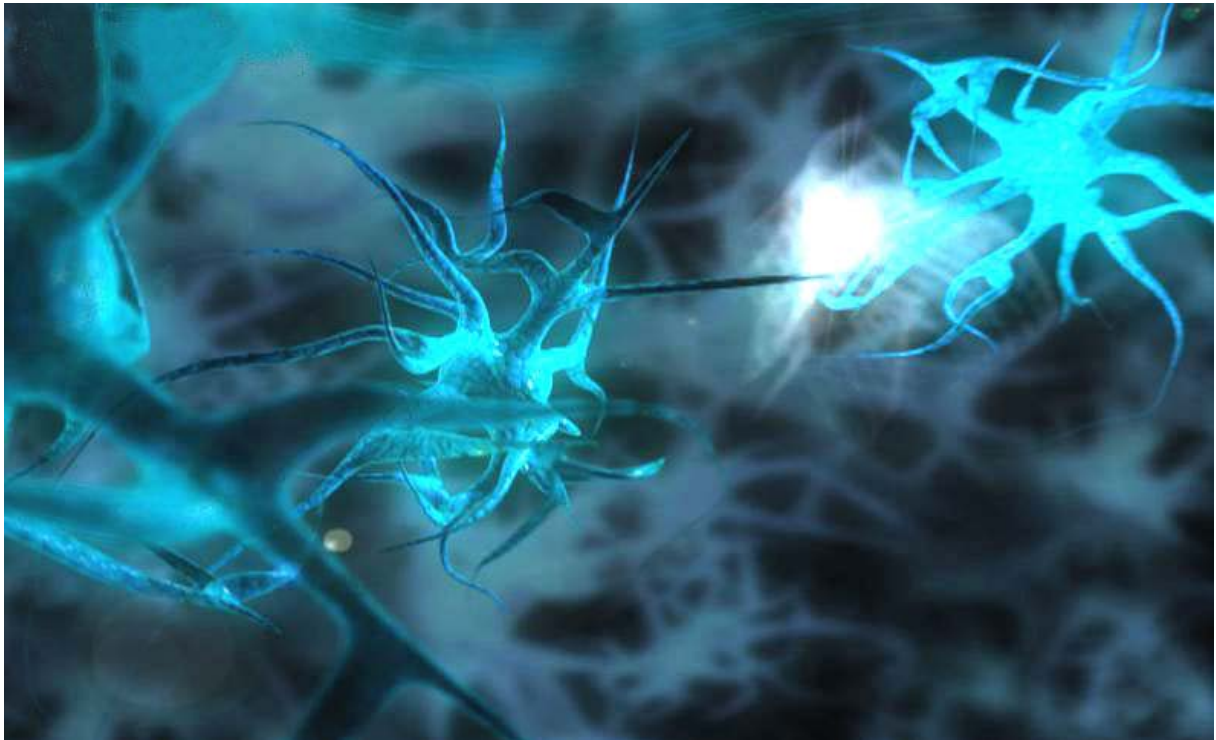


Anwendung von „Neuronalen Netzen“ bei der Programmierung von künstlichen Spielern



[Bild 1]

Thema 10
Seminar „Spiele-KI“
bei Prof. Dr. Iwanowski
von Niels Schwennicke (mi5230)

Gliederung

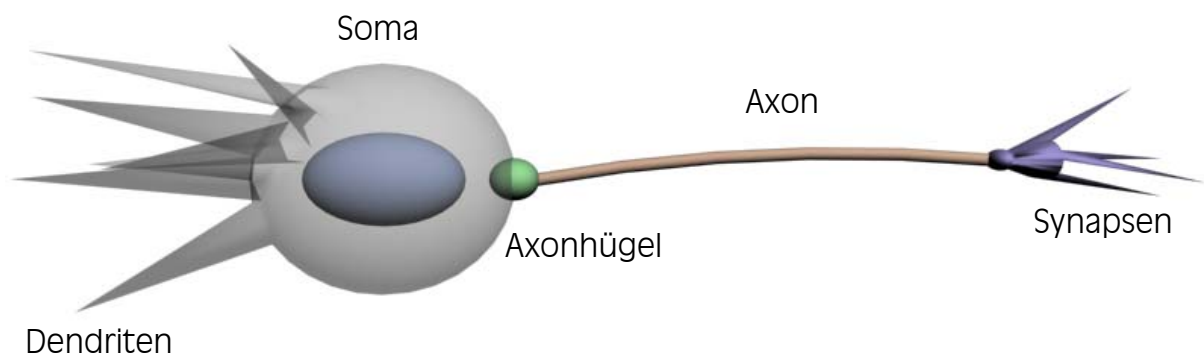
1.	NN: Biologisches Vorbild	<i>S. 3</i>
1.1	Aufbau: Neuron	<i>S. 3</i>
1.2	Funktionsweise: Neuron	<i>S. 4</i>
1.3	Das Nervensystem	<i>S. 4</i>
1.4	Lernen	<i>S. 5</i>
2.	KNN	<i>S. 6</i>
2.1	Aufbau: KN	<i>S. 6</i>
2.2	Funktionsweise: KN	<i>S. 7</i>
2.3	Topologien	<i>S. 9</i>
2.4	Lernen	<i>S. 10</i>
3.	Vergleich: NN vs. KNN	<i>S. 13</i>
4.	KNN: Aufgaben in Spielen	<i>S. 14</i>
5.	Design unseres KNNs	<i>S. 14</i>
6.	KNN in Games	<i>S. 17</i>
7.	Fazit	<i>S. 18</i>
8.	Anhang	<i>S. 19</i>
4.1	Literaturverzeichnis	<i>S. 19</i>
4.2	Bildverzeichnis	<i>S. 19</i>

1. NN: Biologisches Vorbild

Die Anwendung der Technik von **Künstlichen Neuronale Netze (KNN)** in Spielen und allgemein, basiert auf einem grundsätzlichen Verständnis der Thematik, die ihre Wurzeln in der Biologie hat. Aus diesem Grund wird im Folgenden der Ursprung, also ein Blick auf **Natürliche** Neuronale Netze geworfen. Zunächst wird hierbei der Aufbau einer menschlichen Nervenzelle (Neuron) betrachtet.

1.1 Aufbau: Neuron

Die natürliche Nervenzelle besteht prinzipiell aus folgenden wesentlichen Bestandteilen:



[Bild 2]

Die Bestandteile eines natürlichen Neurons lassen sich wie folgt umreißen:

- Dendriten: baumartige Verästelungen, die an dem Neuron hängen und eine Schnittstelle von Neuron **n-1** zu **n** bilden
- Soma: Zellkörper des Neurons, welcher den Zellkern enthält
- Aktionshügel: generiert bei überschweelliger Erregung das Aktionspotential, was einen neuronalen Impuls darstellt
- Axon: schlauchartiges Gebilde, welches neuronale Impulse vom Axonshügel zu den Synapsen leitet
- Synapsen: Schnittstelle vom Neuron **n** zu Neuron **n+1**; hier findet die Übertragung von einem neuronalen Impuls auf die empfangenden Neuronen statt

1.2 Funktionsweise: Neuron

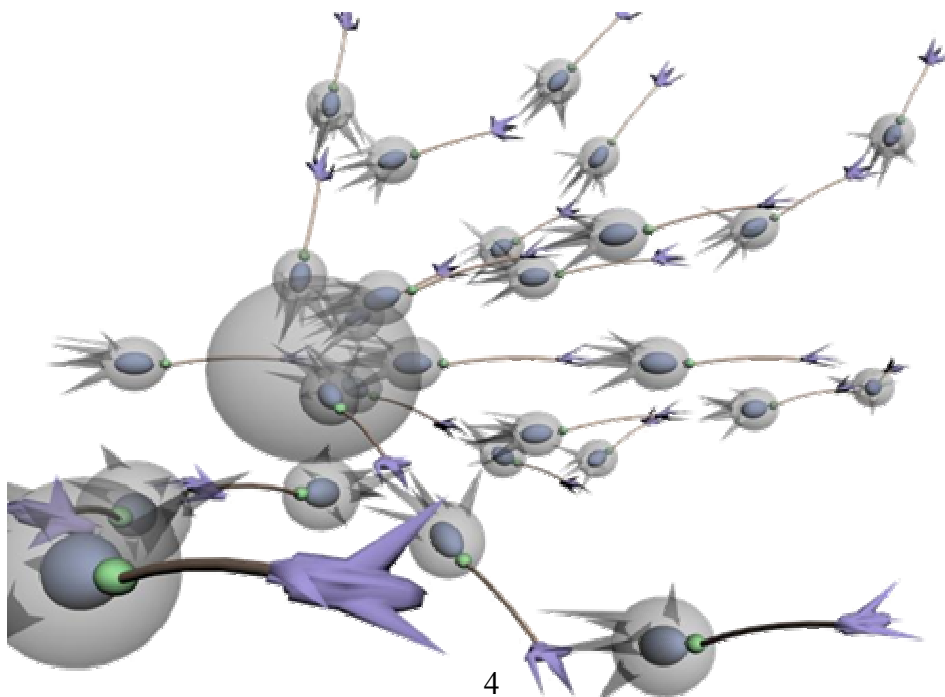
Nervenzellen besitzen ein Membran-Potential, d.h. ein elektisches Ladungsgefälle (Spannung) zwischen Zellinneren, getrennt durch die Zellhaut (Membran) und dem Zelläußeren. Die Ladungsverhältnisse im Ruhezustand (konstante Spannung zwischen den Polen) nennt man **Ruhepotential** und die Ladungsverhältnisse während einer Informationübertragung **Aktionspotential**.

Neuronen können ausgelöst durch einen Reiz (Stimulus) ihr Membranpotential ändern, d.h. Wechsel von Ruhepotential zum Aktionspotential. Diese Potentialänderung entsteht, wenn bspw. Informationen von Sinneszellen (z.B. Sehnerv) oder anderen Neuronen über die Synapsen auf die Dendriten des Neurons n übertragen werden. Diese eintreffenden elektischen Impulse werden von Neuron n „verarbeitet“. Sind die Impulse stark genug, d.h. überschreiten sie einen gewissen Schwellenwert, so setzt das Neuron n über den Axonshügel ein ein Aktionspotential frei, welches über das Axon und die Snapsen auf eine Neuron $n+1$ übertragen wird. Die Synapsen des Neurons n sind mit den Dendriten des Neuons $n+1$ verbunden.

Somit ist die grundsätzliche Funktionsweise erklärt und wir wollen nun den Verbund vieler Neuronen im sogenannten Nervensystem betrachten.

1.3 Das Nervensystem

Durch die eben behandelte Erregbarkeit und Leitfähigkeit sind Neuronen in der Lage, Impulse selektiv weiterzuleiten, und in Verbund befähigt, Informationen zu verarbeiten und ggf. zu speichern. Der Verbund vieler Neuronen wird als Nervensystem bezeichnet. Bei allen Wirbeltieren liegt das zentrale Nervensystem auf der Rückseite des Körpers. Es besteht aus Gehirn und Rückenmark, die mit dem peripheren Nervensystem (Verbindungen zu den Organen) kommunizieren. Allein das Gehirn besteht aus 100 Milliarden Neuronen.



[Bild 3]

Unser Gehirn ist also ein riesiges Netzwerk. So sind bei 100 Milliarden Nervenzellen mit jeweils 10.000 Verbindungen (pro Neuron) die Kombinationsmöglichkeiten nahezu unbegrenzt. Diese Verzweigungen werden durch Lernen, bzw. durch Training des Gehirns, dynamisch angepasst und einzelnen Verbindungen ggf. geschwächt oder auch verstärkt.

1.4 Lernen

Wenn wir etwas ausführen, an eine Sache denken oder lernen ist entscheidend, welche Nervenzellen dabei gemeinsam elektrische Impulse „abfeuern“, also Aktionspotentiale generieren.

Prinzipiell lernt das Gehirn durch Training, also bspw. aktives Memorieren beim Vokabeln lernen. Der gelernte Zusammenhang wird durch Aufbau oder auch „Redesign“ von Verbindungen zwischen den Neuronen erstellt. Zusätzlich werden die Aktivierungsschwellen der einzelnen Neuronen ggf. angepasst.

2. KNN

Im letzten Kapitel haben wir einen Eindruck gewonnen, wie das biologische neuronale Netz funktioniert und strukturiert ist. Nun wollen wir uns den künstlichen Neuronale Netz zuwenden. Hier werden wir zunächst mit den Aufbau eines einzelnen künstlichen Neurons beginnen.

2.1 Aufbau: KN

Das künstliche Neuron besteht im Wesentlichen aus folgenden Bestandteilen:

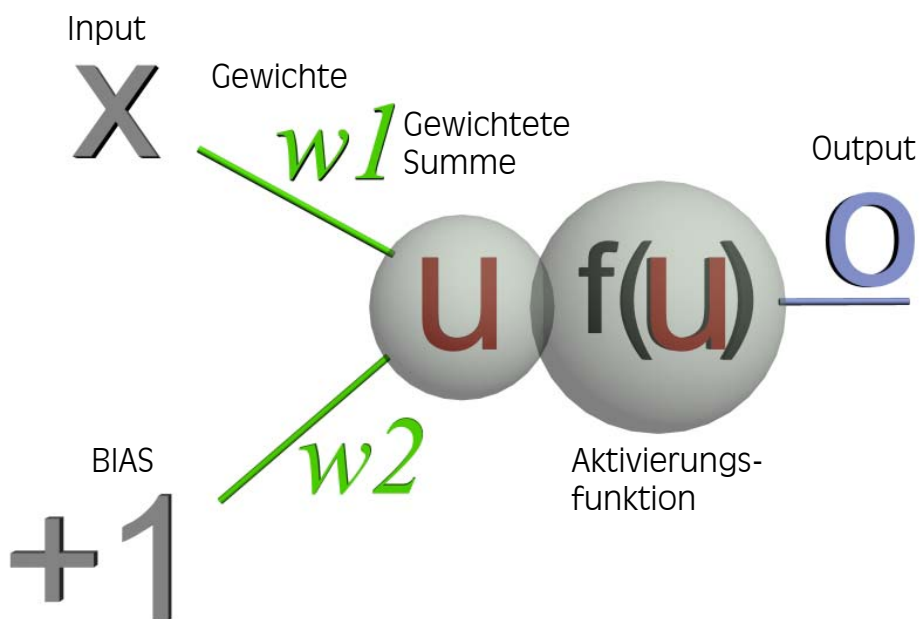


Bild 41

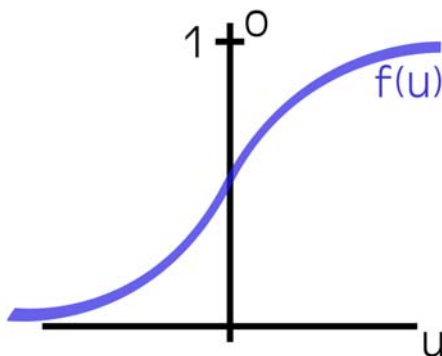
Die Bestandteile lassen sich wie folgt beschreiben:

- Input bzw. Input-Vektor, z.B. $x = (x_1)$: stellt die Werte, die als Eingabe in unser künstliches Neuron fließen, dar
- BIAS: wird als zusätzliche Komponente an unseren Input gehängt und hält meistens den Wert $+1$ oder -1 ; also $x = (x_1, x_2) = (x_1, \text{BIAS})$
- Gewichte bzw. Gewichtsvektor, z.B. $w = (w_1, w_2)$: die Gewichte gilt es durch Training einzustellen, damit das künstliche Neuron optimale Ergebnisse liefert
- Gewichtete Summe, $u = \text{sum}(x_i * w_i)$, mit $i = 1 \dots n$: verknüpft intern die Komponenten des Inputs mit den zugehörigen Komponenten des Gewichtsvektors

- Aktivierungsfunktion, $f(u)$:
verarbeitet in einer vordefinierten Funktion die zuvor ermittelte gewichtete Summe; sie kann bspw. die Form einer **Schritt**-Funktion, **Logistik**-Funktion, etc. haben

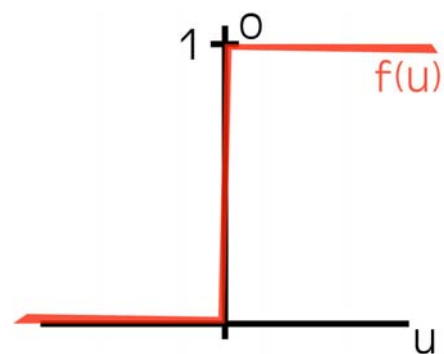
Logistik-Funktion:

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



Schritt-Funktion:

$$f(u) = \begin{cases} 1; & u > 0 \\ 0; & u \leq 0 \end{cases}$$



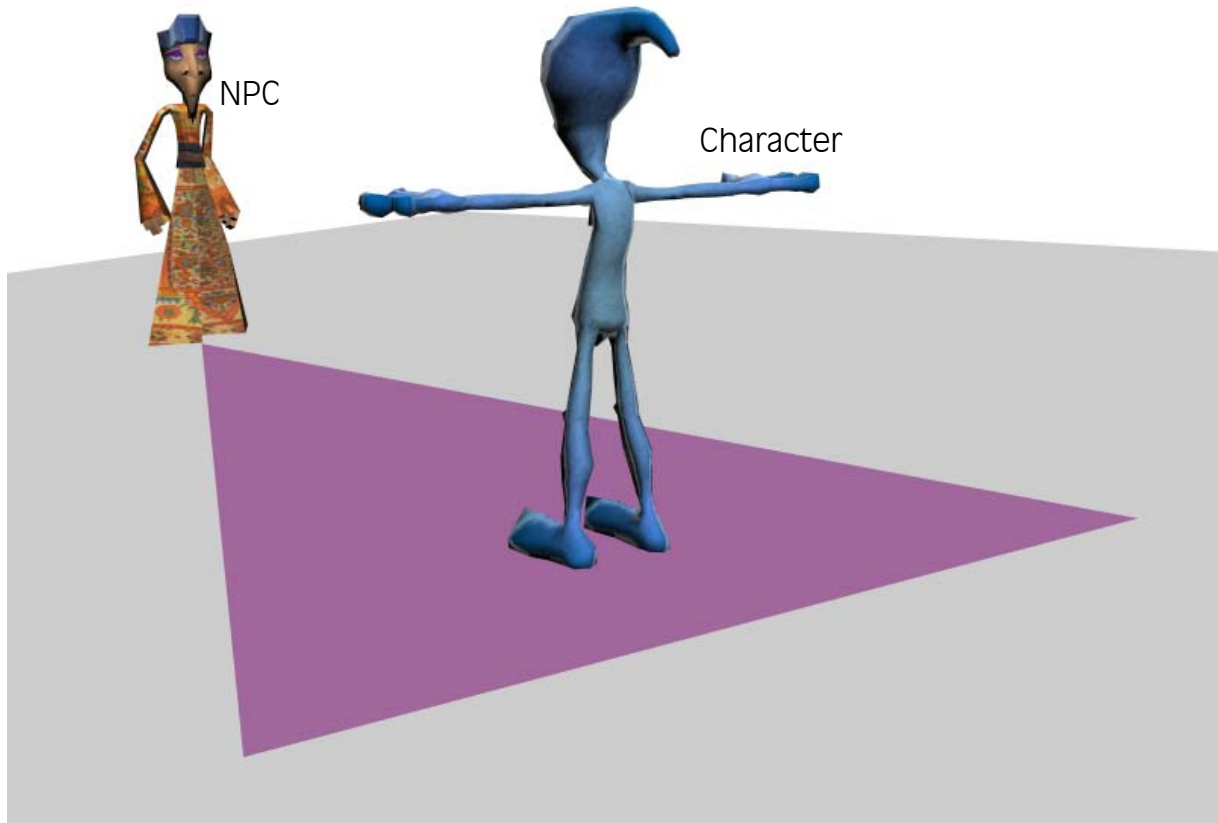
[Bilder 5]

- Output:
Ergebnis der Aktivierungsfunktion und somit der neuronalen Verarbeitung des Inputs

2.2 Funktionsweise: KN

Wir wollen nun an einem Beispiel die Funktionsweise eines einfachen bereits trainierten künstlichen Neurons betrachten.

Wir nehmen beispielhaft die Situation eines NPCs (Non Person Character) in einem Online-Rollenspiel, der einen anderen Character trifft und diesen klassifizieren soll. Die Entscheidung, die der NPC zu treffen hat, ist: „Handelt es sich um einen Freund, also ein anderer NPC oder um einen Feind, nämlich ein menschlicher Spieler?“



[Bild 6]

Als Input x wird die Größe des dem NPC gegenüber stehenden Characters genommen, wobei dieser Wert vor der eigentlichen Verarbeitung seitens des künstlichen Neurons (KN) normalisiert wird. Der Charakter ist **0.9** Längeneinheiten groß.

Die Gewichte sind gegeben, z.B. durch vorhergehendes Training ermittelt und umfassen die Komponenten **w1** und **w2**.

Wir nehmen als Aktivierungsfunktion die **Schritt**-Funktion, die uns als Output die Werte „0“ was FALSE entspricht, bzw. „1“ also TRUE liefert. TRUE heißt aus der Sicht unseres NPCs, das der Character ihm gegenüber auch ein NPC, also ein Freund ist; FALSE entsprechend.

Die Verarbeitung sieht nun, wie folgt aus:

- Input-Vektor mit BIAS:
 $x = (x_1, x_2) = (x_1, \text{BIAS}) = (0.9, +1)$
- Gewichte (gegeben):
 $w = (w_1, w_2) = (-1, 0.8)$
- Gewichte Summe:
 $u = \text{sum}(x_i * w_i) = w_1 * x_1 + w_2 * x_2 = 0.9 * (-1) + 1 * 0.8 = -0.1$
- Aktivierungsfunktion, hier Schritt-Funktion:
 $f(u) = \text{step}(u) = \text{step}(-0.1) = 0$
- Output:
 $o = 0 >>> \text{FALSE}$

Das trainierte künstliche Neuron hat nun den, dem NPC gegenüberstehenden Character, als Feind klassifiziert. Es hat (anders ausgedrückt) die Frage: „*Handelt es sich um einen Freund, also ein anderer NPC?*“ mit **FALSE** beantwortet. Das impliziert, dass es sich um den menschlichen Spieler, also dem Feind handelt.

(Als simple Vorgabe wurde ihr angenommen, dass alle anderen NPCs aus der Sicht des unseres NPCs unter 0.8 Längeneinheiten sind. Feinde, also die menschlichen Spieler im Online-Rollenspiel, seien jedoch größer als 0.8 LE.)

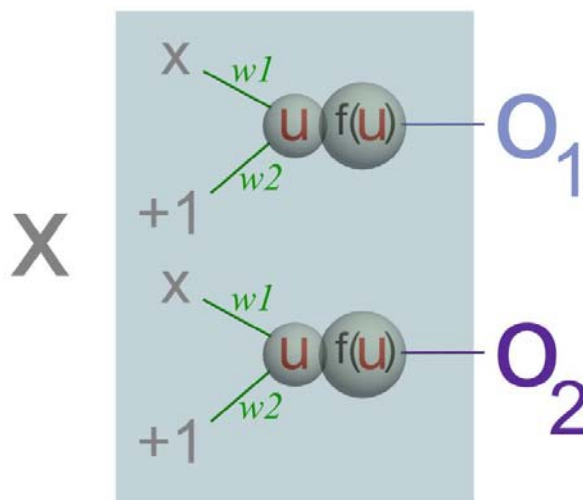
Wir haben nun Funktionsweise eines einfachen künstlichen Neurons anhand eines Beispiels nachvollzogen. Um komplexere Problemstellungen lösen zu können, reicht meistens ein einzelnes künstliches Neurons nicht aus. Hierfür werden bspw. Neuronen hintereinander geschaltet, die dann ein KNN bilden. Diese unterschiedlichen Anordnungen von Neuronen in einem Netz werden als Topologien bezeichnet.

2.3 Topologien

Grundsätzlich unterscheidet man zwischen „**Single-Layer**“-KNNs, also einschichtige künstliche neuronalen Netze und „**Multi-Layer**“-KNNs, also den mehrschichtigen Netzen.

Single-Layer KNNs:

Wie man im *[Bild 7]* erkennt besteht das KNN aus einer Schicht a zwei Neuronen. Der Input x wird in beide Neuronen gegeben und es werden zwei Output generiert o_1 und o_2 .

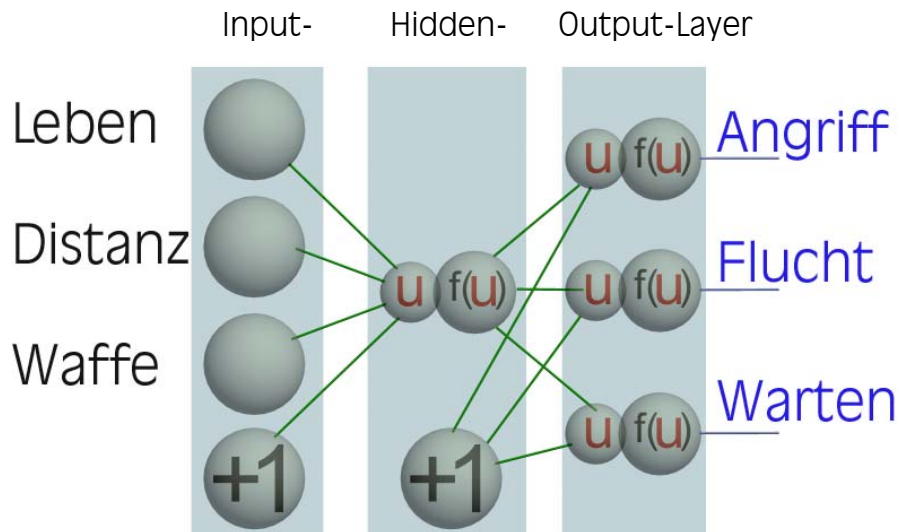


[Bild 7]

Single-Layer KNNs werden für Lösung von linearen Problemen herangezogen; bei komplexeren Aufgaben muss man zusätzliche Schichten einführen.

Multi-Layer KNNs:

Für komplexere Problemlösungen sind in Spielen **3-Layer** Netze typisch, welche folglich aus 3 Schichten bestehen: **Input-**, **Hidden-** und **Output-Layer**. Der Input-Layer (Schicht) hat keine verarbeitende Aufgabe und schiebt den Input einfach weiter zur versteckten Schicht, dem Hidden-Layer. Dort geschieht die erste rechnerische Verarbeitung. Der Output der versteckten Schicht wird weiter in den Output-Layer „gefüttert“, wo der finale Output des KNNs generiert wird.



[Bild 8]

An dieser Stelle sei erwähnt, dass bei der Anwendung von KNNs in Spielen meist sog. **Feedforward**-Netze verwendet werden. „Feedforward“ meint eigentlich nur, dass die Daten von links nach rechts laufen, also von Input- über den Hidden- zum Output-Layer; sie werden „nach vorne gefüttert“. Bei dieser Vorgehensweise wird auf rekursive Laufwege verzichtet.

2.4 Lernen

Prinzipiell ist zu sagen, dass Lernen bei KNNs immer ein Prozess der optimalen Gewichtsfindung ist. Sind die Gewichte erst einmal optimal eingestellt, so wird das KNN zufriedenstellenden Output liefern, vorausgesetzt die Topologie ist ausreichend für die Lösung des Problems „designt“.

Analog zum menschlichen Gehirn lernt ein KNN durch Training, wobei dieses schrittweise vollzogen wird. Man benötigt für das Training Daten, die sogenannten **Trainingssets**. Diese bestehen prinzipiell aus Input x_1 , x_2 , x_3 etc. und gewünschten Outputs d_1 , d_2 , d_3 etc., wobei „d“ für „desired“ steht. Die Trainingsdaten müssen zur Topologie des KNNs passen, d.h. wenn man 3 Eingänge hat braucht man auch 3 Inputs.

Wenn wir das Beispiel des 3-Layer-KNNs auf *[Bild 8]* heranziehen, so könnte ein Trainingsset bspw. so aussehen:

```
double trainingsSet [3][6]{
    //x1      x2      x3      d1      d2      d3
    //Leben  Distanz  Waffe  Angriff  Flucht  Warten
    0.9      0.1      0.3    0.9    0.1     0.1
    0.1      0.1      0.1    0.1     0.9    0.1
    0.7      0.3      0.4    0.1     0.1     0.9
}
```

Wir haben hier drei Inputs, die ersten drei Felder im inneren Array und drei Outputs, die nachfolgenden Felder. Die Werte wurden zuvor normalisiert, d.h. bspw. dass 0.9 bei „Leben“ 90 Prozent der maximalen Lebensenergie bedeutet. Bei der Waffe könnte man annehmen, dass 1.0 die stärkste Waffe bedeutet.

Wenn wir bspw. die Logistik-Funktion als Aktivierungsfunktion nehmen (*siehe [Bilder 5]*), dann kann man erkennen, dass sich diese Funktion asymptotisch an die **u**-Achse bzw. an der gedachten horizontalen Linie bei **o=1** anschmiegt. Die Funktion liefert als Output also nie den diskreten Wert **o=0** bzw. **o=1**. Aus diesem Grund haben wir bei unseren gewünschten Output **d1** (Angriff), in der ersten Zeile, den Werte **0.9** zugewiesen. Dieser Wert kann dem Bool'schen Wert **true** gleichgesetzt werden und bedeutet letztlich, das es wünschenswert ist wenn der NPC, beim gegebenen Input, angreift. Später, wenn das KNN fertig trainiert im Spiel Einsatz findet, wird der höchste Output als Indizierung dass die zugehörige Aktion nun folgen soll, genommen.

Beispiel einer Spielsituation mit dem fertig trainierten KNN:

$$x = (x_1, x_2, x_3) = (0.73, 2.564, 2.7)$$

und der Output der von unseren KNN generiert wird, sei:

$$o = (o_1, o_2, o_3) = (0.58, 0.31, 1.03)$$

dann sagt **0.58** nach der Devise „*the winner takes it all*“ aus, dass der NPC angreifen soll.

Zurück zum Trainings-Prozess selbst:

Beim Training, bzw. ersten Trainingsdurchlauf, „schieben“ wir nun unseren Input-Vektor durch das KNN, welches uns einen Output liefert (die Gewichte des KNNs wurden zuvor mit Zufallswerten initialisiert).

Nun bekommen wir unseren ersten Output-Vektor **o**, dessen Komponenten sich wahrscheinlich stark von denen des „Desired“-Vektor **d** unterscheiden werden. Es wird nun der Fehler **e** ermittelt, der wie folgt aussieht:

$$e = d - o$$

Dieser Wert e fließt nun in den sog. „**Backpropagation**“-Algorithmus ein, der versucht die Gewichte von hinten nach vorne (Output-Layer in Richtung Input-Layer) zu optimieren, so dass sich der tatsächliche Output den gewünschten Output nähert. Dazu werden weitere Fehler berechnet, die bei den Output- und Hidden-Neuronen anliegen. Zunächst der Fehler an den Output-Neuronen:

$$e_{\text{Out}} = e * o * \text{act}'$$

Hier ist act' die Ableitung der Aktivierungsfunktion. Mit diesem Wert lässt sich nun der Fehler an den Neuronen im Hidden-Layer bestimmen:

$$e_{\text{Hid}} = \text{sum}(e_{\text{Out}_i} * w_i) * \text{act}'$$

, mit $i = 1 \dots \# \text{Verbindungen}$

Der Fehler pro Neuron wird u.a., durch die Summierung der gewichteten Verbindungen w_i multipliziert mit dem entstandenen Fehler an den Output-Neuronen gebildet. Das Ergebnis wird im Anschluss mit der Ableitung der Aktivierungsfunktion multipliziert.

Am Input-Layer wird kein Fehler berechnet, da hier keine Verarbeitung stattfindet.

Nun können wir die Gewichte der neuronalen Verbindungen nach folgender Regel anpassen:

$$w_t = w_{t-1} + \text{learnRate} * e_{\text{Hid}} \text{ bzw. } e_{\text{Out}},$$

mit $t = 1 \dots \# \text{Durchläufe}$

Der Begriff „**learnRate**“ oder auch Lernrate ist ein Regler, der bestimmt in welchem Ausmaß die Gewichte angepasst werden. Sie hat typischer Weise einen Wert zwischen **0.0** bis **1.0**.

e_{Hid} wird in der Formel verwendet, wenn man die Gewichte zwischen Input- und Hidden-Layer anpassen will, e_{Out} entsprechend bei der Gewichtsangpassung zwischen Hidden- und Output-Layer.

Der **Back-Propagation**-Algorithmus wird nun nach jedem Trainingsdurchlauf angewendet, solange bis die Gewichte optimal angepasst sind. Das ist der Fall, wenn das KNN eine gewisse Güte erfüllt.

Als Güte-Maß wird der kleinste, mittlere quadratische Fehler, also der „**Least Mean Square Error**“ (**LMS**-Error) herangezogen:

$$\text{lmsError} = \text{sum}(e_j^2) / \# \text{outputNeurons},$$

mit $j = 1 \dots \# \text{outputNeurons}$

Der einfache „ $e = d - o$ “ Fehler wird also für jeden Output quadriert; danach werden die Fehler aufsummiert und der Mittelwert gebildet. Unterschreitet nun der **LMS**-Error, nach einer gewissen Iterationsstufe ein vorher durch den

Entwickler definierten Wert (z.B.: $\epsilon < 0.05$), so gelten die Gewichte als optimal angepasst. Erreicht der LMS-Error nach bspw. 50000 Iterationen nicht diese Grenze, so muss man ein zusätzliches Trainingsabbruch-Kriterium z.B. „Maximale Anzahl der Trainingsiteration“ definieren.

Abschließend sei noch erwähnt, dass prinzipiell zwischen **Online-** und **Offline-**Training unterschieden wird:

- **Online-Training:**
findet statt, wenn wir unser KNN zu Laufzeit, also während des „Spiel-Loops“ trainieren; es werden Spieldaten gesammelt, diese Trainingssets konvertiert und normalisiert und mittels Back-Propagation findet die Gewichtsanzpassung statt.
- **Offline-Training:**
Trainingsdaten werden während der Entwicklungsphase generiert und aufs KNN angewendet; Spieldaten der Entwickler, die ihr eigenes Spiel testen, können gesammelt werden; z.B. bei ein Ralley-Spiel fährt der Entwickler sehr gute Zeiten; diese Daten werden aufbereitet und zum Training der Bot-Autos anschließend eingesetzt

3. Vergleich: NN vs. KNN

Generell kann man sagen, dass künstliche Neuronen eine abstrakte Imitation natürlicher Neuronen sind. Beide erhalten einem Input, welcher beim natürlichen Neuron elektrischen Impulsen sind. Die Gewichte beim künstlichen Neuron könnte man etwa mit der Stärke der Verbindung von den Synapsen der Neuronen $n-1$ zu den Dendriten des Neurons n vergleichen. Die gewichtete Summe ist die Vereinigung der einströmenden elektrischen Impulse beim natürlichen Neuron. Die Aktivierungsfunktion kann mit dem Axonshügel, der bei überschwelliger Erregung ein Aktionpotential absetzt, gleichsetzen.

Bzgl. der Topologien gibt es eigentlich keine Ähnlichkeiten, wenn man von der Tatsache absieht, dass Neuronen hintereinander „geschaltet“ werden können. Die Natur baut ihre eigenen abstrakten Netze.

Beim Lernen gibt es wieder eine Parallele in der Hinsicht, dass man sowohl NN als auch KNN durch iteratives Trainings optimiert. KNNs können beim Lernen jedoch nicht neue Neuronen hinzufügen, wenn einmal die Topologien festgelegt ist, im Gegensatz zu natürlichen NNs.

4. KNN: Aufgaben in Spielen

Der Aufgabenbereich von KNN in Spiel kann man eigentlich in folgende Kategorien aufteilen:

- **Klassifizierung:**
(siehe Beispiel S. 8)
- **Entscheidung:**
soll der NPC angreifen, fliehen oder doch lieber warten in Abhängigkeit von Variablen der Spiel-Umgebung, die als Input in das KNN fließen;
(siehe 3-Layer KNN auf S. 10 unten)
- **Steuerung:**
soll der Bot-Panzer die rechte oder doch die linke Kette, in Abhängigkeit von Variablen der Spiel-Umgebung, bewegen; wird in der engl. Literatur auch als „Robotic Controller“ bezeichnet

5. Design unseres KNNs

Die Vorgehensweise beim Design, Training und schließlich der Implementierung eines KNN in einem Spiel, setzt eine gründliche Planung voraus. Als kleine Daumenregel sollte sich der Entwickler folgende zentrale Fragen stellen und beantworten.

- **Fragen:**
 - I. Welche Aufgaben soll mein KNN im Spiel lösen?
 - II. Wie soll die Topologie aussehen (Anzahl der Schichten und wie viele Neuronen pro Schicht)?
 - III. Welche Aktivierungsfunktionen?
 - IV. Welches Lernverfahren?
 - V. Wann Training?

Nach sorgfältigen Überlegungen sollte man bspw. unter Berücksichtigung des Spiel-Genres, der Ressourcen, der Spielkonsole auf der das Spiel laufen soll, etc. Antworten finden. In unserer Beispiel-Implementierung wollen wir, dass ein NPC in einen Rollenspiel eine Entscheidung trifft, welche Aktion er nächstes ausführt.

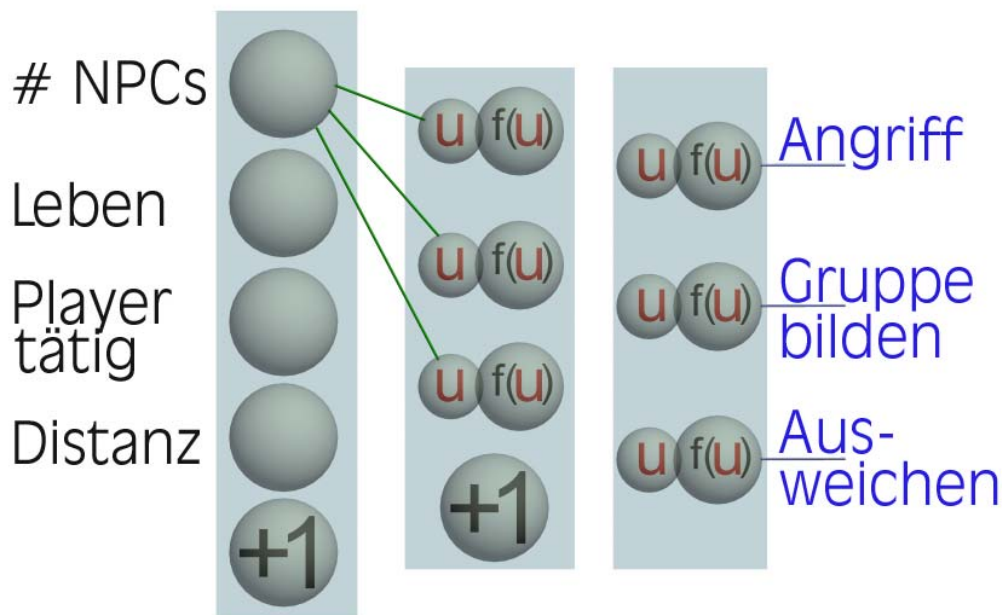
Folgende Situation sei zu lösen:

dem NPC steht einem Feind (menschliche Spieler) gegenüber; der NPC hat diesen Charakter bereits als Feind klassifiziert (*siehe Klassifizierungs-Beispiel S. 8*); der NPC soll als nächstes eine Aktion, wie Angriff, Flucht etc. durchführen; die Entscheidung für eine sinnvolle Aktion seitens des NPCs soll das KNN treffen; wir haben nur wenig Rechenzeit zur Verfügung, da es sich um ein Online-Rollenspiel handelt und der Server nicht unnötig belastet werden soll

Nun können wir unsere Fragen beantworten:

- Antworten: I. Aktions-Entscheidung im RPG
II. 3-Layer Feedforward Netz: $4 > 3 > 3$
VI. Logistik
VII. Back-Propagation
VIII. Offline

Im Anschluss können wir mit dem Design beginnen:



[Bild 9]

(die **grünen** Gewichtsverbindungen sollen stellvertretend für jedes Neuron stehen, d.h. jedes Neuron des Input-Layers ist mit jedem Neuron des Hidden-Layers und jedes Neuron des Hidden-Layers ist wiederum mit jedem Neuron des Output-Layers verbunden)

Nun können wir mit der Implementierung beginnen, die im Folgenden als Pseudo-Code auf *Seite 16* dargestellt ist. Die farbigen Methoden kommunizieren miteinander. Es wurde aus Platzmangel auf die Implementierung der Methoden-

Rümpfe verzichtet. Die Namen der Methoden wurden selbsterklärend ausgewählt, so dass der Zusammenhang zur Theorie sich erschließen müsste.

Herauszugreifen sei bspw. die Tatsache, dass ein Hidden-Layer einen Zeiger auf den „**elternLayer**“ besitzt, welcher den Input-Layer darstellt. Der „**kindsLayer**“ aus der Sicht des Hidden-Layers ist folglich der Output-Layer.

Eine andere wichtige Erkenntnis ist, dass sie das trainierte KNN während des Spiels, also im „**spielLoop()**“, sich wie eine „State-Machine“ sich verhält. Es lernt zur Laufzeit nichts mehr dazu.

```
class Layer {
    int anzahlNeurons;
    // Input, Output, Wunsch,
    // Gewichte, Fehler, ...
    Layer* elternLayer;
    Layer* kindsLayer;
    init(...);
    berechneNeuronOutput(...);
    berechneFehler(...);
    passeGewichteAn();
}
```

```
class KNN {
    Layer inputLayer;
    Layer hiddenLayer;
    Layer outputLayer;
    //...
    setzeInput(...);
    setzeWunsch(...);
    feedforward(...);
    backPropagate();
    //...
}
```

```
KNN gehirn; // unser KNN fuer die Entscheidungsfaellung

Double tSet[14][7]; // das Trainings-Set: 14 Trainingsituationen a 4 Inputs
// und 3 gewuenschte Outputs

trainiereGehirn(){
    for(i=0; i < tSet.length; i++) {
        gehirn.setzeInput(...);
        gehirn.setzeWunsch(...);
        gehirn.feedforward(...);
        gehirn.backPropagate();
    }
}

double lmsError; // mittlerer quadratischer Fehler
int iter; // # maximale Anzahl an Iterationen+

while(lmsError > 0.05 && iter < 50000){
    trainiereGehirn();
}

// im Spiel KNN wie „state-machine“
spielLoop() {
    // ...
    gehirn.feedforward(...);
}
```

6. KNN in Games

Der Einsatz von KNN in Spielen ist nicht sehr weit verbreitet. Wir wollen trotzdem an dieser Stelle zwei bekanntere Spiele vorstellen, die Technik der künstlichen Neuronale Netze zur Lösung von Teilaufgaben nutzten.

Folgende Spiele seien hier erwähnt:

- „Black & White“ für PC:
hier handelt es sich um ein Strategie-Spiel, eine sog. Göttersimulation; man ist ein Gott über ein Volk und muss gegen andere Götter antreten; während des Spiels erschafft man u.a. eine Kreatur, die man erzieht

Die Erziehung dieser soll angeblich mit KNNs, die wohl Online trainiert werden, passieren. In welchem Ausmaß und welche tatsächliche Struktur das KNNs besitzt, konnte man durch Internet-Recherche leider nicht in Erfahrung bringen.

Das Spiel erntete viele Vorschusslorbeeren, war aber im Nachhinein, aus Sicht der Verkaufszahlen, eine Enttäuschung. Der Nachfolger „B&W 2“ sollte die Schmach wieder richten.



Szene aus „B&W 2“

[Bild 10]

- „Colin MacRae 2“ für PS1 (Playstation 1): Ralleyspiel; Fahrten immer nacheinander, um die beste Zeit

Bei diesem Spiel wurde ein KNN als Steuer-System („Robotic Controller“) implementiert. Das Training fand Offline, während der Entwicklungsphase statt. Hierbei wurden Spieldaten, die beim Fahren von zeitlich guten Runden seitens eines Entwicklers entstanden, aufgezeichnet und als in Trainings-Sets konvertiert. Anschließend wurde das Steuerungs-KNN des Bots-Autos mit diesen Trainings-Sets vor allem auf rutschigen Strecken trainiert.



rutschige Strecke

[Bild 11]

Bei diesem Spiel wurden „Feedforward Multilayer-Perceptrons“ als KNN-Architektur verwendet. Das ist praktisch ein mehrschichtiges Netzwerk (*wie auch zuvor besprochen wurden*) mit der Schritt-Funktion also Aktivierungsfunktion in den einzelnen Neuronen.

Das Spiel, bzw. die Spielreihe garantiert akkuraten und realistischen Rennsport für Puristen; wurde aber nie ein „Top-Seller“.

7. Fazit

Die Anwendung von KNNs ist aufgrund ihrer Lernfähigkeit prinzipiell eine spannende Angelegenheit. Dennoch scheuen sich Firmen KNNs einzusetzen, was u.a. an der Tatsache liegt, dass es noch kein absolutes Mainstream-Spiel mit KNNs gab. *„Wozu sollten man diese KNNs nutzen, wenn man auch mit eingesessenen Techniken Geld verdient“*, mögen sich einige Firmen fragen.

Hinzu kommt eine gewisse Scheu vor unberechenbaren Verhaltensweise, die bei Online-Training eventuell passieren könnten ganz zu Schweigen von der stets herrschenden Ressourcen-Knappheit. Dabei können KNN, wenn man sie bewusst und vereinzelt einsetzt eigentlich problemlos funktionieren. Man muss sich halt an gewisse Regeln halten, wie Offline-Training bevorzugen, Online-Training nur bei Single-Layer-KNNs, keine rekursiven Netzstrukturen verwenden sondern Feedforward, etc..

Beherzigt man diese Regeln, gibt es keinen Grund KNNs nicht zu implementieren. Ob und wie sich die Technik in der Spiel-Industrie etablieren wird, entscheidet letztlich der Spieler. Denn er lenkt die Tendenzen und bestimmt somit ob ein „KNN-Spiel“ sich durchsetzen wird oder nicht. Mal abwarten und vielleicht wird „Colin MacRae Dirt“ (Release am 22.06.07) neue Maßstäbe setzen.

8. Anhang

8.1 Literaturverzeichnis

<i>„AI for Game Developers“</i>	David M. Bourg und Glenn Seemann, O’Reilly; 2004
<i>„Learning and Softcomputing“</i>	Vojislav Kecman, Bradford MIT, 2001
<i>„Learning and Adaption“</i>	Artikel 1.1, AI Game Programming WISDOM 1
<i>„The Dark Art of Neural Networks“</i>	Artikel 11.10, AI Game Programming WISDOM 1
<i>„Strategic Decision-Making with Neural Networks and Influence Maps“</i>	Artikel 7.7, AI Game Programming WISDOM 2
<i>„How to Build Neural Networks for Games“</i>	Artikel 11.1, AI Game Programming WISDOM 2
<i>„Practical Algorithms for In Game Learnings“</i>	Artikel 8.1, AI Game Programming WISDOM 3
<i>„A Brief Comparison of Machine Learning Methods“</i>	Artikel 8.2, AI Game Programming WISDOM 3
<i>Neuronale Netze und Softcomputing</i>	http://wwwmath.uni-muenster.de/SoftComputing/lehre/material/wwwnscript/einl.html
<i>diverse Artikel bzgl. Spiele</i>	http://www.gamasutra.com
<i>diverse Artikel bzgl. AI</i>	http://www.generation5.org
<i>Artikel: „Colin McRae 2“</i>	http://www.generation5.org/content/2001/hanna_n.asp
<i>Begriffe</i>	http://en.wikipedia.org/wiki/Main_Page

8.2 Bildverzeichnis

<i>[Bild 1]</i>	http://sharpbrains.files.wordpress.com/2006/09/picneuron.jpg
<i>[Bild 2], [Bild 3], [Bild 4]</i>	selbst erstellt mit 3ds-Max 8.0
<i>[Bilder 5]</i>	selbst erstellt mit Photoshop
<i>[Bild 6]</i>	Tutorial-Model aus 3ds-Max 8.0 und selbst gemodelter Charakter zusammengefügt in Szene in 3ds-Max 8.0
<i>[Bild 7], [Bild 8], [Bild 9]</i>	selbst erstellt mit 3ds-Max 8.0
<i>[Bild 10]</i>	http://www.gamasutra.com/features/20061016/carless_03.shtml
<i>[Bild 11]</i>	http://img222.imageshack.us/img222/721/40217yd.jpg