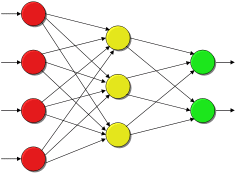
2.3 Das Neuronale Netzwerk

2.3.1 Warum wir uns für neuronale Netze entschieden haben

Neuronale Netze sind vereinfachte Modelle des Gehirns und sind deshalb sehr vielseitig einsetzbar und können viele unterschiedliche Aufgaben erledigen. Ein gutes Beispiel dafür, dass künstliche neuronale Netze auch komplexe Aufgaben wie das Spielen von Brettspiele erledigen kann ist das IBM-Programm Deep Blue oder das von Google DeppMind entwickelte AlphaGo. Bei unserem neuronalen Netzwerk haben wir uns für eine Deep Feedforward Struktur entschieden, auf die ich später noch genauer eingehen werde.

2.3.2 Aufbau eines neuronalen Netzes

Auf der folgenden Abbildung sieht man ein beispielhaftes neuronales Netzwerk.

[[1]](#footnote-1)

Ein neuronales Netz besteht aus Neuronen, die auch Units genannt werden, die miteinander verbunden werden. Es gibt verschiedene Arten von Neuronen. Das abgebildete neuronale Netzt besteht aus Input-Neuronen, welche in der Abbildung rot dargestellt werden, Hidden-Neuronen, welche in der Abbildung gelb dargestellt werden, und Output-Neuronen, welche in der Abbildung grün dargestellt werden. Die Neuronen eines neuronalen Netzes können oft in Layer zusammengefasst werden. Das vorliegende Netz besteht aus einem Input-Layer, einem Hidden-Layer und einem Output-Layer. Da das Netzt nur aus den Eben beschriebenen Bausteinen besteht und nur einen Hidden-Layer besitzt handelt es sich hierbei um ein Netzt mit Feedforward Struktur. Wenn das Netzt mehrere Hidden-Layer besäße, wäre es ein Deep Feedforward Netzt.

Ein Neuronales Netzt bekommt immer Zahlen als Input. Bei einem Feedforeward-Netzt berechnet man den Output indem erst die Input-Neuronen setzt und dann der Layer und die Verbindungen nacheinander updatet. Hierbei geben die Input-Neuronen den, ihnen zugewiesenen, Wert nur an die Verbindungen weiter. Die Verbindungen wiederum multiplizieren den Wert mit einem variablen Wert, welcher als Gewicht bezeichnet wird, und geben das entstandene Produkt dann an die Hidden-Neuronen, des ersten Hidden-Layers, weiter. Wenn ein Hidden-Neuron mehrere Werte, von den Verbindungen, bekommen hat und geupdatet wird, addiert es diese Werte erst zu einer Summe, welche dann eine Aktivierungsfunktion, die auch Transferfunktion genannt wird, durchläuft. Beispiele für eine Aktivierungsfunktion sind

die Sigmoidfunktion[[2]](#footnote-2), eine binäre Schwellwertfunktion[[3]](#footnote-3) oder eine lineare Funktion[[4]](#footnote-4).

Wir benutzen für unser Projekt nur die Sigmeoidfunktion. Nachdem alle Neuronen, eines Hidden-Layers, geupdatet wurden geben alle ihre berechneten Werte an die Verbindungen weiter, welche diese wieder mit einer Variable Multiplizieren und weitergeben. Dieser Vorgang wird solange wiederholt, bis die Werte schließlich an den Output-Layer weitergegeben werden. Die Output-Neuronen funktionieren wie die Hidden-Neuronen bis auf den Unterschied, dass sie den Wert nicht mehr weitergeben. Nachdem diese geupdatet wurden, bilden ihre Werte den Output des Netzes.

2.3.3 Die Optimierung des neuronalen Netzes

Bei der Optimierung von neuronalen Netzen gibt es mehrere Möglichkeiten, von denen mache dem neuronalen Netzt Beispiele geben und das neuronale Netzt sich an diese annähert und sich damit auch einer allgemeinen Lösung annähert und andere keine Beispiel Lösungen und im Bezug auf Brettspiele auch keine Strategie vorgeben. Wir haben uns bei unserem Projekt für einen evolutionären Ansatz entschieden, da auch hier keine Strategie vorgegeben wird. Bei unserer Optimierungsmethode werden mehrere Netze erzeugt, die alle gegeneinander Dame spielen. Nach jedem spiel werden die beiden Netze bewertet, indem ihnen ein Fitnesswert zugeschrieben wird. Wenn alle Netze gegen alle Netze gespielt haben werden die Netze mit den geringsten Fitnesswerten aussortiert und veränderte Versionen der anderen Netze ersetzen diese. Dieser Prozess kann nun sehr oft wiederholt werden. Auf diese Weise erhält man mit der Zeit immer Bessere Neuronale Netze.

2.3.4 Die Umsetzung

Das neuronale Netz besteht aus einer einzelnen Klasse. In ihr befinden sich die Variablen, die die Gewichte, der Verbindungen angeben und die den Bereich angeben, indem die Sigmoidfunktion skaliert und sich die Gewichte der Verbindungen, bei der Initialisierung befinden dürfen. Die folgende Tabelle Zeigt die Methoden der Klasse NN und beschreibt diese.

|  |  |
| --- | --- |
| **Methode** | **Beschreibung** |
| randomWeights | Allen Gewichten ein zufälliger Wert, in einem festgelegten Bereich zugeordnet. |
| vector\_matrix\_multiplication | Die Übergabe, Multiplikation mit den Gewichten und die Addition der Werte, der Neuronen, einer Schicht, haben wir hier durch eine Matrizenmultiplikation, bei der die Werte, der Neuronen einen Vector und die Gewichte eine Matrix bilden, umgesetzt. |
| sigmoid | Ein übergebener Wert wird in eine gestauchte und verschobene Sigmoidfunktion eingesetzt und das Ergebnis wird zurückgegeben. |
| changeAll | Alle Gewichte werden mit zufälligen Werten, in einem festgelegten Bereich, multipliziert. |
| changeFunction | Ein übergebener Wert wird in eine Quadratische Funktion, die für changeAll gebraucht wird, um die Zufallszahl, die für die Veränderung, der Gewichte, benötig wird, zu skalieren, eingesetzt und das Ergebnis wird zurückgegeben. |
| childFrom | Die Gewichte, des neuronalen Netzes, werden zu einer Mischung, der Gewichte, von zwei anderen Netzen. Hierbei wird bei jedem Layer zufällig entschieden, von welchem Netz der Layer übernommen wird. |
| run | Die Inputwerte durlaufen, wie in 2.3.2 beschrieben, mithilfe der anderen Methoden das neuronale Netz. Dieser Prozess wird auch Propagation genannt. |

Um uns das speichern und Übergeben der Struktur und Variablen, die das neuronale Netz bei der Initialisierung benötigt, zu vereinfachen, haben wir die Klasse NNSpecification geschrieben, in der diese Werte gespeichert sind. Unsere neuronalen Netze besitzen 64 Inputneuronen 640 Hiddenneuronen, welche gleichmäßig auf 10 Layer aufgeteilt sind und 64 Outputneuronen. Damit besitzen sie auch 40960 Verbindungen, mit jeweils einem Gewicht.

Um ein neuronales Netz mit dem Brettspiel Dame zu verbinden haben wir die Klasse NNPlayer hinzugefügt. Diese hat erstens die Aufgabe das Feld der aktuellen Spielsituation an das neuronale Netz, das am Zug ist, weiterzugeben, wobei die ersten 32 Werte beschreiben auf welchen Feldern sich eine normale Figuren befinden und welche davon eine eigene und welche eine gegnerische ist, hierbei steht eine 1 für eine eigene normale Figur, eine -1 für eine gegnerische normale Figur und eine 0 für ein leeres Feld oder eine Dame, und die zweiten 32 Werte beschreiben auf welchen Felder sich Damen befinden und welche davon eine eigene und welche eine gegnerische ist, hierbei steht eine 1 für eine eigene Dame, eine -1 für eine gegnerische Dame und eine 0 für ein leeres Feld oder eine normale Figur. Zweitens die Aufgabe die Output Werte in einen Zug zu interpretieren, wobei von den ersten 32 Werten der höchste Wert das Feld angibt, zudem sich eine Figur bewegen soll und von den zweiten 32 Werten der höchste Wert die Figur angibt, die sich dorthin bewegen soll, wobei nur Figuren berücksichtigt werden, für die dieser Zug möglich wäre. Und drittens die Aufgabe das zugewiesene neuronale Netz nach jedem Zug und jedem Spiel zu bewerten. Wir haben uns hierbei dafür entschieden die Bewertung für

Für die Umsetzung des Trainings des neuronalen Netzes haben wir die Klasse TrainingsSession programmiert. Die folgende Tabelle Zeigt die Methoden der Klasse TrainingsSession und beschreibt diese.

|  |  |
| --- | --- |
| Methode | Beschreibung |
| resetPlayer | Die Bewertung, jedes Netzes, wird zurückgesetzt. |
| sortAndCalculateSum | Die Netze werden nach ihrer Bewertung sortiert und die beste und die durchschnittliche Bewertung wird ausgegeben. |
| weightedRandomSelection | Ein zufälliges Netz wird zurückgegeben, allerdings ist es wahrscheinlicher, dass ein Netz mit besserer Bewertung zurückgegeben wird. |
| randomizeArray | Die Netze werden zufällig angeordnet. |
| awaitStopping | Testet, ob Trainiert wird. |
| isRunning | Testet, ob Trainiert wird. |
|  |  |
| train | Diese Methode startet entweder die Methode trainNormal oder die Methode trainMiniMax. |
| trainNormal | Hier haben wir das Training, wie in 2.3.3 beschrieben, umgesetzt. |
| trainMiniMax | Hier trainieren die neuronalen Netze nicht indem sie gegen die jeweils anderen Netze spielen, sondern gegen den MiniMax Algorithmus. |

2.3.5 Fazit zum neuronalen Netzwerk

Die Implementation des neuronalen Netzes und des Trainings hat sich als relativ einfach herausgestellt, da das neuronale Netz nur aus einer Abfolge von mathematischen Operationen und Veränderung von Werten besteht und das Training mit unserem evolutionären Ansatz auch leicht umzusetzen war. Beim Verbinden, des neuronalen Netzes mit dem von uns implementierten Brettspiel Dame, standen wir am Anfang vor Schwierigkeiten, da wir uns entscheiden mussten, wie wir das Umsetzen wollten.

Die Leistung des neuronalen Netzes muss in mehrere Bereiche aufgeteilt werden. Da es erstens eine gewisse Zeit Lernen muss um die Regeln zu verstehen und sehr lange Lernen muss um Züge zu machen, die als Sinnvoll und gut bezeichnet werden können, es aber zweitens für die Ausführung, eines einzelnen Zuges nicht viel Zeit benötigt und es drittens das Potenzial hat innerhalb von sehr kurzer Zeit einen sehr guten oder Perfekten Zug auszuwählen.

1. <http://www.neuronalesnetz.de/units.html> [↑](#footnote-ref-1)
2. **1/(1+e^(-x))** \*m+b [↑](#footnote-ref-2)
3. Ergibt für x<a y1 und für x>=a y2 [↑](#footnote-ref-3)
4. m\*x+b [↑](#footnote-ref-4)