$\delta^{(0)}_{error} = \sum_{i=1}^{m} \frac{\partial E(w,x)}{\partial y^{(0)}_i} \delta^{(0)}_i. \tag{L3.2}$$

Proof. Equation (L3.2) is true by definition. In order to proof (L3.1) we show that

$$\begin{aligned}
\delta^{(l)}_{error} &\overset{(D3)}{=} \sum_{i=1}^{m} \frac{\partial E(w,x)}{\partial y^{(0)}_i} \delta^{(l)}_i \\
&\overset{(L2)}{=} \sum_{i=1}^{m} \frac{\partial E}{\partial y_i} \delta^{(l-1)}_i \tilde{W}^{(l-1)} \frac{\partial F^{(l)}(s^{(l)})}{\partial s^{(l)}} \\
&\overset{(D3)}{=} \delta^{(l-1)}_{error} \tilde{W}^{(l-1)} \frac{\partial F^{(l)}(s^{(l)})}{\partial s^{(l)}}. \quad \square
\end{aligned}$$

With Lemma 3 we have all necessary equations for calculating $\frac{\partial E(w,x)}{\partial W^{(l)}}$. Similar to the back-propagation algorithm we summarize the procedure in the following algorithm.

Error back-propagation algorithm. *Given a multi-layered neural network with k hidden layers, one output layer and weight matrices $W^{(0)}, W^{(1)}, \ldots, W^{(k)}$. In order to calculate $\frac{\partial E(w,x)}{\partial W^{(l)}}$ for $l = 0, 1, \ldots, k$ do the following steps.*

1. *Forward-propagate x through the neural network and obtain the weighted sums $s^{(k)}, s^{(k-1)}, \ldots, s^{(0)}$ and outputs $y^{(k)}, y^{(k-1)}, \ldots, y^{(0)}$.*
2. *Calculate the derivatives $\frac{\partial F^{(0)}(s^{(0)})}{\partial s^{(0)}}, \frac{\partial F^{(1)}(s^{(1)})}{\partial s^{(0)}}, \ldots, \frac{\partial F^{(k)}(s^{(k)})}{\partial s^{(k)}}$.*
3. *Apply* (L3.1) *and* (L3.2) *to evaluate $\delta^{(0)}_{error}, \delta^{(1)}_{error}, \ldots, \delta^{(2)}_{error}$ and*
4. *use* (2.2) *to obtain the sought derivatives $\frac{\partial E(w,x)}{\partial W^{(l)}}, \ldots, \frac{\partial E(w,x)}{\partial W^{(k)}}$.*

Example 5.

We now show in an example how to use the error back-propagation algorithm. Given a neural network with one hidden layer, one output layer of size 1 and weight matrices $W^{(2)}, W^{(1)}, W^{(0)}$. Their activation functions are given by

$$F_i^{(1)}(s^{(1)}) = tanh(s_{h^{(1)}}^{(1)})\ i = 1, \ldots, h^{(1)} \quad \text{and} \qquad F^{(0)}(s^{(0)}) = s^{(0)}.$$

Given some input vector x, we first forward-propagate x through the network according to equations (1.10) and (1.11) and obtain $y^{(2)}, y^{(1)}, y^{(0)}$. Next, we have to determine the derivatives of the activation functions

$$\frac{\partial F^{(0)}(s^{(0)})}{\partial s^{(0)}} = 1 \qquad \text{and}$$
$$\frac{\partial F^{(1)}(s^{(1)})}{\partial s^{(1)}} = \begin{pmatrix} 1 - (y_1^{(1)})^2 & & \\ & \ddots & \\ & & 1 - (y_{h^{(1)}}^{(1)})^2 \end{pmatrix}.$$

In the next step we obtain the error signals

$$\delta_{error}^{(0)} = \frac{\partial E^n}{\partial y^{(0)}} \frac{\partial y^{(0)}}{\partial s^0} = (y^{(0)} - t)^T \quad \text{and}$$
$$\delta_{error}^{(1)} = \delta_{error}^{(0)} \tilde{W}^0 \begin{pmatrix} 1 - (y_1^{(1)})^2 & & \\ & \ddots & \\ & & 1 - (y_{h^{(1)}}^{(1)})^2 \end{pmatrix}.$$

Finally, we are able to calculate the derivatives

$$\frac{\partial E^n}{\partial W^0} = \delta^{0^T} y^{(1)^T} \qquad \text{and} \qquad \frac{\partial E^n}{\partial W^{(1)}} = \delta^{(1)^T} x^T.$$

2.4 Cross-entropy error

Of course there are more possible choice error functions than the mean squared error. The so called *cross-entropy error* and how to evaluate its error signal is described in [Bis95, ch. 6.9]. Recall section 1.2.1 where we introduced the softmax-output function which is classifying data into m different groups by assigning a discrete probability distribution to every input. The softmax activation function is given by

$$F(s) = \frac{1}{\sum_{i=1}^m e^{s_i}} \begin{pmatrix} e^{s_1} \\ \vdots \\ e^{s_m} \end{pmatrix}.$$

The target values for this kind of classification tasks are encoded in the so called 1-of-c scheme. Consider m different classes $C_1, \ldots, C_m$ and a neural network that classifies data

into one of these classes. The target value of some training data point x from class C_j is a m-dimensional vector defined by

$$t_i := \begin{cases} 1 & \text{if } x \in C_i, \\ 0 & \text{otherwise.} \end{cases}$$

For this kind of output it is convenient to use the *cross-entropy error function*, given by

$$E(w, x) = -\sum_{i=1}^{m} t_i ln\left(\frac{y_i}{t_i}\right).$$

In case $t_i = 0$, the expression $t_i ln\left(\frac{y_i}{t_i}\right)$ is not defined, therefore we set

$$t_i ln\left(\frac{y_i}{t_i}\right) := \lim_{t_i \to 0} t_i ln\left(\frac{y_i}{t_i}\right) = 0.$$

Theorem 2 (Error signal of softmax-output)**.** *Given a neural network with softmax-output y and cross-entropy error function. The error signal of an input-target-pair $\{x, t\}$ is given by*

$$\delta_{error} = (y - t)^T.$$

Proof. We first show that

$$\frac{\partial y}{\partial s} = diag(y) - yy^T, \tag{2.3}$$

where $diag(y) = \begin{pmatrix} y_1 & 0 & \dots & 0 \\ 0 & y_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & y_n \end{pmatrix}$.

In case $j = i$ we get

$$\frac{\partial y_i}{\partial s_j} = \frac{\partial F_i(s)}{\partial s_j} = \frac{e^{s_i} \sum_{l=1}^{m} e^{s_l} - e^{s_i} e^{s_i}}{\left(\sum_{l=1}^{m} e^{s_l}\right)^2} = \frac{e^{s_i}}{\sum_{l=1}^{m} e^{s_l}} - \left(\frac{e^{s_i}}{\sum_{l=1}^{m} e^{s_l}}\right)^2 = y_i - {y_i}^2$$

whereas in case $j \neq i$, the derivative

$$\frac{\partial y_i}{\partial s_j} = \frac{\partial F_i(s)}{\partial s_j} = 0 - \frac{e^{s_i} e^{s_j}}{\left(\sum_{l=1}^{m} e^{s_l}\right)^2} = -y_i y_j.$$

By summarizing both cases in matrix notation we obtain (2.3). Now we can calculate

$\delta_{error}^{(0)}$:

$$\begin{aligned}
\delta_{error}^{(0)} &\overset{(D3)}{=} \frac{\partial E}{\partial y}\frac{\partial y}{\partial s} \\
&\overset{(2.3)}{=} -\begin{bmatrix} \frac{t_1}{y_1} & \frac{t_2}{y_2} & \dots & \frac{t_m}{y_m} \end{bmatrix} \left(diag(y) - yy^T\right) \\
&= -t^T + \underbrace{\left(\sum_{l=1}^{m} t_l\right)}_{=1} y^T \\
&= -t^T + y^T \\
&= (y-t)^T. \quad \square
\end{aligned}$$

2.5 Initializing weights

In the previous chapter we saw how to obtain the derivatives of a neural network output with respect to its weights. We are able to use these derivatives to perform gradient descent in order to adapt the network weight according to a given training data set. In practice we have to initialize a set of network weights. Choosing suitable initial weights can be crucial for the performance of the optimization algorithm. We are using random initial weights in order to avoid problems due to symmetries of the network.

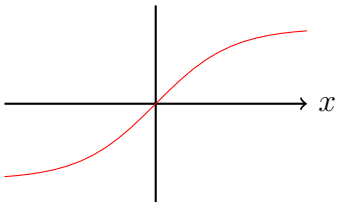


Figure 9: *Tangens hyperbolicus* activation function.

As suggested in [Bis95, p. 261] we have a look at the sigmoidal activation function like tangens hyperbolicus and see that for values close to zero the activation function will be approximately linear. For large values the derivatives of the activation function will be very small which will lead to a flat error surface. In order to make use of the non-linear area of the activation function, the summed inputs should be of order unity. One option to achieve this is to choose the probability distribution such that the weighted sums $s = \langle w, x \rangle$ have expectation 0 and variance 1. We get that by drawing weights from a Gaussian probability distribution with mean $\mu = 0$ and variance σ^2. A reasonable choice of σ^2 depends on the size of the layers as we can see in the following.

For one unit of a layer of size h, the weighted sum is given by

$$s = \sum_{i=1}^{h} w_i x_i.$$

Assuming the inputs are normalized (i.e. $\mathbb{E}(x_i) = 0$ and $\mathbb{E}(x_i^2) = 1$) and all w_i are independently drawn from a normal distribution with mean $\mathbb{E}(w_i) = 0$ and $\mathbb{E}(w_i^2) = \sigma^2$ we get

$$\mathbb{E}(s) = \mathbb{E}\left(\sum_{i=1}^{h} w_i x_i\right) = \sum_{i=1}^{h} \mathbb{E}(w_i)\mathbb{E}(x_i) = 0$$

and

$$\begin{aligned} Var(s) = \mathbb{E}(s^2) - \underbrace{(\mathbb{E}(s))^2}_{=0} &= \mathbb{E}\left(\sum_{i=1}^{h} w_i x_i \sum_{j=1}^{h} w_j x_j\right) \\ &= \sum_{i=1}^{h} \mathbb{E}(w_i^2 x_i^2) + \sum_{i \neq j}^{h} \mathbb{E}(w_i w_j x_i x_j) \\ &= \sum_{i=1}^{h} \mathbb{E}(w_i^2)\mathbb{E}(x_i^2) + \sum_{i \neq j}^{h} \underbrace{\mathbb{E}(w_i)}_{=0} \mathbb{E}(w_j x_i x_j) = \sigma^2 h \end{aligned}$$

If we set $\sigma = \frac{1}{\sqrt{h}}$, then the variance of s is 1. We used that x_i and w_i are uncorrelated.

3 Minimax Algorithm

The minimax algorithm is used for finding an optimal strategy for two-player zero-sum games. In our context two-player zero-sum means that there are two players whose actions alternate and the game is fully observable and deterministic. Each player knows all possible moves the opponent can make (unlike Scrabble for example) and objective values of game states have to be opposite numbers. For example, if some game state is a winning state for player one and therefore has value (+1), then the other player necessarily looses (-1). These conditions hold for Tic-Tac-Toe. Before we explain the minimax algorithm we give some definitions in order to be able to describe Tic-Tac-Toe in a mathematical way.

Definition 3 (Successor)**.** *Let $\mathcal{B}$ be the set of all possible Tic-Tac-Toe boards. A move b is called legal in $B \in \mathcal{B}$ if according to the rules of the game, b is a possible move at board B. $B + b$ is defined as the board resulting performing move b at state B. Furthermore, define the set of successors of B by*

$$S(B) = \{B + b | b \text{ legal in } B\}.$$

In other words, $S(B)$ contains all boards resulting from legal moves in B.

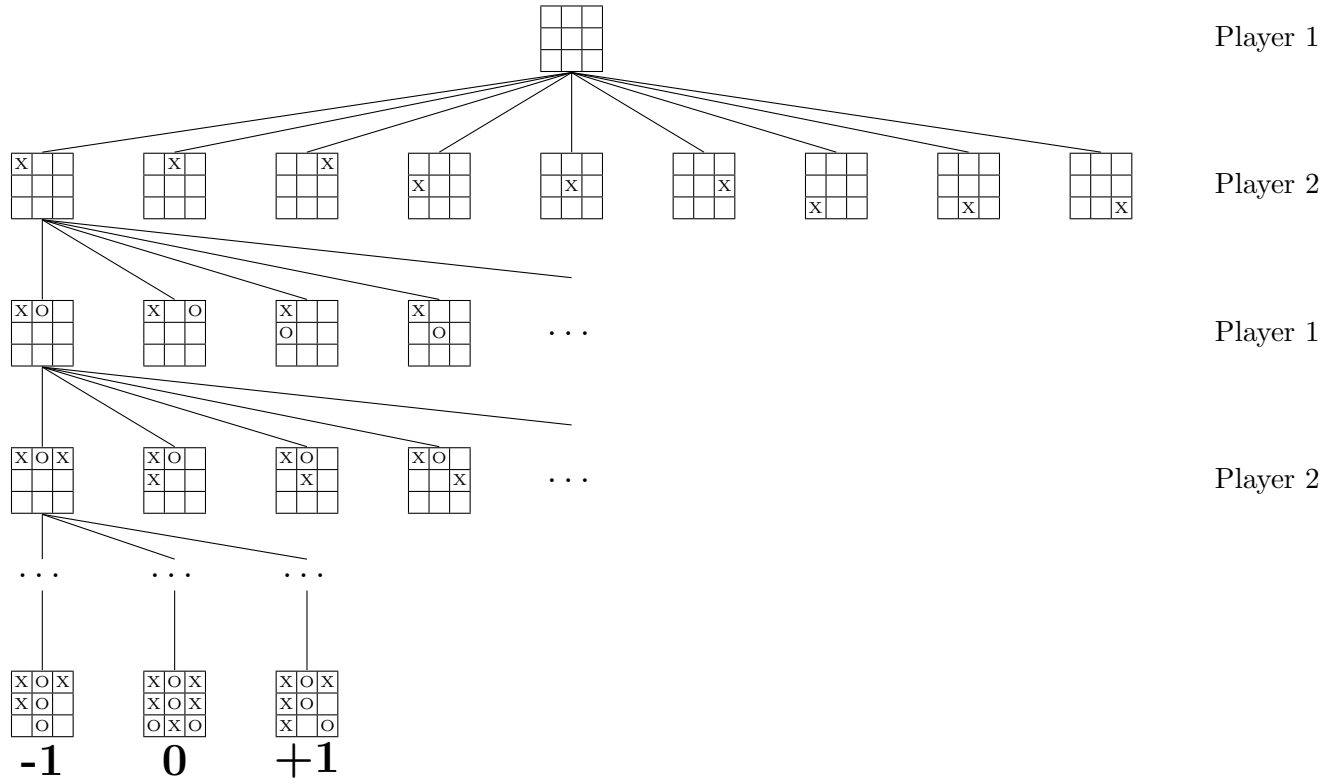
Definition 4 (Terminal boards)**.** *A board $B \in \mathcal{B}$ is called terminal if $S(B) = \emptyset$, i.e. the game ends in B with a win for player "X", win for player "0" or draw.*

Definition 5. *Define $\mathcal{B}_X \subset \mathcal{B}$ as the set of boards where player "X" has to do the next move and $\mathcal{B}_O \subset \mathcal{B}$ as the set of boards where player "O" has to move.*

Definition 6 (Utility function)**.** *Let $\mathcal{T} \subset \mathcal{B}$ be the set of all terminal boards. We define the utility function $U : \mathcal{T} \to \mathbb{R}$ by*

$$U(T) = \begin{cases} 1 & \text{if player "X" wins,} \\ -1 & \text{if player "O" wins and} \\ 0 & \text{otherwise.} \end{cases}$$

Game Tree. A Tic-Tac-Toe game tree is a directed graph whose nodes stand for boards and whose edges represent moves. Assume a game tree starting at some board B. The children of a board are given by its successors $B' \in S(B)$. The leaves of the tree represent the terminal states T having some utility value $U(T)$.

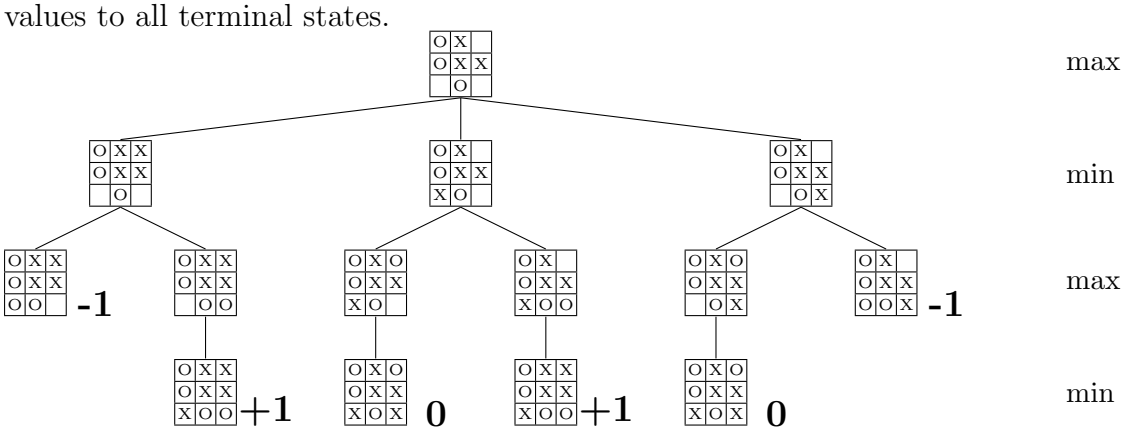


Given that game tree and assuming a perfect playing opponent, we can find the optimal strategy by determining the so called minimax value of each node (see [RN09, ch. 6.2]).

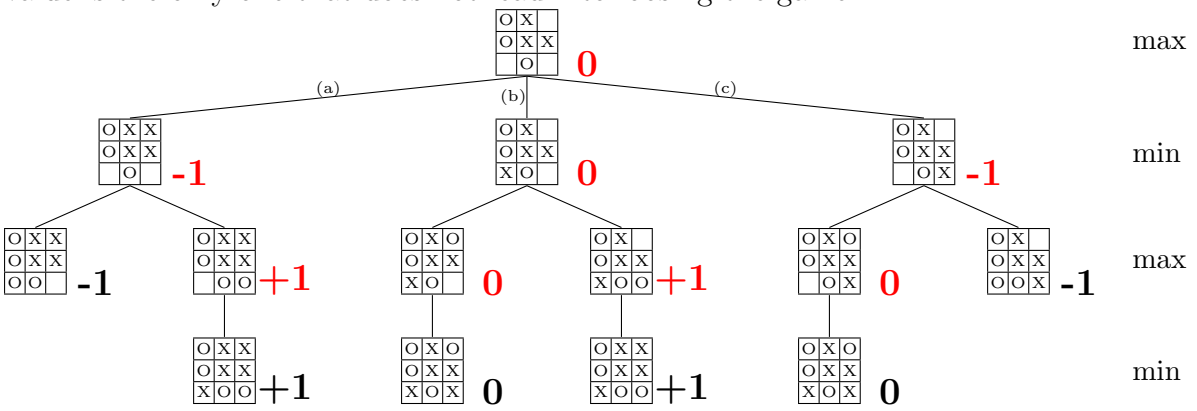
Definition 7. *Let a game be at some state B. The minimax value of that state is defined in a recursive way by*

$$MiniMax(B) = \begin{cases} U(B) & \text{if } B \in \mathcal{T}, \\ \max_{B' \in S(B)} MiniMax(B') & \text{if } B \in \mathcal{B}_X, \\ \min_{B' \in S(B)} MiniMax(B') & \text{if } B \in \mathcal{B}_O. \end{cases} \tag{3.1}$$

Example. Let's have a look at an example of a Tic-Tac-Toe game in state $B =$  . In this state it is player "X"'s turn. First we develop the game-tree and assign the utility values to all terminal states.



We can find the minimax value of  by assigning the minimum/maximum value among siblings to their respective parents. We start at the bottom of the tree and go level by level back to the root. We can see that for player "X" the move with the highest minimax value is the only one that does not lead into loosing the game.



Decision rule. Note how we defined the utility function. It is in player "X"'s interest to choose a move that maximizes the minimax value whereas player "O" benefits from a move leading to a small minimax value. Based on that, the best move b^* at board B is given by

$$b^*(B) = \begin{cases} \underset{b \text{ legal in } B}{\operatorname{argmax}} \, MiniMax(B+b) & \text{if } B \in \mathcal{B}_X \\ \underset{b \text{ legal in } B}{\operatorname{argmin}} \, MiniMax(B+b) & \text{if } B \in \mathcal{B}_O \end{cases}. \tag{3.2}$$

Depth of the game tree. Unfortunately, game trees for games more complex than Tic-Tac-Toe are too big to be searched by minimax algorithm. According to [Wik17],the number of all possible courses of Tic-Tac-Toe amounts to 255169, for chess it is about 10^{25}. If we still want to use the minimax algorithm for more complex games then we have to stop the algorithm at a certain depth of the game tree and return 0 if we do not reach a terminal state. This consideration leads to another definition of the minimax value.

Definition 8. *Let a game be at some state $B \in \mathcal{B}$. The minimax value of depth $d \in \mathbb{N}_0$ at state B is defined as*

$$MiniMax(B,d) = \begin{cases} U(B) & \text{if } B \in \mathcal{T} \\ 0 & \text{if } d{=}0 \\ \max_{B' \in S(B)} MiniMax(B', d-1) & \text{if } B \in \mathcal{B}_X \\ \min_{B' \in S(B)} MiniMax(B', d-1) & \text{if } B \in \mathcal{B}_O \end{cases} \tag{3.3}$$

We can say that a player $\mathcal{M}$ playing according to minimax values of depth d plans d moves ahead. The greater d the better decisions player $\mathcal{M}$ can make. Therefore we will also use the term *intelligence* of a Mini-Max player.

4 Neural Networks for Classifying Tic-Tac-Toe Boards

4.1 Network architecture

Now we have introduced all tools we need to design a neural network that classifies Tic-Tac-Toe boards. In chapter 1 we introduced how to use neural networks, whereas in chapter 3 we formalized playing Tic-Tac-Toe. In this chapter we will design a neural network that takes Tic-Tac-Toe boards as inputs.
Let $\mathcal{B}$ be the set of all possible Tic-Tac-Toe boards. The minimax function introduced in chapter 3 assigns values $+1, 0, -1$ to each board. Therefore we define three classes: let C_1 contain all boards with minimax value -1, C_2 contain all boards with minimax value 0 and C_3 contain all boards with minimax value $+1$. Boards in C_1 are those which (assuming a perfect playing opponent) lead to a success of player "X", in C_2 are boards leading to a draw and in C_3 are boards that end with play "O" winning. We will use a neural network with softmax-output to approximate the function

$$V : \mathcal{B} \to \mathbb{R}^3 \qquad V(B) = \begin{cases} (1,0,0)^T & \text{if } x \in C_1, \\ (0,1,0)^T & \text{if } x \in C_2 \text{ and} \\ (0,0,1)^T & \text{if } x \in C_3. \end{cases} \tag{4.1}$$

Encoding of boards. In order to use Tic-Tac-Toe boards as neural network inputs, we have to translate these boards into numerical vectors. A Tic-Tac-Toe board has 9 slots $b_1, \ldots, b_9$ which are either empty or marked with "X" or "O". Therefore, to a board B we assign a vector $x \in \mathbb{R}^9$:

$$\begin{array}{|c|c|c|}\hline b_1 & b_2 & b_3 \\ \hline b_4 & b_5 & b_6 \\ \hline b_7 & b_8 & b_9 \\ \hline \end{array} \mapsto (x_1, x_2, \ldots, x_9) \qquad \text{with } x_i = \begin{cases} -1.35 & \text{if } b_i = \text{"X"}, \\ 0.45 & \text{if } b_i = \text{" "} \text{ and} \\ 1.05 & \text{if } b_i = \text{"O"}. \end{cases} \tag{4.2}$$

These three numbers $-1.35, 0.45, 1.05$ might seem odd. First, note that we do not assign 0 to empty slots of Tic-Tac-Toe boards, because for zero-valued inputs the network does not make use of its weight parameters. Second, recall that in chapter 2.5 we justified how to initialize network weights. We made the assumption that the inputs have mean close to 0 and variance close to 1. By encoding the boards as introduced in (4.2) we ensure this assumption is satisfied, because experiments show that

$$\sum_{x \in \mathcal{B}} \sum_{i=1}^{9} x_i \approx 0 \qquad \text{and} \qquad \sum_{x \in \mathcal{B}} \sum_{i=1}^{9} x_i^2 \approx 1. \tag{4.3}$$

Network Architecture. Next we choose sizes for the hidden layers. Experiments with different numbers and sizes of hidden layers revealed that a network with two hidden layers of size 60 is doing well (see table 1 on page 28 for more details) for a neural network that classifies Tic-Tac-Toe boards. As described in chapter 2.5 we initialize the weights by drawing from a normal distribution depending on the layer sizes. That is

$$w^2_{i,j} \sim \mathcal{N}(0, \frac{1}{\sqrt{9}}) \qquad w^1_{i,j} \sim \mathcal{N}(0, \frac{1}{\sqrt{60}}) \qquad w^0_{i,j} \sim \mathcal{N}(0, \frac{1}{\sqrt{60}}) \tag{4.4}$$

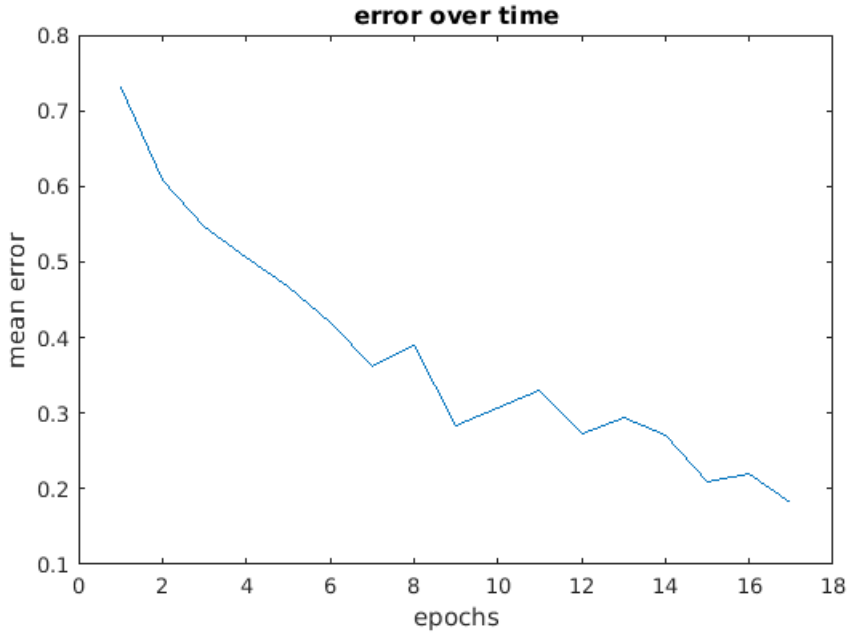


Figure 10: This plot shows the mean training error $E(W)$ over time using stochastic gradient descent, learning rate $\gamma = 0.005$, momentum parameter $\mu = 0.4$ and no regularization. The training set contains all Tic-Tac-Toe boards. We stop training when the mean error is below $\epsilon = 0.2$.

As activation functions we choose *tangens hyperbolicus* for both hidden layers and softmax-output for the output layer. The output $y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$ of the softmax layer is interpreted as a probability distribution over the three classes C_1, C_2, C_3. Therefore we categorize the input to the class C_{i^*} with highest probability, i.e.

$$i^* = \arg\max_i y_i. \tag{4.5}$$

4.2 Training

First we split the whole set of Tic-Tac-Toe boards randomly in a training set $\mathcal{B}_{training}$ and a test set $\mathcal{B}_{test}$. Let $N_{training} = |\mathcal{B}_{training}|$ denote the size of the training set. For weight optimization on the training set we use stochastic gradient descent. For stochastic gradient descent we update weights after every sample shown to the network. In order to measure how many iterations are needed to reach a satisfying training error, we introduce the term epoch. We say that updating the weights by $N_{training}$ training samples is equivalent to one epoch. We stop training when the mean training error $E(W)$ is less than some tolerance ϵ. Experiments show that after about 20-50 epochs the network converges to a state where all boards in the test group are classified correctly.
Let $N = |\mathcal{B}|$ be the number of all Tic-Tac-Toe boards, and $N_{training} = |\mathcal{B}_{training}|$ the size of the training set. In order to check the generalization ability of the network we test how

layer sizes	*testrate*	*epochs*
40	0.86	200
60	0.89	200
80	0.87	200
$20-20$	0.85	200
$40-40$	0.89	200
$60-60$	0.93	187
$80-40$	0.90	200
$80-80$	0.91	200
$20-40-20$	0.84	200
$40-80-40$	0.91	76

Table 1: The table lists rates of correct classification on test data depending on the network architecture used during the training. The network was trained on 50% of the data with learning rate $\gamma = 0.05$, momentum $\mu = 0.4$ and regularization parameter $\lambda = 0.002$. Training was stopped after 200 epochs if the training error did not decrease below the desired tolerance $\epsilon = 0.04$.

many boards from $\mathcal{B}_{test}$ are classified correctly. The choice of the architecture of a neural network is crucial for the generalization ability. Therefore we test the generalization ability for several network architectures. The results summarized in Table 1 justify the choice of layer sizes in chapter 4.1.
Not only the network architecture, but also optimization parameters have influence on the generalization ability of the network. The first one is the tolerance ϵ which influences how good we adjust the weights to the training set. The second one is the regularization parameter. Results for different choices of parameters are presented in Table 2.
The best results were obtained with tolerance $\epsilon = 0.05$ and regularization parameter $\lambda = 0.002$. Now we check the generalization ability of this network based on the size of the training set.

Symmetry of Tic-Tac-Toe boards. By making use of the symmetry of Tic-Tac-Toe boards we can increase the generalization ability even more. As explained in [Tap11, p.184] a rectangle has eight symmetries. A Tic-Tac-Toe board therefore has up to eight equivalent game states which can be obtained by rotations and reflections.
So instead of classifying a board B by (4.5) we average over the network outputs of all boards symmetric with respect to B. Let $B^{(1)}, B^{(2)}, \dots, B^{(8)}$ be the eight symmetrically equivalent boards to B (including B itself) resulting from transformations shown in figure 11 and let $y^{(1)}, \dots, y^{(8)}$ be the network outputs of these boards. Denoting

$$\bar{y} = \frac{1}{8} \sum_{i=1}^{8} y^{(i)},$$

	regularization parameter				
tolerance	$\lambda = 0$	$\lambda = 0.001$	$\lambda = 0.002$	$\lambda = 0.003$	$\lambda = 0.004$
$\epsilon = 0.03$	0.85	0.90	-	-	-
$\epsilon = 0.05$	0.85	0.88	0.92	0.92	-
$\epsilon = 0.1$	0.83	0.87	0.89	0.91	-
$\epsilon = 0.2$	0.81	0.83	0.85	0.85	0.85
$\epsilon = 0.3$	0.78	0.80	0.83	0.82	0.83

Table 2: The table lists rates of correct classification on test data depending on the regularization parameter λ and tolerance ϵ used during the training. The network was trained on 50% of the data and tested on the other 50%. Empty entries imply that the training error did not decrease below the desired tolerance.

we then categorize B to class C_i where

$$i^* = \arg\max_i \bar{y}_i. \tag{4.6}$$

.

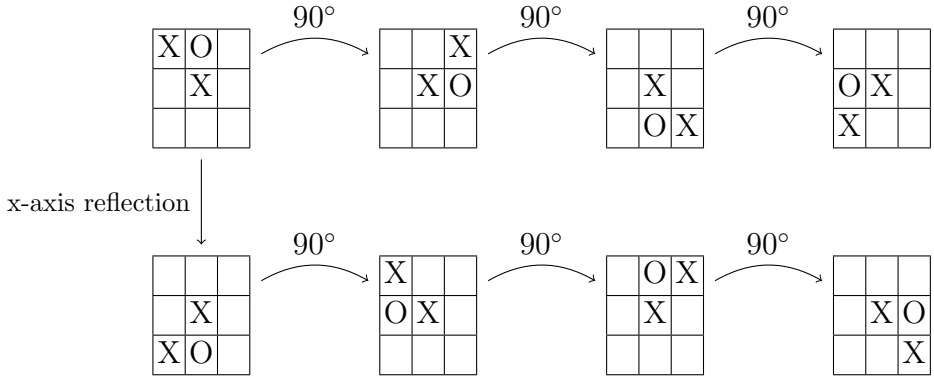


Figure 11: The eight symmetries of a Tic-Tac-Toe board can be obtained by one reflection and six rotations of 90°

Comparing the standard classification (4.5) and the symmetric classification (4.6) we see that the symmetric classification performs much better, especially on small training sets. The rate of correctly classified test boards in $\mathcal{B}_{test}$ depending on ratio $\frac{N_{training}}{N}$ is summarized in figure 12.

The results show that a network trained on only 20% of all boards is able to categorize about 88% of boards it has not seen before.

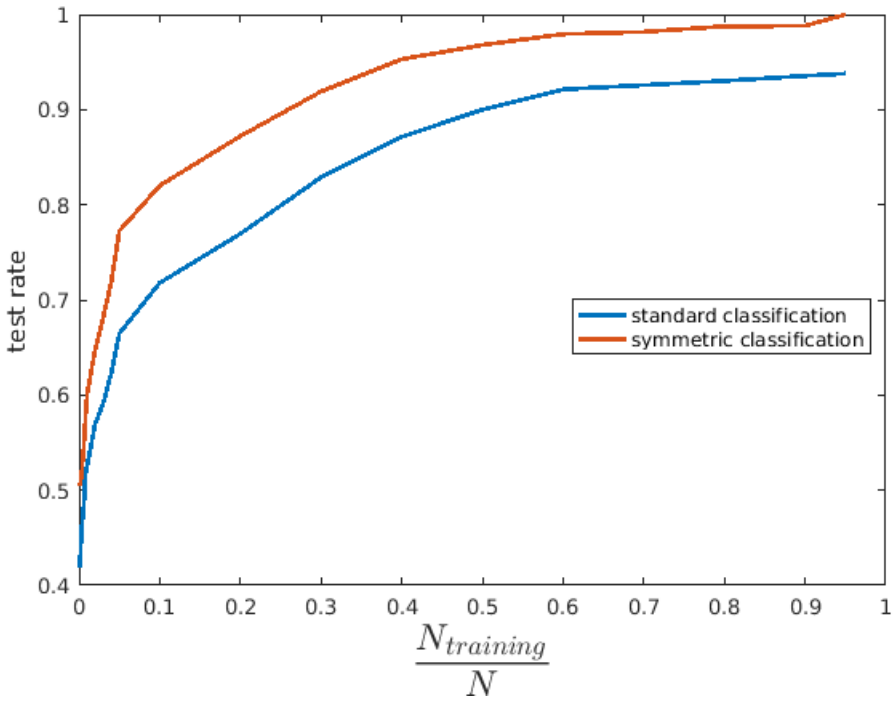


Figure 12: This plot shows how many Tic-Tac-Toe boards from the test group were classified correctly, depending on the size of the training set. Each training was done by stochastic gradient descent with batch size 1, learning rate $\gamma = 0.005$, momentum parameter $\mu = 0.4$, regularization parameter $\lambda = 0.002$. In case the mean error did not decrease below $\epsilon = 0.05$ the training stopped after 200 epochs.

5 Dynamical Learning

5.1 Reinforcement-learning approach

5.1.1 Learning procedure

In chapter 1 we derived the minimax algorithm and a decision rule based on the minimax values of game states. In this chapter we define different artificial Tic-Tac-Toe players who choose their moves according to a certain strategy. Some of these players will be able to adapt their strategy based on the outcomes of completed games they played. We start with the minimax player who chooses its move based on the minimax decision rule from chapter 1.

Definition 9 (Minimax-player)**.** *Let $B \in \mathcal{B}$ be the current state of a Tic-Tac-Toe game. A minimax-player $\mathcal{M}_d$ of intelligence d is a Tic-Tac-Toe player who chooses its move according to the minimax decision rule, i.e.*

$$b^*(B) = \begin{cases} \underset{b \text{ legal in } B}{\operatorname{argmax}} \, MiniMax(B+b,d) & \text{if } B \in \mathcal{B}_X \\ \underset{b \text{ legal in } B}{\operatorname{argmin}} \, MiniMax(B+b,d) & \text{if } B \in \mathcal{B}_O \end{cases} \tag{5.1}$$

As we saw in the previous chapter, a neural network can be trained for classifying Tic-Tac-Toe boards. Based on the network's output we can define another decision rule.

Decision rule for neural network player.

Definition 10 (Softmax decision rule)**.** *Given a Tic-Tac-Toe board B and a neural network $\mathcal{N}$ with softmax-output layer of size 3. The network output is denoted by $\mathcal{N}(B) = \begin{pmatrix} \mathcal{N}_1(B) \\ \mathcal{N}_2(B) \\ \mathcal{N}_3(B) \end{pmatrix}$. The defensive softmax decision rule in state B given neural network $\mathcal{N}$ is defined by*

$$b^*(B) = \begin{cases} \underset{b \text{ legal in } B}{\operatorname{argmin}} \, \mathcal{N}_3(B+b) & \text{if } B \in \mathcal{B}^X \\ \underset{b \text{ legal in } B}{\operatorname{argmin}} \, \mathcal{N}_1(B+b) & \text{if } B \in \mathcal{B}^O \end{cases}, \tag{5.2}$$

whereas the offensive softmax decision rule is defined by

$$b^*(B) = \begin{cases} \underset{b \text{ legal in } B}{\operatorname{argmax}} \, \mathcal{N}_1(B+b) & \text{if } B \in \mathcal{B}^X \\ \underset{b \text{ legal in } B}{\operatorname{argmax}} \, \mathcal{N}_3(B+b) & \text{if } B \in \mathcal{B}^O. \end{cases} \tag{5.3}$$

Definition 11 (Neural network player)**.** *Let $B \in \mathcal{B}$ be the current state of a Tic-Tac-Toe game and $\mathcal{N}$ a neural network with three-dimensional softmax-output. A defensive (offensive) neural network player $\mathcal{P}_\mathcal{N}$ is a Tic-Tac-Toe player who chooses its move according to the defensive (offensive) softmax decision rule in state B given neural network N.*

Updating strategies

Definition 12 (game)**.** *A game* $G := (B_1, B_2, \ldots, B_n)$ *is a sequence of Tic-Tac-Toe boards.*

In the following we describe by an algorithm how to train a neural network player based on the outcomes of matches against a minimax player: We start with initializing a neural network $\mathcal{N}$ with softmax-output as described in chapter 1.2.1. Afterwards we simulate m games $G_1, \ldots, G_m$ between neural network player $P_{\mathcal{N}}$ and minimax player M_d of intelligence d. We furthermore define the set of seen boards $\mathcal{B}_{seen} := \bigcup_{i=1}^{m} G_i$ which contains all visited boards during the realization of the games. After playing m games we stop and train the neural network on the set of seen boards and target values $t(B)$ that contain information about the outcomes of the games played.
In the beginning the set $\mathcal{B}_{seen}$ is empty. When adding an unseen board to this set we also initialize it as a "neutral" board. This means the initial target value of a new board is an uniform distribution over the three classes C_1 (winning boards for player "X"), C_3 (winning board for player "O"). We can interpret these target values as estimates of the outcome probabilities at specific game state. After adding boards to the training set $\mathcal{B}_{seen}$ we evaluate all games $G_1, \ldots, G_m$ we played and update these estimates according to the outcome of the game.
This idea leads to following algorithm.

Definition 13 (Target update algorithm)**.** *Given a training set* $\{B, t(B)\}$ *with* $B \in \mathcal{B}_{seen}$ *and a game* $G = (B_1, \ldots, B_N)$ *and some update rate* $\alpha \in [0, 1]$*:*

1. *For* $i = 1, \ldots, N$*:*
   - *if* $B_i \notin \mathcal{B}_{seen}$ *then set* $t(B_i) := (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})^T$
2. $\mathcal{B}_{seen} = \mathcal{B}_{seen} \cup \{B_1, \ldots, B_N\}$
3. *Set* $t(B_N) = \begin{cases} (1,0,0)^T & \text{if player X is the winner of G} \\ (0,1,0)^T & \text{if G ends draw} \\ (0,0,1)^T & \text{if player O is the winner of G} \end{cases}$
4. *For* $i = N-1, \ldots, 1$*:*
   - *set* $t(B_i) = (1 - \alpha^{N-i})t(B_i) + \alpha^{N-i} t(B_N)$

The important part of the algorithm is step 4. Here we update the target values of boards depending on the outcome of that game. The target values show estimates of outcome probabilities. For example, if a game ends draw then for all boards visited during that game we increase the estimate of a draw and decrease the estimates of the winning probabilities of player "X" and player "O".

Boards:	B_i	$\cdots$	B_{N-2}	B_{N-1}
target values	$\begin{pmatrix}0.3\\0.44\\0.26\end{pmatrix}$		$\begin{pmatrix}0.4\\0.2\\0.4\end{pmatrix}$	$\begin{pmatrix}0.33\\0.33\\0.33\end{pmatrix}$
updated target values	$(1-\alpha^{N-i})\begin{pmatrix}0.3\\0.44\\0.26\end{pmatrix}+\alpha^{N-1}\begin{pmatrix}0\\0\\1\end{pmatrix}$		$(1-\alpha^2)\begin{pmatrix}0.4\\0.2\\0.4\end{pmatrix}+\alpha^2\begin{pmatrix}0\\0\\1\end{pmatrix}$	$(1-\alpha^1)\begin{pmatrix}0.4\\0.2\\0.4\end{pmatrix}+\alpha^1\begin{pmatrix}0\\0\\1\end{pmatrix}$

Figure 13: Illustration of step 4 in target update algorithm. Assume we want to update target values after a game $G = \{B_1, \ldots, B_N\}$ where player "X" lost the game. Following the algorithm we first update $t(B_N)$ according to the outcome. After that, for $i = 1, \ldots, N-1$ we calculate a convex combination of $t(B_i)$ and $t(B_{N-1})$ depending on rate α. Boards appearing late in the game get new target values which are close to the outcome. Target values of boards in the beginning of the game will only be slightly changed.

After updating the target values we retrain the network based on $\mathcal{B}_{seen}$ and the board's new respective target values. We train the network until the training error is below some tolerance $\epsilon \geq 0$.

Definition 14 (Reinforcement-learning algorithm)**.** *Given simulation size m, learning error ϵ and update rate $\alpha \in [0, 1]$,*

1. *initialize a neural network $\mathcal{N}$ with random weights according to* (4.4)*,*
2. *simulate m games of $P_{\mathcal{N}}$ vs. $P_{\mathcal{M}}$ and obtain $G_1, \ldots, G_m$,*
3. *update the training set $\mathcal{B}_{seen}$ and its target values by $G_1, \ldots, G_m$ with update rate α according to target update algorithm,*
4. *train the network until $E(w) \leq \epsilon$ and*
5. *go back to step 2.*

Repeating this procedure over and over again, we hope that the target values converge to good estimates of the outcome of a game. When the neural network player is at an unseen game state, the neural network hopefully is able to generalize from visited boards and therefore gives a reasonable estimate.
Note that (for example after loosing a game) we not only increase the loosing probability for the boards the neural-net-player saw but also reduce the loosing probability of the other player. So the network is not only considering the experience of one player but also the results of the opponent player. This suggests that a network playing against a smart opponent improves faster than a network playing against a less skilled opponent.

5.1.2 Results

In the following a neural network player $\mathcal{P}_{\mathcal{N}}$ was trained by reinforcement-learning algorithm with update rate $\alpha = 0.4$. Results show that the performance of the learning neural

	learning by playing against						
opponent	$\mathcal{P}_{\mathcal{M}_0}$	$\mathcal{P}_{\mathcal{M}_1}$	$\mathcal{P}_{\mathcal{M}_2}$	$\mathcal{P}_{\mathcal{M}_3}$	$\mathcal{P}_{\mathcal{M}_4}$	$\mathcal{P}_{\mathcal{M}_5}$	$\mathcal{P}_{\mathcal{M}_6}$
$\mathcal{P}_{\mathcal{M}_0}$	0	0	0	0	0	0	0
$\mathcal{P}_{\mathcal{M}_1}$	100	100	100	100	200	100	100
$\mathcal{P}_{\mathcal{M}_2}$	-	2300	500	400	300	300	500
$\mathcal{P}_{\mathcal{M}_3}$	-	2500	600	400	400	800	400
$\mathcal{P}_{\mathcal{M}_4}$	-	2900	800	400	500	400	600
$\mathcal{P}_{\mathcal{M}_5}$	-	2800	600	400	400	300	400
$\mathcal{P}_{\mathcal{M}_6}$	-	-	-	-	-	-	-

Table 3: The table lists the number of games needed for a learning player to perform equally or better than his opponent. Missing entries indicate that the learning agent failed to reach a level where it outperforms his opponent

network player is dependent on how good the opponent is. The reinforcement-learning algorithm was run with Minimax-player $\mathcal{P}_{\mathcal{M}_d}$ with different search depths d. The target values and the network weights were updated after every 100 games. The network learning was stopped if the training error was lower than $\epsilon = 0.1$. After 200 epochs, if the training error was still greater than ϵ the training was stopped as well.
In the following plots we see the performance of $\mathcal{P}_{\mathcal{N}}$ against Minimax-players during the training procedure. The results show that a player that is learning by playing against a less skilled opponent ($\mathcal{P}_{\mathcal{M}_0}$) performs very well against $\mathcal{P}_{\mathcal{M}_0}$ and $\mathcal{P}_{\mathcal{M}_1}$. But its performance against smarter opponents improves only during the first 200 games and then stays at a low level. Whereas a player $\mathcal{P}_{\mathcal{N}}$ that is learning from games against smart opponent is improving much faster. In table 3 we show the number of games needed for learning players to perform better than their respective opponents. We can see two effects overlap: The first one is that a player learning from a smart opponent (minimax player with search depth $3-6$) improves faster than by learning from a stupid opponent (minimax player with search depth 0 or 1). The second observation we can make is that learning by playing against a perfect opponent $\mathcal{P}_{\mathcal{M}_6}$ leads to slower learning than learning from $\mathcal{P}_{\mathcal{M}_4}$. A reason for that may be the bad exploration of game states when playing against a perfect opponent. There are many game states that do not appear in these games, but in games against other opponents.

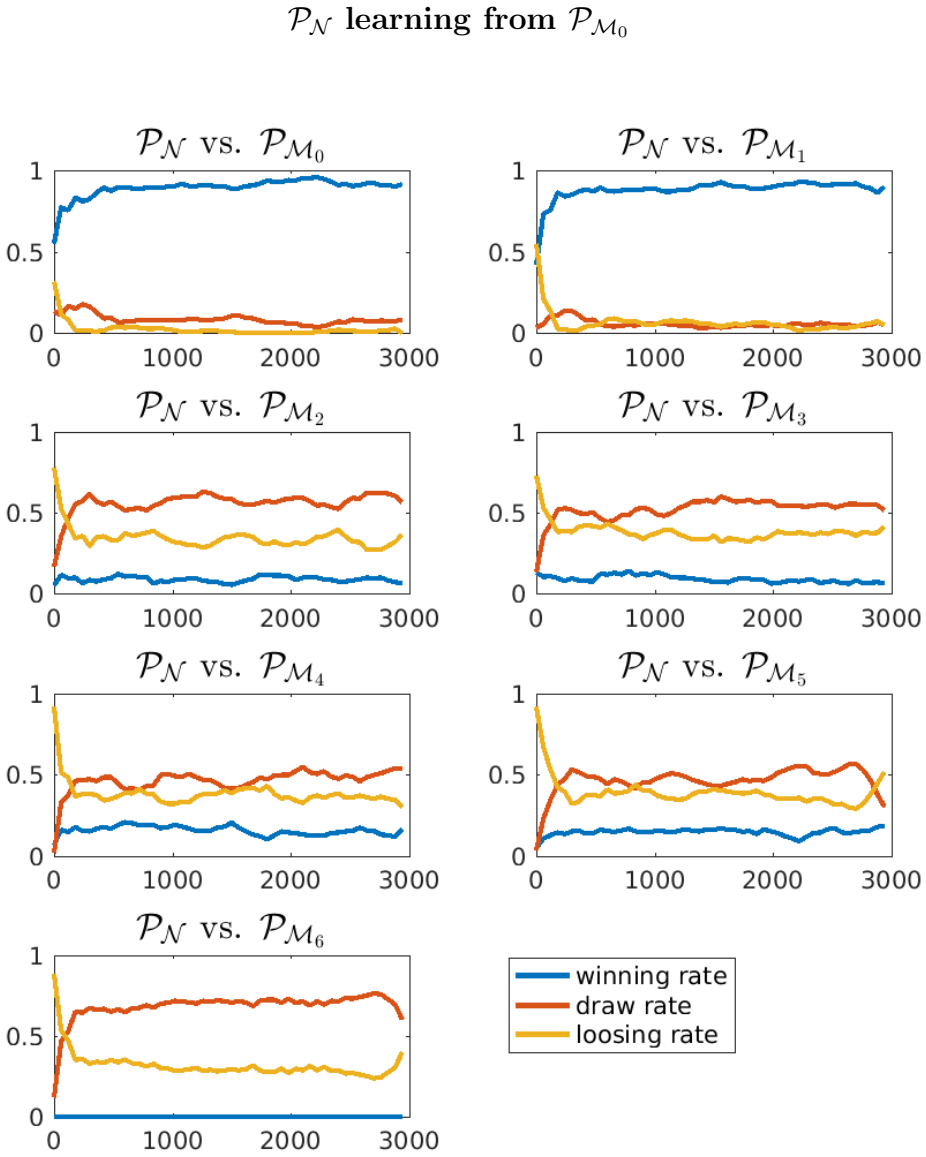

$\mathcal{P}_\mathcal{N}$ learning from $\mathcal{P}_{\mathcal{M}_0}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_0}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_1}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_2}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_3}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_4}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_5}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_6}$
0
0.5
1
0
1000
2000
3000
winning rate
draw rate
loosing rate

Figure 14

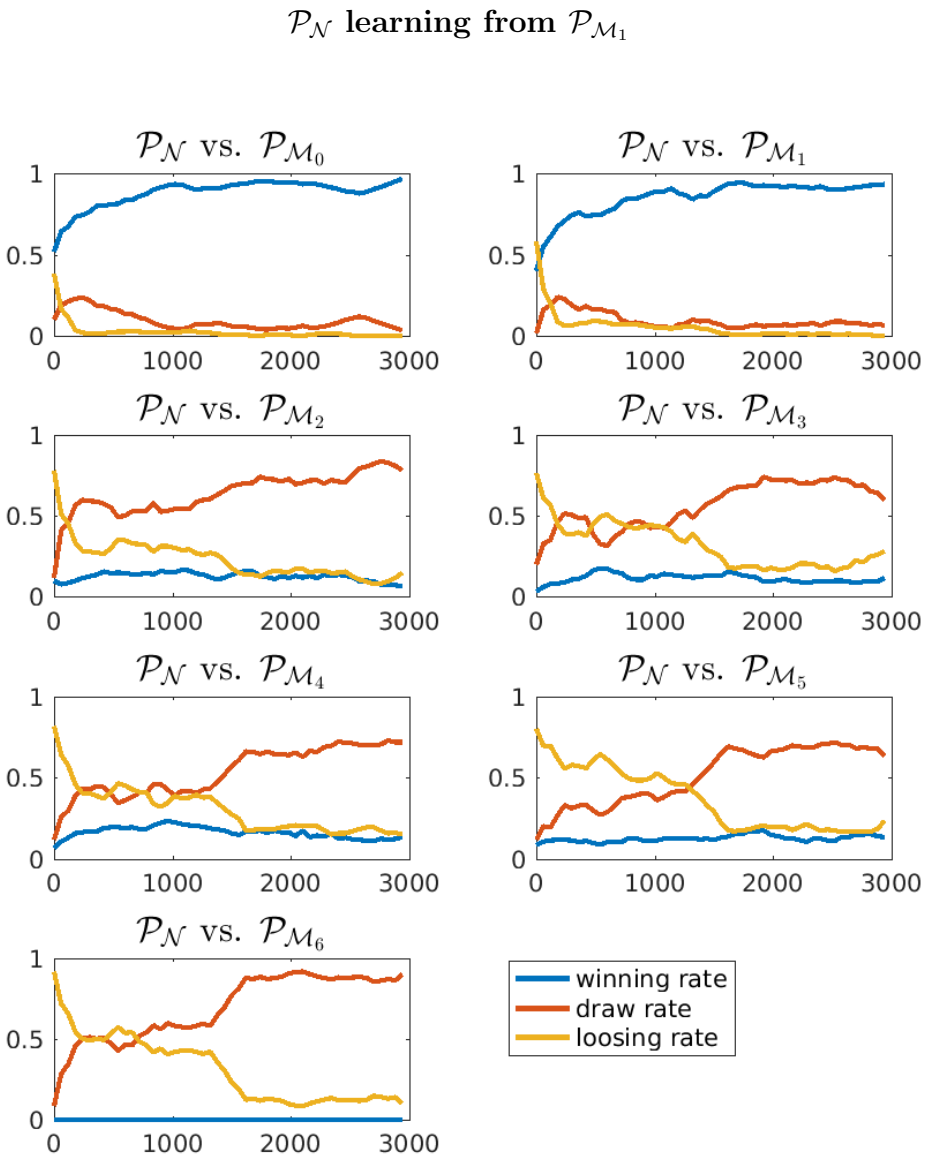

$\mathcal{P}_\mathcal{N}$ learning from $\mathcal{P}_{\mathcal{M}_1}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_0}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_1}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_2}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_3}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_4}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_5}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_6}$
0
0.5
1
0
1000
2000
3000
winning rate
draw rate
loosing rate

Figure 15

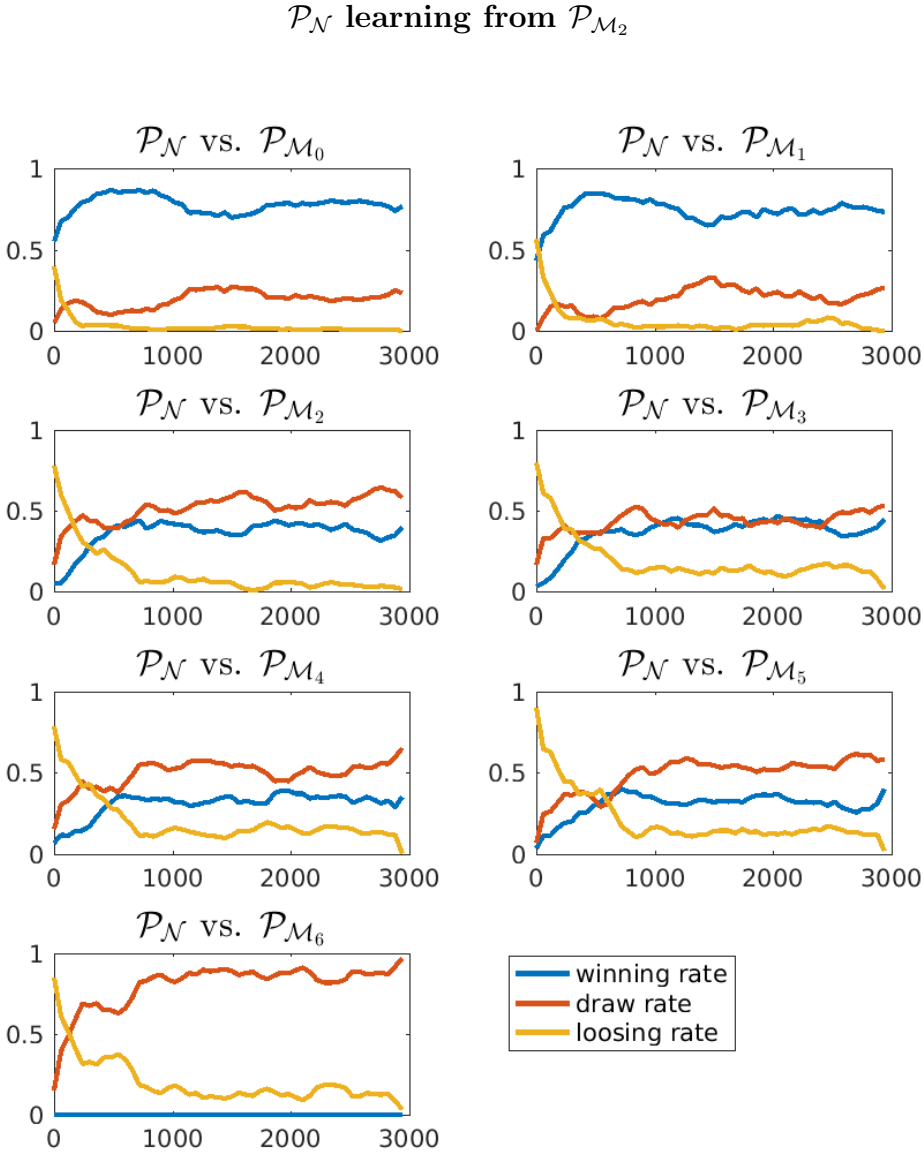

$\mathcal{P}_\mathcal{N}$ learning from $\mathcal{P}_{\mathcal{M}_2}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_0}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_1}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_2}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_3}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_4}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_5}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_6}$
0
0.5
1
0
1000
2000
3000
winning rate
draw rate
loosing rate

Figure 16

$\mathcal{P}_{\mathcal{N}}$ learning from $\mathcal{P}_{\mathcal{M}_3}$

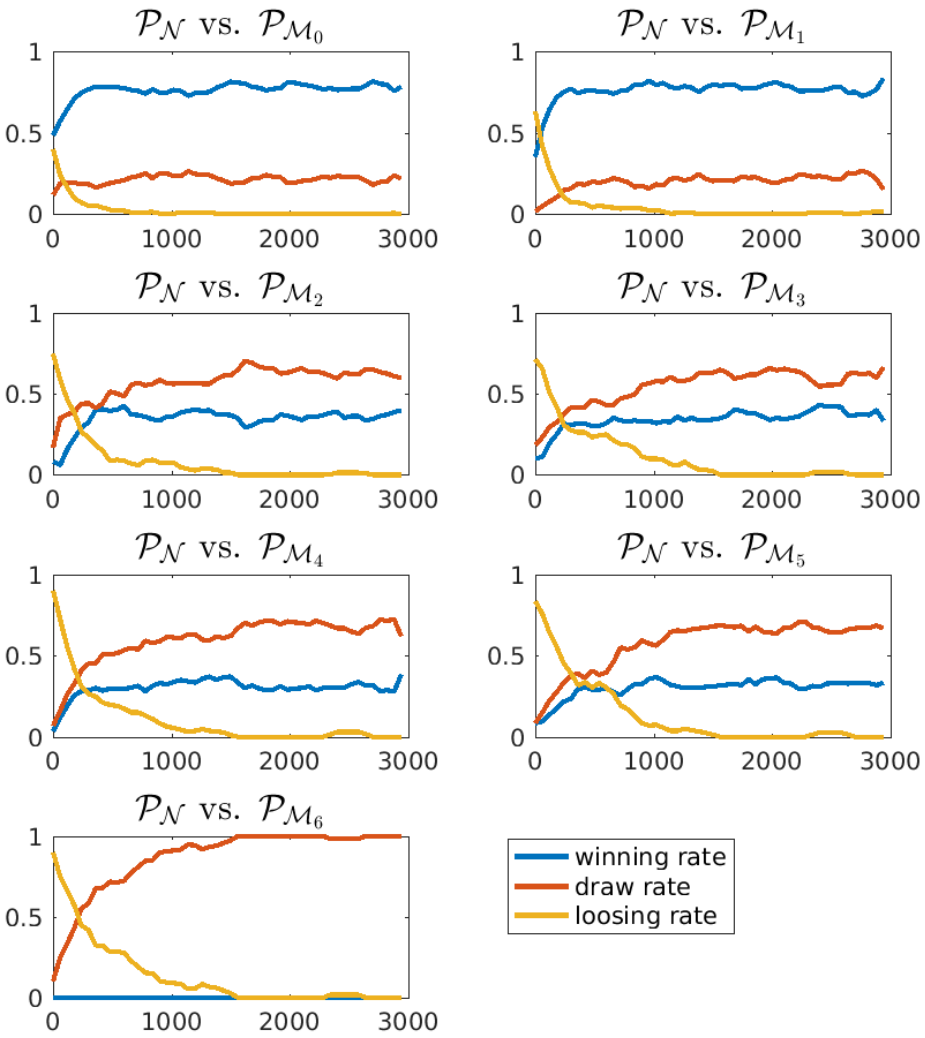


Figure 17

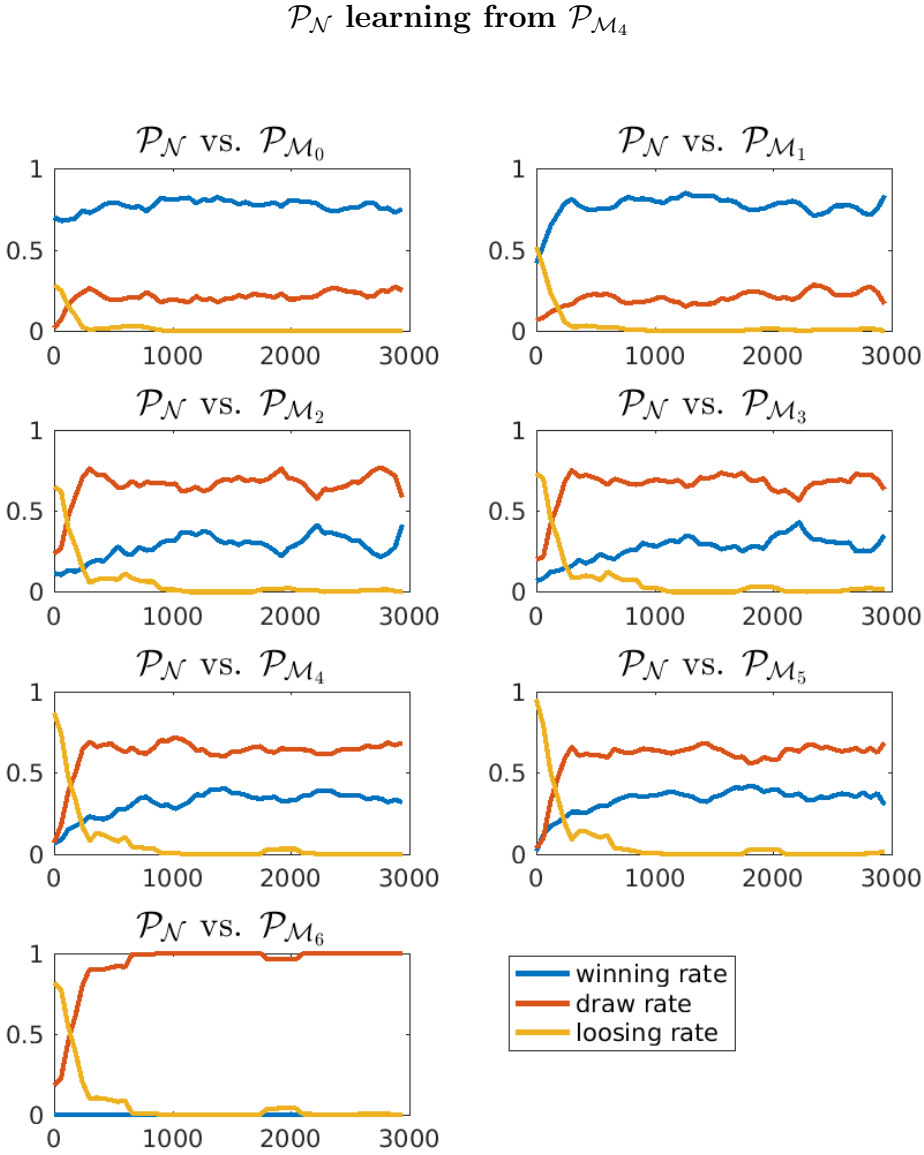

$\mathcal{P}_\mathcal{N}$ learning from $\mathcal{P}_{\mathcal{M}_4}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_0}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_1}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_2}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_3}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_4}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_5}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_6}$
0
0.5
1
0
1000
2000
3000
winning rate
draw rate
loosing rate

Figure 18

$\mathcal{P}_{\mathcal{N}}$ learning from $\mathcal{P}_{\mathcal{M}_5}$

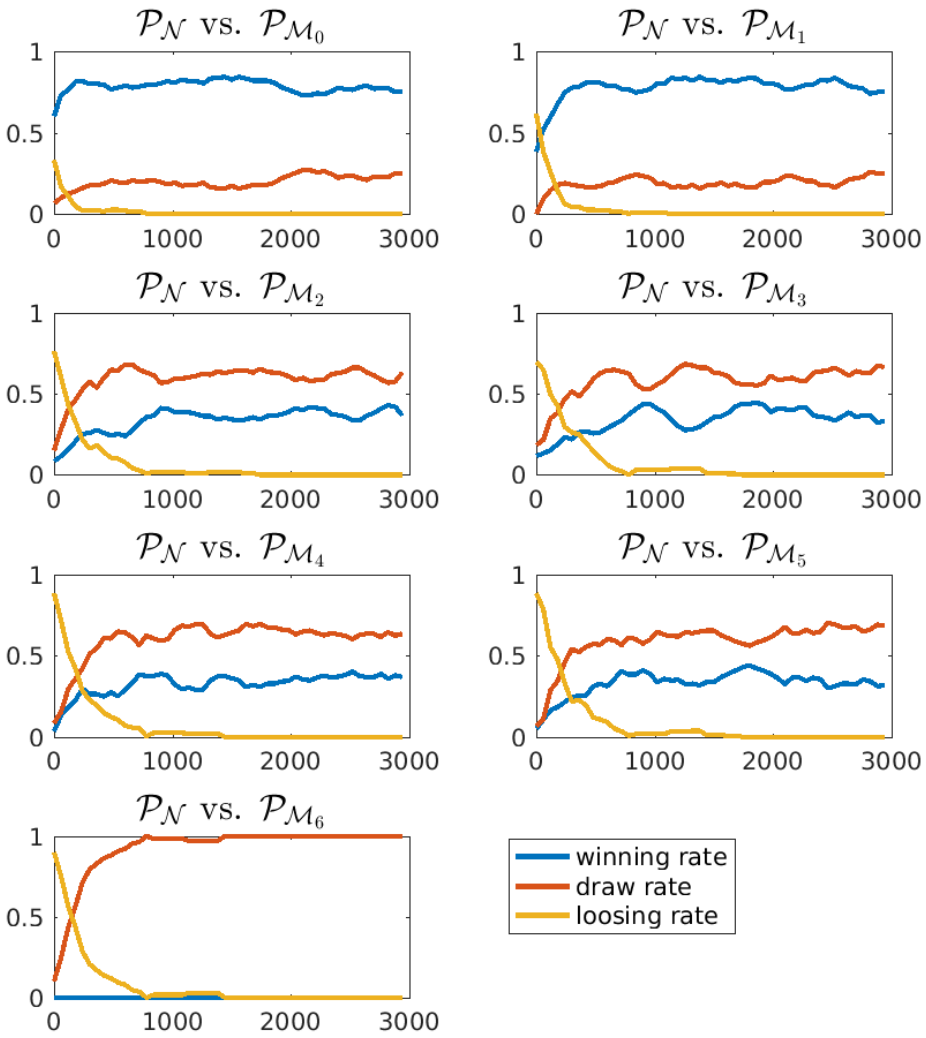


Figure 19

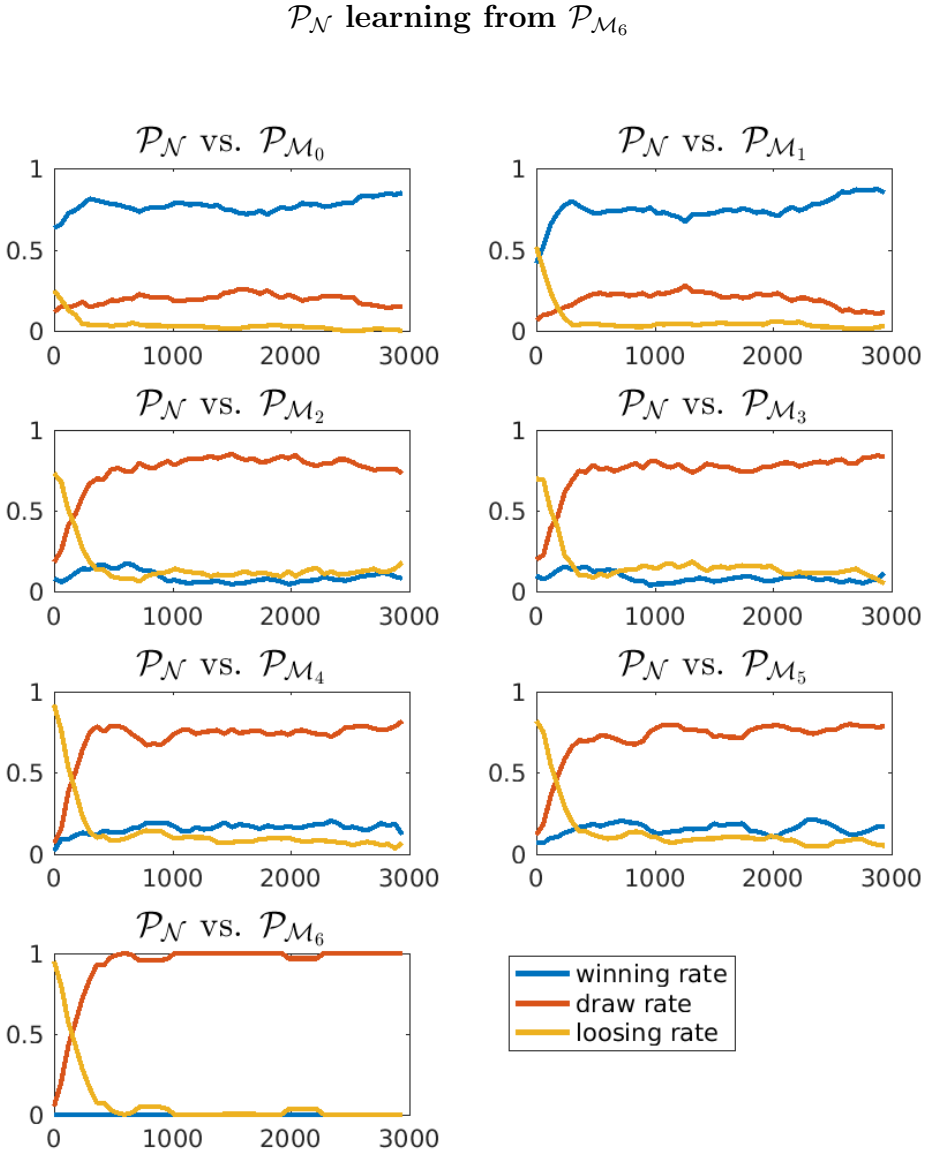

$\mathcal{P}_\mathcal{N}$ learning from $\mathcal{P}_{\mathcal{M}_6}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_0}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_1}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_2}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_3}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_4}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_5}$
$\mathcal{P}_\mathcal{N}$ vs. $\mathcal{P}_{\mathcal{M}_6}$
0
0.5
1
0
1000
2000
3000
winning rate
draw rate
loosing rate

Figure 20

5.2 A Bellman approach

In this chapter we will try another approach of training a neural network that is able to play Tic-Tac-Toe. We will formulate a bellman-like equation for Tic-Tac-Toe and then try to approximate the solution of this equation by a neural network. But first we define reward functions to evaluate single boards.

Definition 15 (Tic-Tac-Toe reward functions)**.** *Let $\mathcal{B}$ be the set of all Tic-Tac-Toe boards. The reward function r^X on the set of boards $\mathcal{B}^X$ is defined by*

$$r^X(B^X) := \begin{cases} 1 & \textit{if Player "X" wins} \\ -1 & \textit{if Player "O" wins} \\ 0 & \textit{otherwise} \end{cases}$$

whereas the reward function defined on the set $\mathcal{B}^O$ is given by

$$r^O(B^O) := \begin{cases} 1 & \textit{if Player "O" wins} \\ -1 & \textit{if Player "X" wins} \\ 0 & \textit{otherwise} \end{cases}.$$

And therefore the reward function on the whole set $\mathcal{B}$ is

$$r(B) = \begin{cases} r^X(B) & \textit{if } B \in \mathcal{B}^X \\ r^O(B) & \textit{if } B \in \mathcal{B}^O \end{cases}.$$

Assuming that each player is performing the move that maximizes its reward (and therefore minimizes the opponent's reward) we can give a bellman-like functional equation which defines a function that returns the value of a board B.

Definition 16 (Value Function)**.** *Let $\mathcal{B}$ be the set of all Tic-Tac-Toe boards and r a reward function. The Value Function is defined as the solution of the functional equation*

$$V(B) = r(B) - \max_{B' \in S(B)} V(B'). \tag{5.4}$$

Clearly, the best move b^* according to V for a player at board B is given by

$$b^*(B) = \underset{b \textit{ legal in } B}{\operatorname{argmax}} V(B + b).$$

Unfortunately, it is not possible to find an explicit solution of (5.4). We will try to approximate a solution in terms of a neural network $V(B, w)$. In case we have a good approximation we can choose the optimal moves based on the neural network outcome by

$$\begin{aligned} b^*(B, w) &:= \underset{b \textit{ legal in } B}{\operatorname{argmax}} V(B, w) \\ &\approx \underset{b \textit{ legal in } B}{\operatorname{argmax}} V(B). \end{aligned}$$

We know that values $V(B, w)$ for $B \in \mathcal{B}$ need to fulfill equation (5.4). Hence, given a set of boards $B_1, \ldots, B_N$ we can define the *bellman-error function*

$$E(B_i, w) := \frac{1}{N} \sum_{i=1}^{N} \|V(B_i, w) - \left[r(B_i) - \max_{B' \in S(B_i)} V(B', w) \right] \|^2 + \mathcal{R}(w), \tag{5.5}$$

where $\mathcal{R}(w)$ is some regularization term.
In order to use stochastic gradient descent for minimizing $E(B_i, w)$ we need to determine the derivative of the error

$$\begin{aligned} \nabla_w E(B_i, w) &= \nabla_w \|V(B_i, w) - \left[r(B_i) - \max_{B' \in S(B_i)} V(B', w) \right] \|^2 \\ &= 2 \left(V(B_i, w) - \left[r(B_i) - \max_{B' \in S(B_i)} V(B_i', w) \right] \right) \\ &\quad \left(\nabla_w V(B_i, w) + \nabla_w \max_{B' \in S(B_i)} V(B', w) \right). \end{aligned} \tag{5.6}$$

In chapter 1.4 we derived an algorithm to calculate the derivative of a neural network with respect to its weights w. We can use this algorithm to determine $\nabla_w V(B_i, w)$. After proving the next lemma we will see how to calculate

$$\nabla_w \max_{B' \in S(B_i)} V(B', w). \tag{5.7}$$

Lemma 4. *Assume $\mathcal{A} = \{a_1, \ldots, a_m\}, a \in \mathcal{A}, w \in \mathbb{R}^m$ and $F : \mathcal{A} \times \mathbb{R}^m \to \mathbb{R}$. If F is smooth in w then*

$$\nabla_w \left[\max_{a \in \mathcal{A}} F(a, w) \right] v = (\nabla_w V)(\operatorname*{argmax}_{a \in \mathcal{A}}, w)v$$

Proof. By smoothness we know that for all $w, v \in \mathbb{R}^m$ there exist a $\epsilon_0 > 0$ such that

$$\begin{aligned} a^* &= \operatorname*{argmax}_{a \in \mathcal{A}} F(a, w) \\ &= \operatorname*{argmax}_{a \in \mathcal{A}} F(a, w + \epsilon v) \end{aligned} \tag{5.8}$$

for all $\epsilon \in (0, \epsilon_0]$.

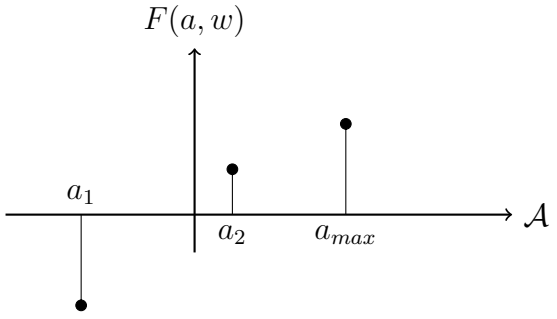


Figure 21: Since $\mathcal{A}$ is a finite set, a_{max} remains the argument of the minimum after small changes of the neural network weights.

From (5.8) follows that

$$\max_{a\in\mathcal{A}} F(a,w) = F(a^*,w)$$

and

$$\max_{a\in\mathcal{A}} F(a,w+\epsilon v) = F(a^*,w+\epsilon v)$$

for all $\epsilon \in (0,\epsilon_0]$.
Hence we have

$$\begin{aligned}\nabla_w \left(max_{a\in\mathcal{A}} F(a,w)\right) v &= \lim_{\epsilon\to 0} \frac{\max_{a\in\mathcal{A}} F(a,w+\epsilon v) - \max_{a\in\mathcal{A}} F(a,w)}{\epsilon} \\ &= \lim_{\epsilon\to 0} \frac{F(a^*,w+\epsilon v) - F(a^*,w)}{\epsilon} \\ &= \left(\nabla_w F\right)(a^*,w)v \\ &= \left(\nabla_w F\right)\left((\operatorname{argmax}_{a\in\mathcal{A}} F(a,w),w\right)v. \quad \square\end{aligned}$$

After applying Lemma 4 to (5.7) we get

$$\nabla_w \max_{B'\in S(B_i)} V(B',w) = \left(\nabla_w V\right)\left(\operatorname{argmax}_{B'\in S(B_i)} V(B',w), w\right). \tag{5.9}$$

Setting $B^* := \operatorname{argmax}_{B'\in S(B_i)} V(B',w)$ and putting (5.9) back in (5.6) we get the result

$$\nabla_w E(B_i,w) = 2\big[V(B_i,w) - r(B_i) + V(B^*,w)\big]\left(\nabla_w V(B_i,w) + \nabla_w V(B^*,w)\right). \tag{5.10}$$

With (5.10) we can apply gradient descent method in order to minimize the bellman error (5.6) of the neural network V.

5.2.1 Learning procedure

Similar to the dynamical learning in chapter 5.1.1, we first have to choose a suitable network architecture. The function we want to approximate has values in $[-1,1] \subset \mathbb{R}$, therefore we choose *tangens hyperbolicus* as activation function for the output and both hidden layers. Experiments in chapter 4.2 showed good generalization properties for networks with two hidden layers of size 60. Therefore we use the same layer sizes as in chapter 4.2.
Before testing a dynamical learning procedure we apply stochastic gradient descent to minimize bellman error (5.6) not only on boards the learning agent is visiting but on the whole set of Tic-Tac-Toe boards $\mathcal{B}$. The training is summarized in figure 22.
After adapting network weights, the network hopefully approximates the solution of equation 5.4. Now we define a player which chooses its move based on the outcome of this neural net.

Definition 17 (Bellman player)**.** *Let $B \in \mathcal{B}$ be the current state of a Tic-Tac-Toe game and V a neural network with weights w approximating the solution of 5.4. A Bellman player $\mathcal{P}_\mathcal{B}$ is a Tic-Tac-Toe player who chooses move b^*, where*

$$b^*(B) = \operatorname*{argmax}_{b \text{ legal in } B} V(B+b,w). \tag{5.11}$$

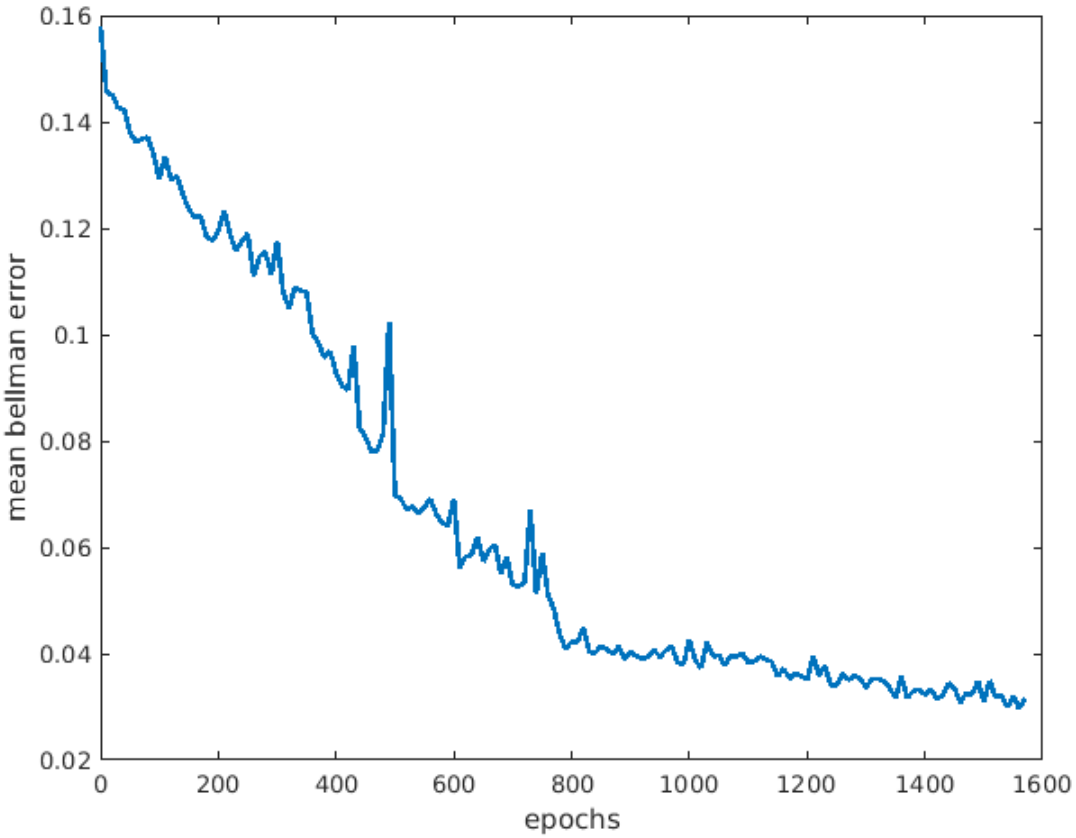


Figure 22: Training of neural network to approximate solution of equation (5.4) with learning rate $\gamma = 0.001$, momentum $\mu = 0.4$ and without regularization. The plot shows the bellman error (5.6) over time.

The trained network is then used to simulate games of a bellman-player against minimax-players. The performance of the network is summarized in figure 23.

The dynamical learning procedure is similar to the one in chapter 5.1.1. We first initialize a neural network with random weights. Then, based on this network a bellman-player is playing several games against a minimax-player. In the next step we train the network based on the boards visited during these games.

Definition 18 (Bellman-learning algorithm)**.** *Given simulation size m, learning error ϵ and a minimax-player $P_{\mathcal{M}_d}$,*

1. *initialize a neural network V with random weights according to* (4.4)*,*
2. *simulate m games of P_B vs. $P_{\mathcal{M}_d}$ and obtain $G_1, \ldots, G_m$ and*
3. *train the neural network V by stochastic gradient descent to minimize bellman-error* (5.6) *with training samples randomly drawn from $G_1, \ldots, G_m$. Stop if $E(w) < \epsilon$ and*
4. *go back to step 2.*

$\mathcal{P}_{\mathcal{B}}$ trained on the set of all possible Tic-Tac-Toe boards.

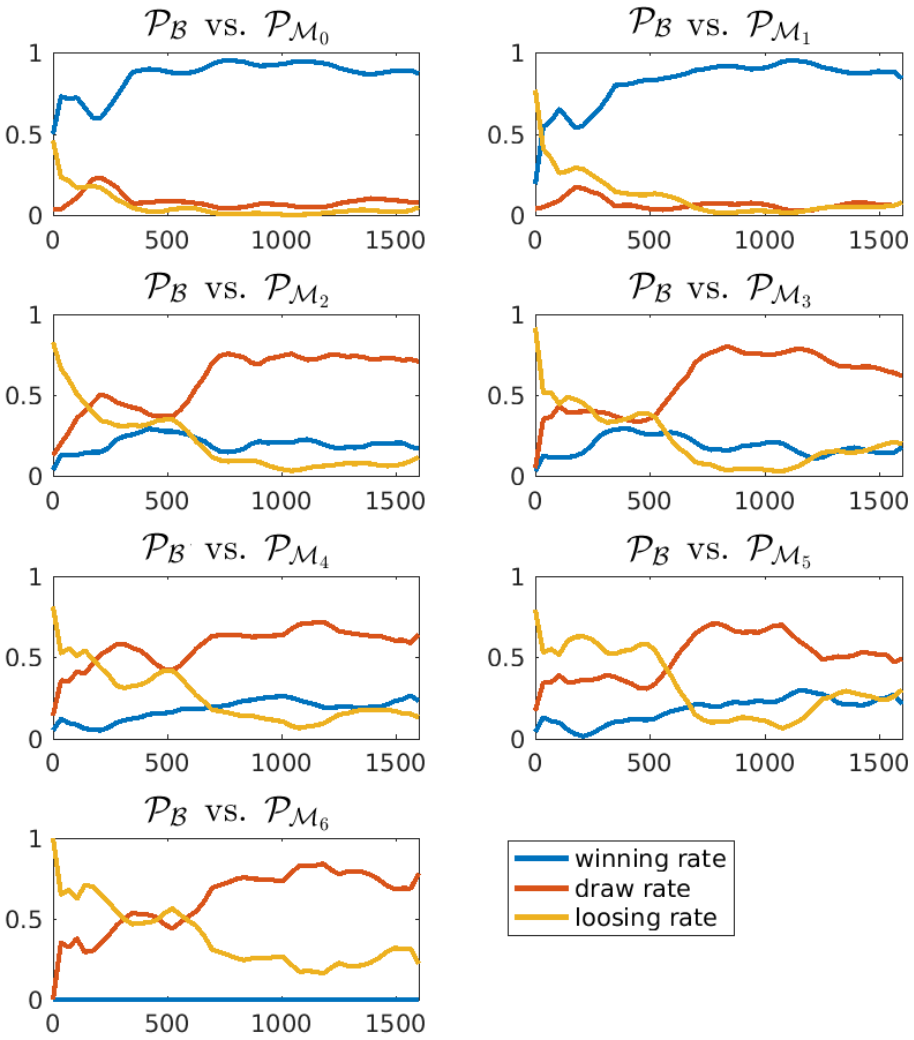


Figure 23: The plot shows the performance of a bellman-player using network V against minimax-players versus the number of epochs the network was trained.

5.2.2 Results

The bellman-learning algorithm was applied to minimax-opponents of intelligence $d = 0, 3, 5$, simulation size $m = 200$ and learning error $\epsilon = 0.05$. If the desired learning error was not obtained after 200 epochs, then the training was stopped as well. The training was done with learning rate $\gamma = 0.001$, momentum $\mu = 0.4$ and without regularization. Results are presented in figure 24 to 26.
We can see that the learning agent outperforms $\mathcal{P}_{\mathcal{M}_0}$ after 200 games and $\mathcal{P}_{\mathcal{M}_1}$ after about 1000 games. But when playing against minimax-players with intelligence greater than 2, the learning agent does not compete very well. The intelligence of the opponent the bellman-player is learning from does not play a major role in the learning behavior. This is because the learning procedure is based only on the boards visited during the games and not on the outcome.

5.3 Conclusion

We saw that the reinforcement-learning approach worked well. Nevertheless, learning against very smart opponents led to bad state exploration whereas training against inexperienced opponents came along with slow learning behavior. For a trade-off of these two effects in further experiments, one could try a setting in which an agent is learning from opponents with random intelligence.

The bellman-learning approach yielded good results when the bellman-error (5.6) was minimized on the set of all possible Tic-Tac-Toe boards. In case the bellman-error was minimized only on boards visited during simulation of games against minimax opponents, the performance of the learning agent increased only very slowly to a low level. Similar to the reinforcement-learning approach the exploration of game states is influencing the learning behavior.

Getting good results in a not very complex game like Tic-Tac-Toe does not mean this method works for other games. Additional investigations could involve applying the learning procedures presented in this thesis to more complex two-player zero-sum games like Connect Four.

$\mathcal{P}_{\mathcal{B}}$ learning from $\mathcal{P}_{\mathcal{M}_0}$

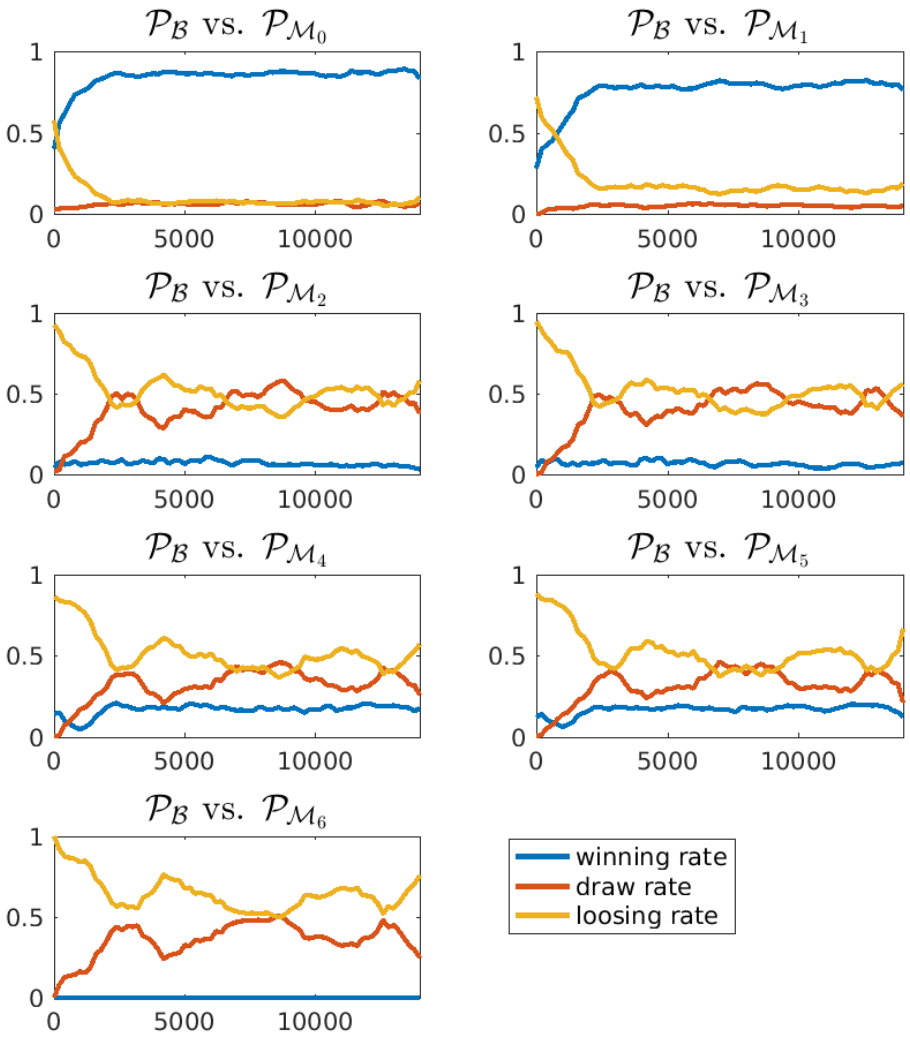


Figure 24

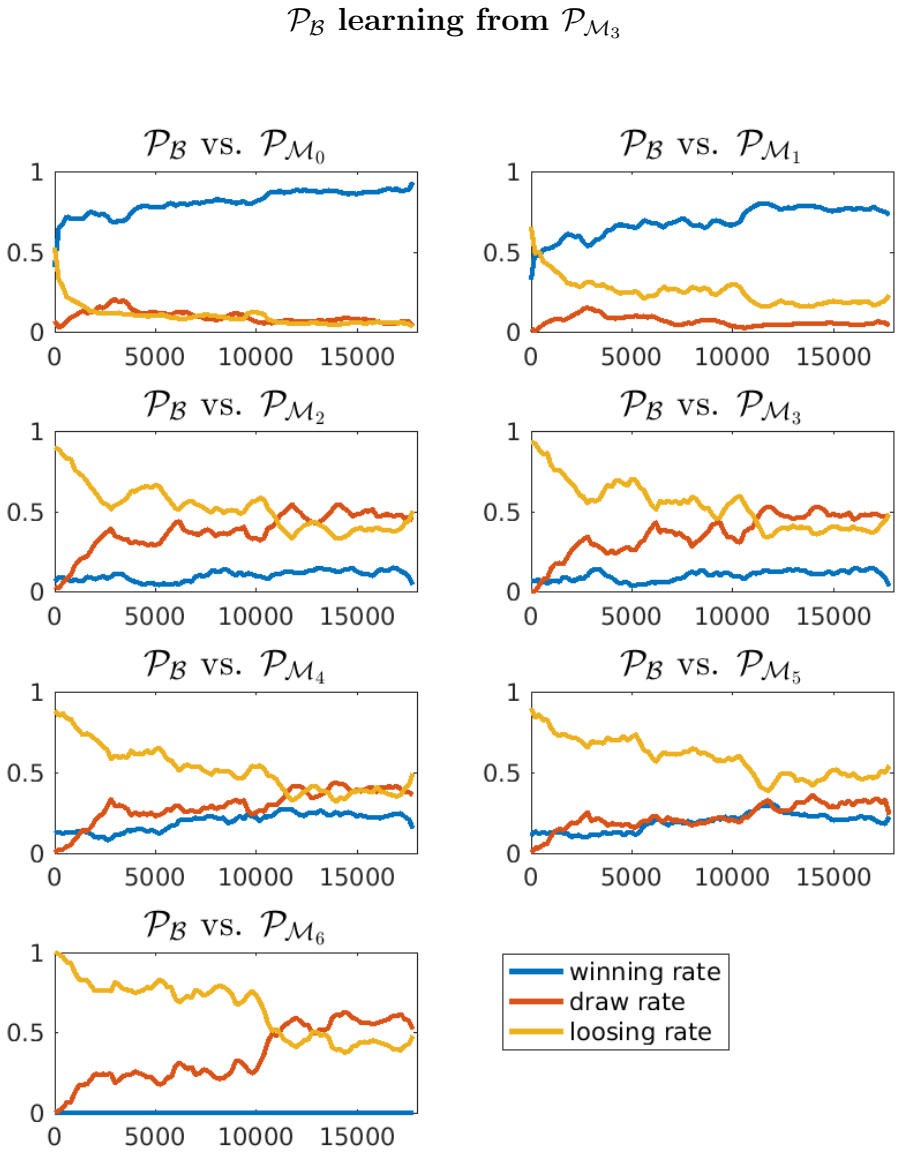

$\mathcal{P}_\mathcal{B}$ learning from $\mathcal{P}_{\mathcal{M}_3}$
$\mathcal{P}_\mathcal{B}$ vs. $\mathcal{P}_{\mathcal{M}_0}$
$\mathcal{P}_\mathcal{B}$ vs. $\mathcal{P}_{\mathcal{M}_1}$
$\mathcal{P}_\mathcal{B}$ vs. $\mathcal{P}_{\mathcal{M}_2}$
$\mathcal{P}_\mathcal{B}$ vs. $\mathcal{P}_{\mathcal{M}_3}$
$\mathcal{P}_\mathcal{B}$ vs. $\mathcal{P}_{\mathcal{M}_4}$
$\mathcal{P}_\mathcal{B}$ vs. $\mathcal{P}_{\mathcal{M}_5}$
$\mathcal{P}_\mathcal{B}$ vs. $\mathcal{P}_{\mathcal{M}_6}$
1
0.5
0
0
5000
10000
15000
winning rate
draw rate
loosing rate

Figure 25

$\mathcal{P}_{\mathcal{B}}$ learning from $\mathcal{P}_{\mathcal{M}_6}$

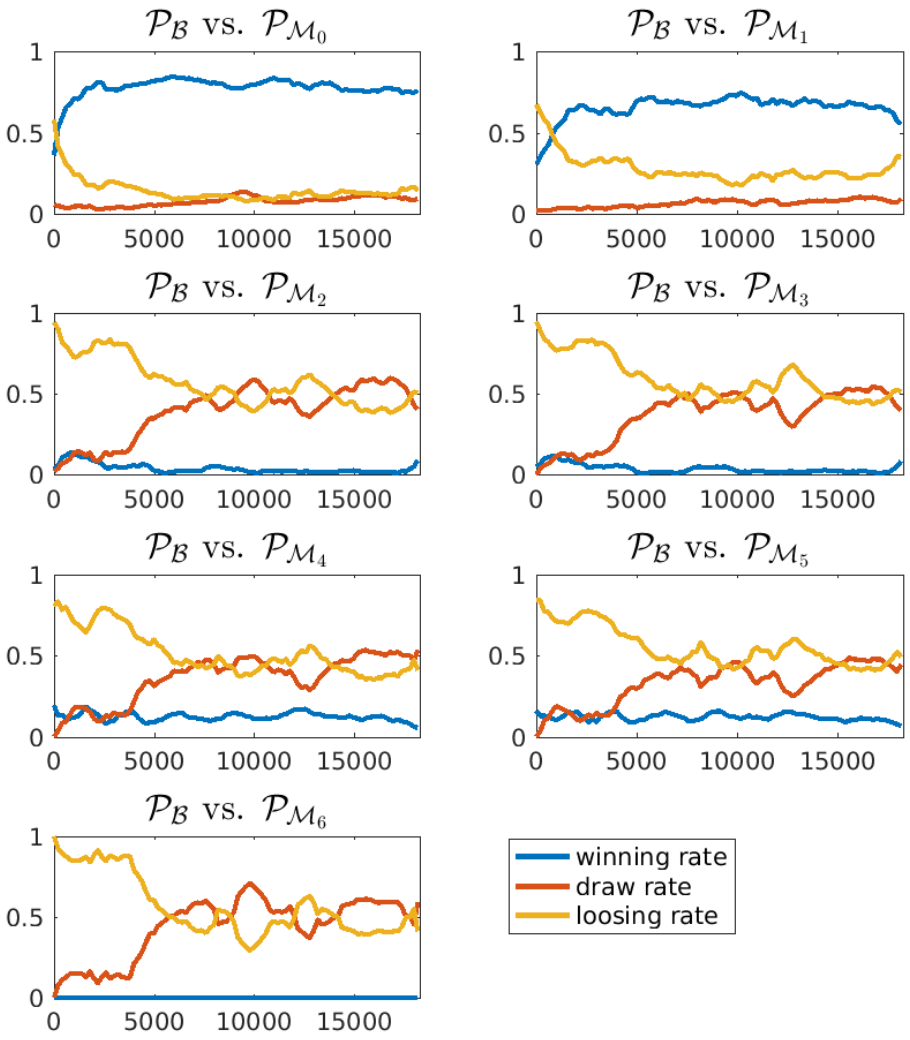


Figure 26

References

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.

[Bot12] Léon Bottou. *Stochastic Gradient Descent Tricks*, pages 421–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[KvdS96] Ben Kröse and Patrick van der Smagt. An introduction to neural networks, 1996.

[Nie15] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[NWS14] Deanna Needell, Rachel Ward, and Nati Srebro. Stochastic gradient descent, weighted sampling, and the randomized kaczmarz algorithm. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1017–1025. Curran Associates, Inc., 2014.

[RN09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[Tap11] Kristopher Tapp. *Symmetry: A Mathematical Exploration*. Springer, 2012 edition, 2011.

[Wik17] Wikipedia. Game complexity — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Game%20complexity&oldid=774441613`, 2017. [Online; accessed 26-April-2017].