Hoffmann-von-Fallersleben-Schule
Gymnasium
- Offene Ganztagsschule -

Facharbeit Informatik

Künstliche neuronale Netze

Sebastian Morr

Fachlehrer: Herr Ebeling

Eingereicht am 4. April 2008

Inhaltsverzeichnis

1 Einleitung

> *„Die Natur hat uns 3,7 Milliarden Jahre Forschungszeit voraus. Wir wären dumm, wenn wir versuchten, etwas anderes aus dem Boden zu stampfen, das auch nur annähernd so interessant ist.“*

Diese Worte stammen von Toby Simpson, dem Kreativchef der Softwarefirma Cyberlife Technologies.[1] Er ist Mitentwickler des Computerspiels „Creatures“, in dem intelligente und selbstständig lernende künstliche Lebensformen vorkommen. Er führt aus, anstatt das Verhalten von natürlichen Organismen nur zu simulieren, sei es erfolgversprechender, sie direkt nachzubauen.

Genau diesen Ansatz verfolgt man bei den **künstlichen neuronalen Netzen** (kurz: neuronale Netze oder KNN), die sich Strukturen des Gehirns zum Vorbild nehmen, um lernfähige Programme zu erzeugen. Ein künstliches neuronales Netz besteht - kurzgefasst - aus vielen verbundenen künstlichen Nervenzellen, die aus einer gegebenen Eingabe eine passende Ausgabe erzeugen können.

In dieser Arbeit möchte ich das Konzept und die Funktionsweise von künstlichen neuronalen Netzen darstellen. Dabei werde ich mich auf die Darstellung vorwärts verketteter Netze und des geläufigsten Lernverfahrens konzentrieren. Außerdem sollen über die Forschungsgeschichte informiert und Anwendungsbereiche sowie Vor- und Nachteile von neuronalen Netzen aufgezeigt werden. Auf Basis der erarbeiteten Erkenntnisse werde ich ein eigenes neuronales Netz entwickeln und es in eine Beispielanwendung integrieren.

Zum ersten Mal auf das Thema gestoßen bin ich bei der Beschäftigung mit künstlicher Intelligenz in Computerspielen, die mit herkömmlicher Programmierung nur schwer erreichbar ist. Ich habe mich für dieses Thema entschieden, da mich der Gedanke von lernfähigen und intelligenten Programmen fasziniert und ich überzeugt bin, dass dieses Forschungsgebiet in Zukunft große Möglichkeiten eröffnen wird.

2 Geschichte

Die erste Idee zur Konstruktion künstlicher Neuronen wird dem Amerikaner Warren McCulloch zugeschrieben. Zusammen mit seinem Schüler Walter Pitts entwarf er 1943 ein einfaches Modell einer Nervenzelle, das

[1] (o. V.): Zehntausend stürzten ab. In: DER SPIEGEL, 23/1998 vom 01.06.1998, S. 194

McCulloch-Pitts-Neuron, das noch heute einen wichtigen Grundbaustein für viele Netze darstellt.

Frank Rosenblatt griff 1957 das Konzept wieder auf und entwickelte ein KNN, das in der Lage war, verschiedene Buchstaben einzulesen und zu erkennen. Es handelte sich dabei um ein Netz aus zwei Schichten von Neuronen, das Rosenblatt **Perzeptron** nannte und aus 400 Fotozellen und 512 Regelwiderständen zusammenbaute. Das Perzeptron wird in Abschnitt 3.3.2 näher erläutert.

1969 wies Marvin Minsky nach, dass zweischichtige Netze nicht für alle Aufgaben verwendet werden können, und versetzte dem Forschungsgebiet damit einen schweren Schlag. Ein Jahrzehnt lang herrschte „neuronale Eiszeit“[Lossau92], bis John Hopfield ein Netz entwarf, das die von Minsky aufgezeigten Beschränkungen überwinden konnte.

Was folgte, war ein Boom der neuen Forschungsrichtung der Neuroinformatik. Teilweise versuchte man sich an sogenannten Neuro-Chips, Computerchips, die mithilfe von eingebauten Neuronennetzen flexibler werden sollten. Neue Netzarchitekturen entstanden und mit dem Backpropagation-Lernverfahren (siehe Abschnitt 3.4.2), das zuerst von Paul Werbos formuliert wurde, wurde 1974 ein weiterer wichtiger Meilenstein gelegt.

Inzwischen ist die Forschung in der Neuroinformatik fast unüberschaubar geworden. Es wurde etliche neue Netztypen und Lernverfahren entwickelt, die zu vielfältigsten Zwecken eingesetzt werden können, und die Möglichkeit, neuronale Netze im Computer zu simulieren, lässt sie zunehmend leistungsstärker werden.

Quellen: [Lossau92], [SchHaGa90]

3 Theorie

3.1 Biologisches Neuron

Da sich das Konzept der künstlichen neuronalen Netze an biologischen Vorbildern orientiert, soll hier zunächst ein Überblick über Aufbau und Funktionsweise eines natürlichen Neurons, einer Nervenzelle, gegeben werden.

Das Neuron ist die Grundeinheit eines Nervensystems und hat die Aufgabe, Erregungen aufzunehmen und weiterzuleiten. Sie ermöglichen sowohl Wahrnehmung der Umgebung als auch Kommunikation innerhalb eines Lebewesens. Neuronen befinden sich insbesondere in Sinnesorganen, dem Rückenmark und im Gehirn, wobei das menschliche Gehirn

mindestens 14 Milliarden davon besitzt [MiSch97, S. 322].

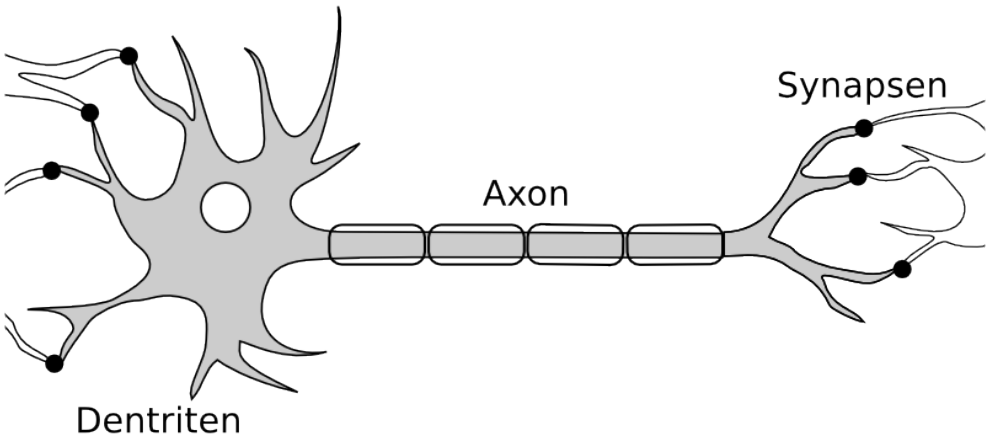


Abbildung 1: Bestandteile einer Nervenzelle

Die **Dendriten** sind weitverzweigte Ausläufer der Zellen und nehmen Erregungen von anderen Neuronen auf. Diese Reize addieren sich und sobald ein bestimmter Schwellwert überschritten ist, „feuert“ das Neuron und erzeugt wiederum eine Erregung. Diese wird dann über das sogenannte **Axon** geleitet und an die Dendriten der nächsten Neuronen weitergereicht, wobei die **Synapsen** als Verbindungsstelle fungieren.

Quellen: [MiSch97], [SchHaGa90], [LaeCl01]

3.2 Künstliches Neuron

Nach dem Vorbild der biologischen Neuronen lässt sich nun ein vereinfachtes Modell eines Neurons konstruieren:

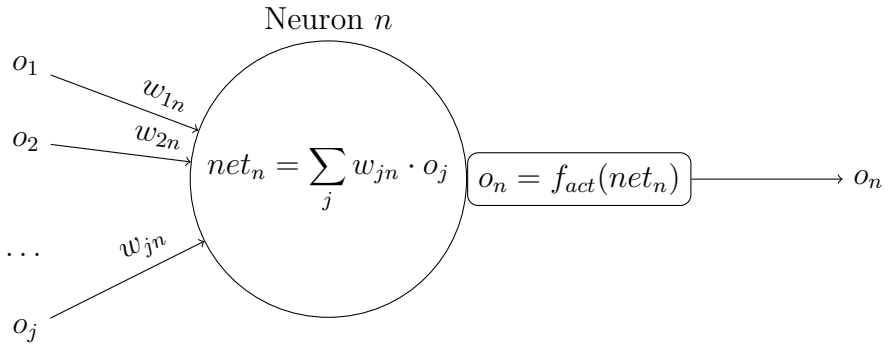


Abbildung 2: künstliches Neuron

Wiederum erhält das Neuron Eingabeinformationen, die von anderen Neuronen stammen (in der Grafik o_1, o_2 usw.). Die Stärke dieser Eingaben wird reguliert durch die jeweiligen **Gewichte** (w_{1n}, w_{2n} usw.).[2] Oft

[2]Die Indizes stehen hierbei jeweils für das Neuron: o_j ist der Ausgabewert des Neurons j, w_{jn} ist das Gewicht der Verbindung zwischen Neuronen j und n.

wird das Neuron noch mit einem weiteren Neuron verbunden, das stets den Ausgabewert 1 besitzt, um auszuschließen, dass es überhaupt keine Eingabe erhält. Dieses Neuron wird **Bias** genannt (englisch für „Verzerrung“, da es die Eingabeinformationen nach oben oder unten verzerrt).

Die **Propagierungsfunktion** net_n addiert die gewichteten Eingaben und fasst somit die Informationen zusammen, die aus dem Netz ins Neuron kommen.

Dieser Wert wird der **Aktivierungsfunktion** f_{act} übergeben, die daraus den neuen Ausgabewert o_n des Neurons ermittelt. Typischerweise verwendet man hier eine sigmoide [3] Funktion, wie die logistische Funktion $f(x) = \frac{1}{1+e^{-x}}$, die bei negativen Werten gegen 0, bei positiven Werten gegen 1 geht:

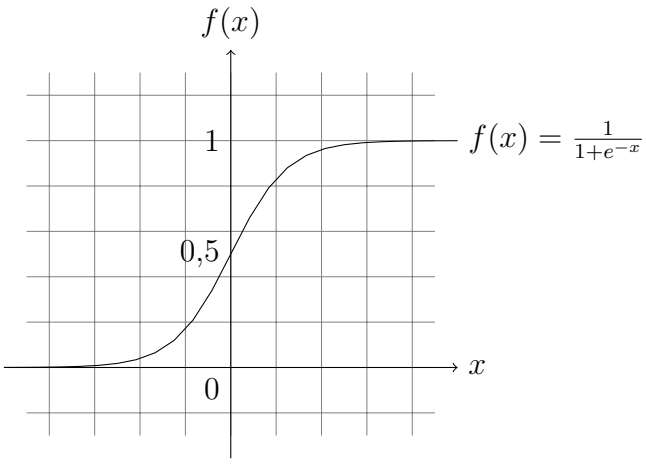


Abbildung 3: logistische Funktion

Wie das biologische Neuron „feuert“ das Neuron also nur, wenn die Eingabeinformation groß genug ist.

Quellen: [LaeCl01], [Callan03], [SchHaGa90]

3.3 Netzarchitekturen

3.3.1 Allgemeines

Die Idee der künstlichen neuronalen Netze basiert nun darauf, viele dieser beschriebenen Neuronen zu einem großen Netz zu verknüpfen. Einigen Neuronen wird dann eine Eingabe gegeben (diese Eingaben nennt man auch **Muster**); diese berechnen dann ihre Ausgaben und geben sie an die Neuronen weiter, mit denen sie verbunden sind. So werden die Werte durch das gesamte Netz weitergereicht. Wenn die letzten Neuronen,

[3] „S-förmige“, vergleiche Abbildung 3

die Ausgabeneuronen, ihre Werte erhalten haben, ist der Vorgang abgeschlossen.

Einige Möglichkeiten, die Neuronen zu verbinden, die sogenannten **Netztypen** oder -**architekturen**, sollen nun vorgestellt werden.

3.3.2 Perzeptron

Das Perzeptron ist einer der einfachsten Netztypen und besteht aus nur zwei Schichten, der **Eingabe**- und der **Ausgabeschicht**, wobei die jeweilige Anzahl der Neuronen in den Schichten frei festgelegt werden kann. Alle Neuronen einer Schicht sind mit allen Neuronen der anderen verbunden:

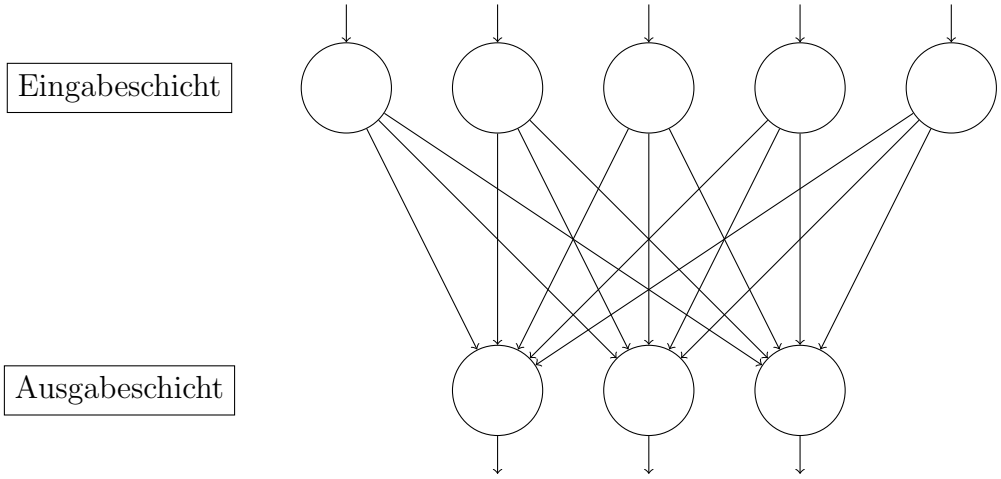


Abbildung 4: Perzeptron

Der Name leitet sich vom englischen *percept*, „wahrnehmen“, ab und stammt daher, dass sich das Perzeptron recht gut zum Erkennen und Zuordnen unterschiedlicher Muster eignet[4].

Quellen: [SchHaGa90], [LaeCl01], [Lossau92]

3.3.3 Mehrschichtige vorwärts verkettete Netze

Wie in Abschnitt 2 beschrieben, gibt es Probleme, die sich mit einem zweischichtigen Netz nicht lösen lassen, etwa eine Darstellung des logischen XOR[5]. Um dieses Manko zu beheben, kann ein Perzeptron nun zu einem mehrschichtigen Netz erweitert werden (siehe Abbildung 5).

Zu einem mehrschichtigen Netz gehört neben Eingabe- und Ausgabeschicht noch mindestens eine **innere Schicht**, die Komplexität und Speicherkapazität des Netzes erhöht. Hierbei gilt die Regel, dass alle Neuronen einer Schicht mit allen Neuronen der folgenden verbunden werden.

[4] Vgl. Abschnitte 2 und 4.1

[5] Eine Verknüpfung von zwei Wahrheitswerten, die dann wahr ist, wenn **genau einer der beiden** Werte wahr ist.

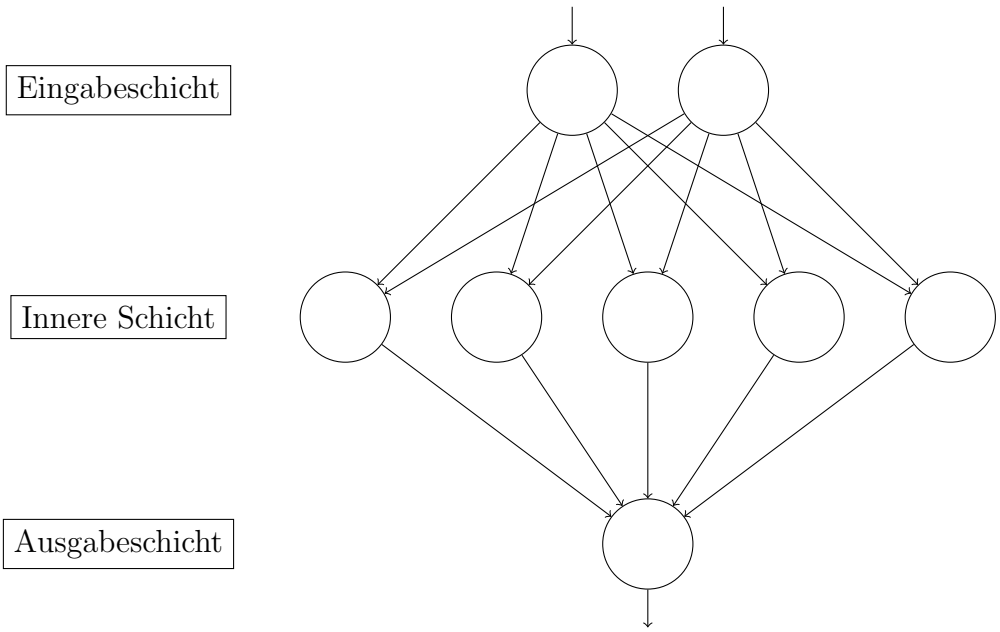


Abbildung 5: mehrschichtiges Netz

Quellen: [SchHaGa90], [LaeCl01], [Callan03]

3.3.4 Sonstige Netztypen

Neben den vorwärts verketteten Netzen gibt es noch eine Vielzahl von anderen Netztypen, von denen zwei wichtige kurz vorgestellt werden sollen.

Rückgekoppelte Netze werden immer dann eingesetzt, wenn die Reihenfolge der Muster eine Rolle spielt. Sie besitzen Verbindungen, die von der Ausgabeschicht wieder in höhere Schichten führen. Beim nächsten Durchlauf berücksichtigt das Netz die Ausgabe des vorherigen Durchlaufes, was eine Art „Erinnerungsvermögen“ ermöglicht.

Selbstorganisierende Karten bestehen zumeist aus einer zweidimensionalen Schicht, deren Neuronen alle miteinander verbunden sind. Sie lernen unüberwacht (siehe folgender Abschnitt) und ordnen jedem Muster einen bestimmten Punkt auf der Karte zu. Liegen für zwei Muster die Punkte nah zusammen, sind sie einander ähnlich. Sie sind bei der Klassifizierung oder der Routenplanung hilfreich.

Quellen: [LaeCl01], [SchHaGa90]

3.4 Lernverfahren

3.4.1 Unterteilung

Die Lernverfahren zum Training neuronaler Netze lassen sich in drei Kategorien einteilen:

Beim **überwachten Lernen** benötigt der Benutzer einige Trainingssätze, welche Eingabedaten und auch die dafür erwarteten Ausgabedaten enthalten. Für jedes Eingabemuster vergleicht man nun die tatsächliche Ausgabe des Netzes mit der erwarteten Ausgabe. Treten Fehler auf, werden die Verbindungsgewichte entsprechend angepasst. Ein solches Lernverfahren ist sehr effizient.

Kann der Benutzer die absolut richtige Ausgabe für ein Muster selbst nicht ermitteln, kann er **bestärkendes Lernen** einsetzen. Hierbei wird dem Netz nur mitgeteilt, ob seine Ausgabe gut oder schlecht war, und es lernt anhand dieser groben Informationen.

Ist das gewünschte Ergebnis gar nicht bekannt, kommt **unüberwachtes Lernen** zum Einsatz, etwa bei der Klassifizierung von Mustern. Das Netz kann hierbei eigenständig Regeln aufstellen und die Eingabemuster verschiedenen Kategorien zuordnen.

Quellen: [Callan03], [LaeCl01]

3.4.2 Backpropagation

Das Backpropagation-Lernen ist der verbreitetste [SchHaGa90, S. 90] Algorithmus zum überwachten Lernen mehrschichtiger vorwärts verketteter Netze. Er beinhaltet zwei Durchläufe: zunächst einen Vorwärtsdurchlauf von der Eingabe- zur Ausgabeschicht, bei dem die Ausgabe des Netzes bestimmt wird. Die Ausgabe wird mit einem Zielmuster verglichen und die Fehlerwerte der einzelnen Ausgabeneuronen bestimmt. Diese Fehler werden nun in einem Rückwärtsdurchlauf zurückpropagiert[6], und die Verbindungsgewichte werden so angepasst, dass das Netz beim nächsten Anlegen des Musters eine passendere Ausgabe erzeugt.

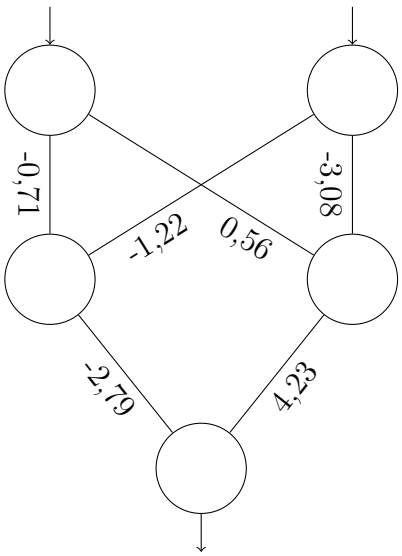
Als Beispiel sei in Abbildung 6 ein Netz gezeigt, das das logische XOR verwirklichen soll. Die Zahlen neben den Verbindungen stehen hierbei für die (zunächst zufällig gewählten) Verbindungsgewichte; Die Bias-Neuronen werden in diesem Beispiel außer Acht gelassen.

Wie die Tabelle mit den Ausgaben o für verschiedene Eingaben i_1 und i_2 zeigt, liefert das Netz noch nicht das gewünschte Verhalten.

Im folgenden Schritt werden daher die Verbindungsgewichte angepasst. Wiederholt man diese Schritte, das Anlegen eines Musters ans Netz, das Bestimmen der Ausgabe und das Anpassen der Gewichte, oft genug, zeigt das Netz nach einiger Zeit das gewünschte Verhalten. Auf diese Weise kann das Netz lernen.

Das Fehlersignal eines Ausgabeneurons, eine Größe, die über die Beteiligung eines Ausgabewertes am Gesamtfehler Auskunft gibt, wird hierbei

[6] „zurück-verbreitet“, woher sich auch der Name der Methode ableitet



i_1	i_2	o
0	0	0,67
0	1	0,39
1	0	0,85
1	1	0,49

Abbildung 6: untrainiertes XOR-Netz

über die Formel

$$\delta_j = f'_{act}(net_j) \cdot (t_j - o_j)$$

bestimmt. f'_{act} steht für die Ableitung der Aktivierungsfunktion nach dem Eingabewert. Sie wird verwendet, um zu bestimmen, wie stark sich der Ausgabewert des Neurons verändert, wenn man den Eingabewert erhöht. net_j ist wieder die Eingabeinformation des Neurons j, o_j die Ausgabe und t_j ist der erwartete Ausgabewert. $t_j - o_j$ beschreibt also den Fehlerwert, die Differenz zwischen erwartetem und tatsächlichem Ausgabewert. Ist also der Fehlerwert hoch und $f'_{act}(net_j)$ ebenfalls, ist auch das Fehlersignal hoch.

Der Fehlerwert eines inneren Neurons kann nicht direkt bestimmt werden, da kein Wert für die erwartete Ausgabe vorliegt. Man bedient sich deshalb der gewichteten Fehlersignale der Neuronen, mit denen das Neuron verbunden ist, denn zu deren Fehler hat es ja - je nach Gewicht der Verbindungen - mehr oder weniger beigetragen:

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

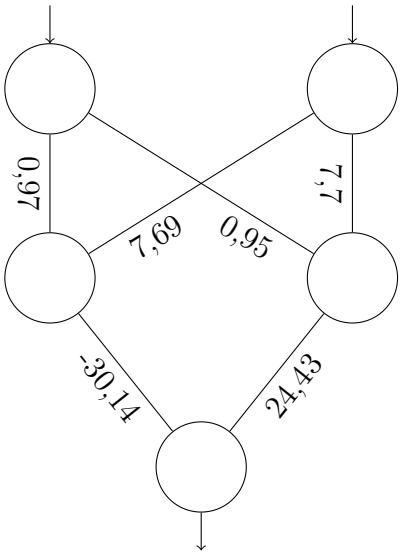
Die Gewichte der eingehenden Verbindungen werden folgend mit der Formel

$$\Delta w_{ij} = \eta \cdot o_j \cdot \delta_j$$

angepasst. η gibt die **Lernrate** an, die oft auf Werte zwischen 0 und 1 gesetzt wird, um zu große Sprünge zu vermeiden. Bei hohem Fehlersignal und hohem Ausgabewert (welcher ja aussagt, dass das Neuron tatsächlich zu dem Fehler beigetragen hat) wird das Gewicht entsprechend stark verändert.

Eine ausführliche Herleitung der Formeln findet sich in [LaeCl01], Seite 191ff.

Das Beispielnetz wurde wiederholt mit den gewünschten Ausgaben trainiert und zeigt - mit tolerierbaren Abweichungen von etwa 0,1 - das richtige Verhalten.



i_1	i_2	o
0	0	0,03
0	1	0,92
1	0	0,91
1	1	0,12

Abbildung 7: trainiertes XOR-Netz

Quellen: [Callan03], [SchHaGa90], [LaeCl01]

4 Praxis

4.1 Vorteile

Ein Vorteil von neuronalen Netzen besteht in ihrer **Generalisierungsfähigkeit**. Sobald ein Netz trainiert ist, kann es nicht nur mit den Trainingsdaten, sondern auch mit unbekannten Mustern umgehen. Wenn Netzgröße und -typ geeignet gewählt wurden, kann ein neuronales Netz abstrahieren, indem es aus den Trainingssätzen Regeln erarbeitet und anwendet.

Die **Fehlertoleranz** eines neuronalen Netzes ist hilfreich, wenn ein Teil des Netzes ausfällt oder beschädigt wird. Da sein Wissen im gesamten Netz verteilt ist, vermindert sich dann seine Leistung, es arbeitet jedoch oft noch sinnvoll weiter.

Neben der **Lernfähigkeit**, auf der ja das gesamte Konzept der neuronalen Netze beruht, besticht es auch durch seine **Anpassungsfähigkeit**. Wenn die Ansprüche ans Netz steigen oder sie sich verändern, kann es jederzeit entsprechend trainiert werden.

Ein neuronales Netz bietet zudem **Modellfreiheit**. Der Benutzer muss ein Problem nicht durch ein Modell beschreiben oder gar programmieren, sondern kann das Netz nur anhand von Beispielen trainieren.

Quellen: [Lossau92], [Callan03], [SchHaGa90]

4.2 Nachteile

Bei der Verwendung eines zu großen Netzes besteht die Gefahr, dass es sich die Trainingsmuster schlichtweg merkt, anstatt zu abstrahieren. Man bezeichnet das als **Überanpassung**; das Netz ist dann nicht mehr flexibel und kann mit unbekannten Mustern nur schlecht umgehen. Auch bei zu oft wiederholtem Training tritt dieses Phänomen auf.

Neuronale Netze sind nicht für alle Anwendungen geeignet: Im Vergleich zur herkömmlichen Programmierung sind sie um ein Vielfaches langsamer und somit **ineffektiv**. Dieser Nachteil könnte in Zukunft mit sich weiter verbessernder Rechnerleistung verschwinden.

Schließlich tritt sowohl bei Optimierungsproblemen als auch generell die Schwierigkeit auf, dass ein neuronales Netz **keine Garantie auf das bestmögliche Ergebnis** liefert. So findet ein Backpropagation-Netz möglicherweise nur ein lokales Fehlerminimum, während eine andere Kombination der Gewichte ein weit besseres Ergebnis erzielen würde.

Quellen: [Lossau92], [Callan03], [SchHaGa90]

4.3 Anwendungsbereiche

Ein wichtiges Einsatzgebiet von neuronales Netzen ist die Mustererkennung und -zuordnung. Wie beschrieben, kann ein Netz nach entsprechendem Training auch mit zuvor unbekannten Mustern zuverlässig umgehen und sie oft richtig einordnen. Dies ist etwa im Bereich der Schrift- und Spracherkennung hilfreich, um ein Programm nicht explizit auf Handschrift und Stimme eines bestimmten Benutzers abstimmen zu müssen. Auch für Spracherzeugung wurden neuronale Netze schon erfolgreich angewandt [Lossau92, S. 109].

Bei bestimmten Sensoren, wie etwa Geruchssensoren, ist eine Mustererkennung ebenso wichtig wie bei der Gesichts- oder Spracherkennung.

Mithilfe geeigneter Trainingssätze kann ein Netz so trainiert werden, dass es eine bestimmte Funktion approximiert. Weiterhin ist es dazu in der Lage, eine Funktion in einem unbekannten Wertebereich fortzuführen, was man sich etwa bei der Vorhersage von Aktienkursen zunutze macht. Ein Netz wurde mehrere Jahre lang mit dem Dollar-Mark-Wechselkurs, einigen Zinssätzen und Inflationsraten trainiert und war

daraufhin in der Lage, den Kurs besser vorherzusagen als Finanzexperten [Lossau92, S. 118]. Auch für Frühwarnsysteme (z.B. Tsunami-Warnung), Wettervorhersagen und sogar in der Medizin (Überwachung von Atmung und Herzschlag eines Patienten) lassen sich neuronale Netze anwenden.

Neuronale Netze können auch für die Steuerung von Robotern eingesetzt werden. Das Netz enthält in einem solchen Fall Daten, die seine Sensoren liefern (etwa das Bild einer Kamera), als Eingabe und erzeugt als Ausgabe Anweisungen für die Steuerung von Motoren. Das Training kann durch Nachahmung eines Menschen oder durch bestärkende oder unüberwachte Lernverfahren erfolgen. Neuronale Netze stellen schließlich auch effektive Autopiloten in der Luft- und Raumfahrt dar.

Quellen: [Lossau92], [SchHaGa90], [Callan03]

5 Implementierung

5.1 Ziel

Um die Funktionsweise von neuronalen Netzen überprüfen und besser nachvollziehen zu können, nahm ich mir vor, eine einfache Bibliothek[7] zu schreiben, die mehrschichtige vorwärts verkettete Netze und deren Training per Backpropagation implementiert, und diese anhand einer Beispielanwendung zu testen.

Als Ziel hatte ich trainierbare virtuelle Agenten[8] vor Augen, denen eine bestimmte Verhaltensweise antrainiert werden kann. Anschaulicher ausgedrückt: eine Simulation, in der man „Ameisen“ beibringen kann, ihr Futter zu sammeln.

5.2 Bibliothek

Die verwendete Programmiersprache ist **Ruby**, eine relativ junge objektorientierte Skriptsprache[9], die 1993 von dem Japaner Yukihiro Matsumoto entworfen wurde. Ich entschied mich für diese Sprache, da sie eine schnelle Softwareentwicklung ermöglicht, leicht schreib- und lesbar sowie erweiterbar ist und einer Open-Source-Lizenz unterliegt.

[7] Ein Programmodul zur Lösung einer bestimmten Aufgabe

[8] Bezeichnung von künstlichen Wesen in der KI-Forschung

[9] Objektorientierung bedeutet, die Funktionalität eines Programms in verschiedene Klassen aufzuteilen, die reale oder abstrakte Objekte nachbilden können. In diesem Fall wären Objekte vom Typ *Ameise* oder vom Typ *Neuron* möglich. Eine Skriptsprache wird im Gegensatz zu einer Compilersprache nicht in ein Programm übersetzt, sondern Zeile für Zeile eingelesen und ausgeführt.

Das Programm basiert auf einer selbstgeschriebenen KNN-Bibliothek, die es ermöglicht, Netze zu erstellen und zu trainieren. Diese Bibliothek kann in das Hauptprogramm eingebunden und dort benutzt werden. Durch Datenkapselung braucht der Anwender nur drei Befehle der Schnittstelle zu kennen: Der Befehl

```
net = Net.new([2,4,3])
```

erzeugt beispielsweise ein Netz mit 2 Eingabeneuronen, einer inneren Schicht mit 4 Neuronen und 3 Ausgabeneuronen (die Werte sind willkürlich gewählt). Das Netz kann nun beliebig trainiert werden:

```
net.train([3.14,42], [0,0,1])
```

Das erste Argument stellt die Eingabe dar, das zweite die gewünschte Ausgabe. Das Netz führt nun Backpropagation-Lernen durch. Wurde das Netz genügend trainiert, kann per

```
output=net.eval([1,2])
```

die Ausgabe für ein gegebenes Muster bestimmt werden (in diesem Fall könnte die Ausgabe etwa `[0.168,0.171,0.829]` sein).

Intern ist die Bibliothek in Klassen aufgebaut, die die Struktur des Netzes widerspiegeln: Die Klasse `Net` setzt sich aus mehreren Instanzen der Klasse `Layer` zusammen, welche wiederum eine bestimmte Anzahl von Neuronen (`Neuron`) enthalten. Die Neuronen zweier angrenzender Schichten sind durch Verbindungen der Klasse `Link` verbunden. Eine genauere Beschreibung der Funktionsweise findet sich im Quelltext und der Dokumentation (siehe auch Anhang B.2).

5.3 Aufbau

Das Netz einer Ameise braucht zwei Eingänge und zwei Ausgänge: Die relative Position des nächsten Zieles dient als Eingabe, als Ausgabe werden die Bewegungsgeschwindigkeit sowie eine Drehung generiert.

Die Ameisen benötigen kein Erinnerungsvermögen, denn sie sollen in gleichen Situationen gleich reagieren; ich entschied mich daher für die Verwendung eines vorwärts verketteten Netzes.

Für die Bestimmung der Netzgröße gibt es keine allgemeinen Regeln [LaeCl01, S. 217], es gilt jedoch der Grundsatz, die Netze möglichst klein zu halten, um Abstraktion zu ermöglichen und Überanpassung zu vermeiden (siehe dazu Abschnitt 4.2). Eine angemessene Größe kann durch simples Ausprobieren herausgefunden werden. In diesem Fall ist eine innere Schicht unbedingte Voraussetzung, 5 Neuronen haben sich als günstige Anzahl herausgestellt.

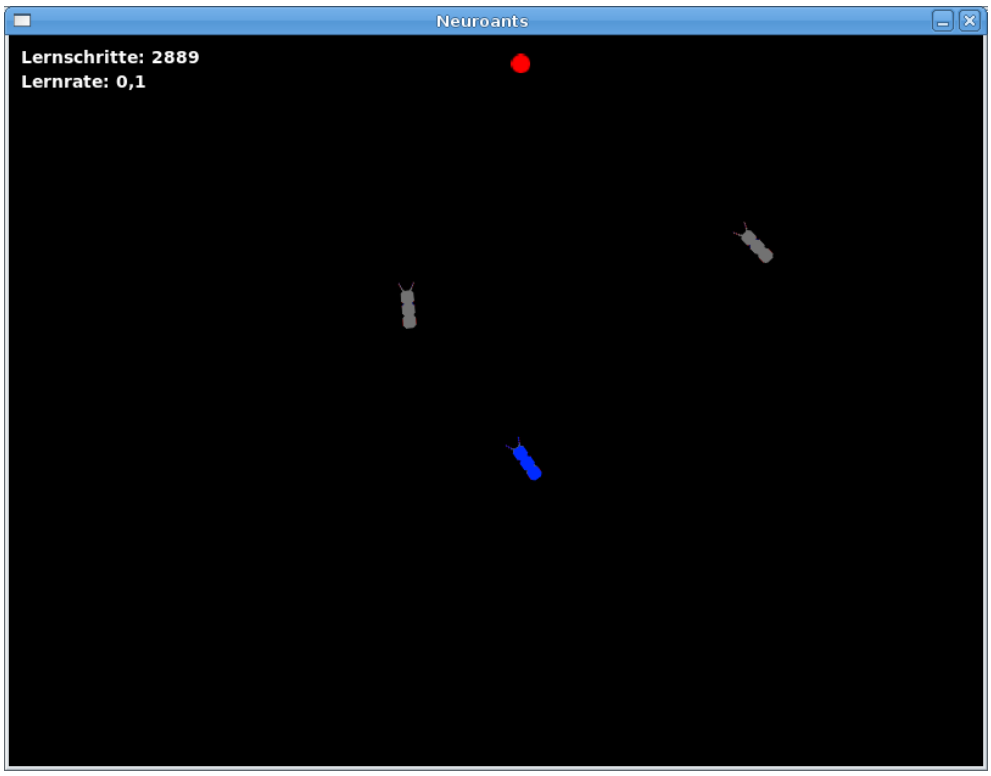


Abbildung 8: Screenshot des Programms

5.4 Funktionsweise

Als Lernmechanismus bot sich das Backpropagation-Verfahren an. Die Trainingssätze werden erzeugt, indem ein menschlicher Benutzer das erwünschte Verhalten „vormacht“.

Meine Grundidee war dabei, den Benutzer eine der Ameisen steuern zu lassen und dessen Vorwärts- und Drehbewegung dann als perfektes, erwünschtes Verhalten anzunehmen. Die Netze werden danach auf Ein- und Ausgaben des menschlichen Benutzers trainiert und bringen so nach einiger Zeit die richtigen Ergebnisse hervor.

5.5 Resultat

Anders als bei einer fest einprogrammierten Verhaltensweise ist es mit neuronalen Netzen möglich, den Ameisen etwas beizubringen *während das Programm läuft* oder sie sogar „umzutrainieren“. So können sie etwa lernen, das Futter vorwärts- oder auch rückwärtslaufend einzusammeln, es zu umkreisen oder sich einfach auf der Stelle zu drehen. Ich stellte fest, dass sie auch differenziertere Verhaltensweisen lernen können, beispielsweise nur Rechtskurven zu benutzen oder sich in Schleifen ihrem Ziel zu nähern.

Exemplarisch sei in Abbildung 9 eine Fehlerkurve aufgeführt, die beim Training einer Ameise auf das Sammeln von Futter entstand.

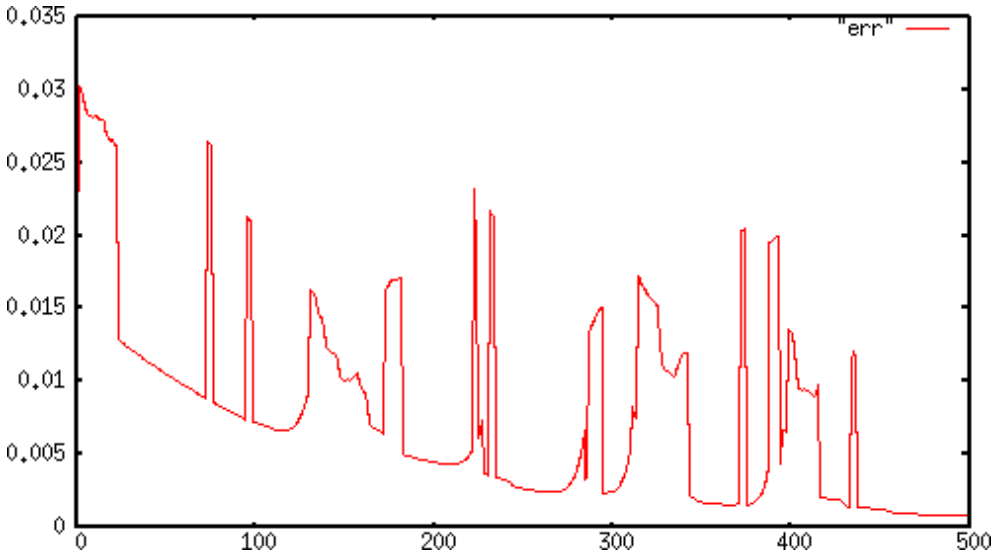


Abbildung 9: quadrierter Fehler

Dargestellt ist $\sum_j (t_j - o_j)^2$, die Summe der quadrierten Fehler der einzelnen Ausgabeneuronen und ihre Entwicklung im Laufe von 500 Lernschritten (die Fehler werden quadriert, um zu vermeiden, dass sich positive und negative Fehler aufheben).

Das Rauschen führe ich auf eigene Ungenauigkeiten bei der Steuerung und Sprünge im Lernverhalten des Netzes zurück. Dennoch ist eine generelle Tendenz des Fehlers gegen Null zu erkennen.

Die angesprochene Abhängigkeit von Fehlern des Benutzers kann durch eine geringere Lernrate vermindert werden (im Beispiel wurde $\eta = 0,25$ gesetzt). In einem solchen Fall werden die Gewichte in jedem Lernzyklus nur schwach angepasst, was die resultierende Steuerung der Ameisen genauer, das Lernen jedoch langsamer macht. Bei hohen Lernraten wird die Steuerung des Benutzers sofort imitiert.

Diese Beispielanwendung ist einfach und eignet sich eher als Demonstration von künstlicher Intelligenz in Computerspielen. Die Funktionsweise ist jedoch die gleiche wie in größeren Anwendungen. Ich bin sicher, man könnte einen mobilen Roboter mit einem neuronalen Netz versehen, ihn mit einer Kamera ausstatten und ihn darauf trainieren, farblich markante Objekte anzusteuern. In diesem Fall würde sich anbieten, für jeden Pixel des Kamerabildes ein Eingabeneuron vorzusehen und beispielsweise die Intensität des gesuchten Farbtons als Eingabe zu wählen. Auch komplexere Anwendungsmöglichkeiten wären denkbar, etwa ein Fahrzeug, das einer Straße folgt, oder ein Roboterarm, der Bewegungen erlernt.

Eine Erweiterung des Programms wäre ebenso denkbar. Indem man den Ameisen weitere Eingabeinformationen gibt, etwa die relative Po-

sition der nächsten anderen Ameise oder eines „Feindes“, könnte man komplexere Verhaltensweisen ermöglichen.

6 Fazit und Ausblick

Bei den Vorbereitungen zu dieser Arbeit habe ich neuronale Netze als beeindruckende Technik zur Erzeugung künstlicher Intelligenz kennengelernt. Sie sind immer dann hilfreich, wenn das Lösen eines Problems mit herkömmlicher Programmierung entweder zu aufwändig oder zu unflexibel ist. Ich werde auch weiterhin in diesem Bereich experimentieren, vor allem in der Richtung des bestärkenden Lernens.

Ich konnte die Funktionsweise der verbreitetsten Netzarchitektur vorstellen und diese in einer Simulation umsetzen. Außerdem habe ich einen Einblick in die schon heute möglichen Anwendungen gegeben sowie Stärken und Schwächen zusammengefasst.

Beim Ausprobieren meiner Implementierung von neuronalen Netzen musste ich feststellen, dass die Lerngeschwindigkeit bei einigen tausend Verbindungen so stark abnimmt, dass das Netz fast unbrauchbar wird. Bei der momentanen rapiden Verbesserung der Rechengeschwindigkeit von Computern ist jedoch abzusehen, dass in Zukunft immer größere, komplexere Netze simuliert und entsprechend anspruchsvolle Aufgaben gelöst werden können.

Hugo de Garis plante um die Jahrtausendwende ein Netz aus knapp 75 Millionen Neuronen zur Steuerung eines Roboters und hoffte, dieser würde etwa so intelligent wie eine Katze.[10] Auch wenn das Projekt aus finanziellen Gründen aufgegeben wurde, zeigt es doch die Möglichkeiten, die künstliche neuronale Netze künftig eröffnen könnten.

[10] Hugo de Garis - Utah-Brain Project. 23.10.2003.
http://www.setiai.com/archives/000026.html. Stand: 14.03.2008

A Literatur

[Callan03] Callan, Robert: Neuronale Netze im Klartext. 1. Aufl., München 2003

[LaeCl01] Lämmel, Uwe/Cleve, Jürgen: Künstliche Intelligenz. 1. Aufl., Wismar 2001, S. 171-289

[Lossau92] Lossau, Norbert: Wenn Computer denken lernen. Neuronale Netzwerke. 1. Aufl., Berlin 1992

[MiSch97] Miriam, Wolfgang/Scharf Karl-Heinz et al.: Biologie heute SII. Ein Lehr- und Arbeitsbuch. 3. Aufl., Hannover 1997

[SchHaGa90] Schöneburg, Eberhard/Hansen, Nikolaus/Gawelczyk, Andreas: Neuronale Netzwerke. Einführung, Überblick und Anwendungsmöglichkeiten. 2. Aufl., München 1990

B Hinweise zum Datenträger

Das Programm ist direkt von der CD-ROM lauffähig, es empfiehlt sich jedoch, die Dateien auf die Festplatte zu kopieren, um die Ausführungsgeschwindigkeit zu erhöhen. Es wurde auf die Lauffähigkeit unter Windows XP überprüft.

B.1 Übersicht über die enthaltenen Dateien

- `run.bat` startet das Programm
- `neuroants.rb` ist der Quelltext des Hauptprogramms
- `net.rb` stellt die KNN-Bibliothek dar
- `engine.rb` enthält wiederverwendbare, allgemeine Hilfsklassen
- `COPYING.txt` enthält die freie Lizenz des Programms, die GNU General Public License Version 3
- `ruby.exe` ist der Interpreter der Programmiersprache Ruby
- `gosu.dll` ist die verwendete Multimediabibliothek
- der Ordner `media\` enthält im Programm verwendete Grafiken
- `doc\` enthält die Dokumentation, beginnend mit `index.html`
- `quellen\` enthält die Internet- und Zeitschriftenquellen, auf die im Text verwiesen wurden

B.2 Bedienung und Dokumentation des Programms

Die blaue Ameise ist die vom Menschen gesteuerte, die grauen sind künstliche Intelligenzen. Das „Futter“ wird durch den roten Punkt dargestellt.

Am oberen Bildschirmrand werden die Anzahl der Lernzyklen sowie die Lernrate angezeigt.

Die Steuerung erfolgt über die Pfeiltasten. Die Ameisen werden dabei nur trainiert, wenn der Benutzer eine Bewegung vornimmt, um zu vermeiden, dass sie „Stillstehen“ erlernen. Escape beendet das Programm, Bild Hoch und Bild Runter erhöhen/verringern die Lernrate.

Es ist für die Ameisen möglich, sich außerhalb der Fensterränder zu bewegen, um ihren Bewegungspielraum nicht einzuschränken.

Die Dokumentation zum Programm kann in der Datei `doc/index.html` eingesehen werden, es handelt sich dabei um eine automatisch generierte Zusammenstellung der Kommentare aus dem Quelltext.[11] Ein Klick auf `[Source]` unter jeder Methode zeigt den jeweiligen Quelltext mit weiteren Erläuterungen. In der Box `Classes` oben in der Mitte können die verschiedenen Klassen ausgewählt werden.

Natürlich kann auch direkt der Quelltext in den `*.rb`-Dateien mit einem beliebigen Texteditor gelesen (und verändert) werden, unter Windows ist dabei etwa der „Editor“ geeignet. So können die einzelnen Parameter wie Ameisenanzahl, Bewegungsgeschwindigkeit oder die Netzgröße angepasst werden, dabei ist die Datei `neuroants.rb` entscheidend.

B.3 Syntax

Zuletzt noch einige Anmerkungen zur Syntax von Ruby, die das Lesen des Quelltextes erleichtern sollen. Dabei soll auf die bedeutendsten Unterschiede zu C-ähnlichen Sprachen eingegangen werden.

- Klammern um Argumente können bei Eindeutigkeit weggelassen werden. Beispiel: `print "Hello World"`.
- Ruby besitzt eingebaute Arrays, die die Syntax `[element1, element2, ...]` besitzen.
- In Ruby ist komplett objektorientiert, auch Zahlen oder Strings sind Objekte. So wird z.B. `"Neuronale Netze".reverse.downcase` möglich. (Ausgabe: `"ezten elanoruen"`)

[11] Da es sich um UTF-8-codierten Text handelt, ist bei Verwendung des Internet Explorers 6 eine Umstellung erforderlich: Ansicht → Codierung → (Mehr →) Unicode.

- `@` vor Variablen deklariert diese als Attribute, sie werden dann dauerhaft der Klasse zugeordnet.
- Ausdrücke wie

  ```
  array.each do |i|
      ...
  end
  ```

  oder in der Kurzschreibweise `array.each{|i| ... }` iterieren über `array`, wobei der Iterator mit der Bezeichnung `i` versehen wird. Mit diesen sogenannten Blöcken werden elegante Konstruktionen wie `5.times{|i| ... }` oder `array.sort_by{|s| s.size}` möglich. Nächste Entsprechung in anderen Sprachen wäre die `for`-Schleife.
- Negative Indizes hinter Arrays (`array[-n]`) beziehen sich auf das n-te Element *von hinten*.
- `attr_reader` und `attr_accessor` definieren in Klassen Methoden zum Lesen bzw. zum Lesen und Schreiben von Attributen.
- In einer Methode ist der letzte R-Wert gleichzeitig Rückgabewert. Gültiges Beispiel:

  ```
  def double n
      n*2
  end
  ```

C Verwendete Hilfsmittel

Geschrieben mit dem Texteditor Vim 7.1.56[12] unter dem Betriebssystem GNU/Linux, Kernel 2.6.22-14[13], gesetzt mit dem Textsatzprogramm LaTeX unter Verwendung der Grafikbibliothek PGF/TikZ 2.00[14] und des Vektor-Zeichenprogramms Inkscape 0.45.1[15].

Alle Grafiken wurden selbst erstellt, wobei sich einige an Abbildungen aus der verwendeten Literatur anlehnen.

Das Programm basiert auf der Programmiersprache Ruby 1.6.8[16] und der Multimedia-Bibliothek gosu 0.7.8.5[17]. Die Klassenstruktur orientiert sich an einem Vorschlag aus [LaeCl01].

[12] http://www.vim.org/
[13] http://www.gnu.org/
[14] http://sourceforge.net/projects/pgf/
[15] http://inkscape.org/
[16] http://www.ruby-lang.org/de/
[17] http://code.google.com/p/gosu/

D Veröffentlichungseinverständnis

Hiermit erkläre ich, dass ich damit einverstanden bin, wenn die von mir verfasste Facharbeit der schulinternen Öffentlichkeit zugänglich gemacht wird.

Sebastian Morr

E Versicherung der selbstständigen Erarbeitung

Hiermit versichere ich, dass ich die Arbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Facharbeit, die im Wortlaut oder im wesentlichen Inhalt aus anderen Werken (auch aus dem Internet) entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe. Verwendete Informationen aus dem Internet sowie Artikel aus Zeitungen und Zeitschriften sind dem Lehrer vollständig auf CD-ROM zur Verfügung gestellt worden.

Sebastian Morr