Max-Planck-Gymnasium Bielefeld

Klasse 11/Q1
Schuljahr 17/18
2. Halbjahr

Facharbeit

im Grundkurs Informatik

Entwicklung einer Künstlichen Intelligenz (KI) für Tic-Tac-Toe auf der Basis eines Neuronalen Netzes mit evolutionärem Lern-Algorithmus

Verfasser: Jonas Keller
Kursleiter: Herr Gehrke
Bearbeitungszeit: 06.02.2018 bis 26.03.2018
Abgabetermin: 26.03.2018

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit soll anhand des einfachen Strategiespiels Tic-Tac-Toe das evolutionäre Lernen einer künstlichen Intelligenz (KI) auf der Basis eines künstlichen Neuronalen Netzes (KNN) unter Verwendung der Programmiersprache Java beschrieben werden.

1.1 Was ist Tic-Tac-Toe?

Tic-Tac-Toe ist ein einfaches Zweipersonen-Strategiespiel. Zwei Spieler setzen abwechselnd ihr Symbol (Kreis oder Kreuz) auf ein 3x3 Felder großes Spielfeld in ein freies Feld. Der Spieler, der sein Symbol als erstes in drei Felder senkrecht, waagerecht oder diagonal in einer Reihe gesetzt hat, gewinnt. Wenn alle neun Felder gefüllt sind, ohne dass ein Spieler gewonnen hat, geht das Spiel unentschieden aus. [1]

1.2 Was ist ein neuronales Netz?

1.2.1 Biologisches Vorbild

Da KNNs ein Vorbild in der Natur haben, wird nachfolgend kurz auf die biologische Nervenzelle und ein biologisches Neuronales Netz eingegangen. [2] [3]

Alle Tiere verfügen über Biologische Neuronale Netze, die aus miteinander z.T. sehr komplex verbundenen Nervenzellen (Neuronen) bestehen.[4] Jedes Neuron besteht aus einem Zellkörper, einem Axon, vielen Dendriten und Synapsen.[5]

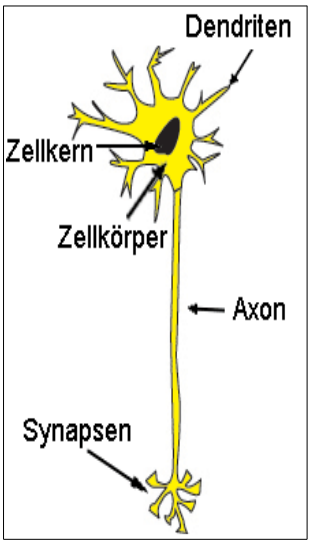


Abb. 1: Schematischer Aufbau einer Biologischen Nervenzelle[5]

Die Dendriten sammeln die (elektrischen) Signale von anderen Neuronen und leiten sie zum Zellkörper[6]. Wird dort ein bestimmter Schwellenwert überschritten, „feuert“ die Nervenzelle, d. h., ein elektrischer Impuls wird über das Axon an weitere Neuronen weitergeleitet. Die Stelle, an denen das Axon

1 Wikipedia: Tic-Tac-Toe, https://de.wikipedia.org/wiki/Tic-Tac-Toe (Abgerufen am: 24.02.2018)
2 Aufbau künstlicher Neuronen, http://www2.chemie.uni-erlangen.de/projects/vsc/chemoinformatik/erlangen/datenanalyse/nn_kneuron.html (Abgerufen am: 24.02.2018)
3 Kristian, Alex, Künstliche neuronale Netze in C#, http://www.codeplanet.eu/tutorials/csharp/70-kuenstliche-neuronale-netze-in-csharp.html (Abgerufen am: 24.03.2018)
4 Rigoll, Gerhard, 1994, Neuronale Netze, Expert Verlag, Renningen-Malmsheim, S. 32
5 Girwidz R.,LMU München, Das Neuron, http://www.didaktikonline.physik.uni-muenchen.de/piko/material/medizintechnik/medizintechnik/neuron1/neuron.html (Abgerufen am: 25.03.2018)
6 Kristian, Alex, Künstliche neuronale Netze in C#, http://www.codeplanet.eu/tutorials/csharp/70-kuenstliche-neuronale-netze-in-csharp.html (Abgerufen am: 24.03.2018)

eines Neurons mit den Dendriten eines weiteren Neurons verbunden ist, wird Synapse genannt. Über diese Synapsen werden die Neuronen miteinander vernetzt. Diese Synapsen fungieren als Schalter, die die Intensität eines Signals beim Weiterleiten ändern. Das Ausmaß dieser Änderung hängt von der Synapsenstärke ab. Diese Stärke ist variabel, was es einem Netz aus Nervenzellen, also z.B. einem Gehirn, möglich macht, zu lernen. Diese Fähigkeit zu lernen kann mithilfe eines mathematischen Modells nachgebildet werden, das nachfolgend beschrieben wird. [7]

1.2.2 Aufbau eines künstlichen Neurons

Wie sein biologisches Vorbild besteht auch ein KNN aus Neuronen, die miteinander verbunden sind [8] [9]:

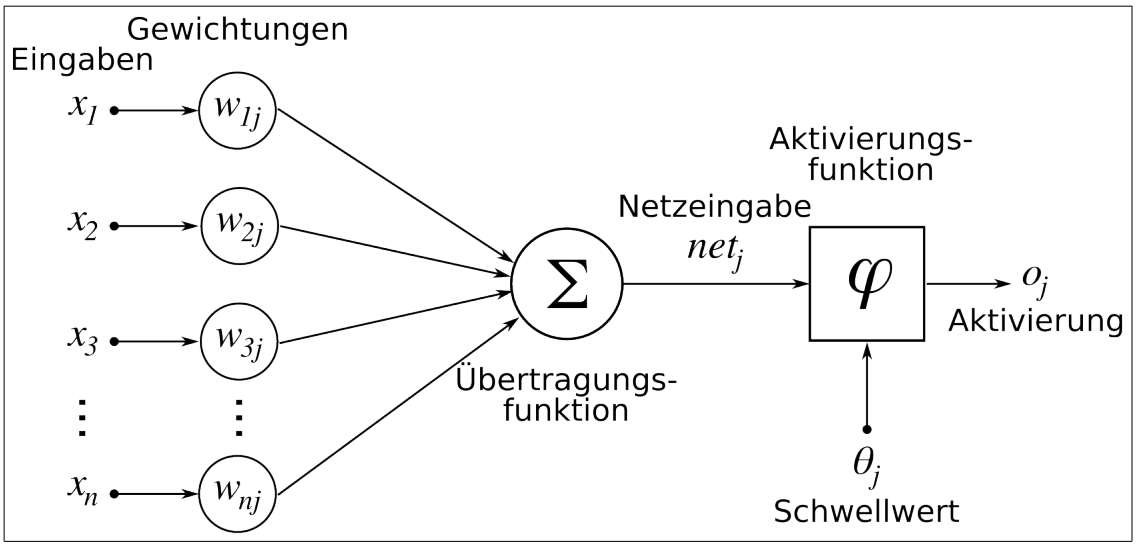


Abb. 2: Schematische Darstellung eines künstlichen Neurons[10]

In Abbildung 2 ist ein Schema eines künstlichen Neurons dargestellt. In jedem Neuron werden die Eingänge $x_1 \dots x_n$, mit den Gewichtungen $w_{1j} \dots w_{nj}$ multipliziert (gewichtet) und in der Übertragungsfunktion aufaddiert, wobei j den Index des betrachteten Neurons und n den Index der Eingabe darstellt. Dieser Wert wird dann Netzeingabe net_j genannt:

$$net_j = \sum_{i=1}^{n} x_n w_{nj}$$

7 Aufbau künstlicher Neuronen, http://www2.chemie.uni-erlangen.de/projects/vsc/chemoinformatik/erlangen/datenanalyse/nn_kneuron.html (Abgerufen am: 24.02.2018)
8 Rigoll, Gerhard, 1994, Neuronale Netze, Expert Verlag, Renningen-Malmsheim, S. 42ff
9 10.12: Neural Networks: Feedforward Algorithm Part 1 - The Nature of Code, https://www.youtube.com/watch?v=qWK7yW8oS0I (Abgerufen am: 18.03.2018)
10 Wikipedia: Künstliches neuronales Netz, https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz (Abgerufen am: 18.03.2018)

Die Netzeingabe wird dann in die Aktivierungsfunktion $\varphi(x)$, die im Prinzip der Kern des künstlichen Neurons ist, gegeben. Als Aktivierungsfunktion wird häufig die Sigmoidfunktion verwendet:

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

Sie staucht die Eingangswerte auf das Intervall $[0;1]$. In dieser Arbeit wird allerdings eine andere Aktivierungsfunktion verwendet. Diese als Softsign bezeichnete Funktion wird im Abschnitt 2.2.3 behandelt.

Wie das Neuron bis jetzt beschrieben wurde, kann es jedoch noch nicht zu einem speziellen Verhalten bei einem 0-Input gebracht werden. Durch das Anpassen der Gewichte kann die Netzeingabe net_j nicht verändert werden. Deshalb wird zu der Netzeingabe noch der Schwellenwert θ_j (in Abbildung 2 als „Schwellwert“ bezeichnet und manchmal auch Bias genannt) aufaddiert. Er sorgt dafür, dass dem Neuron auch für einen 0-Input ein bestimmtes Verhalten antrainiert werden kann. Der Schwellenwert ist auch als y-Achsenabschnitt vorstellbar. Daraus ergibt sich dann für die Aktivierung o_j :

$$o_j=\varphi\left(\sum_{i=1}^{n} x_i w_{ij}+\theta_i\right)$$

1.2.3 Aufbau eines Feedforward-Netzes

Ein KNN kann verschiedene Architekturen haben. In dieser Arbeit werden lediglich sogenannte Feedforward-Netze behandelt. Diese sind folgendermaßen aufgebaut[11] [12]:

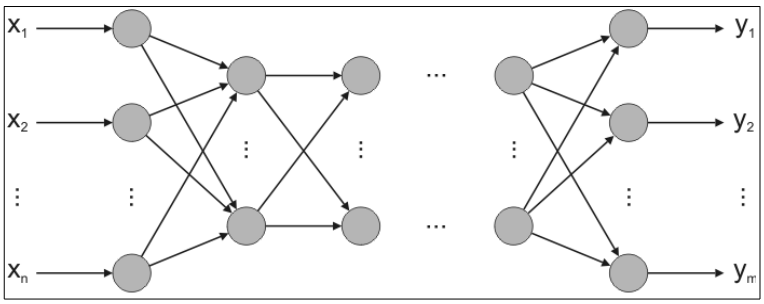


Abb. 3: Struktur eines Feedforward-Netzes[13]

Ein Feedforward-Netz ist in Schichten aufgebaut, die jeweils eine bestimmte Anzahl an Neuronen beinhalten, wie schematisch in Abbildung 3 dargestellt ist.

11 Rigoll, Gerhard, 1994, Neuronale Netze, Expert Verlag, Renningen-Malmsheim, S. 47
12 10.12: Neural Networks: Feedforward Algorithm Part 1 - The Nature of Code, https://www.youtube.com/watch?v=qWK7yW8oS0I (Abgerufen am: 18.03.2018)
13 Künstliche Neuronale Netze, http://pi.informatik.uni-siegen.de/Arbeitsgebiete/ci/knn/ (Abgerufen am: 18.03.2018)

Ein Kreis symbolisiert hier ein Neuron, so wie es in Abschnitt 1.2.2 (Abb. 2) beschrieben ist. Alle Schichten sind vollständig mit der jeweils nächsten Schicht verbunden, d. h., dass jedes Neuron seine Ausgabe (Output) an jedes Neuron der nächsten Schicht weiter gibt (in Abbildung 3 durch Pfeile repräsentiert), in der der Output dann als Eingabe (Input) verarbeitet wird, wie in Abschnitt 1.2.2 beschrieben.
Es gibt drei Arten von Schichten (Layer): die Eingabe-Schicht (Input-Layer), die versteckten Schichten (Hidden-Layer) und die Ausgabe-Schicht (Output-Layer) [14]. Jedes Feedforward-Netz hat einen Input-Layer, einen Output-Layer und dazwischen beliebig viele Hidden-Layer. Der Input-Layer ist im Grunde nur dazu da, die Inputs an die nächste Schicht zu verteilen, hier finden keine Berechnungen statt. Die Input-Werte werden dann Schicht für Schicht immer weiter durch das KNN „geschoben“, bis sie am Output-Layer angekommen sind. Daher auch der Name „Feedforward-Netz“.

1.2.4 Matrixschreibweise des Feedforward-Netzes

Um einfacher und effizienter mit KNNs arbeiten zu können, lassen sich die Gewichtungen eines KNNs als Matrizen zwischen einer aktuellen Schicht und einer Schicht davor darstellen [15]. Wenn die Neuronen der Schicht davor mit n indexiert werden und die der aktuellen Schicht mit j , lässt sich folgende Gewichtsmatrix aufstellen:

$$\begin{bmatrix} w_{11} & w_{21} & \cdots & w_{n1} \\ w_{12} & & & w_{n2} \\ \vdots & & & \vdots \\ w_{1j} & w_{2j} & \cdots & w_{nj} \end{bmatrix}$$

Wenn jetzt die Inputs, die Schwellenwerte und die Outputs der aktuellen Schicht mit Ein-Spalten-Matrizen bzw. Vektoren dargestellt werden, kann folgende Formel für den Output-Vektor der aktuellen Schicht aufgestellt werden[16] [17]:

$$\begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_j \end{bmatrix} = \varphi\left(\begin{bmatrix} w_{11} & w_{21} & \cdots & w_{n1} \\ w_{12} & & & w_{n2} \\ \vdots & & & \vdots \\ w_{1j} & w_{2j} & \cdots & w_{nj} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_j \end{bmatrix} \right)$$

14 Rigoll, Gerhard, 1994, Neuronale Netze, Expert Verlag, Renningen-Malmsheim, S. 41
15 Rigoll, Gerhard, 1994, Neuronale Netze, Expert Verlag, Renningen-Malmsheim, S. 59
16 Rigoll, Gerhard, 1994, Neuronale Netze, Expert Verlag, Renningen-Malmsheim, S. 57-60
17 10.12: Neural Networks: Feedforward Algorithm Part 1 - The Nature of Code, https://www.youtube.com/watch?v=qWK7yW8oS0I (Abgerufen am: 18.03.2018)

1.2.5 Implementation des KNNs

Das KNN und auch der restliche Code für diese Arbeit wurde in Java geschrieben und lässt sich dokumentiert im Anhang und auf Github[18] finden. Da diese Arbeit aber zu umfangreich ausfiele, wenn die Implementation eines KNNs im Detail erläutert werden würde, wird hier darauf verzichtet. Der dokumentierte Code kann im Anhang und auf Github nachgelesen werden.

1.2.6 Lernmethoden

Um KNNs zu trainieren, gibt es verschiedene Lernmethoden. In dieser Arbeit wird nur das evolutionäre Lernen betrachtet, das in Abschnitt 2.2.1 genauer erläutert wird.

1.3 Aufgabenstellung / Motivation

Es gibt bereits „unbesiegbare“ effiziente KIs für Tic-Tac-Toe[19]. Daher stellt sich die Frage, weshalb in dieser Arbeit eine Künstliche Intelligenz (KI) auf Basis eines Neuronalen Netzes entwickelt werden soll. Das Ziel dieser Arbeit war es nicht, lediglich eine weitere KI für Tic-Tac-Toe zu entwickeln. Diese Arbeit hat zwei andere Ziele:
Zum Einen soll eine KI für Tic-Tac-Toe entwickelt werden, die von einem menschlichen Gegner (Spieler) noch besiegt werden kann. Der menschliche Spieler soll noch Spaß am Spielen haben und auch gegen einen Computer (mit KI) gewinnen können. Sie soll eher eine KI werden, die eventuell sogar „menschliche“ Fehler macht. Zum Anderen soll anhand von Tic-Tac-Toe gezeigt werden, wie das Lernen von KNNs mithilfe eines evolutionären Algorithmus funktioniert. Damit soll es möglich sein, KNNs auch für Probleme zu verwenden, für die es noch keine Lösung gibt und für die andere Ansätze zum Lernen von KNNs nicht funktionieren.

2 Durchführung

Um eine KI zu entwickeln, die Tic-Tac-Toe spielen soll, reicht schon ein sehr kleines Programm, denn Tic-Tac-Toe ist nicht sonderlich kompliziert, wie in Abschnitt 1.1 bereits beschrieben wurde. Trotzdem wird auf die Implementation nicht genauer eingegangen wird, um diese Arbeit nicht zu umfangreich zu gestalten.

18 https://github.com/Jonicho/TicTacToeKI
19 Abdolsaheb, Ahmad, How to make your Tic Tac Toe game unbeatable by using the minimax algorithm, https://medium.freecodecamp.org/how-to-make-your-tic-tac-toe-game-unbeatable-by-using-the-minimax-algorithm-9d690bad4b37 (Abgerufen am: 25.03.2018)

2.1 Verbindung von Tic-Tac-Toe und einem KNN

Damit das KNN Tic-Tac-Toe spielen kann, muss zwischen Spiel und KNN eine Verbindung bestehen. Bei einem menschlichen Spieler wäre diese Verbindung die GUI („graphical user interface“, englisch für „graphische Benutzeroberfläche“), bei der der Mensch über einen Bildschirm das Spielfeld sehen kann und mit der Maus oder der Tastatur seinen nächsten Zug an den Computer übermittelt. In dem Programm dieser Arbeit wird der Einfachheit halber die Konsole als „GUI“ genutzt.

Für diese Verbindung wird die abstrakte Klasse `Player` verwendet, welche im Folgenden näher betrachtet wird. Sie ist im Code folgendermaßen notiert:

```
public abstract class Player {
	public abstract void init(boolean firstPlayer);
	public abstract int[] turn(int[][] field);
	public abstract void finish(int winPlayer, int[][] field);
}
```

Am Anfang eines jeden Spiels wird die Methode `init` aufgerufen, dessen Parameter `firstPlayer` darüber informiert, ob ein `Player`-Objekt der erste oder zweite Spieler ist. Außerdem können hier Initialisierungen von internen Variablen vorgenommen werden. In jedem Zug wird die Methode `turn` aufgerufen und ihr ein 2-dimensionales Array übergeben, welches das aktuelle Spielfeld darstellt. Wie genau das Spielfeld mithilfe dieses Arrays dargestellt wird, wird in Abschnitt 2.1.1 genauer beschrieben. Die Methode `turn` soll dann ein Array zurückgeben, in welchem die x und y Koordinate des Feldes gespeichert ist, auf das das eigene Symbol gesetzt werden soll. Am Ende eines Spiels wird die Methode `finish` auf beiden `Player`-Objekten aufgerufen, die die Nummer des Spielers, der gewonnen hat, und noch einmal das Spielfeld übergibt. Falls keiner der beiden Spieler gewonnen hat, das Spiel also mit einem Unentschieden ausging, wird als Nummer des Spielers, der gewonnen hat, 0 übergeben.

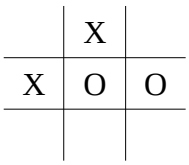
2.1.1 Codierung des Spielfeldes

Damit das KNN Tic-Tac-Toe spielen kann, muss es „wissen“, wie das Spielfeld aussieht. Das Spielfeld hat 9 Felder, die entweder leer sind, mit dem eigenen Symbol (im Beispiel unten X) oder mit dem Symbol des Gegners (im Beispiel unten O) belegt sein können. Um diese Informationen in das KNN zu geben, müssen sie in Zahlen repräsentiert werden. Das geschieht wie folgt:

- Leeres Feld: 0
- Feld mit gegnerischem Symbol: -1
- Feld mit eigenem Symbol: 1

Beispiel:

Feld:



Input-Array:

[0, 1, 0, 1, -1, -1, 0, 0, 0]

2.1.2 Interpretation des Netzwerk-Outputs

Das KNN hat 9 Outputs. Jeder Output entspricht einem Feld des Spielfeldes. Auf das Feld, dessen Output am höchsten ist, wird das Symbol des KNNs gesetzt. Was passiert, wenn dieses Feld schon besetzt ist, das KNN also einen illegalen Zug macht, wird in Abschnitt 2.2.2 behandelt.

2.2 Lernen des KNNs

Wenn man das KNN jetzt spielen lassen würde, wäre das Ziel dieser Arbeit noch nicht erreicht. Denn das KNN würde komplett zufällige Züge machen, da es zufällig initialisiert wurde. Das KNN muss erst einmal lernen, wie Tic-Tac-Toe gespielt und wie gewonnen wird. Dazu wird in dieser Arbeit das evolutionäre Lernen verwendet:

2.2.1 Evolutionäres Lernen

2.2.1.1 Vorbild in der Natur

Wie der Name schon andeutet, hat das evolutionäre Lernen die Evolution von Organismen (Lebewesen) als Vorbild, in der hin und wieder zufällige Veränderungen (Mutationen) passieren. Diese Mutationen können sowohl positive als auch negative Auswirkungen für das entsprechende Lebewesen haben. Hat die Mutation positive Auswirkungen, überlebt das Lebewesen eventuell besser und/oder kann mehr Nachkommen erzeugen. Als Ergebnis setzt sich diese Mutation durch. Dieses Prinzip ist auch als natürliche Selektion oder „survival of the fittest“ bekannt [20].

2.2.1.2 Funktionsweise

Das evolutionäre Lernen startet mit einer bestimmten Anzahl von KNNs, die am Anfang noch zufällig initialisiert sind. Um zu vergleichen, wie gut bzw. schlecht die KNNs ihre Aufgabe erfüllen, hat jedes KNN einen Wert (Score), welcher angibt, wie gut das KNN ist. Um dieses zu messen, wird das KNN getestet und ihm anhand einiger vom Anwendungsfall abhängigen Faktoren ein Score gegeben. Am Beispiel Tic-Tac-Toe

20 Wikipedia: Evolution, https://de.wikipedia.org/wiki/Evolution, (Abgerufen am: 25.03.2018)

wären diese Faktoren beispielsweise, wie oft das KNN gewinnt oder verliert. Nachdem der Test dann an allen KNNs einmal durchgeführt wurde, werden sie anhand des Scores sortiert.
Die KNNs, die am schlechtesten abgeschnitten haben, also den geringsten Score haben, werden jetzt aussortiert, also gelöscht. Sie haben dann nicht „überlebt“.
Danach kommt der wichtigste Teil des evolutionären Lernens, nämlich die Mutation. Für die frei gewordenen Plätze wird jetzt jeweils ein zufälliges KNN aus den übrigen, also besten KNNs ausgewählt, kopiert und dann leicht zufällig verändert, also mutiert.
Anschließend werden wieder alle KNNs getestet und dieser Durchgang (im Folgenden Iteration genannt) wiederholt sich so lange bis die KNNs zufriedenstellende Scores erreicht haben. Falls der Score absolut ist, er sich also bei einem erneuten Testen des selben KNNs nicht ändert, kann in der folgenden Iteration auf das erneute Testen der KNNs, die nicht verändert wurden, verzichtet werden, um die Performance zu steigern.

2.2.1.3 Implementation in Java

Um das evolutionäre Lernen zu implementieren, wurde für diese Arbeit die Klasse `EvolutionalTrainer` erstellt. Sie hat folgende Attribute:

```
private ArrayList<EvolutionalNeuralNetwork> networks;
private int keepAmount;
private double mutationRate;
private double lastHighscore;
```

Das Attribut `networks` beinhaltet eine Liste an Objekten vom Typ `EvolutionalNeuralNetwork`, der später noch genauer beschrieben wird. Das Attribut `keepAmount` gibt die Anzahl der KNNs an, die nach der Sortierung nicht gelöscht werden und damit mit der Länge der Liste `networks` implizit die Anzahl der KNNs die anhand eines schlechten Scores gelöscht werden. Das Attribut `mutationRate` gibt an, wie stark die KNNs mutiert werden sollen. Außerdem gibt das Attribut `lastHighscore` den höchsten Score der letzten Iteration an.
Um einem KNN einen Score zuordnen zu können, hat die Klasse `EvolutionalTrainer` eine innere Klasse `EvolutionalNeuralNetwork`. Diese hat die folgenden Attribute:

```
private final NeuralNetwork neuralNetwork;
private double score;
private boolean tested = false;
```

Die ersten beiden Attribute sind selbsterklärend, das dritte Attribut `tested` hält fest, ob das KNN bereits getestet und ihm ein Score gegeben wurde, um es nicht unnötig noch einmal testen zu müssen, wie bereits in 2.2.1.2 beschrieben.

Außerdem hat diese Klasse noch die Methode `mutate`, die den Parameter `mutationRate` entgegennimmt. `mutationRate` stellt die Mutationsrate m dar, welche genutzt wird, um die Zufallszahl r zufällig aus dem Intervall $[-m;m]$ zu generieren. Die Zufallszahl r wird anschließend mit dem ursprünglichen Wert x_{alt} des Gewichts bzw. des Schwellenwerts multipliziert, um bei kleinen Werten eine zu große Veränderung und bei großen Werten eine zu kleine Veränderung zu verhindern. Dieses Produkt wird dann auf x_{alt} aufaddiert, was für diese Summe x_{neu} folgende Formel ergibt:

$$x_{neu} = x_{alt} + (r * x_{alt}) \quad | \quad r \stackrel{\text{def}}{=} [-m;m]$$

Die Klasse `EvolutionalTrainer` besitzt des Weiteren auch einen Konstruktor, der die folgenden Parameter entgegennimmt:

```
NeuralNetwork seedNetwork,
int networkAmount,
int keepAmount,
boolean randomize
```

Der Parameter `seedNetwork` stellt das KNN dar, welches als Vorlage für die KNNs dient, deren Menge der Parameter `networkAmount` festlegt. Der Wert des Parameters `keepAmount` wird dem oben bereits erläuterten Attribut `keepAmount` zugewiesen. Mit dem Parameter `randomize` kann festgelegt werden, ob die Gewichte und Schwellenwerte der KNNs am Anfang randomisiert werden sollen. Wenn ein KNN von Grund auf trainiert werden soll, ergibt es Sinn, hier **`true`** zu übergeben, um mit zufälligen Werten zu beginnen. Soll hingegen ein schon bestehendes KNN weiter trainiert oder verbessert werden, sollte **`false`** übergeben werden, damit die Gewichte und Schwellenwerte des KNNs nicht überschrieben werden.

Der wichtigste Teil der Klasse `EvolutionalTrainer` ist die Methode `doIteration`, die eine Iteration durchführt und folgendermaßen aussieht:

```
public void doIteration(NeuralNetworkTester nnt, boolean useMultiThreading) {
	Thread[] threads = new Thread[networks.size()];
	for (int i = 0; i < networks.size(); i++) {
		if (!networks.get(i).tested) {
			int index = i;
			NeuralNetwork nn = networks.get(index)
					.getNeuralNetwork();
			threads[i] = new Thread(() -> {
				networks.get(index).setScore(nnt.test(nn));
			});
			if (useMultiThreading) {
				threads[i].start();
			} else {
				threads[i].run();
			}
		}
	}
	if (useMultiThreading) {
		for (int i = 0; i < threads.length; i++) {
			try {
				if (threads[i] != null) {
					threads[i].join();
				}
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
	sortNetworks();
	lastHighscore = networks.get(0).getScore();
	generateNewNetworks();
}
```

Der erste Parameter ist ein Interface, mit dem die KNNs mit der Methode `test`, die ein KNN entgegennimmt und den erreichten Score zurück gibt, getestet werden können. Mit dem zweiten Parameter `useMultiThreading` kann bestimmt werden, ob Multithreading genutzt werden soll, was eine Iteration auf Computern mit mehr als einem Prozessorkern deutlich beschleunigt.

Am Anfang der Methode wird zunächst ein Thread-Array angelegt, in das für jedes KNN ein Thread gespeichert werden kann. Dann wird für jedes KNN, das noch nicht getestet wurde, ein neuer Thread angelegt und dem Array hinzugefügt, der, wenn Multithreading genutzt werden soll, gestartet wird. Dadurch wird das Testen des KNNs in einem parallelen Thread ausgeführt und blockiert den aktuellen Thread nicht. Falls Multithreading nicht genutzt werden soll, wird der Thread nicht gestartet, sondern im aktuellen Thread ausgeführt. Danach wird, falls Multithreading benutzt wurde, noch darauf gewartet, dass alle Threads fertig sind. Am Ende der Methode werden die KNNs noch nach ihrem Score sortiert und der höchste Score zwischengespeichert.

Ganz zum Schluss wird außerdem noch die Methode `generateNewNetworks` aufgerufen, die die schlechtesten KNNs durch eine mutierte Variante der restlichen KNNs ersetzt. Sie sieht wie folgt aus:

```
private void generateNewNetworks() {
	for (int i = keepAmount; i < networks.size(); i++) {
		int randIndex = (int) (Math.random() * Math.random() *
		keepAmount);
		networks.set(i, new EvolutionalNeuralNetwork(
			networks.get(randIndex)
			.getNeuralNetwork().getCopy()));
		networks.get(i).mutate(mutationRate);
	}
}
```

Wie zusehen ist, fängt die **for**-Schleife bei dem Index `keepAmount` an, um nur die KNNs zu ersetzen, die nicht behalten werden sollen. Es wird dann der zufällige Index `randIndex` zwischen 0 und `keepAmount` generiert. Mit `Math.random()` wird hier zweimal multipliziert, damit der Index häufiger näher bei 0 ist, um die besseren KNNs mit einer höheren Wahrscheinlichkeit auszuwählen. Mit einer Kopie des KNNs, das mit dem Index `randIndex` indexiert ist, wird dann das KNN mit dem Index `i` überschrieben. Diese Kopie wird anschließend noch mit der Mutationsrate mutiert.
Abschließend lässt sich in der Klasse `EvolutionalTrainer` noch die Methode `resetTested`, die von allen KNNs die Information, dass sie schon getestet wurden zurücksetzt, sodass in der nächsten Iteration alle KNNs erneut getestet werden. Diese Methode sollte aufgerufen werden, wenn der Testalgorithmus sich geändert hat, der Score also nicht mehr absolut ist (Siehe Abschnitt 2.2.1.2).

2.2.2 Umgang mit Illegalen Zügen

Dass das KNN am Anfang komplett zufällige Züge macht, führt unweigerlich auch dazu, dass es Züge generieren wird, die nicht auf das Spiel anwendbar sind, da laut den Regeln von Tic-Tac-Toe nicht in ein schon besetztes Feld gesetzt werden darf. In dieser Arbeit wird dies als illegaler Zug bezeichnet. Um also mit illegalen Zügen umzugehen, gibt es mehrere Möglichkeiten.

Möglichkeit 1:

Falls das KNN einen illegalen Zug macht, verliert es. Das führt auf lange Sicht dazu, dass das KNN lernt, keine illegalen Züge mehr zu machen, da sich, um auf den evolutionären Aspekt zurückzukommen, eher die Versionen des KNNs durchsetzen, die weniger illegale Züge machen.

Möglichkeit 2:

Statt aus allen Feldern das Feld auszuwählen, dessen Output am höchsten ist, wird nur aus den Feldern, auf denen ein Symbol gesetzt werden könnte, das ausgewählt, dessen Output am höchsten ist. Es werden also alle Outputs, die zu einem illegalen Zug führen

würden, ignoriert. Das führt dazu, dass das KNN nicht erst lernen muss, keine illegalen Züge zu machen, wodurch das Lernen deutlich schneller geschieht. Deshalb wird diese Variante in dieser Arbeit verwendet.

2.2.3 Wahl der Aktivierungsfunktion

Als Aktivierungsfunktion wird meistens auf die Sigmoidfunktion genutzt, die in Abschnitt 1.2.2 schon einmal erwähnt wurde:

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

Die Sigmoidfunktion kommt dem Verhalten eines echten Neurons relativ nah. Sie ist sehr gut differenzierbar, was für einige Lernmethoden wichtig ist. Abgeleitet lässt sich die Sigmoidfunktion wie folgt darstellen[21]:

$$\varphi'(x)=\varphi(x)*(1-\varphi(x))$$

Diese Ableitung lässt sich sehr schnell berechnen, wenn zuvor bereits der Wert der Sigmoidfunktion berechnet wurde. Dadurch wird das Lernen mit Lernmethoden, die diese Ableitung benötigen, sehr effizient.

Allerdings benötigt das evolutionäre Lernen keine Ableitung, weshalb folgende Aktivierungsfunktion mit der Bezeichnung Softsign[22], die die Eingangswerte auf das Intervall $[-1;1]$ staucht, besser geeignet ist:

$$\varphi(x)=\frac{x}{1+|x|}$$

Damit sie allerdings näherungsweise so wie die Sigmoidfunktion verläuft und die Eingangswerte auch auf das Intervall $[0;1]$ staucht, wird sie folgendermaßen modifiziert:

$$\varphi(x)=\frac{0{,}5*x}{1+|x|}+0{,}5$$

Diese Funktion lässt sich zwar nicht so gut differenzieren, kann aber deutlich schneller berechnet werden.

2.2.4 Anpassung des evolutionären Lernens an Tic-Tac-Toe

Jetzt könnte ein KNN trainiert werden, indem man es gegen eine bereits perfekte KI spielen lässt. Allerdings soll diese Arbeit zeigen, wie das evolutionäre Lernen auf Probleme angewendet werden kann, für die noch keine andere oder nur eine schlechte

21 Wikipedia: Sigmoidfunktion, https://de.wikipedia.org/wiki/Sigmoidfunktion (Abgerufen am: 25.03.2018)

22 Wikipedia: Activation function, https://en.wikipedia.org/wiki/Activation_function (Abgerufen am: 07.03.2018)

Lösung zur Verfügung steht. Deshalb sollen die KNNs gegen andere anfangs zufällige KNNs antreten. Das läuft dann folgendermaßen ab:
Es werden mehrere Gruppen von KNNs gebildet, die nur innerhalb einer Gruppe sortiert und durch mutierte KNNs ausgetauscht werden.
Das Testen des KNNs erfolgt dann, indem es gegen KNNs antritt, die nicht in der eigenen Gruppe sind. Gegen die KNNs in der eigenen Gruppe tritt es nicht an, da es gegen sich selbst bzw. gegen eine nur leicht veränderte Variante von sich immer ungefähr gleich gut sein wird, weshalb die Qualität daran nicht gemessen werden kann.
Eine Gruppe wird dann mehrfach (im Code dieser Arbeit 100-mal) mithilfe der anderen Gruppen trainiert, sodass sich die KNNs in dieser Gruppe an das Niveau der anderen Gruppen anpassen können. Dann wird die nächste Gruppe trainiert und so weiter.

2.2.5 Implementation des Lernens an Tic-Tac-Toe

Um den oben beschriebenen Vorgang zu implementieren, wurde die Klasse `Training` erstellt. In dieser findet sich die Methode `train`, in der jetzt KNNs trainiert werden können, Tic-Tac-Toe zu spielen. Sie sieht aus wie folgt:

```
public void train() {
	NeuralNetwork n = new NeuralNetwork(
		ActivationFunction.SOFTSIGN_NORM, 9, 18, 18, 9);
	evolutionalTrainers = new ArrayList<EvolutionalTrainer>();
	for (int i = 0; i < 20; i++) {
		evolutionalTrainers.add(
			new EvolutionalTrainer(n, 10, 5, true));
		evolutionalTrainers.get(i).setMutationRate(0.2);
	}
	int wholeIterations = 0;
	while (true) {
		if (evolutionalTrainerIndex == 0) {
			wholeIterations++;
		}
		System.out.println("Training " + evolutionalTrainerIndex
			+ " at iteration " + wholeIterations);
		EvolutionalTrainer evolutionalTrainer =
			evolutionalTrainers
			.get(evolutionalTrainerIndex);
		ArrayList<NeuralNetwork> opponentNetworks =
			getOpponentNetworks();
		evolutionalTrainer.resetTested();
		for (int i = 0; i < 100; i++) {
			evolutionalTrainer.doIteration((nn) ->
				getNeuralNetworkTester(
				opponentNetworks).test(nn), true);
		}
		if (evolutionalTrainerIndex
			== evolutionalTrainers.size() - 1) {
			saveBestNetworks();
		}
		evolutionalTrainerIndex = ++evolutionalTrainerIndex
			% evolutionalTrainers.size();
	}
}
```

Am Anfang der Methode wird ein neues KNN erstellt, das als Vorlage dient. Es wird mit der Aktivierungsfunktion initialisiert, die in Abschnitt 2.2.3 erarbeitet wurde und hat 9 Neuronen im Input-Layer, 2 Hidden-Layer mit jeweils 18 Neuronen und 9 Neuronen im Output-Layer. Dann werden 20 Gruppen mit jeweils 10 KNNs erstellt. Die Anzahl der KNNs, die nach der Sortierung nicht gelöscht werden, beträgt 5. Außerdem werden die KNNs randomisiert und die Mutationsrate auf 0.2 gesetzt. Dann wird die **while**-Schleife ausgeführt. Am Anfang jedes Durchgangs wird der Index der aktuellen Gruppe sowie die Anzahl der bisher durchführten Durchgänge der **while**-Schleife ausgegeben, damit kontrolliert werden kann, wie weit das Training ist und abgeschätzt werden kann, wie schnell die KNNs trainiert werden. In der Methode `getOpponentNetworks` werden dann die jeweils ersten zwei KNNs aller Gruppen außer der aktuellen Gruppe in einer Liste gespeichert und diese durchmischt. Da die KNNs aus einer Gruppe nach einem Durchlauf der **while**-Schleife gegen KNNs antreten, die durch das Training leicht verändert wurden, und der Score dann nicht mehr absolut ist, wird die Methode `resetTested` aufgerufen (Siehe Abschnitt 2.2.1.2). Danach werden 100 Iterationen mit dem `NeuralNetworkTester`, den `getNeuralNetworkTester` zurück gibt, durchgeführt. Am Ende eines Schleifendurchgangs werden, wenn die aktuelle Gruppe die Letzte ist, die besten KNNs gespeichert. Außerdem wird hier der Index der aktuellen Gruppe erhöht, bzw. auf 0 gesetzt, sofern die aktuelle Gruppe die Letzte war.

Betrachtet werden muss noch die Methode `getNeuralNetworkTester`, da hier festgelegt ist, wie der Score berechnet wird:

```
private NeuralNetworkTester getNeuralNetworkTester(ArrayList<NeuralNetwork>
opponentNetworks) {
	return new NeuralNetworkTester() {
		@Override
		public double test(NeuralNetwork nn) {
			TicTacToe ttt = new TicTacToe();
			NeuralNetworkPlayer nnp = new NeuralNetworkPlayer(
				nn);
			ttt.setPlayer1(nnp);
			for (int i = 0; i < opponentNetworks.size(); i++) {
				ttt.setPlayer2(new NeuralNetworkPlayer(
					opponentNetworks.get(i)));
				ttt.setStartPlayer(i % 2 == 0);
				ttt.run();
			}
			double score = (nnp.getWins()
					+ nnp.getDraws() * 0.5 - nnp.getIlls())
					/ (double) opponentNetworks.size();
			return score;
		}
	};
}
```

Die Methode `test` von dem `NeuralNetworkTester`, der zurückgegeben wird, legt zu Beginn ein `TicTacToe` Objekt und ein `NeuralNetworkPlayer` Objekt an. Letzteres wird mit dem übergebenen KNN initialisiert und dem Tic-Tac-Toe Spiel als Spieler 1 hinzugefügt. Dann tritt das zu testende KNN gegen jedes Gegner-KNN einmal an, wobei es immer abwechselnd anfängt. Mit der Anzahl der gewonnenen Spiele s_{gew}, der Anzahl der unentschiedenen Spiele s_{un}, der Anzahl der Spiele s_{ill}, in denen das KNN illegale Züge gemacht hat, und der Gesamtanzahl s_{ges} der Spiele lässt sich folgende Formel für den Score x aufstellen:

$$x = \frac{s_{gew} + s_{un} * 0.5 - s_{ill}}{s_{ges}}$$

Die Formel wurde so gewählt, dass gewonnene Spiele einen positiven Einfluss, unentschiedene Spiele einen kleineren positiven Einfluss und Spiele, in denen ein illegaler Zug gemacht wurde, einen negativen Einfluss haben, sodass die KNNs, die am häufigsten gewinnen, den höchsten Score erhalten. Dieser Score x wird dann am Ende der Methode `test` zurückgegeben.

3 Auswertung

3.1 Ergebnisse

Als Ergebnis des oben dargestellten Lernprozesses wurde ein KNN erzielt, welches die Ziele dieser Arbeit gut widerspiegelt. Es wurden fast 900 Iterationen durchgeführt. Einige Beispiel-Spiele, in denen gegen das KNN gespielt wurde, finden sich im Anhang, an denen zu erkennen ist, dass das KNN häufig so setzt, dass es gewinnt. Ebenso ist erkennbar, dass das KNN oft an die Position setzt, auf die der Gegner setzen müsste, um zu gewinnen. Das KNN hat also gelernt, wie Tic-Tac-Toe gespielt und gewonnen wird. Allerdings setzt das KNN nicht immer so, dass es gewinnt, was dem Ziel dieser Arbeit entspricht, keine perfekte KI zu entwickeln.

3.2 Bewertung der Ergebnisse

Trotz des Erreichens des Zieles dieser Arbeit kann darüber diskutiert werden, ob die Ergebnisse als positiv zu betrachten sind; denn das KNN ist, auch nach einigen Stunden an Training, nicht immer in der Lage, gut zu spielen. Andere Algorithmen, wie beispielsweise der Minimax-Algorithmus,[23] müssen überhaupt nicht trainiert werden

23 Fox, Jason, Tic Tac Toe: Understanding the Minimax Algorithm, https://www.neverstopbuilding.com/blog/2013/12/13/tic-tac-toe-understanding-the-minimax-algorithm13 (Aufgerufen am: 25.03.2018)

und spielen in jedem Fall den bestmöglichen Zug. Als Lösung für das Problem, dass es nach kurzer Zeit nur noch wenig Spaß macht, gegen eine unbesiegbare KI zu spielen, könnten Möglichkeiten betrachtet werden, die KI gezielt schlechter agieren zu lassen. Zum Beispiel, indem statt des von der KI vorgeschlagenen Zuges ein anderer Zug zufällig ausgewählt wird.
Da das KNN trotzdem ein wenig gelernt hat, Tic-Tac-Toe zu spielen, konnte in dieser Arbeit anhand des Beispieles Tic-Tac-Toe gezeigt werden, wie der Algorithmus des evolutionären Lernens funktioniert.

3.3 Ausblick

Wie in Abschnitt 3.1 aufgezeigt wurde, sind die Ergebnisse nicht perfekt. Das war zwar auch ein Ziel dieser Arbeit, jedoch könnte es für andere Einsatzmöglichkeiten sinnvoll sein, die KNNs noch weiter zu verbessern. Eine Möglichkeit, dies zu erreichen, ist, den evolutionären Algorithmus um das sogenannte Crossover zu erweitern. Hierzu werden alle Gewichte und Schwellenwerte als eine Liste aus Zahlen betrachtet, die Chromosom genannt wird. Dann werden zwei oder mehr „Eltern“- Chromosomen ausgewählt und Abschnitte aus diesen Chromosomen ausgetauscht. Diese neu kombinierten Chromosomen werden „Kinder“ genannt. So können verschiedene Eigenschaften verschiedener KNNs miteinander kombiniert werden [24].

24 Zhang, Byoung-Tak, 1992, Lernen durch genetisch-neuronale Evolution, infix Verlag, S. 134ff

4 Literaturverzeichnis

1. 10.12: Neural Networks: Feedforward Algorithm Part 1 - The Nature of Code, https://www.youtube.com/watch?v=qWK7yW8oS0I (Abgerufen am: 18.03.2018)
2. Abdolsaheb, Ahmad, How to make your Tic Tac Toe game unbeatable by using the minimax algorithm, https://medium.freecodecamp.org/how-to-make-your-tic-tac-toe-game-unbeatable-by-using-the-minimax-algorithm-9d690bad4b37 (Abgerufen am: 25.03.2018)
3. Fox, Jason, Tic Tac Toe: Understanding the Minimax Algorithm, https://www.neverstopbuilding.com/blog/2013/12/13/tic-tac-toe-understanding-the-minimax-algorithm13 (Aufgerufen am: 25.03.2018)
4. Girwidz R.,LMU München, Das Neuron, http://www.didaktikonline.physik.uni-muenchen.de/piko/material/medizintechnik/medizintechnik/neuron1/neuron.html (Abgerufen am: 25.03.2018)
5. Kristian, Alex, Künstliche neuronale Netze in C#, http://www.codeplanet.eu/tutorials/csharp/70-kuenstliche-neuronale-netze-in-csharp.html (Abgerufen am: 24.03.2018)
6. Rigoll, Gerhard, 1994, Neuronale Netze, Expert Verlag, Renningen-Malmsheim
7. Uni Erlangen, Chemie: Aufbau künstlicher Neuronen, http://www2.chemie.uni-erlangen.de/projects/vsc/chemoinformatik/erlangen/datenanalyse/nn_kneuron.html (Abgerufen am: 24.02.2018)
8. Uni Siegen, Informatik: Künstliche Neuronale Netze, http://pi.informatik.uni-siegen.de/Arbeitsgebiete/ci/knn/ (Abgerufen am: 18.03.2018)
9. Wikipedia: Activation function, https://en.wikipedia.org/wiki/Activation_function (Abgerufen am: 07.03.2018)
10. Wikipedia: Evolution, https://de.wikipedia.org/wiki/Evolution, (Abgerufen am: 25.03.2018)
11. Wikipedia: Künstliches neuronales Netz, https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz (Abgerufen am: 18.03.2018)
12. Wikipedia: Sigmoidfunktion, https://de.wikipedia.org/wiki/Sigmoidfunktion (Abgerufen am: 25.03.2018)
13. Wikipedia: Tic-Tac-Toe, https://de.wikipedia.org/wiki/Tic-Tac-Toe (Abgerufen am: 24.02.2018)
14. Zhang, Byoung-Tak, 1992, Lernen durch genetisch-neuronale Evolution, infix Verlag

5 Anhang

5.1 Spiele eines Trainierten Netzwerks

X: KNN O: menschlicher Gegner
Spiel 1:

Zug 1:

	X	

Zug 2:

	X	
		O

Zug 3:

	X	
	X	
		O

Zug 4:

	X	
	X	
	O	O

Zug 5:

	X	
	X	
X	O	O

Zug 6:

	X	O
	X	
X	O	O

Zug 7:

	X	O
	X	X
X	O	O

Zug 8:

	X	O
O	X	X
X	O	O

Zug 9:

X	X	O
O	X	X
X	O	O

Spiel 2:

Zug 1:

	O	

Zug 2:

	X	
	O	

Zug 3:

	X	
	O	
		O

Zug 4:

	X	
	O	
	X	O

Zug 5:

	X	O
	O	
	X	O

Zug 6:

	X	O
	O	
X	X	O

Zug 7:

	X	O
	O	O
X	X	O

Spiel 3:

Zug 1:

	X	

Zug 2:

	O	
	X	

Zug 3:

	O	
	X	
	X	

Zug 4:

O	O	
	X	
	X	

Zug 5:

O	O	
	X	
X	X	

Zug 6:

O	O	O
	X	
X	X	

Spiel 4:

Zug 1:

	O	

Zug 2:

	X	
	O	

Zug 3:

	X	
	O	
	O	

Zug 4:

	X	
	O	
X	O	

Zug 5:

	X	O
	O	
X	O	

Zug 6:

X	X	O
	O	
X	O	

Zug 7:

X	X	O
	O	O
X	O	

Zug 8:

X	X	O
	O	O
X	O	X

Zug 9:

X	X	O
O	O	O
X	O	X

Spiel 5:

Zug 1:

	X	

Zug 2:

O	X	

Zug 3:

	X	
O	X	

Zug 4:

	X	
O	X	
	O	

Zug 5:

	X	
O	X	
X	O	

Zug 6:

	X	
O	X	O
X	O	

Zug 7:

	X	X
O	X	O
X	O	

Spiel 6:

Zug 1:

	O	

Zug 2:

	O	
	X	

Zug 3:

	O	
	X	
O		

Zug 4:

	O	
	X	
O	X	

Zug 5:

	O	O
	X	
O	X	

Zug 6:

	O	O
	X	
O	X	X

Zug 7:

	O	O
O	X	
O	X	X

Zug 8:

X	O	O
O	X	
O	X	X

5.2 Code

Github: https://github.com/Jonicho/TicTacToeKI

de.jrk.neuralnetwork.Matrix.java:

```java
package de.jrk.neuralnetwork;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Matrix {
    private final double[][] data;

    /**
     * Erzeugt ein Matrix mit {@code rows} Zeilen und {@code cols} Spalten.
     *
     * @param rows
     *            Die Anzahl der Zeilen.
     * @param cols
     *            Die Anzahl der Spalten.
     */
    public Matrix(int rows, int cols) {
        data = new double[rows][cols];
    }

    /**
     * Gibt den Wert an der Position {@code i,j} zurück.
     *
     * @param i
     *            Der Zeilenindex.
     * @param j
     *            Der Spaltenindex.
     * @return Der Wert an der Position {@code i,j}.
     */
    public double get(int i, int j) {
        return data[i][j];
    }

    /**
     * Gibt die Anzahl der Zeilen zurück.
     *
     * @return Die Anzahl der Zeilen.
     */
    public int getRows() {
        return data.length;
    }

    /**
     * Gibt die Anzahl der Spalten zurück.
     *
     * @return Die Anzahl der Spalten.
     */
    public int getCols() {
        return data[0].length;
    }

    /**
     * Gibt die Summe dieser Matrix und der Matrix {@code m} zurück. Die übergebene
     * Matrix muss die gleiche Größe wie diese Matrix haben.
     *
     * @param m
     *            Die Matrix, die addiert werden soll.
     * @return Die Summenmatrix.
     */
    public Matrix add(Matrix m) {
        if (getRows() != m.getRows() || getCols() != m.getCols()) {
            throw new IllegalArgumentException(
                    "The size of this Matrix does not match the size of the given Matrix!");
        }
        return map((x, i, j) -> x + m.get(i, j));
    }
```

```java
/**
 * Gibt die Produktmatrix der Matrixmultiplikation von dieser Matrix und der
 * Matrix {@code m} zurück.
 *
 * @param m
 *            Die Matrix mit der multipliziert werden soll.
 *
 * @return Die Produktmatrix.
 */
public Matrix multiply(Matrix m) {
    if (getCols() != m.getRows()) {
        throw new IllegalArgumentException(
                "The columns of this Matrix do not match the rows of the given Matrix!");
    }
    return new Matrix(getRows(), m.getCols()).map((x, i, j) -> {
        double sum = 0;
        for (int k = 0; k < getCols(); k++) {
            sum += get(i, k) * m.get(k, j);
        }
        return sum;
    });
}

/**
 * Wendet die übergebene {@link MapFunction} {@code function} auf alle Werte
 * dieser Matrix an und gibt eine Matrix mit den Ergebnissen der
 * {@link MapFunction} zurück.
 *
 * @param function
 *            Die Funktion, die auf alle Werte angewendet werden soll.
 * @return Die Ergebnismatrix.
 */
public Matrix map(MapFunction function) {
    Matrix result = new Matrix(getRows(), getCols());
    for (int i = 0; i < result.getRows(); i++) {
        for (int j = 0; j < result.getCols(); j++) {
            result.data[i][j] = function.function(get(i, j), i, j);
        }
    }
    return result;
}

/**
 * Funktion, die in der Methode {@link Matrix#map(MapFunction) map} genutzt
 * wird.
 */
public interface MapFunction {
    double function(double x, int i, int j);
}

/**
 * Gibt eine Kopie dieser Matrix zurück.
 *
 * @return Eine Kopie dieser Matrix.
 */
public Matrix getCopy() {
    return (Matrix) clone();
}

@Override
protected Object clone() {
    return map((x, i, j) -> x);
}

@Override
public String toString() {
    return Arrays.deepToString(data);
}

/**
 * Erzeugt eine 2-dimensionale Matrix aus dem 2-dimensionalen Array
 * {@code array}.
 *
 * @param array
 *            Das 2-dimensionalen Array aus dem eine Matrix erzeugt werden soll.
 * @return Die 2-dimensionale Matrix.
```

```
     */
    public static Matrix from2DArray(double... array) {
        return new Matrix(array.length, 1).map((x, i, j) -> array[i]);
    }

    /**
     * Erzeugt eine Matrix aus einem String, der von {@link #toString() toString}
     * zurück gegeben wurde.
     *
     * @param string
     *            Ein String, der eine Matrix repräsentiert.
     * @return Die Matrix, die durch den String repräsentiert wurde.
     */
    public static Matrix fromString(String string) {
        Pattern pattern = Pattern.compile("\\[([^\\[\\]]*)\\]");
        Matcher matcher = pattern.matcher(string);
        ArrayList<double[]> g = new ArrayList<double[]>();
        int cols = 0;
        while (matcher.find()) {
            String[] s = matcher.group(1).split(", ");
            if (g.size() == 0) {
                cols = s.length;
            } else if (s.length != cols) {
                throw new IllegalArgumentException("Invalid matrix string!");
            }
            double[] d = new double[s.length];
            for (int i = 0; i < s.length; i++) {
                d[i] = Double.parseDouble(s[i]);
            }
            g.add(d);
        }
        return new Matrix(g.size(), cols).map((x, i, j) -> g.get(i)[j]);
    }
}
```

de.jrk.neuralnetwork.ActivationFunction.java:

```
package de.jrk.neuralnetwork;

public class ActivationFunction {
    private ActivationFunction() {
    }

    /**
     * Eine Aktivierungsfunktion.<br>
     * Folgende Aktivierungsfunktionen sind möglich:<br>
     * <br>
     * {@link #IDENTITY}: <code>&phi;(x)=x</code><br>
     * {@link #SIGMOID}: <code>&phi;(x)=1/(1+e^x)</code><br>
     * {@link #TANH}: <code>&phi;(x)=tanh(x)</code><br>
     * {@link #SOFTSIGN}: <code>&phi;(x)=x/(1+|x|)</code><br>
     * {@link #SOFTSIGN_NORM}: <code>&phi;(x)=0.5*x/(1+|x|)+0.5</code><br>
     */
    public static final String IDENTITY = "identity", SIGMOID = "sigmoid", TANH = "tanh",
            SOFTSIGN = "softsign", SOFTSIGN_NORM = "softsign_norm";

    /**
     * Gibt das Ergebnis der Aktivierungsfunktion {@code function} zurück.<br>
     * Folgende Aktivierungsfunktionen sind möglich:<br>
     * <br>
     * {@link #IDENTITY}: <code>&phi;(x)=x</code><br>
     * {@link #SIGMOID}: <code>&phi;(x)=1/(1+e^x)</code><br>
     * {@link #TANH}: <code>&phi;(x)=tanh(x)</code><br>
     * {@link #SOFTSIGN}: <code>&phi;(x)=x/(1+|x|)</code><br>
     * {@link #SOFTSIGN_NORM}: <code>&phi;(x)=0.5*x/(1+|x|)+0.5</code><br>
     *
     * @param function
     *            Die Aktivierungsfunktion.
     * @param x
     *            Das x.
     * @return Das Ergebnis.
     */
    public static double function(String function, double x) {
        switch (function) {
        case IDENTITY:
            return x;
```

```
        case SIGMOID:
            return 1 / (1 + Math.exp(-x));
        case TANH:
            return Math.tanh(x);
        case SOFTSIGN:
            return x / (1 + Math.abs(x));
        case SOFTSIGN_NORM:
            return 0.5 * x / (1 + Math.abs(x)) + 0.5;
        default:
            throw new IllegalArgumentException(
                    "Activation function \"" + function + "\" does not exist!");
        }
    }
}
```

de.jrk.neuralnetwork.NeuralNetwork.java:

```
package de.jrk.neuralnetwork;

import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class NeuralNetwork {
    private final Matrix[] weights;
    private final Matrix[] biases;
    private final Matrix[] activations;
    private final String activationFunction;

    /**
     * Erzeugt ein neues neuronales Netzwerk mit der angegebenen Anzahl von
     * Neuronen.
     *
     * @param neurons
     *            Ein Array mit der Anzahl von Neuronen für jede Schicht.
     */
    public NeuralNetwork(int... neurons) {
        this(ActivationFunction.SOFTSIGN_NORM, neurons);
    }

    /**
     * Erzeugt ein neues neuronales Netzwerk mit der angegebenen Anzahl von Neuronen
     * und der angegebenen Aktivierungsfunktion.
     *
     * @param activationFunction
     *            Die Aktivierungsfunktion, die dieses Netzwerk nutzen soll.
     * @param neurons
     *            Ein Array mit der Anzahl von Neuronen für jede Schicht.
     */
    public NeuralNetwork(String activationFunction, int... neurons) {
        weights = new Matrix[neurons.length - 1];
        biases = new Matrix[neurons.length - 1];
        activations = new Matrix[neurons.length - 1];
        this.activationFunction = activationFunction;
        for (int i = 1; i < neurons.length; i++) {
            weights[i - 1] = new Matrix(neurons[i], neurons[i - 1]);
            biases[i - 1] = new Matrix(neurons[i], 1);
        }
    }

    /**
     * Randomisiert die Gewichte und Schwellenwerte dieses Netzwerkes in dem
     * Interval [-range, range].
     *
     * @param range
     *            Die Größe des Intervals in dem die zufälligen Werte liegen sollen.
     */
    public void randomize(double range) {
        for (int l = 0; l < weights.length; l++) {
            weights[l] = weights[l].map((x, i, j) -> Math.random() * 2 * range - range);
            biases[l] = biases[l].map((x, i, j) -> Math.random() * 2 * range - range);
        }
    }

    /**
     * Führt den Feedforward Algorithmus mit der gegebenen Input-Matrix aus und gibt
```

```
     * die Output-Matrix zurück. Die Größe der Input-Matrix muss genau der Anzahl
     * der Neuronen in der ersten Schicht entsprechen.
     *
     * @param inputs
     *            Die Input-Matrix.
     * @return Die Output-Matrix.
     */
    public Matrix feedforward(Matrix inputs) {
        for (int a = 0; a < activations.length; a++) {
            activations[a] = weights[a].multiply(a == 0 ? inputs : activations[a - 1])
                    .add(biases[a])
                    .map((x, i, j) -> ActivationFunction.function(activationFunction, x));
        }
        return activations[activations.length - 1].getCopy();
    }

    /**
     * Gibt die Gewichts-Matrizen für jede Schicht in einem Array zurück.
     *
     * @return Ein Array mit den Gewichts-Matrizen für jede Schicht.
     */
    public Matrix[] getWeights() {
        return weights;
    }

    /**
     * Gibt die Schwellenwert-Matrizen für jede Schicht in einem Array zurück.
     *
     * @return Ein Array mit den Schwellenwert-Matrizen für jede Schicht.
     */
    public Matrix[] getBiases() {
        return biases;
    }

    /**
     * Gibt die Aktivierungs-Matrizen für jede Schicht in einem Array zurück.
     *
     * @return Ein Array mit den Aktivierungs-Matrizen für jede Schicht.
     */
    public Matrix[] getActivations() {
        return activations;
    }

    /**
     * Gibt die Aktivierungsfunktion zurück, die dieses Netzwerk nutzt.
     *
     * @return Die Aktivierungsfunktion.
     */
    public String getActivationFunction() {
        return activationFunction;
    }

    /**
     * Erstellt eine Kopie dieses Netzwerkes.
     *
     * @return Eine Kopie dieses Netzwerkes.
     */
    public NeuralNetwork getCopy() {
        return (NeuralNetwork) clone();
    }

    @Override
    protected Object clone() {
        int[] neurons = new int[getWeights().length + 1];
        NeuralNetwork nn = new NeuralNetwork(neurons);
        for (int i = 0; i < getWeights().length; i++) {
            nn.getWeights()[i] = getWeights()[i].getCopy();
        }
        for (int i = 0; i < getBiases().length; i++) {
            nn.getBiases()[i] = getBiases()[i].getCopy();
        }
        for (int i = 0; i < getActivations().length; i++) {
            nn.getActivations()[i] = getActivations()[i] == null ? null
                    : getActivations()[i].getCopy();
        }
        return nn;
```

```
    }

    @Override
    public String toString() {
        String result = activationFunction + ":";
        for (int i = 0; i < weights.length; i++) {
            result += "{" + weights[i] + ";" + biases[i] + "}";
        }
        return result;
    }

    /**
     * Erzeugt ein Neuronales Netzwerk aus einem String, der von {@link #toString()
     * toString} zurück gegeben wurde.
     *
     * @param string
     *            Ein String, der ein Neuronales Netzwerk repräsentiert.
     * @return Das Neuronale Netzwerk, das durch den String repräsentiert wurde.
     */
    public static NeuralNetwork fromString(String string) {
        String activationFunction = string.split(":")[0];
        string = string.split(":")[1];
        ArrayList<Matrix> weights = new ArrayList<Matrix>();
        ArrayList<Matrix> biases = new ArrayList<Matrix>();
        Pattern pattern = Pattern.compile("\\{([^\\{\\}]*)\\}");
        Matcher matcher = pattern.matcher(string);
        while (matcher.find()) {
            String[] layer = matcher.group(1).split(";");
            weights.add(Matrix.fromString(layer[0]));
            biases.add(Matrix.fromString(layer[1]));
        }
        int[] neurons = new int[weights.size() + 1];
        neurons[0] = weights.get(0).getCols();
        for (int i = 1; i < neurons.length; i++) {
            neurons[i] = weights.get(i - 1).getRows();
        }
        NeuralNetwork result = new NeuralNetwork(activationFunction, neurons);
        for (int i = 0; i < weights.size(); i++) {
            result.getWeights()[i] = weights.get(i);
            result.getBiases()[i] = biases.get(i);
        }
        return result;
    }
}
```

de.jrk.neuralnetwork.training.EvolutionalTrainer.java:

```
package de.jrk.neuralnetwork.training;

import java.util.ArrayList;
import de.jrk.neuralnetwork.NeuralNetwork;

public class EvolutionalTrainer {
    private ArrayList<EvolutionalNeuralNetwork> networks;
    private int keepAmount;
    private double mutationRate;
    private double lastHighscore;

    /**
     * Erzeugt einen neues Objekt zum evolutionärem Lernen von
     * {@link NeuralNetwork}s.
     *
     * @param seedNetwork
     *            Das {@link NeuralNetwork}, das Vorlage dient.
     * @param networkAmount
     *            Anzahl der {@link NeuralNetwork}s.
     * @param keepAmount
     *            Anzahl der {@link NeuralNetwork}s, die in
     *            {@link #generateNewNetworks() generateNewNetworks} behalten
     *            werden.
     * @param randomize
     *            Ob die {@link NeuralNetwork}s zu Beginn randomisiert werden
     *            sollen.
     */
    public EvolutionalTrainer(NeuralNetwork seedNetwork, int networkAmount, int keepAmount,
            boolean randomize) {
```

```
    if (networkAmount < 2) {
        throw new IllegalArgumentException("The amount of networks must not be less than 2!");
    }
    if (keepAmount >= networkAmount) {
        throw new IllegalArgumentException(
                "The amount of keep networks has to be less than the amount of networks!");
    }
    this.keepAmount = keepAmount;
    networks = new ArrayList<EvolutionalNeuralNetwork>(networkAmount);
    for (int i = 0; i < networkAmount; i++) {
        networks.add(new EvolutionalNeuralNetwork(seedNetwork.getCopy()));
    }
    if (randomize) {
        for (EvolutionalNeuralNetwork neuralNetworkWithScore : networks) {
            neuralNetworkWithScore.getNeuralNetwork().randomize(1);
        }
    }
}

/**
 * Führt eine Iteration aus, in der die {@link NeuralNetwork}s mit dem
 * {@link NeuralNetworkTester} {@code nnt} getestet und nach Score sortiert
 * werden und anschließend {@link #generateNewNetworks() generateNewNetworks}
 * aufgerufen wird.
 *
 * @param nnt
 *            Der {@link NeuralNetworkTester}, mit dem die
 *            {@link NeuralNetwork}s getestet werden.
 * @param useMultiThreading
 *            Ob Multithreading genutzt werden soll.
 */
public void doIteration(NeuralNetworkTester nnt, boolean useMultiThreading) {
    Thread[] threads = new Thread[networks.size()];
    for (int i = 0; i < networks.size(); i++) {
        if (!networks.get(i).tested) {
            int index = i;
            NeuralNetwork nn = networks.get(index).getNeuralNetwork();
            threads[i] = new Thread(() -> {
                networks.get(index).setScore(nnt.test(nn));
            });
            if (useMultiThreading) {
                threads[i].start();
            } else {
                threads[i].run();
            }
        }
    }
    if (useMultiThreading) {
        for (int i = 0; i < threads.length; i++) {
            try {
                if (threads[i] != null) {
                    threads[i].join();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    sortNetworks();
    lastHighscore = networks.get(0).getScore();
    generateNewNetworks();
}

/**
 * Die besten {@link #keepAmount} {@link NeuralNetwork}s werden behalten, die
 * restlichen werden durch eine mutierte Variante von zufällig ausgewählten
 * behaltenen {@link NeuralNetwork}s ersetzt.
 */
private void generateNewNetworks() {
    for (int i = keepAmount; i < networks.size(); i++) {
        int randIndex = (int) (Math.random() * Math.random() * keepAmount);
        networks.set(i, new EvolutionalNeuralNetwork(
                networks.get(randIndex).getNeuralNetwork().getCopy()));
        networks.get(i).mutate(mutationRate);
    }
}
```

```java
/**
 * Markiert alle {@link NeuralNetwork}s als ungetestet. Sollte aufgerufen
 * werden, wenn der Test-Algorithmus geändert wurde.
 */
public void resetTested() {
    for (EvolutionalNeuralNetwork evolutionalNeuralNetwork : networks) {
        evolutionalNeuralNetwork.tested = false;
    }
}

/**
 * Sortiert die {@link NeuralNetwork}s absteigend nach Score.
 */
private void sortNetworks() {
    networks.sort((n1, n2) -> (int) Math.signum(n2.getScore() - n1.getScore()));
}

/**
 * Gibt den letzten höchsten Score zurück.
 *
 * @return Der letzte höchste Score.
 */
public double getHighscore() {
    return lastHighscore;
}

/**
 * Setzt die Rate, mit der die {@link NeuralNetwork}s in
 * {@link #generateNewNetworks() generateNewNetworks} mutiert werden.
 *
 * @param mutationRate
 *            Die Mutationsrate.
 */
public void setMutationRate(double mutationRate) {
    this.mutationRate = mutationRate;
}

/**
 * Gibt eine Kopie des {@link NeuralNetwork}s mit dem höchsten Score zurück.
 *
 * @return Das beste {@link NeuralNetwork}.
 */
public NeuralNetwork getBestNetwork() {
    sortNetworks();
    return networks.get(0).getNeuralNetwork().getCopy();
}

/**
 * Gibt alle {@link NeuralNetwork}s in einer Liste zurück.
 *
 * @return Eine Liste mit allen {@link NeuralNetwork}s.
 */
public ArrayList<NeuralNetwork> getNetworks() {
    ArrayList<NeuralNetwork> result = new ArrayList<NeuralNetwork>();
    for (int i = 0; i < networks.size(); i++) {
        result.add(networks.get(i).getNeuralNetwork());
    }
    return result;
}

/**
 * Ein Interface, das in
 * {@link EvolutionalTrainer#doIteration(NeuralNetworkTester, boolean)
 * doIteration} verwendet wird, um ein {@link NeuralNetwork} zu testen.
 */
public interface NeuralNetworkTester {
    public double test(NeuralNetwork nn);
}

/**
 * Klasse, die ein {@link NeuralNetwork} und einen Score hält.
 */
public class EvolutionalNeuralNetwork {
    private final NeuralNetwork neuralNetwork;
    private double score;
```

```
        private boolean tested = false;

        public EvolutionalNeuralNetwork(NeuralNetwork neuralNetwork) {
            this.neuralNetwork = neuralNetwork;
        }

        public NeuralNetwork getNeuralNetwork() {
            return neuralNetwork;
        }

        public double getScore() {
            return score;
        }

        public void setScore(double score) {
            this.score = score;
            tested = true;
        }

        /**
         * Mutiert das {@link NeuralNetwork}.
         *
         * @param mutationRate
         *            Die Rate, die angibt, wie stark das {@link NeuralNetwork} mutiert
         *            werden soll.
         */
        public void mutate(double mutationRate) {
            for (int w = 0; w < neuralNetwork.getWeights().length; w++) {
                neuralNetwork.getWeights()[w] = neuralNetwork.getWeights()[w].map(
                        (x, i, j) -> x + ((Math.random() * mutationRate * 2) - mutationRate) * x);
            }
            for (int b = 0; b < neuralNetwork.getBiases().length; b++) {
                neuralNetwork.getBiases()[b] = neuralNetwork.getBiases()[b].map(
                        (x, i, j) -> x + ((Math.random() * mutationRate * 2) - mutationRate) * x);
            }
        }
    }
}
```

de.jrk.tictactoe.players.Player.java:

```
package de.jrk.tictactoe.players;

import de.jrk.tictactoe.TicTacToe;

/**
 * Ein Spieler für {@link TicTacToe}
 */
public abstract class Player {
    /**
     * Wird am Anfang jedes Spiels aufgerufen.
     *
     * @param firstPlayer
     *            Gibt an, ob dieser Spieler der erste Spieler ist.
     */
    public abstract void init(boolean firstPlayer);

    /**
     * Wird in jedem Zug aufgerufen.
     *
     * @param field
     *            Das Spielfeld.
     * @return Ein Array mit den Koordinaten, wo dieser Spieler hinsetzen soll.
     */
    public abstract int[] turn(int[][] field);

    /**
     * Wird am Ende jedes Spiels aufgerufen.
     *
     * @param winPlayer
     *            Die Nummer des Spielers, der gewonnen hat.
     * @param field
     *            Das Spielfeld.
     */
    public abstract void finish(int winPlayer, int[][] field);
}
```

de.jrk.tictactoe.players.ConsolePlayer.java:

```java
package de.jrk.tictactoe.players;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import de.jrk.tictactoe.players.Player;

public class ConsolePlayer extends Player {
    private boolean firstPlayer;

    @Override
    public void init(boolean firstPlayer) {
        this.firstPlayer = firstPlayer;
    }

    @Override
    public int[] turn(int[][] field) {
        for (int j = 0; j < field[0].length; j++) {
            for (int i = 0; i < field.length; i++) {
                if (i != 0) {
                    System.out.print(" ");
                }
                System.out.print(field[i][j]);
            }
            System.out.println();
        }
        System.out.println("Player " + (firstPlayer ? "1" : "2") + ", type pos:");
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String line[] = null;
        try {
            line = br.readLine().split(",");
        } catch (IOException e) {
            e.printStackTrace();
        }
        int[] result = { Integer.parseInt(line[0].trim()), Integer.parseInt(line[1].trim()) };
        return result;
    }

    @Override
    public void finish(int winPlayer, int[][] field) {
        for (int j = 0; j < field[0].length; j++) {
            for (int i = 0; i < field.length; i++) {
                if (i != 0) {
                    System.out.print(" ");
                }
                System.out.print(field[i][j]);
            }
            System.out.println();
        }
        System.out.println("Player " + winPlayer + " won!");
    }
}
```

de.jrk.tictactoe.players.NeuralNetworkPlayer.java:

```java
package de.jrk.tictactoe.players;

import de.jrk.neuralnetwork.Matrix;
import de.jrk.neuralnetwork.NeuralNetwork;

public class NeuralNetworkPlayer extends Player {
    private boolean firstPlayer;
    private int wins = 0;
    private int loses = 0;
    private int draws = 0;
    private int ills = 0;
    public NeuralNetwork nn;

    /**
     * Konstruiert einen neuen Spieler auf Basis des übergebenen
     * {@link NeuralNetwork} {@code nn};
```

```
     *
     * @param nn
     *            Das {@link NeuralNetwork}.
     */
    public NeuralNetworkPlayer(NeuralNetwork nn) {
        this.nn = nn;
    }

    @Override
    public void init(boolean firstPlayer) {
        this.firstPlayer = firstPlayer;
    }

    @Override
    public int[] turn(int[][] field) {
        double[] inputs = new double[9];
        int a = 0;
        for (int i = 0; i < field.length; i++) {
            for (int j = 0; j < field[i].length; j++) {
                int fieldPart = field[i][j];
                inputs[a] = 0;
                if (fieldPart == 0) {
                    a++;
                    continue;
                } else {
                    inputs[a] = (fieldPart == 1 && firstPlayer)
                            || (fieldPart == 2 && !firstPlayer)
                            ? 1 : -1;
                    a++;
                }
            }
        }
        Matrix outputs = nn.feedforward(Matrix.from2DArray(inputs));
        int[] pos = null;
        double posProp = -1;
        int p = 0;
        for (int i = 0; i < field.length; i++) {
            for (int j = 0; j < field[i].length; j++) {
                if (outputs.get(p, 0) > posProp && (field[i][j] == 0)) {
                    pos = new int[] { i, j };
                    posProp = outputs.get(p, 0);
                }
                p++;
            }
        }
        if (field[pos[0]][pos[1]] != 0) {
            ills++;
        }
        return pos;
    }

    /**
     * @return Wie oft dieser Spieler seit der Instantiierung bzw. dem letztem
     *         Aufruf von {@link #resetRecord() resetRecord} gewonnen hat.
     */
    public int getWins() {
        return wins;
    }

    /**
     * @return Wie oft dieser Spieler seit der Instantiierung bzw. dem letztem
     *         Aufruf von {@link #resetRecord() resetRecord} verloren hat.
     */
    public int getLoses() {
        return loses;
    }

    /**
     * @return Wie oft dieser Spieler seit der Instantiierung bzw. dem letztem
     *         Aufruf von {@link #resetRecord() resetRecord} unentschieden gespielt
     *         hat.
     */
    public int getDraws() {
        return draws;
    }
```

```
    /**
     * @return Wie oft dieser Spieler seit der Instantiierung bzw. dem letztem
     *         Aufruf von {@link #resetRecord() resetRecord} einen illegalen Zug
     *         gemacht hat hat.
     */
    public int getIlls() {
        return ills;
    }

    /**
     * Setzt die Statistiken zurück.
     */
    public void resetRecord() {
        wins = 0;
        loses = 0;
        draws = 0;
        ills = 0;
    }

    @Override
    public void finish(int winPlayer, int[][] field) {
        if ((winPlayer == 1 && firstPlayer) || (winPlayer == 2 && !firstPlayer)) {
            wins++;
        } else if (winPlayer != 0) {
            loses++;
        } else {
            draws++;
        }
    }
}
```

de.jrk.tictactoe.TicTacToe.java:

```
package de.jrk.tictactoe;

import de.jrk.tictactoe.players.Player;

public class TicTacToe implements Runnable {
    private int[][] field;
    private Player player1;
    private Player player2;
    private boolean started;
    private boolean startPlayer;

    /**
     * Setzt Spieler 1.
     *
     * @param player
     */
    public void setPlayer1(Player player) {
        if (!started) {
            player1 = player;
        }
    }

    /**
     * Setzt Spieler 2.
     *
     * @param player
     */
    public void setPlayer2(Player player) {
        if (!started) {
            player2 = player;
        }
    }

    /**
     * Setzt den Startspieler. {@code true} steht für Spieler 1, {@code false} für
     * Spieler 2.
     *
     * @param startPlayer
     */
    public void setStartPlayer(boolean startPlayer) {
        this.startPlayer = startPlayer;
    }
```

```java
/**
 * Führt eine Runde des Spiels Tic-Tac-Toe aus.
 */
@Override
public void run() {
    started = true;
    field = new int[3][3];
    player1.init(true);
    player2.init(false);
    boolean currentPlayer = startPlayer;
    int winPlayer;
    while ((winPlayer = getWinPlayer()) == 0 && !isFieldFull()) {
        int[] pos = (currentPlayer ? player1 : player2).turn(getFieldCopy());
        if (field[pos[0]][pos[1]] == 0) {
            field[pos[0]][pos[1]] = currentPlayer ? 1 : 2;
        } else {
            winPlayer = currentPlayer ? 2 : 1;
            break;
        }
        currentPlayer = !currentPlayer;
    }
    player1.finish(winPlayer, field);
    player2.finish(winPlayer, field);
    started = false;
}

/**
 * Gibt eine Kopie des aktuellen Spielfeldes zurück.
 *
 * @return Eine Kopie des aktuellen Spielfeldes.
 */
private int[][] getFieldCopy() {
    int[][] result = new int[3][3];
    for (int i = 0; i < result.length; i++) {
        for (int j = 0; j < result[0].length; j++) {
            result[i][j] = field[i][j];
        }
    }
    return result;
}

/**
 * Berechnet welcher Spieler gewonnen hat. Gibt {@code 0} zurück, wenn (noch)
 * kein Spieler gewonnen hat.
 *
 * @return Die Nummer des Spielers, der gewonnen hat.
 */
private int getWinPlayer() {
    int currentPlayer = field[0][0];
    if (currentPlayer != 0 && field[1][1] == currentPlayer && field[2][2] == currentPlayer) {
        return currentPlayer;
    }
    if (currentPlayer != 0 && field[0][1] == currentPlayer && field[0][2] == currentPlayer) {
        return currentPlayer;
    }
    if (currentPlayer != 0 && field[1][0] == currentPlayer && field[2][0] == currentPlayer) {
        return currentPlayer;
    }
    currentPlayer = field[2][0];
    if (currentPlayer != 0 && field[1][1] == currentPlayer && field[0][2] == currentPlayer) {
        return currentPlayer;
    }
    if (currentPlayer != 0 && field[2][1] == currentPlayer && field[2][2] == currentPlayer) {
        return currentPlayer;
    }
    currentPlayer = field[2][2];
    if (currentPlayer != 0 && field[1][2] == currentPlayer && field[0][2] == currentPlayer) {
        return currentPlayer;
    }
    currentPlayer = field[0][1];
    if (currentPlayer != 0 && field[1][1] == currentPlayer && field[2][1] == currentPlayer) {
        return currentPlayer;
    }
    currentPlayer = field[1][0];
    if (currentPlayer != 0 && field[1][1] == currentPlayer && field[1][2] == currentPlayer) {
        return currentPlayer;
```

```
        }
        return 0;
    }

    /**
     * Berechnet, ob das Spielfeld voll ist.
     *
     * @return Ob das Spielfeld voll ist.
     */
    private boolean isFieldFull() {
        for (int i = 0; i < field.length; i++) {
            for (int j = 0; j < field[i].length; j++) {
                if (field[i][j] == 0) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

de.jrk.tictactoe.Training.java:

```
package de.jrk.tictactoe;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;

import de.jrk.neuralnetwork.ActivationFunction;
import de.jrk.neuralnetwork.NeuralNetwork;
import de.jrk.neuralnetwork.training.EvolutionalTrainer;
import de.jrk.neuralnetwork.training.EvolutionalTrainer.NeuralNetworkTester;
import de.jrk.tictactoe.players.NeuralNetworkPlayer;

public class Training {
    private File saveFile = new File("nets" + System.currentTimeMillis() / 1000 + ".txt");
    private ArrayList<EvolutionalTrainer> evolutionalTrainers;
    private int evolutionalTrainerIndex;

    /**
     * Trainiert 20 Gruppen á 10 {@link NeuralNetwork}s, Tic-Tac-Toe zu spielen.
     * Speichert nach jeder Iteration das jeweils Beste {@link NeuralNetwork} aus
     * jeder Gruppe in eine Datei.
     */
    public void train() {
        NeuralNetwork n = new NeuralNetwork(ActivationFunction.SOFTSIGN_NORM, 9, 18, 18, 9);
        evolutionalTrainers = new ArrayList<EvolutionalTrainer>();
        for (int i = 0; i < 20; i++) {
            evolutionalTrainers.add(new EvolutionalTrainer(n, 10, 5, true));
            evolutionalTrainers.get(i).setMutationRate(0.2);
        }
        int wholeIterations = 0;
        while (true) {
            if (evolutionalTrainerIndex == 0) {
                wholeIterations++;
            }
            System.out.println(
                    "Training " + evolutionalTrainerIndex + " at iteration " + wholeIterations);
            EvolutionalTrainer evolutionalTrainer = evolutionalTrainers
                    .get(evolutionalTrainerIndex);
            ArrayList<NeuralNetwork> opponentNetworks = getOpponentNetworks();
            evolutionalTrainer.resetTested();
            for (int i = 0; i < 100; i++) {
                evolutionalTrainer.doIteration(
                        (nn) -> getNeuralNetworkTester(opponentNetworks).test(nn), true);
            }
            if (evolutionalTrainerIndex == evolutionalTrainers.size() - 1) {
                saveBestNetworks();
            }
            evolutionalTrainerIndex = ++evolutionalTrainerIndex % evolutionalTrainers.size();
        }
    }
```

```
    /**
     * Gibt einen {@link NeuralNetworkTester} zurück, um zu testen, wie gut ein
     * {@link NeuralNetwork} gegen die {@link NeuralNetwork}s
     * {@code opponentNetworks} ist.
     *
     * @param opponentNetworks
     *            Die {@link NeuralNetwork}s gegen die das zu testende
     *            {@link NeuralNetwork} spielt.
     * @return Der {@link NeuralNetworkTester}.
     */
    private NeuralNetworkTester getNeuralNetworkTester(ArrayList<NeuralNetwork> opponentNetworks) {
        return new NeuralNetworkTester() {
            @Override
            public double test(NeuralNetwork nn) {
                TicTacToe ttt = new TicTacToe();
                NeuralNetworkPlayer nnp = new NeuralNetworkPlayer(nn);
                ttt.setPlayer1(nnp);
                for (int i = 0; i < opponentNetworks.size(); i++) {
                    ttt.setPlayer2(new NeuralNetworkPlayer(opponentNetworks.get(i)));
                    ttt.setStartPlayer(i % 2 == 0);
                    ttt.run();
                }
                double score = (nnp.getWins() + nnp.getDraws() * 0.5 - nnp.getIlls())
                        / (double) opponentNetworks.size();
                return score;
            }
        };
    }

    /**
     * Gibt die Gegner-{@link NeuralNetwork}s in einer Liste zurück.
     *
     * @return Die Gegner-{@link NeuralNetwork}s in einer Liste.
     */
    private ArrayList<NeuralNetwork> getOpponentNetworks() {
        ArrayList<NeuralNetwork> opponentNetworks = new ArrayList<NeuralNetwork>();
        for (int i = 0; i < evolutionalTrainers.size(); i++) {
            if (i == evolutionalTrainerIndex)
                continue;
            opponentNetworks.addAll(evolutionalTrainers.get(i).getNetworks().subList(0, 2));
        }
        Collections.shuffle(opponentNetworks);
        return opponentNetworks;
    }

    /**
     * Speichert das jeweils Beste {@link NeuralNetwork} aus jeder Gruppe in eine
     * Datei.
     */
    private void saveBestNetworks() {
        try {
            FileWriter fw = new FileWriter(saveFile);
            String s = "";
            for (EvolutionalTrainer evt : evolutionalTrainers) {
                s += evt.getBestNetwork().toString() + "\n";
            }
            fw.write(s);
            fw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

de.jrk.tictactoe.Main.java:

```
package de.jrk.tictactoe;

public class Main {
    public static void main(String[] args) {
        new Training().train();
    }
}
```

Versicherung zur selbstständigen Arbeit

Hiermit erkläre ich, dass ich die Facharbeit selbstständig angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel verwendet habe.

_____

Datum, Unterschrift