Neuronale Netze zur Auswertung von Delay Line Detektoren

Bachelor-Arbeit

Verfasser

Christian Janke

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Fachbereich 13 - Physik
Institut für Kernphysik

Erklärung nach § 30 (11) Ordnung für den BA- und MA-Studiengang

Hiermit erkläre ich, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderen fremden Texten entnommen wurden, sind von mir als solche kenntlich gemacht worden. Ferner erkläre ich, dass die Arbeit nicht - auch nicht auszugsweise - für eine andere Prüfung verwendet wurde.

Christian Janke

Frankfurt, den 25. September 2012

Information zu dieser Arbeit:

Autor:	Christian Janke
Geburtsdatum:	15.09.1989
E-Mail-Adresse:	c.janke@atom.uni-frankfurt.de
Matrikelnummer:	3 969 847
Abgabedatum:	7.11.2012
Erstgutachter:	Prof. Dr. Reinhard Dörner
Zweitgutachter:	Dr. Achim Czasch

Inhaltsverzeichnis

Einleitung 1

1. Der Detektor 2
1.1. Micro-channel plate . . . 2
1.2. Delay-Line Anode . . . 2
1.3. Signalumwandlung . . . 3
1.4. Signalerzeugung . . . 5

2. Neuronale Netze 6
2.1. Das künstliche Neuron . . . 6
2.2. Topologie . . . 8
2.3. Lernen . . . 11
2.3.1. Backpropagation of Error . . . 12

3. Technische Details 17
3.1. verwendete Programme . . . 17
3.2. Ausgabe . . . 17

4. Single Layer Perceptron 19
4.1. Implementation . . . 19
4.2. Single Hit . . . 19
4.2.1. Erste Implementation . . . 19
4.2.2. Verbesserungen . . . 20
4.2.3. Interpretation . . . 22
4.3. Double Hit . . . 25

5. Multi Layer Perceptron 29
5.1. Die ersten Implementationen . . . 30
5.2. Pruning . . . 31
5.3. 2 SLP = 1 MLP? . . . 31

6. Zusammenfassung & Ausblick 33

A. Literaturverzeichnis 34

B. Anhang **35**
B.1. Quellcode zur Signalerzeugung 35
B.2. Quellcode zur Gewichtserzeugung 36
B.3. Quellcode zur Berechnung der SLP-Ausgabe 36
B.4. Quellcode zum Trainieren des SLP 37

Einleitung

In der Atom- und Molekülphysik werden häufig Multichannelplate Detektoren mit Delay-Line-Auslese eingesetzt. Um eine große Präzision und eine hohe Reaktionsrate zu erhalten, ist es wichtig, dass *alle* Daten genau analysiert werden können. Die aktuelle Methode der Datenanalyse stößt dabei auf Probleme, wenn mehrere Teilchen kurz hintereinander auf den Detektor treffen. In dieser Arbeit wird versucht, ein neuronales Netz so zu trainieren, dass es eine bessere Datenanalyse liefert. Hierzu wird im ersten Kapitel der Detektoraufbau beschrieben, um zu verstehen, woher die einzelnen Signale kommen und wie sie zu interpretieren sind. Im zweiten Kapitel wird dann die Theorie vorgestellt, auf der neuronale Netze basieren. Das dritte Kapitel gibt einen kurzen Überblick über die benutzte Technik. Im vierten und fünften Kapitel werden die Ergebnisse dieser Arbeit vorgestellt. Im abschließenden sechsten Kapitel werden die Ergebnisse zusammengefasst und ein Ausblick auf weitere mögliche Projekte gegeben.

1. Der Detektor

Für die Experimente der Atomphysik-Gruppe der Goethe Universität Frankfurt werden vor allem Delay-Line Detektoren eingesetzt. Die Detektoren bestehen dabei aus Micro-channel plates (MCP) und Delay-Line Anoden. Auftreffende Elektronen und Ionen lösen aus den MCPs Sekundärelektronen aus, die zur Anode hin beschleunigt und dort detektiert werden.

1.1. Micro-channel plate

Micro-channel plates bestehen aus einer Bleiglasplatte, die oben und unten metallbeschichtet ist. In der Platte befinden sich schmale ($\sim$3-25 μm dicke) Kanäle mit einer Halbleiteroberfläche, die leicht geneigt sind. Fliegt ein Elektron von oben in die MCP, so trifft es auf die Halbleiterwand und schlägt dort weitere Sekundärelektronen aus. Durch die an den Metallschichten angelegte Spannung, werden auch diese Elektronen nach unten beschleunigt und stoßen wiederum an die Wand. Aus einem einzelnen Elektron kann so ein ganzer Elektronenschauer werden. Abbildung 1.1 zeigt schematisch den Aufbau.

Jedes auftreffende Teilchen schlägt insgesamt circa 1000 Sekundärelektronen aus. Um das Signal zu verstärken, können mehrere MCP übereinander verbaut werden. Dabei ist darauf zu achten, dass die Kanäle gegeneinander geneigt sind, um ein Ionenfeedback in den Kanälen zu vermeiden. Strukturen aus zwei bis drei MCP haben sich dabei bewährt und werden bevorzugt verwendet. Zwischen den beiden äußeren Metallschichten gibt es einen kurzzeitigen Spannungseinbruch, wenn Sekundärelektronen ausgeschlagen werden. Dieses Signal wird für die Flugzeitberechnung der Teilchen verwendet.

1.2. Delay-Line Anode

Unterhalb der MCP befinden sich zwei bis drei Delay-Line Anoden. Eine Anode besteht dabei aus zwei Drähten (Layer), die parallel auf einen Rahmen gewickelt werden. Bei dieser spulenähnlichen Wicklung gehört folglich jeder zweite Draht zur gleichen Leitung. Diese Zweidrahtleitung (Lecherleitung) ist nötig, um Dispersionseffekte während der Laufzeit zu minimieren.

Treffen nun Elektronen auf einen Draht, so entsteht ein Ladungsüberschuss. Dieser fließt zu beiden Seiten hin ab und kann an beiden Enden der Leitung gemessen werden.

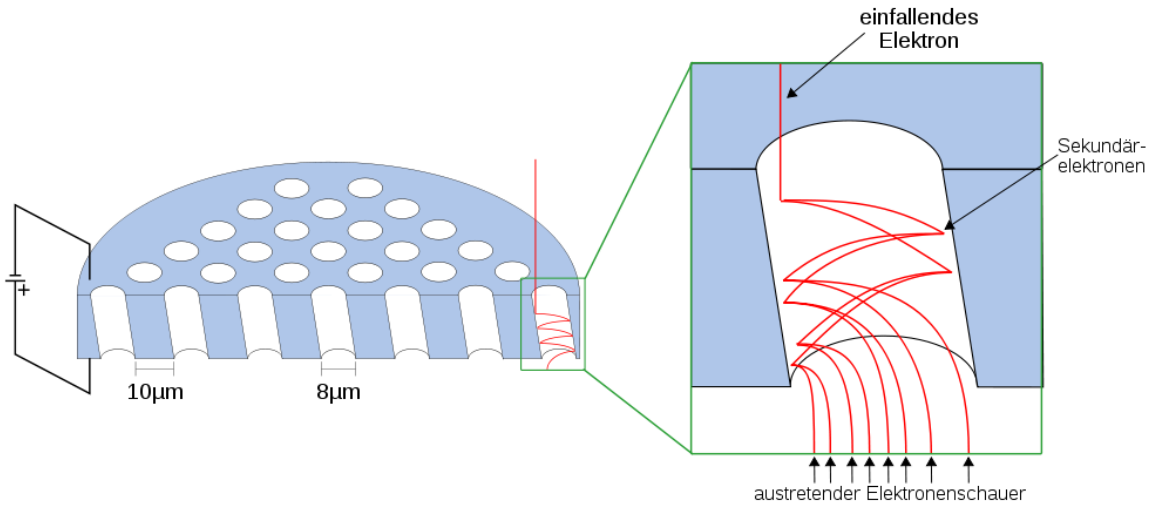


Abbildung 1.1.: MCP. Ein einfallendes Elektron trifft auf die Seitenwand und schlägt dort Sekundärelektronen aus. Diese werden nach unten beschleunigt und schlagen weitere Elektronen aus. Ein auftreffendes Teilchen erzeugt so einen Schauer mit etwa 1000 Sekundärelektronen. Quelle: wikimedia.org

Der zeitliche Unterschied zwischen den zwei Signalen hängt dabei direkt mit dem Auftreffpunkt zusammen (siehe Abbildung 1.2). Über die getroffene Wicklung (und damit der zeitliche Unterschied) kann der Ort in einer Dimension bestimmt werden. Legt man nun zwei Anoden 90° versetzt übereinander, so kann aus der Kombination der Signale ein zweidimensionaler Ort bestimmt werden. Um die Datenanalyse zu verbessern, werden bei der Hexanode drei Anoden jeweils 60°-versetzt übereinander angebracht[1].

In Abbildung 1.3 ist ein Bild eines Detektors mit Hexanode abgedruckt. Für eine detailliertere Beschreibung des Detektors sei auf Literatur, wie zum Beispiel [JCC+02] oder [JMUP+98], verwiesen.

1.3. Signalumwandlung

Aus dem Detektor können sieben Signale ausgelesen werden: Die Spannungsänderung am MCP, sowie zwei Signale von jeder der drei Anoden. Mithilfe eines Constant Fraction Discriminators (CFD) können diese sieben analogen Signale in binäre Signale und diese wiederum mit einem TDC (time-to-digital-converter) in Zeitwerte umgewandelt werden. Aus den sechs Zeiten der Hexanode lässt sich der Auftreffort des Teilchens bestimmen. Dies geht aber nur dann genau, wenn die Signale immer relativ gleichförmig

[1] In dieser Arbeit wird nur auf den Hexanoden-Typ eingegangen.

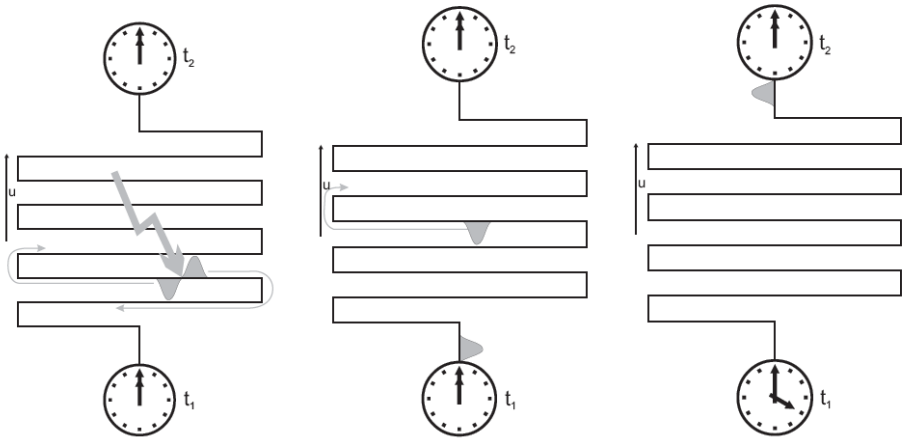


Abbildung 1.2.: Prinzip der Delay-Line Anode. Ein auftreffender Elektronenschauer bildet einen Ladungsüberschuss, der zu beiden Seiten hin abfließt (links). Wenn die Ladung ein Ende erreicht, wird eine Stoppuhr gestartet (Mitte). Aus der Differenz der zwei Zeitsignale kann dann auf den Ort zurück geschlossen werden (rechts). Quelle: [Mec06, Abbildung: 3.13]

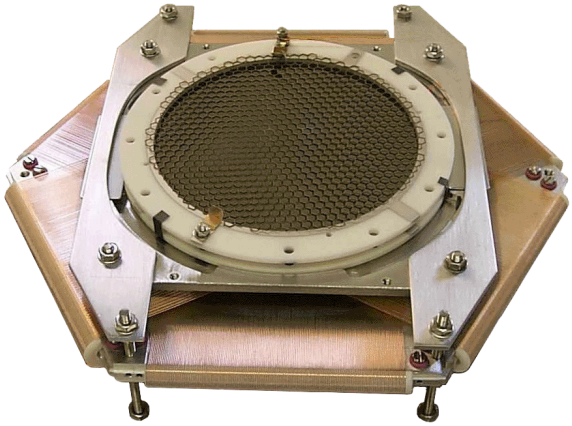


Abbildung 1.3.: Bild eines Detektors mit MCP und Hexanode. Quelle: RoentDek Handels GmbH

sind[2]. Ein weiteres großes Problem tritt auf, wenn zwei Teilchen kurz nacheinander auf den Detektor treffen und sich die Signale (auf einem Layer) überlagern.

Um dieses Problem zu umgehen, können die Signale aus dem Detektor mithilfe eines Analog-to-Digital-Converter (ADC) digitalisiert und dann gespeichert werden. Bei einer typischen Abtastrate von 1ns und einer Zeitdauer von 120ns [3] werden so aber aus 7 Signalen $7 \cdot 120 = 840$ Werte.

Die Analyse der Detektorsignale wird aber durch Störfaktoren sehr erschwert. Es kann vorkommen, dass der Detektor zusätzliche Signale erzeugt oder richtige Signale verloren gehen. Außerdem kann es passieren, dass die Signale auf einer Drahtebene dicht aufeinander folgen, sodass sie ineinander laufen und von der Elektronik nicht mehr als zwei Signale erkannt werden. Auch Rauschen auf den Leitungen und Reflexionen von Steckverbindungen erschweren die Analyse. Diese Probleme können mit iterativen Algorithmen nur schwer gelöst werden und sollen nun mithilfe von neuronalen Netzen behoben werden, da diese sehr robust gegenüber Störungen sein sollen.

1.4. Signalerzeugung

Um das Problem für diese Arbeit zu vereinfachen, werden die Signale des Detektors simuliert. Der Quellcode hierzu befindet sich im Anhang B.1. Es wird ein x,y-Paar erzeugt, das innerhalb eines Kreises mit Radius `MCP_RADIUS` liegen muss. Danach werden diesen x,y-Werte zu Laufzeiten auf den sechs Layern umgerechnet und gaußförmige Signale generiert. Der Mittelpunkt des MCP-Signals wird dabei fest auf 60 [4] gesetzt. Die berechneten Werte werden im Eingabevektor `input_layer` gespeichert.

[2] Die Amplitude ist für CFD irrelevant, aber die Anstiegszeiten, beziehungsweise Signalformen, müssen immer etwa gleich sein.

[3] typische Zeit, die ein Signal von einem Ende bis zum anderen Ende der Anode benötigt (durch die Bauart bedingt)

[4] Als fester Wert wird die Hälfte der Abstastlänge gewählt.

2. Neuronale Netze

Künstliche neuronale Netze werden verwendet, um Eingabevektoren mithilfe von künstlichen Neuronen in Ausgabevektoren umzurechnen. Mit diesem Maschinenlernverfahren können Probleme der Regression, sowie der Klassifikation gelöst werden. Die Dimensionen der Vektoren sind dabei dem Problem anzupassen und sind im Allgemeinen nicht gleich groß. Ziel dieser Arbeit ist es, ein neuronales Netz zu entwickeln, das die 840 Werte des Delay-Line Detektors in eine x-y-Ortsinformation umwandelt[1].

2.1. Das künstliche Neuron

Jedes neuronale Netz wird aus künstlichen Neuronen aufgebaut. Ein Neuron ist dabei eine Einheit, die einen Eingabevektor ($\vec{i}$) mithilfe einer Propagierungsfunktion in eine Zahl (net) umrechnet und diese Zahl nach Anwendung einer Aktivierungsfunktion (f_{act}) weitergibt ($output, o$). Als Propagierungsfunktion empfiehlt sich die gewichtete Summe. Für jedes Neuron wird nun ein Gewichtsvektor benötigt, dessen Komponenten die einzelnen Gewichte sind. Mehrere Neuronen können in Schichten[2] angeordnet werden. Die Gewichtsvektoren von diesen Neuronen lassen sich dann zu einer Matrix zusammenfassen, sodass sich die Netzeingabe eines Neuron j aus dieser Schicht wie folgt berechnet:

$$net_j = \sum_n \omega_{nj} i_n \tag{2.1}$$

Die einfachste Aktivierungsfunktion ist die Identität. Der Eingabevektor wird zusammengefasst und direkt weitergegeben. Um die möglichen Ausgabewerte zu beschränken wird oft die Heaviside-Funktion benutzt. Erreicht die Netzeingabe einen bestimmten Schwellwert, so wird 1 ausgegeben (das Neuron *feuert*), ansonsten 0[3]. Das Problem dieser Funktion ist, dass sie nicht stetig differenzierbar ist. Vor allem das Fehlen der Ableitung bereitet später Probleme. Als Ausweg bieten sich sigmoidale Funktionen an, da diese die Stufenfunktion gut approximieren können. Am beliebtesten sind dabei der Tangens Hyperbolicus und die Fermi-Funktion (logistische Funktion).

[1] In dieser Arbeit wird dieselbe Notation wie in [Kri07] verwendet.
[2] Der Begriff Schicht wird im nächsten Abschnitt genauer beschrieben.
[3] Die entspricht der 1943 vorgeschlagenen McCulloch-Pitts-Zelle [MP43]

$$sig(t) = \frac{1}{1 + e^{-(t-s)/T}} = \frac{1}{2} \cdot \left(1 + \tanh\left(\frac{t-s}{2T}\right)\right) \tag{2.2}$$

Beide Funktionen wurden in Gleichung 2.2 um den (Temperatur-) Parameter T erweitert, um die Steigung um den Nullpunkt variieren zu können. Außerdem wurde der Schwellwert s abgezogen, um den Wendepunkt zu verschieben.

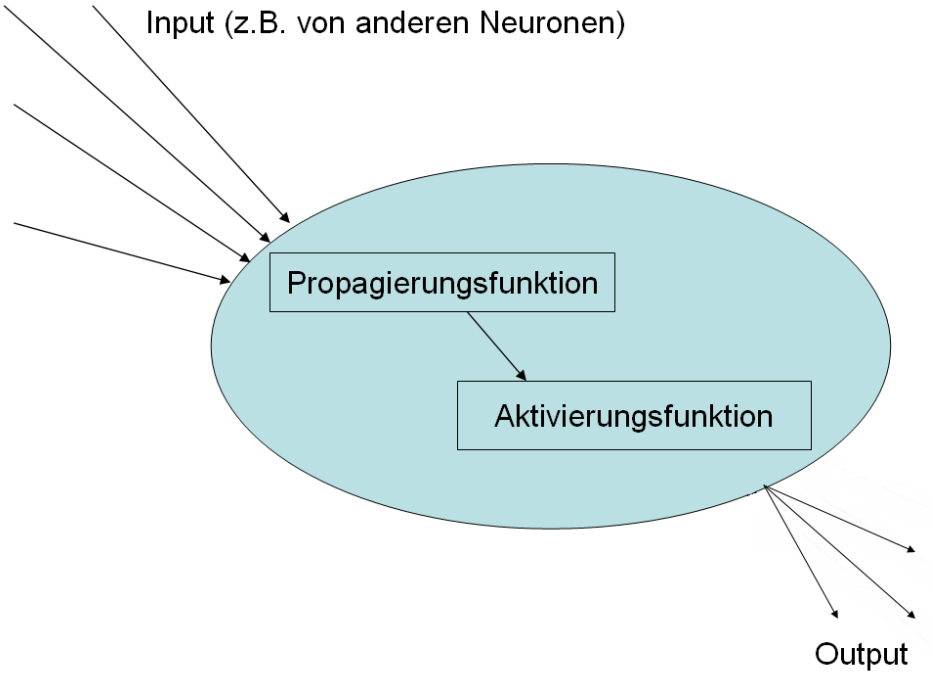


Abbildung 2.1.: Veranschaulichung eines Neuron. Der eingehende Input wird mithilfe von Propagierungs- und Aktivierungsfunktion zu einem Output umgerechnet.

Festzuhalten bleibt, dass sich Neuronen in den folgenden Punkten unterscheiden (können):

- Propagierungsfunktion
- Gewichtsvektor
- Aktivierungsfunktion
- Schwellwert
- Temperaturparameter

Für ein perfektes Neuronales Netz müssten alle diese Parameter für jedes Neuron einzeln gefunden werden. Da dies nicht möglich ist, werden einige Vereinfachungen gemacht. Als erstes wird für alle Neuronen die gewichtete Summe als Propagierungsfunktion angenommen. Weiterhin werden die Aktivierungsfunktion und der Temperaturparameter für alle Neuronen einer Schicht gleich definiert. Als letzte Vereinfachung wird der Schwellwert mit in den Gewichtsvektor aufgenommen. Hierzu wird der Eingangsvektor um einen *Bias* genannten Eintrag erweitert. Dieser ist immer 1, aber das Gewicht zum Neuron ist der negative Schwellwert. Mithilfe dieser Tricks müssen später nur noch die Gewichtsvektoren aller Neuronen angepasst werden.

In Schaubildern werden Neuronen gerne als Kreise dargestellt. Im Inneren steht dabei die Propagierungsfunktion (oben) und die Aktivierungsfunktion (unten). Abbildung 2.2 zeigt eine Auswahl von Neuronen.

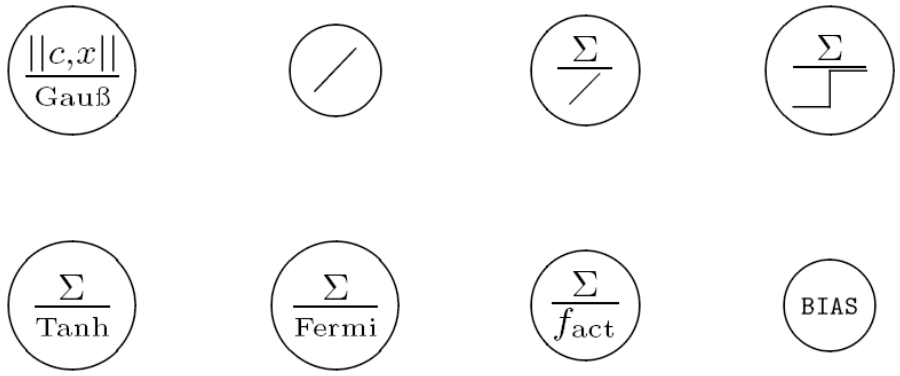


Abbildung 2.2.: Auswahl einiger Neuronen. Die gerade Linie steht dabei für die Identität, das große Sigma für die gewichtete Summe. Quelle: [Kri07, Abb. 3.10]

2.2. Topologie

Um aus Neuronen ein Netz zu bilden, werden eine gewisse Anzahl von Neuronen in Schichten angeordnet. Die Zahl der Neuronen in der ersten Schicht entspricht genau der Dimension des Eingabevektors. Diese Schicht wird daher Eingabeschicht oder Inputlayer genannt. Für die letzte Schicht werden genau so viele Neuronen verwendet wie für den Ausgabevektor benötigt werden. Weitere Zwischenschichten können optional eingefügt werden, die Größe ist dabei frei wählbar. Es empfiehlt sich, in allen Schichten bis auf die Ausgabeschicht ein Bias-Neuron hinzuzufügen, um das Trainieren zu vereinfachen.

Alle Neuronen einer Schicht sind mit allen Neuronen der Vorgängerschicht verknüpft[4]. Verknüpft heißt in diesem Fall, dass der Output der Vorgängerschicht dem Input der nachfolgenden Schicht entspricht. Abbildung 2.3 zeigt ein einfaches Beispiel:

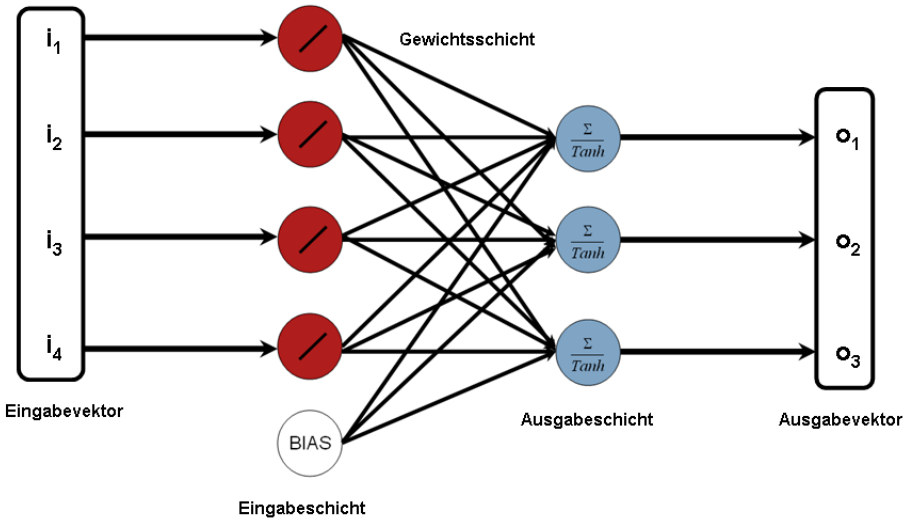


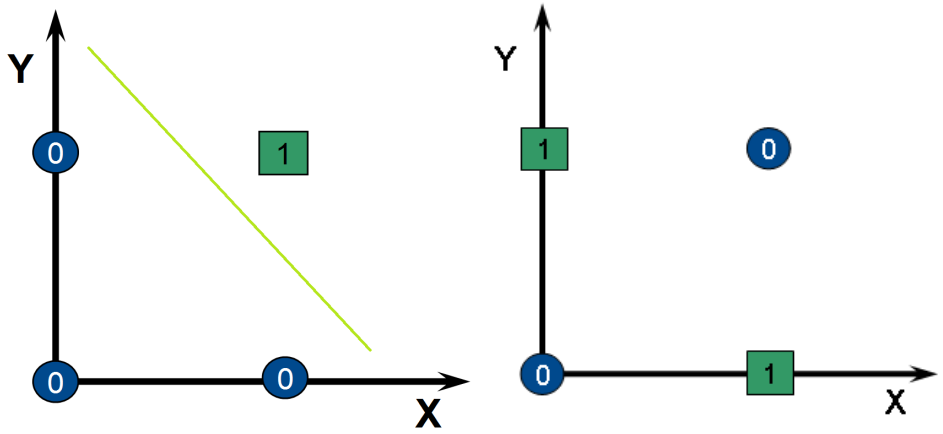
Abbildung 2.3.: Einfaches Modell eines neuronalen Netzes mit 4-dimensionalem Eingabevektor und 3D Ausgabevektor. Der Input der roten Neuronen ist jeweils eine Komponente des Eingabevektors, der Input der blauen Neuronen ist der Output aller vorherigen Neuronen. Die Inputvektoren der blauen Neuronen unterscheiden sich nicht, die verschiedenen Gewichtsvektoren sorgen aber für unterschiedliche Netzeingaben.

Die roten Eingangsneuronen geben an die Outputneuronen ihren Input unverändert weiter (Identität), die Outputneuronen bilden daraus die gewichtete Summe und wenden den Tangens Hyperbolicus an. Zusätzlich wurde ein Bias-Neuron hinzugefügt, das wie oben besprochen als Output immer 1 hat und dessen (negatives) Gewicht als Schwellwert fungiert.

Diese Form des neuronalen Netzes wird einlagiges[5] Perzeptron (Single Layer Perceptron, SLP) genannt und wurde schon 1958 von Frank Rosenblatt publiziert [Ros58]. Mithilfe dieses einfachen Modells ist es schon möglich, komplexe Aufgaben, wie zum Beispiel Echo-Echtzeitfilterung [WH60, S. 96-104], auszuführen. 1969 konnten aber Marvin Minsky und Seymour Papert zeigen, dass ein einlagiges Perzeptron verschiedene Probleme (wie zum Beispiel den XOR-Operator) nicht lösen können [MP69]. Dies liegt daran, dass ein einlagiges Perzeptron nur ein linearer Klassifikator ist und nur Probleme lösen kann, die linear separierbar sind.

[4]In dieser Arbeit werden nur vollverknüpfte Feedforward-Netze betrachtet. Eine kurze Übersicht über andere Topologien gibt [Kri07, Kap. 3.3]

[5]Einlagig bezieht sich hierbei auf die Zahl der Gewichtsschichten.



(a) AND (b) XOR

Abbildung 2.4.: Graphische Darstellung der AND- und XOR-Operatoren. Für den AND-Operator gibt es mehrere Graden, die die Ergebnisse trennen, für den XOR-Fall kann keine Gerade gefunden werden, die die Kreise von den Quadraten trennt.

Um dieses Problem zu lösen, kann man weitere Schichten von Neuronen einfügen (Multi Layer Perceptron, MLP). Abbildung 2.5 zeigt ein zweilagiges Perzeptron, das das XOR-Problem löst. War es mit einem einlagigen Perzeptron nur möglich, lineare Probleme zu separieren, also Hyperebenen im Ausgaberaum zu beschreiben, so kann ein zweilagiges Perzeptron konvexe Polygone beschreiben. Durch Hinzufügen einer weiteren Schicht kann dann jede beliebige Menge im Ausgaberaum beschrieben werden.

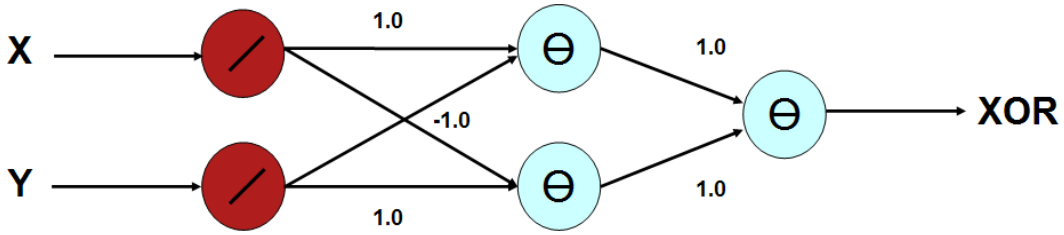


Abbildung 2.5.: Zweilagiges Perzeptron, das einen XOR-Operator simuliert. Die Zahlen an den Pfeilen geben die jeweiligen Gewichte an.

Dreilagige Perzeptrons können also theoretisch alle Probleme lösen. Das größte Problem dabei ist, die optimale Zahl der Neuronen in den versteckten Schichten zu finden. Zu viele Neuronen verlängern das Lernen extrem, zu wenige verhindern die Klassifizierung.

2.3. Lernen

Wie in Kapitel 2.1 beschrieben, werden für jedes Neuron im neuronalen Netz alle Parameter bis auf den Gewichtsvektor festgesetzt. Um neue (Eingabe-) Muster zu lernen, müssen also die Gewichte jedes einzelnen Neurons geändert werden. Hierzu werden zu Beginn der Trainingsphase alle Gewichte zufällig initialisiert. In welchem Wertebereich dies geschieht, ist wieder dem Problem anzupassen. Es empfiehlt sich aber keine Gewichte nahe 0 zuzulassen. Aus diesem Grund wird die Funktion `getRandomInitialWeight()` benutzt, die sich im Anhang B.2 befindet. Die Verteilung der dabei entstehenden Gewichte zeigt Abbildung 2.6.

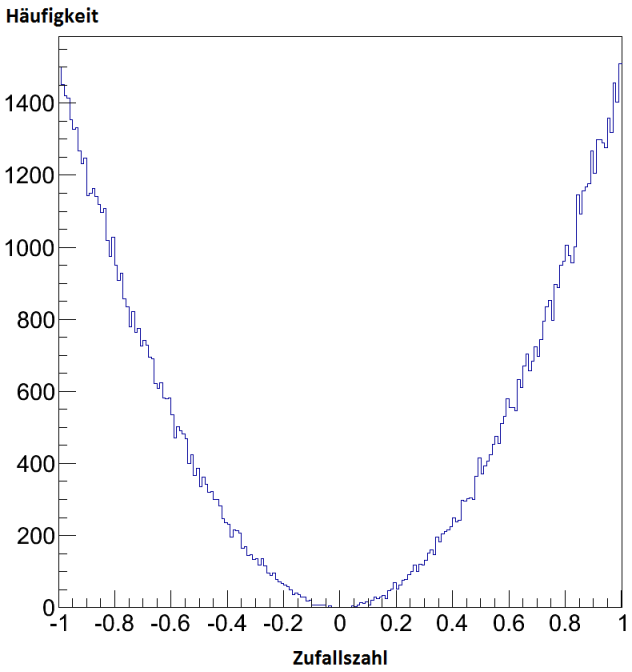


Abbildung 2.6.: Beispielhafte Häufigkeitsverteilung von Zufallszahlen im Intervall [-1;1], die durch die Funktion `getRandomInitialWeight()` erzeugt wurden.

Um neuronale Netze zu trainieren gibt es verschiedene Ansätze. In dieser Arbeit wird nur das überwachte Lernen angewendet. Für eine kurze Beschreibung des unüberwachten und des bestärkenden Lernens sei wieder auf [Kri07, Kap. 4.1] verwiesen.

Beim überwachten Lernen wird das neuronale Netz mit einem Trainingssatz von Daten trainiert. Ein einzelnes Trainingsmuster besteht dabei aus Eingabevektor und dazugehörigem Ausgabevektor. Dem neuronalen Netz wird nun ein Trainingsmuster gegeben und die Ausgabe des Netzes berechnet. Wenn die Ausgabe des Netzes (o_j) mit der gewünschten Ausgabe (t_j, teaching) übereinstimmt, wird nichts verändert, ansons-

ten werden die Gewichte so angepasst, dass dieses Trainingsmuster das nächste Mal besser erkannt wird. Hierfür wird die Differenz zwischen Output und Teachingoutput berechnet und mit zwei Faktoren multipliziert auf das alte Gewicht addiert. Der erste Faktor ist dabei der Input des Neurons (i_i), der zweite eine Lernrate (`gain()`). Die Gleichungen 2.3 und 2.4 fassen dies noch einmal zusammen.

$$\omega_{ij}^{neu} = \omega_{ij}^{alt} + \Delta\omega_{ij} \tag{2.3}$$

$$\Delta\omega_{ij} = gain() \cdot (t_j - o_j) \cdot i_i \tag{2.4}$$

Die Lernrate `gain()` gibt dabei an, wie schnell ein Netz lernt. Ein hoher Wert ($\sim$ 0,9) sorgt dafür, dass neue Muster schnell gelernt, alte aber auch schnell vergessen werden. Ein zu niedriger Wert macht das Trainieren (= Mustererlernen) schwierig und zeitaufwändig. Ein zu hoher Wert verringert das ¨Gedächtnis¨ des neuronalen Netzes.

Es empfiehlt sich also die Lernrate mit der Zeit zu variieren. Am Anfang eine hohe Lernrate führt zu schnellerem Lernen, ein Absinken mit der Zeit verhindert das Vergessen von Trainingsmustern. In dieser Arbeit werden dafür zwei Funktionen verwendet: `gain()` und `eta()`. Die Funktion `gain()` erlaubt dabei einen kontinuierlichen Abfall von einem Startwert a zu einem Endwert e über N Schritte. Bei der Funktion `eta()` gibt es z gleich große Stufen vom Anfangswert a zum Endwert e. Gleichung 2.5 und 2.6 zeigen die Definition der beiden Funktionen.

$$gain(n) = a \cdot \left(\frac{e}{a}\right)^{\frac{n}{N}} \tag{2.5}$$

$$\begin{aligned} intervall &= \lfloor N/z \rfloor \\ eta(n) &= a - (a - e) \cdot \frac{\lfloor n/intervall \rfloor}{z - 1} \end{aligned} \tag{2.6}$$

Die Abbildungen 2.7 und 2.8 zeigen den Verlauf der Lernraten noch einmal anhand von Beispielzahlen.

Die Funktion `gain()` wird für die Single-Layer-Perzeptrons (siehe Kapitel 4) benutzt, die Funktion `eta()` für die zweilagigen Multi-Layer-Perzeptrons (Kapitel 5).

2.3.1. Backpropagation of Error

Die Gewichtsveränderung wie sie in Gleichung 2.4 beschrieben wird, ist nur möglich, wenn der gewünschte teaching output t_j bekannt ist. Wie sieht aber der richtige Output für Eingangsneuronen in einem MLP aus?

Um das Gesamtproblem zu lösen, muss man den Fehler des neuronalen Netzes minimieren. Für neuronale Netz kann man aber verschiedene Fehlerfunktionen (E) definieren. Am einfachsten ist hierbei die quadratische Abweichung:

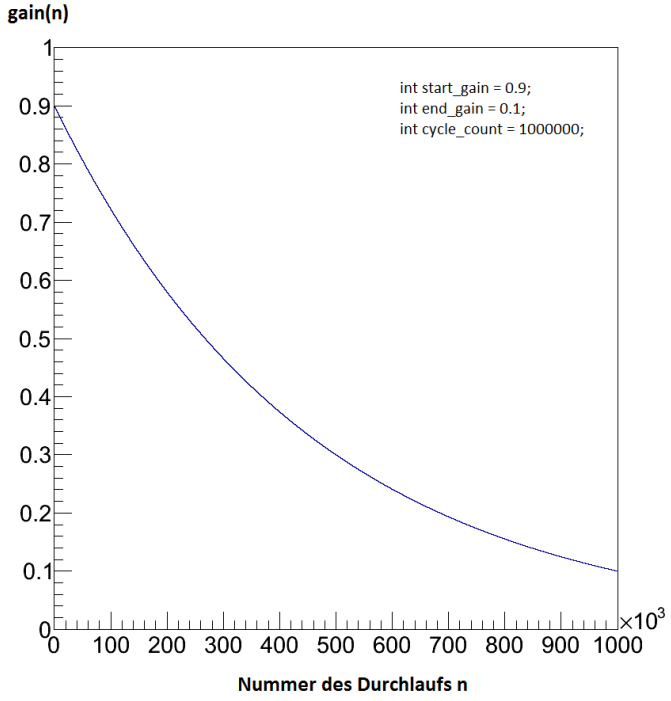


Abbildung 2.7.: Die Funktion `gain()` mit einem Startwert von 0,9 und einem Endwert von 0,1 über 1 Mio. Zyklen.

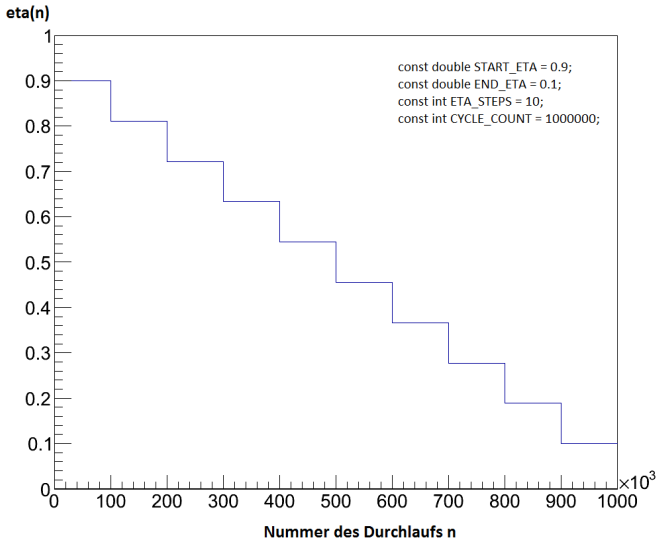


Abbildung 2.8.: Die Funktion `eta()` mit einem Startwert von 0,9 und einem Endwert von 0,1 über 1 Mio. Zyklen und 10 Stufen.

$$E = \sum_{i=1}^{N} \left(0,5 \sum_{k=1}^{\Omega} (t_{k,i} - o_{k,i})^2 \right) \tag{2.7}$$

Für alle N Trainingsmuster des Trainingssatzes wird die quadratische Abweichung des tatsächlichen Outputs vom gewünschten Output über jedes Ausgabeneuron summiert. Der Faktor 0,5 ist dabei Konvention, um die anschließende Rechnung zu vereinfachen. Die beste[6] Konfiguration des neuronalen Netzes (zu den Trainingsdaten) erreicht man nun, wenn man das globale Minimum der Fehlerfunktion findet. Hierzu empfiehlt sich ein Gradientenverfahren[7], bei dem die Änderung der Gewichte in Richtung der größten Abnahme der mehrdimensionalen Fehlerfunktion erfolgt.

Die Änderung eines jeden Gewichtes ist dabei proportional zur partiellen Ableitung der Fehlerfunktion.

$$\Delta\omega_{ij} \sim \frac{\partial E}{\partial \omega_{ij}} \tag{2.8}$$

Als Proportionalitätsfaktor wird η eingeführt, welches nachher durch die oben beschriebene Funktion `eta()` ersetzt wird.

Mithilfe der Kettenregel kann die partielle Ableitung berechnet werden:

$$\Delta\omega_{ij} = \eta \cdot \frac{\partial E}{\partial net_j} \cdot \frac{\partial net_j}{\partial \omega_{ij}} \tag{2.9}$$

$$= \eta \cdot \delta_j \cdot \frac{\partial net_j}{\partial \omega_{ij}} \tag{2.10}$$

Der zweite Term dieser Gleichung lässt sich leicht mit der Definition der Netzeingabe (Gleichung 2.1) berechnen.

$$\frac{\partial net_j}{\partial \omega_{ij}} = \frac{\partial}{\partial \omega_{ij}} \left(\sum_n \omega_{nj} o_n \right) = o_i \tag{2.11}$$

Der in Gleichung 2.10 definierte Term δ_j lässt sich wiederum mit der Kettenregel vereinfachen.

[6] Es kann lediglich nur die beste Konfiguration für die vorhandenen Trainingsdaten gefunden werden. Diese Konfiguration muss aber nicht die beste Konfiguration für das Problem an sich sein. Durch viele Trainingsdaten kann diese Abweichung aber minimiert werden.

[7] Ein Problem von Gradientenverfahren ist, dass sie oft nur lokale Minima finden. Verändern der Startbedingungen kann dieses Problem aber teilweise umgehen. Der größte Vorteil der Gradientenverfahren ist ihre Schnelligkeit.

$$\begin{aligned}\delta_j &= \frac{\partial E}{\partial net_j} \qquad (2.12)\\ &= \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}\\ &= \frac{\partial E}{\partial o_j} \cdot \frac{\partial}{\partial net_j}\left(f_{act}(net_j)\right)\\ &= \frac{\partial E}{\partial o_j} \cdot f'_{act}(net_j) \qquad (2.13)\end{aligned}$$

Hier wird ersichtlich, warum die Aktivierungsfunktion stetig differenzierbar sein muss. Für die partielle Ableitung wird die Ableitung der Aktivierungsfunktion benötigt.

Der erste Term in Gleichung 2.13 ist von der Position des Neuron j im neuronalen Netz abhängig. Gehört das Neuron zur Ausgabeschicht, so ist die Fehlerfunktion E genau die in Gleichung 2.7 beschriebene und kann leicht differenziert werden. Hierbei wird auch ersichtlich, warum der Faktor 0,5 ergänzt wurde. Er kürzt sich mit der 2 der Ableitung und es kann ohne Vorfaktor weitergerechnet werden.

$$\frac{\partial E}{\partial o_j} = \frac{\partial}{\partial o_j} 0,5 \sum_{k=1}^{\Omega} (t_k - o_k)^2 = -(t_k - o_k) \qquad (2.14)$$

Ist j ein inneres Neuron, so bestimmt sein Output die nachfolgenden Neuronen mit und die Fehlerfunktion für dieses Neuron (und damit für das dazugehörige Gewicht) ist eine Funktion der folgenden Netzeingaben.

$$\frac{\partial E}{\partial o_j} = \frac{\partial E(net_1, net_2 ...)}{\partial o_j} = \sum_n \left(\frac{\partial E}{\partial net_n} \cdot \frac{\partial net_n}{\partial o_j} \right) \qquad (2.15)$$

Die Summe über n bezieht sich dabei auf alle Neuronen der nachfolgenden Schicht. Es fällt auf, dass der erste Term von Gleichung 2.15 genau der Definition des δ aus Gleichung 2.12 entspricht. Der zweite Term lässt sich wieder leicht berechnen.

$$\begin{aligned}\frac{\partial E}{\partial o_j} &= \sum_n \left(\delta_n \cdot \frac{\partial net_n}{\partial o_j} \right)\\ &= \sum_n \left(\delta_n \cdot \frac{\partial}{\partial o_j} (\sum_i \omega_{in} o_i) \right)\\ &= \sum_n \delta_n \cdot \omega_{jn} \qquad (2.16)\end{aligned}$$

Mit den Ergebnissen aus den Gleichungen 2.9, 2.13, 2.14 und 2.16 folgt schließlich für die Gewichtsveränderung:

$$\begin{aligned} \Delta\omega_{ij} &= \eta \cdot \delta_j \cdot o_i \\ \delta_j &= \begin{cases} f'_{act}(net_j) \cdot (t_j - o_j), & \text{wenn } j \text{ in Ausgabeschicht} \\ f'_{act}(net_j) \cdot \sum_n \delta_n \cdot \omega_{jn}, & \text{sonst.} \end{cases} \end{aligned} \tag{2.17}$$

Für die Gewichtsänderung einer inneren Schicht wird also immer die Änderung der nachfolgenden Schicht benötigt. Daher kommt auch der Name des Verfahrens. Die Änderungen werden von hinten (Ausgabeschicht) nach vorne (Eingabeschicht) berechnet und angewendet: *Backpropagation of Error*.

3. Technische Details

In dieser Arbeit werden zwei Ansätze vorgestellt, um die simulierten Daten mithilfe von neuronalen Netzen in Ortsinformationen umzuwandeln: Ein einfaches Single Layer Perzeptron, das im nächsten Kapitel beschrieben wird und ein Multi Layer Perzeptron mit einer versteckten Schicht, das im 5. Kapitel behandelt wird.

3.1. verwendete Programme

In dieser Arbeit wird absichtlich auf vorhandene Bibliotheken und fertige Software zu dem Thema neuronale Netze verzichtet. Durch das Programmieren von Grund auf, ist es jeder Zeit möglich Änderungen leicht vorzunehmen. Leider steigt dadurch auch die Zahl der Fehler, die gerade bei neuronalen Netzen schwierig zu finden sind. Dies liegt daran, dass ein fertiges neuronales Netz quasi eine Blackbox ist. Man gibt einen Vektor ein und es kommt ein Ergebnis raus. Warum das Ergebnis so aussieht, kann in den meisten Fällen nicht nachvollzogen werden.

Alle Programme sind in C++ programmiert. Diese Sprache wurde gewählt, weil sie leicht zu verstehen ist und fertige Programme relativ schnell laufen. Zur Visualisierung der Ergebnisse wird das ROOT-Paket des CERN verwendet.

3.2. Ausgabe

Jedes der in dieser Arbeit beschriebenen neuronalen Netze gibt als Output einen zweidimensionalen Array aus. Jede Zelle steht dabei für die Wahrscheinlichkeit, dass das Teilchen an diesem Ort auf den Detektor getroffen ist. Zur Ausgabe werden diese Werte nun in ein zweidimensionales Histogramm geschrieben. Da ROOT die Graustufen relativ zu Minimum und Maximum wählt, kann auf eine Normierung des Outputs verzichtet werden. Abbildung 3.1 gibt ein kleines Beispiel.

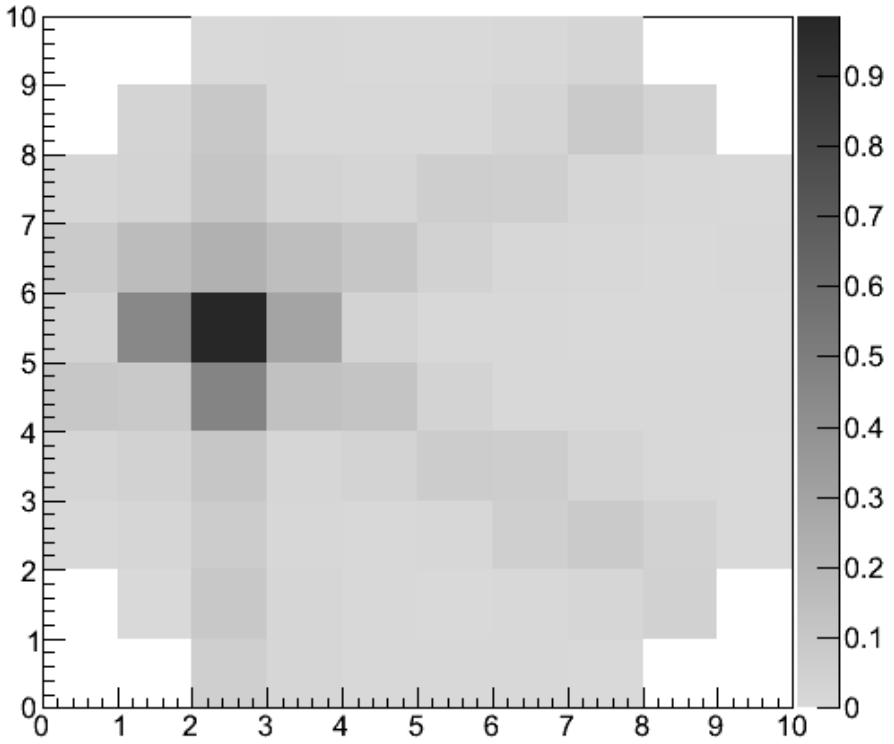


Abbildung 3.1.: Beispiel-Ausgabe eines neuronalen Netzes. Der Detektor wurde in jede Richtung in 10 gleich große Abschnitte aufgeteilt. Die Graufärbung gibt die Wahrscheinlichkeit an, dass das Teilchen in dieser Zelle auf den Detektor getroffen ist. Schwarz steht für eine hohe Wahrscheinlichkeit, weiß für eine geringe.

4. Single Layer Perceptron

4.1. Implementation

Im ersten Ansatz für ein Single Layer Perzeptron werden die Schichten von Neuronen mithilfe von zwei Arrays im Double-Format dargestellt. `double input_layer [7] [SAMPLERATE]`[1] steht dabei für den Eingabevektor, der wiederum gleichzeitig die Ausgabe der Eingabeschicht ist, da Eingabeneuronen keine Propagierungsfunktion haben (es gibt keine vorherigen Gewichte) und die Identität als Aktivierungsfunktion gewählt wurde. Die Ausgabe des neuronalen Netzes wird in `double output_layer [OUTPUT_LAYER_CELL_COUNT] [OUTPUT_LAYER_CELL_COUNT]` abgelegt. Die Auflösung des Netzes wird zu Beginn in `OUTPUT_LAYER_CELL_COUNT` festgelegt und kann leicht variiert werden. Es fällt auf, dass Eingabe- und Ausgabeschicht keine eindimensionale Schichten sind, sondern zweidimensionale Arrays. Für die Realisierung des Netzes ist dies nicht relevant, Eingabe- und Ausgabevektor können so aber leichter ausgelesen werden.

Zum Trainieren wird dann eine Schleife aufgerufen, deren genaue Anzahl an Durchläufen in `cycle_count` festgelegt ist. Zu Beginn des Trainings werden die Gewichte nicht initialisiert, da beim Neusetzen der Gewichte nur das Intervall von 0 bis 1 erlaubt ist. In jedem Schleifendurchlauf wird dann ein Datensatz generiert und das neuronale Netz damit trainiert. Dazu wird, wie oben beschrieben, die Ausgabe des neuronalen Netzes mithilfe von `calculateOutput(int status)` (siehe Anhang B.3) berechnet und danach mit `trainieren()` (vgl. Anhang B.4) die Gewichte angepasst.

4.2. Single Hit

4.2.1. Erste Implementation

Beim ersten Programm wurde für die Ausgabeneuronen die Stufenfunktion als Aktivierungsfunktion gewählt. Um einen ersten Eindruck für gute Parametereinstellungen zu bekommen wurden die Parameter[2] zufällig gewählt (innerhalb eines definierten Bereichs) und das neuronale Netz damit trainiert. Hinterher wurde die Güte des Netzes mithilfe von 10.000 zufälligen Hits getestet. Die Ausgabe des neuronalen Netzes wurde dabei in sechs Klassen kategorisiert:

richtig nur die richtige Zelle feuert

[1] 7 entspricht dabei der Zahl der Signale, `SAMPLERATE` ist die Länge der Signale.
[2] Schwellwert, Startlernrate, Endlernrate, Zahl der Trainingsdurchläufe

richtig mit Nachbarn die richtige Zelle leuchtet, aber auch Zellen in direkter Nachbarschaft geben eine 1

richtig mit entfernten Nachbarn die richtige Zelle leuchtet, aber auch andere Zellen, die weiter entfernt sind, geben ein Signal

falsch mit richtigen Nachbarn die richtige Zelle leuchtet nicht, aber eine der Nachbarzellen

komplett falsch nur entfernte Zelle leuchten

kein Signal wenn keine Zelle eine 1 ausgibt

Beispielhafte Bilder für diese Kategorien sind in Abbildung 4.1 gezeigt.

Insgesamt wurden knapp 800 neuronale Netze auf diese Weise kreiert und getestet. Die abschließende Sortierung erfolgte nach der Häufigkeit von richtigen Mustern. Die besten Einstellungen sind in Tabelle 4.1 zusammengefasst.

Durchläufe	Schwellwert	StartGain	EndGain	richtig	kein Signal
830.000	0,89	0,0091	0,0011	88,60%	4,70%
1.000.000	0,92	0,2535	0,0011	86,30%	5,20%
551.200	0,71	0,7732	0,0007	83,00%	–
100.000	0,81	0,7185	0,0011	81,60%	9,10%
1.000.000	0,88	0,3883	0,0033	80,40%	10,50%
410.500	0,58	0,9635	0,002	80,10%	10,20%

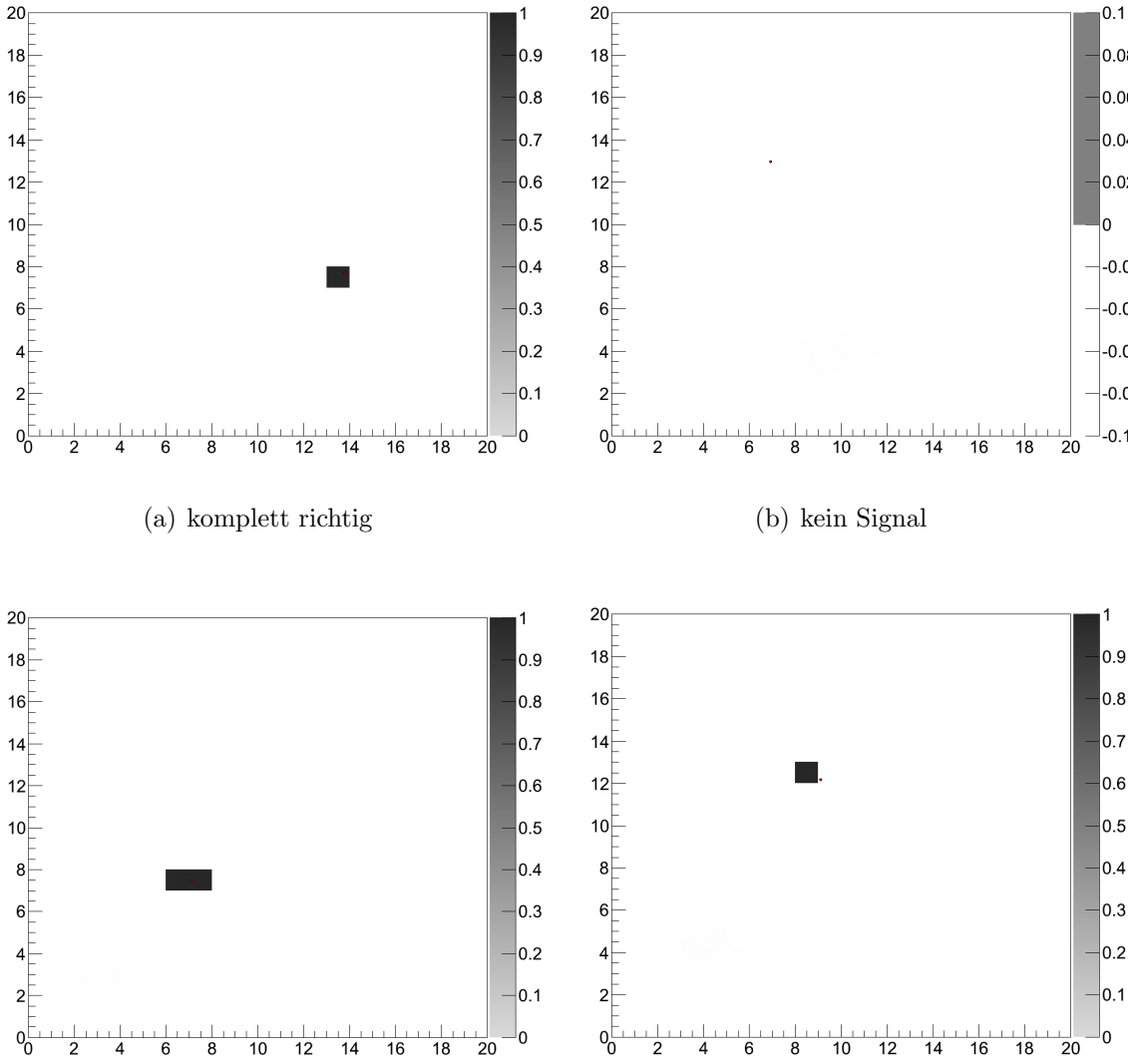
Tabelle 4.1.: Die besten Parametereinstellungen aus der Testreihe mit 788 neuronalen Netzen.

Es fällt auf, dass vor allem eine hohe Anzahl an Trainingsdurchläufen und hohe Schwellwerte zu einem guten Ergebnis führten. Man kann auch erkennen, dass eine niedrige Lernrate zum Schluss das Ergebnis verbessert, da alte Muster nicht so leicht vergessen werden.

4.2.2. Verbesserungen

Die binäre Ausgabe der Outputneuronen macht es schwierig die Fehlerrate zu minimieren (wie in Tabelle 4.1 gezeigt, konnte kein Netz mehr als 90% der Testmuster richtig erkennen). Als nächstes wurde dann mit der besten gefundenen Einstellung das neuronale Netz trainiert, als Aktivierungsfunktion wurde aber zur Ausgabe die Identität gewählt. Abbildung 4.2 a) zeigt eine beispielhafte Ausgabe.

Der Bereich um den tatsächlichen Auftreffpunkt ist relativ dunkel und der Ort kann nur schwer bestimmt werden. Um die Auflösung weiter zu verbessern, wurden im nächsten Schritt die Ausgabewerte einfach potenziert. Die beste Auflösung ergibt sich mit der kubischen Funktion und ist in Abbildung 4.2 b) an einem Beispiel gezeigt.



(a) komplett richtig

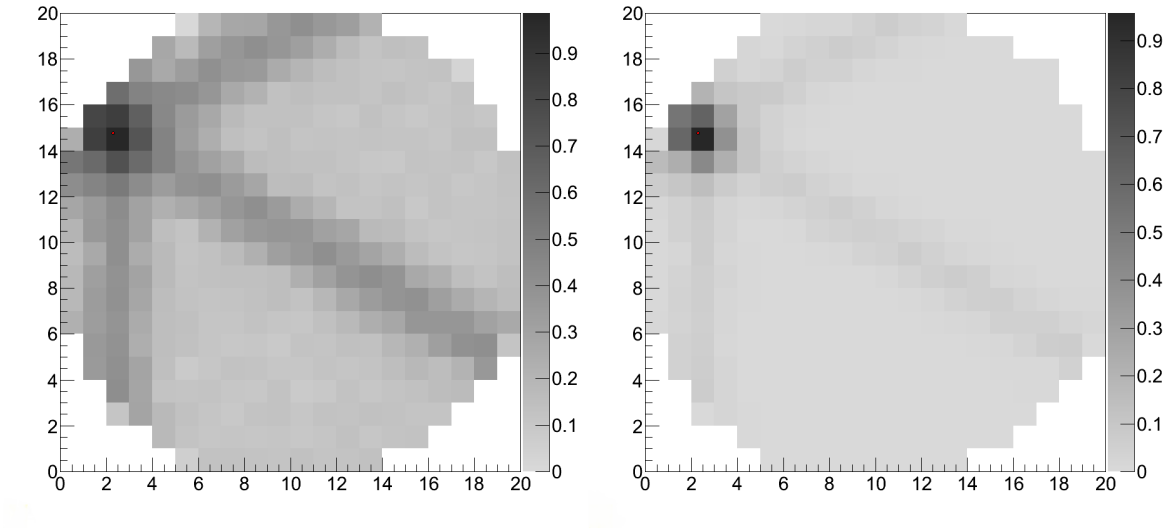
(b) kein Signal

(c) richtig mit Nachbar

(d) falsch mit richtigen Nachbarn

Abbildung 4.1.: Beispielausgaben des ersten SLP. Die vier Bilder geben die häufigsten möglichen Ergebnisse an. Der kleine Punkt zeigt jeweils den tatsächlichen Auftreffpunkt des Teilchens auf dem Detektor an.

Nach der Veränderung der Aktivierungsfunktion (zur Ausgabe der Daten) konnten auch die vorher nicht richtig erkannten Daten besser verarbeitet werden. Der Fall, dass es kein Signal gibt, tritt gar nicht mehr auf und in allen anderen Fällen kann über eine geschickte Interpretation des Outputs der Auftreffort interpoliert werden.



(a) Identität (b) Kubisch

Abbildung 4.2.: Beispielhafte Ausgabe eines SLP das mit der Stufenfunktion als Aktivierungsfunktion trainiert wurde, für die Ausgabe aber links die Identität und rechts die kubische Funktion gewählt wurde.

4.2.3. Interpretation

Das neuronale Netz liefert keine direkten x-y-Ortsinformationen für das Teilchen. Jedes verarbeitete Muster ergibt lediglich eine Wahrscheinlichkeit, dass eine Zelle auf dem Detektor von einem Teilchen getroffen wurde. Diese Ausgabe des neuronalen Netzes muss nun mit einem zweiten Algorithmus interpretiert werden. Die einfachste Möglichkeit für den Single Hit-Fall ist eine Box zunehmen, die genauso groß ist wie eine Zelle und deren Mittelpunkt auf den Mittelwert aller Zellen zu setzen, deren Ausgabewert größer als 50% des Maximums ist. Mit dieser Methode lagen alle getesteten Auftreffpunkte innerhalb der Box, es gehen aber Informationen verloren, da die Ausgabe des Netzes genauer ist.

Um die Interpretation zu verbessern, kann über die gewählten Zellen (die einen Ausgabewert über 50% des Maximums haben) eine *gewichtete* Mittelwertbildung gemacht werden. Um diesen Mittelpunkt kann dann ein relativ kleiner Kreis gelegt werden. Abbildung 4.3 zeigt die beiden Methoden an einem Beispiel.

Mithilfe des gewichteten Mittelwertes kann der Auftreffpunkt sehr genau bestimmt werden, wie Abbildung 4.4 zeigt. Wählt man den Kreisradius zu 0,25 (Bins) so liefert das neuronale Netz in 99,7% aller 10.000 Trainingsdaten den richtigen Ort.

Die meisten Tests der neuronalen Netze wurden mit einer Auflösung von 20x20 gemacht, da hierbei die Auftrefforte gut berechnet werden konnten, die Rechenzeit aber nicht zu groß wurde. Höhere Auflösungen können auch berechnet werden, wie Abbil-

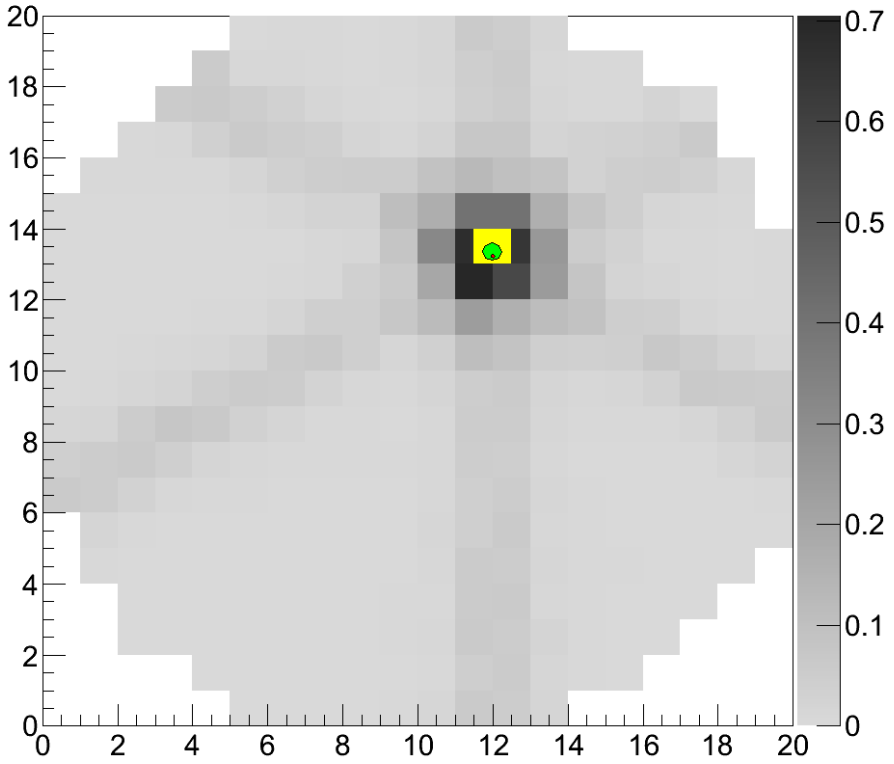


Abbildung 4.3.: Ausgabe eines SLP, die auf zwei Arten interpretiert wurde. Das gelbe Quadrat entspricht der Box-Methode (einfache Mittelwertbildung), der grüne Kreis ergibt sich mithilfe der gewichteten Mittelwertbildung. Aufgrund der Interpretation konnte der Auftreffpunkt richtig erkannt werden, obwohl er nicht mit dem Maximum der Ausgabe übereintrifft.

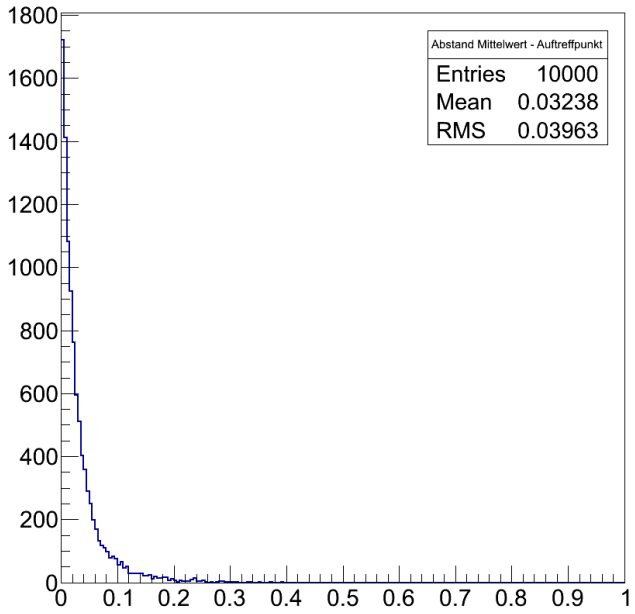


Abbildung 4.4.: Abstand des gewichteten Mittelwertes zum tatsächlichen Auftreffpunkt. Die x-Achse gibt nicht den Radius, sondern die Fläche der dazugehörigen Kreisscheibe an.

dung 4.5 zeigt. Der Kreisradius für die Interpretation muss dann nur entsprechend angepasst werden[3].

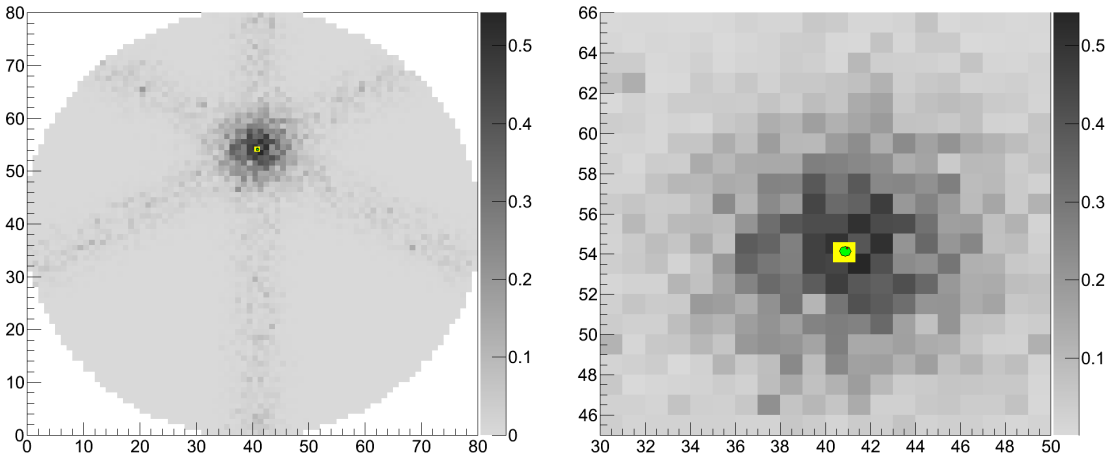


Abbildung 4.5.: Beispielhafte Ausgabe eines SLP, dessen Auflösung auf 80x80 erhöht wurde. Im rechten Bild ist der Auftreffpunkt vergrößert dargestellt.

4.3. Double Hit

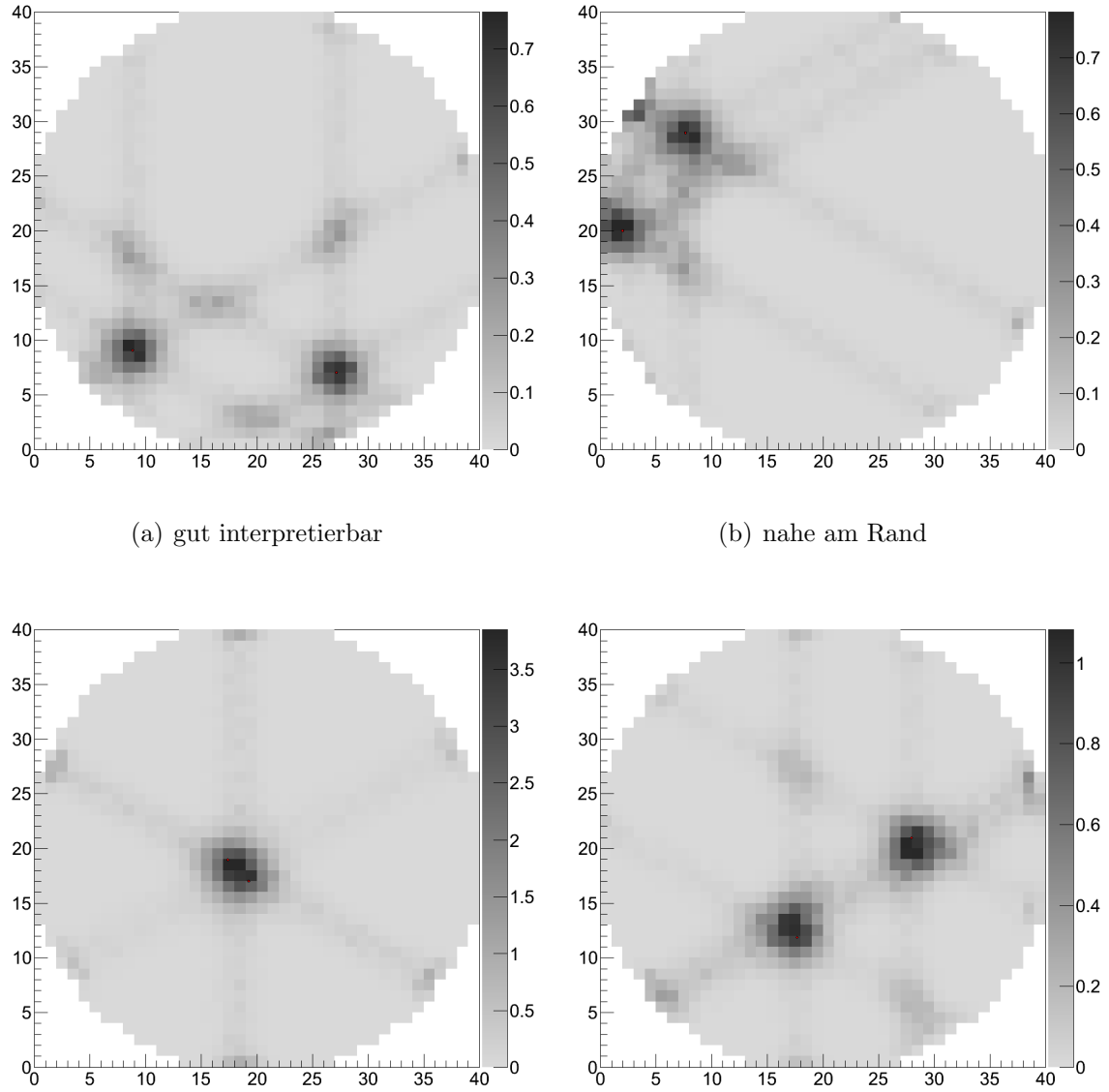
Bisher wurde nur der Fall betrachtet, wenn *ein* Teilchen den Detektor trifft (Single Hit). Dieser Fall ist relativ einfach und kann auch ohne neuronale Netze gelöst werden. Die Hauptaufgabe der neuronalen Netze soll aber die Detektion von Multi Hits sein. Aus diesem Grund wurde im nächsten Schritt ein SLP wie im vorherigen Abschnitt trainiert, zur Ausgabe wurden dann die Signale von zwei verschiedenen Hits aufsummiert. Dies entspricht dem physikalischen Fall, wenn zwei Elektronen perfekt gleichzeitig auf den Detektor treffen. Diese Vereinfachung ist physikalisch nicht unbedingt sinnvoll, gibt aber erste Eindrücke über das Verhalten der neuronalen Netze in diesem Fall.

Erstaunlicherweise sehen die Ausgaben des neuronalen Netzes immer noch sehr sinnvoll aus (siehe Abbildung 4.6). Die Frage ist nun, wie man von der Ausgabe des Netzes zu den tatsächlichen Auftrefforten kommt.

Wie in der Abbildung zu sehen, können Hits, die weit auseinander liegen, genauso interpretiert werden, wie im Single Hit Fall. Kommen die Auftreffpunkte sich aber nahe (4.6c) ist eine Unterscheidung zum Single Hit nicht möglich[4].

[3] Der Radius ist in ROOT relativ zur Bingröße angegeben. Eine höhere Binzahl verkleinert also den absoluten Radius des Kreises.

[4] Die Höhe des Maximums ist bei zwei Teilchen natürlich höher und somit ein Indiz für mehrere



(a) gut interpretierbar

(b) nahe am Rand

(c) geringer Abstand

(d) leicht versetzt auf einem Layer

Abbildung 4.6.: Beispielhafte Ausgabe eines SLP. Die Signale von zwei zeitgleichen Hits wurden addiert und dann erst vom SLP verarbeitet.

Auf allen Visualisierungen der SLP-Ausgaben ist zu sehen, dass von jedem Maximum Linien im 60° Winkel abgehen. Dies zeigt die Funktionsweise des SLP. Aus den

Teilchen. Wirkliche Messwerten können aber auch in der Höhe schwanken, sodass dies kein guter Indikator für die Teilchenzahl ist.

Signalen werden die getroffenen Layer berechnet und dann die Signale der drei Layer übereinander gelegt. Am Schnittpunkt ist das Signal am stärksten und man sieht das Maximum. Abbildung 4.7 verdeutlicht das Problem noch einmal. Nachdem alle Zellen, deren Ausgabe kleiner als 32% des Maximums ist, entfernt wurden, sieht man die verbleibende Detektorstruktur[5].

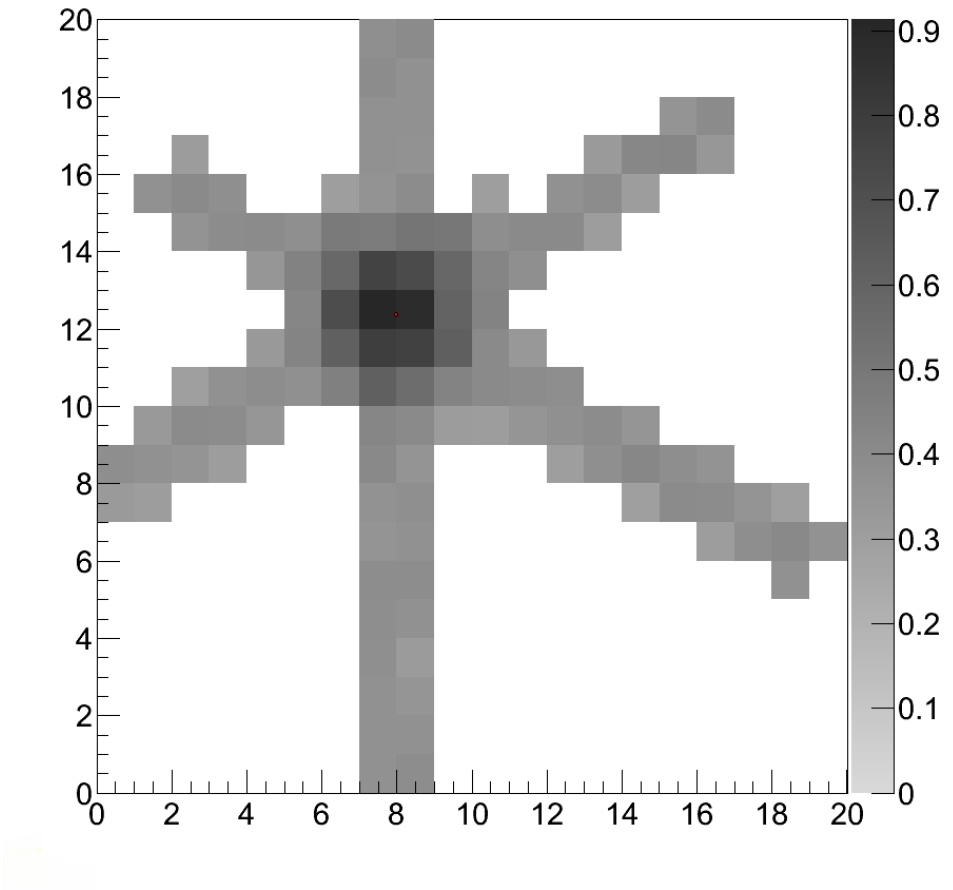


Abbildung 4.7.: Ausgabe eines SLP bei dem alle Einträge, die kleiner als 32% des Maximums sind, entfernt wurden. Die Detektorstruktur ist gut zu erkennen.

Beim Single Hit ist dies kein Problem, da immer nur ein Layer getroffen wird. Treffen aber zwei Teilchen auf den Detektor, können diese Linien sich nahe kommen und so die Interpretation der Ausgabe sehr erschweren (siehe 4.6 d).

Mithilfe der im Rahmen dieser Arbeit getesteten SLP konnten die Auftrefforte daher

[5]Dieser Effekt soll verschwinden, wenn der Teachingoutput zum Training des SLP nicht nur eine 1 enthält, sondern eine Gauß-Kurve um den Einschlag. Näheres dazu findet sich in der Bachelor-Arbeit von Dennis Schmidt [Sch12].

nicht genau bestimmt werden, da die Ausgabe des SLP nicht gut interpretiert werden konnte. Die Hoffnung liegt deswegen bei den MLP, da die weitere Schicht die Ergebnisse des SLP verbessern könnte.

5. Multi Layer Perceptron

Bei der Implementation der SLP wurde immer ein fester Schwellwert für die Neuronen gewählt. Um die MLP flexibler zu machen, wird wie in Kapitel 2.2 besprochen ein Bias-Neuron eingeführt. Damit das Trainingsverfahren keine Rücksicht auf *ein* andersartiges Neuron neben muss, werden statt der Arrays mit `double` Einträgen nun Arrays von `Neuronen` benutzt. Hierzu werden neben der abstrakten Klasse `Neuron` noch drei Unterklassen `Lin_Neuron`, `Bias_Neuron` und `Hidden_Neuron` definiert. Abbildung 5.1 zeigt das entsprechende UML[1]-Diagramm.

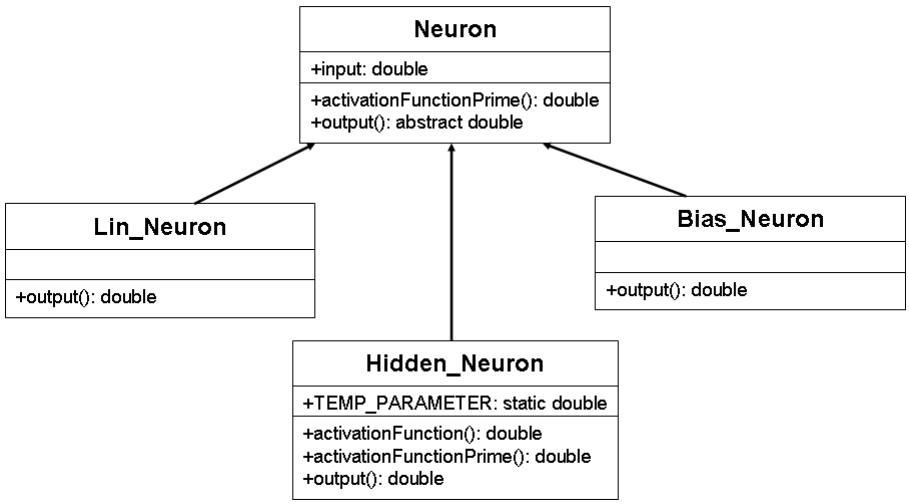


Abbildung 5.1.: UML-Diagramm der verwendeten Neuronenklassen.

Jede Unterklasse muss dabei die Funktion `output()` neu definieren, da diese nur virtuell in der Oberklasse definiert ist. Dieser objektorientierte Ansatz macht das Verändern der Neuronen in jeder Schicht viel einfacher. Als weitere Vereinfachung sind die Neuronen jetzt in eindimensionalen Schichten angeordnet und nicht in zweidimensionalen Arrays. Zum Training des Netzes wird das in Kapitel 2.3.1 besprochene Backpropagation of Error-Verfahren benutzt.

[1] Unified Modeling Language. Ein Standard zur Darstellung von objektorientierten Klassen.

5.1. Die ersten Implementationen

Zu Beginn musste wieder ein grober Bereich für die Parameter des neuronalen Netzes gefunden werden. Da das Training eines MLP sehr viel länger dauert, konnten nicht so viele Tests mit unterschiedlichen Parametern durchgeführt werden. Die Zahl der Neuronen in der versteckten Schicht wurde am Anfang auf 200 und für die Neuronen der Ausgabeschicht auf 100 festgesetzt[2].

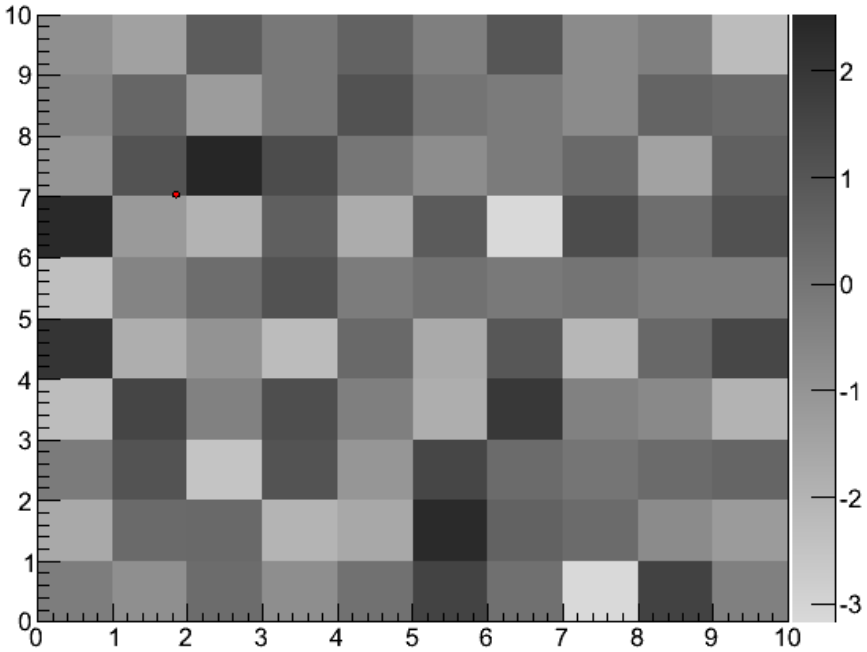


Abbildung 5.2.: Ausgabe eines der ersten MLP. Es ist keine Struktur zu erkennen.

Die ersten MLP konnten das Single Hit-Problem überhaupt nicht lösen (siehe Abbildung 5.2). Unabhängig von den Parameterwerten, sah die Ausgabe für jedes Eingabemuster etwa gleich aus. Erst nach vielen Tests konnten bessere Ergebnisse erzielt werden (vgl. Abbildung 5.3).

Im Rahmen dieser Arbeit war es aber nicht möglich, diese Art der MLP soweit zu verbessern, dass eine genaue Ortsinformation für Single Hits auslesbar ist.

[2]Der Zeitbedarf zum Lernen stieg bei Messungen linear zur Anzahl der versteckten Neuronen.

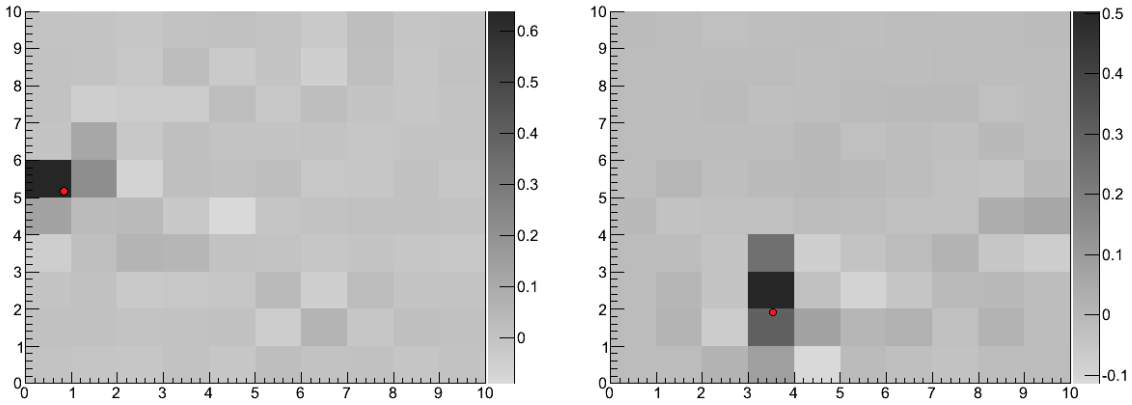


Abbildung 5.3.: Beispielausgabe eines MLP. Links stimmen das Maximum der Ausgabe und der Auftreffort überein, rechts kann auch mit einer Mittelwertsbildung der Ort nicht bestimmt werden.

5.2. Pruning

Um die Auflösung des MLP zu verbessern wurde im nächsten Schritt versucht, Vorwissen in die Struktur des neuronalen Netzes einzubauen. Dazu wird die versteckte Schicht in drei Unterschichten aufgeteilt und jede dieser drei Schichten bekommt als Input das MCP-Signal, sowie die zwei Signale *einer* Anode. Dadurch soll jede Unterschicht das Signal einer Anode wiederspiegeln. Die Ausgabeschicht setzt diese Signale dann zu einem Auftreffpunkt zusammen.

Mithilfe dieses Beschneiden (Pruning) des neuronalen Netzes sollten sich die Ergebnisse verbessern. Aber auch diese Variante hat die Ausgaben nicht verändert.

5.3. 2 SLP = 1 MLP?

Die in Kapitel 4 vorgestellten SLP liefern immer die besseren Ergebnisse, als die MLP aus diesem Kapitel. Lediglich die Interpretation der Double Hit-Ausgaben war schwierig. Warum kann diese Aufgabe nicht auch von einem SLP übernommen werden? Um dies testen, wird ein SLP, wie in Abschnitt 4.2 beschrieben, mit Daten von Single Hits trainiert. Danach werden mit dem Netz Double Hit Signale verarbeitet und der Output dann als Input für ein zweites SLP benutzt (Double Single Layer Perceptron, DSLP). Dieses wird mit den veränderten Eingangswerten auf die gleich Art wie das erste trainiert. Diese Vorgehensweise hat den Vorteil, dass das komplizierte Backpropagation of Error-Verfahren nicht nötig ist, da immer nur eine Gewichtsschicht trainiert wird. Die Ergebnisses dieses MLP aus zwei einzeln trainierten SLP sind in Abbildung 5.4 gezeigt.

Leider liefert auch dieses neuronale Netz noch keine guten Ergebnisse für Double Hit-

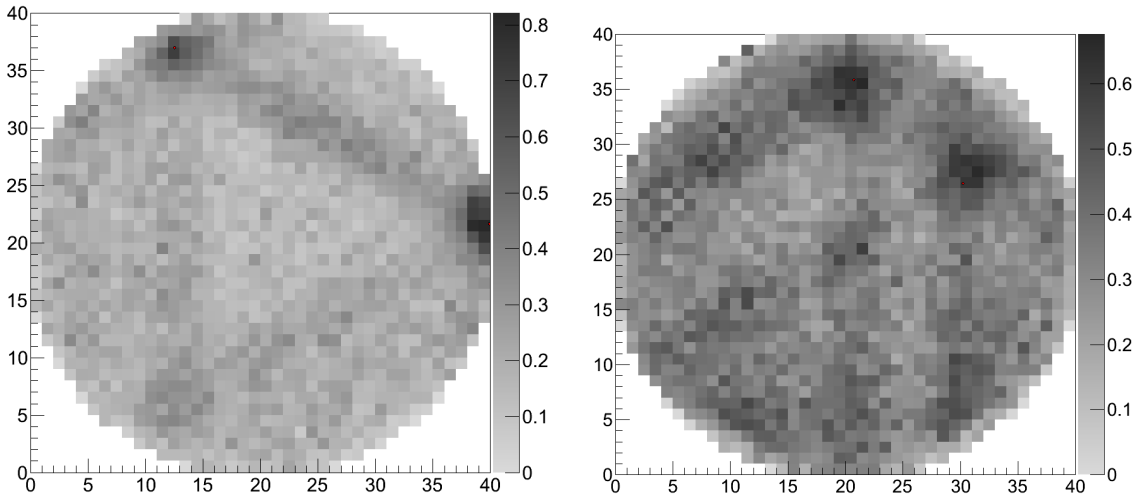


Abbildung 5.4.: Ausgabe eines DSLP. Die Ausgabe eines trainierten SLP wird als Input für ein weiteres SLP benutzt. Links sind die beiden Hits gut zu erkennen, rechts ist eine Interpretation schwierig.

Signale. Durch besseres Trainieren der ersten und der zweiten Schicht ist es vielleicht möglich die Ausgabe zu verbessern.

6. Zusammenfassung & Ausblick

Nachdem die ersten beiden Kapitel einen theoretischen Überblick gegeben haben, wurde in den Kapiteln 4 und 5 die Ergebnisse dieser Arbeit vorgestellt. Mit den Single Layer Perzeptrons, die in Kapitel 4 erklärt wurden, ist es möglich Single Hit Events genau zu berechnen. Durch erhöhten Trainingsaufwand müsste es möglich sein, eine Auflösung von weniger als $1mm$ zu erreichen. Auch die zwei Einschläge bei Double Hits konnten grafisch gut getrennt werden, lediglich eine automatisierte Interpretation gestaltet sich schwierig.

Die im 5. Kapitel beschriebenen Multi Layer Perzeptron lieferten überwiegend schlechte Ergebnisse. Erst durch langwieriges Training (Faktor 100 größer als bei den SLP) konnten gute Ergebnisse erzielt werden. Möglicherweise können bessere Parametereinstellungen gefunden werden, sodass die Auflösung und Interpretation der Daten besser wird.

Die besten Ergebnisse lieferte das DSLP aus dem letzten Abschnitt. Eine Verbesserung der beiden einzelnen SLP ermöglicht sicherlich eine genauere Vorhersagekraft und ist daher ein guter Kandidat für weitere Untersuchungen.

Dem gegenüber stehen aber die Vereinfachungen, die in dieser Arbeit angenommen wurden. Bei allen beschriebenen Tests waren immer alle sieben Signale perfekt vorhanden und es gab keine Art von Rauschen[1]. Es bleibt zu prüfen, wie robust die beschriebenen neuronalen Netze gegenüber echten Daten sind.

Im Rahmen einer nachfolgenden Arbeit kann die Anwendung der SLP und der MLP sicherlich verbessert werden. Auch eine Verarbeitung von echten Daten müsste dann untersucht werden.

Der größte und wichtigste Schritt stellt aber die Einbeziehung der MCP-Signale dar. In dieser Arbeit sind alle Teilchen immer zum gleichen Zeitpunkt auf den Detektor getroffen. Eine Änderung der Auftreffzeit verschiebt alle Werte und erschwert die Auswertung. Am besten wäre eine Lösung, die kontinuierliche Daten verarbeitet und neben der Ortsinformation, die Zeitinformation für alle Teilchen ausgibt.

[1] Bei einem kleinen Test war das Fehlen von ein oder zwei Signalen für die SLP kein Problem.

A. Literaturverzeichnis

[JCC+02] Jagutzki, O. ; Cerezo, A. ; Czasch, A. ; Dörner, R. ; Hattass, M. ; Huang, M. ; Mergel, V. ; Spillmann, U. ; Ullmann-Pfleger, K. ; Weber, T. ; Schmidt-Böcking, H. ; Smith, G. D. W.: Multiple Hit Readout of a Microchannel Plate Detector With a Three-Layer Delay-Line Anode. In: *IEEE Transaction on Nuclear Science* (2002)

[JMUP+98] Jagutzki, O. ; Mergel, V. ; Ullmann-Pfleger, K. ; Spielberger, L. ; Meyer, U. ; Doerner, R. ; Schmidt-Boecking, H. W.: Fast position and time-resolved read-out of micro-channelplates with the delay-line technique for single-particle and photon-detection. In: *Proc. SPIE.* 3438 (1998), October

[Kri07] Kriesel, D.: *Ein kleiner Überblick über Neuronale Netze.* 2007. – erhältlich auf http://www.dkriesel.com (aufgerufen am 15.5.2012)

[Mec06] Meckel, M.: *Strong-Field Ionization of Aligned Oxygen*, Goethe Universität Frankfurt, Diplomarbeit, 2006

[MP43] McCulloch, W. ; Pitts, W.: A logical calculus of the ideas immanent in nervous activity. In: *Bulletin of Mathematical Biophysics* (1943)

[MP69] Minsky, M. ; Papert, S.: Perceptrons. An Introduction to Computional Geometry. In: *MIT Press* (1969)

[Ros58] Rosenblatt, F.: The perceptron - a probablistic model for information storage and organization in the brain. In: *Psychological Review 65* (1958)

[Sch12] Schmidt, D.: *Bachelor-Arbeit.* 2012. – noch nicht veröffentlicht

[WH60] Widrow, B. ; Hofft, M.: Adaptive switching circuits. In: *Proceedings WESCON* (1960)

B. Anhang

B.1. Quellcode zur Signalerzeugung

```
//hit.x = x-Wert des Einschlags (globale Variable)
//hit.y = y-Wert des Einschlags (globale Variable)
//input_layer = Eingabevektor (globale Variable)

double timeA[7];  //Ort des Maximum des Signals
double f = 2.0; //Umrechnungsfaktor Strecke -> Zeit
double width = 10;
// FWHM, full width at half maximum in nanoseconds


// Erstellen eines erlaubten Einschlags
while (1) {
  hit.x = dzufall(-MCP_RADIUS, MCP_RADIUS);
  hit.y = dzufall(-MCP_RADIUS, MCP_RADIUS);
  if (hit.x*hit.x + hit.y*hit.y < MCP_RADIUS*MCP_RADIUS)
    break;
}

// Festsetzen des MCP-Signals und berechnen der Mittelpunkte
timeA[6] =    60.0; // MCP signal
timeA[0] =    0.5 *hit.x*f + timeA[6];
timeA[1] =   -0.5 *hit.x*f + timeA[6];
timeA[2] =    0.25*(hit.x - hit.y*sqrt(3.))*f + timeA[6];
timeA[3] =   -0.25*(hit.x - hit.y*sqrt(3.))*f + timeA[6];
timeA[4] =    0.25*(hit.x + hit.y*sqrt(3.))*f + timeA[6];
timeA[5] =   -0.25*(hit.x + hit.y*sqrt(3.))*f + timeA[6];

// Gauss-Verteilung um Mittelpunkt erstellen
for (int ch=0; ch < 7; ch++) {
  for (int i=0; i < SAMPLERATE; i++) {
    double dx = i-timeA[ch];
    input_layer[i + ch*SAMPLERATE]->input =
```

```
        exp(-dx*dx/(width*2.35));
    }
}
```

B.2. Quellcode zur Gewichtserzeugung

```
double dzufall( double von, double bis)
// Generiert eine double Zufallszahl im Intervall [von, bis]
{
  double r = (double) rand() / RAND_MAX;
  return von + r * (bis - von);
}

double getRandomInitialWeight()
// Berechnet ein zufaelliges Gewicht zwischen
//   -START_WEIGHT_RANGE und +START_WEIGHT_RANGE,
//   aber nicht in der Nähe von null
{
  double x;
  while(1)
  {
    x = dzufall(-START_WEIGHT_RANGE, START_WEIGHT_RANGE);
    if ((x != 0) && (pow(x/START_WEIGHT_RANGE, 2.) >=
            dzufall(0.,1.)))
      break;
  }
  return x;
}
```

B.3. Quellcode zur Berechnung der SLP-Ausgabe

```
calculateOutput(int status)
// Berechnet den output_layer aus den Gewichten und dem
//    input_layer
{
  // Gewichte * input_layer = output_layer;

  // Durchgehen aller Gewichte
  for (int i=0; i < OUTPUT_LAYER_CELL_COUNT; i++)
  {
    for (int j=0; j < OUTPUT_LAYER_CELL_COUNT; j++)
    {
```

```
      double z = 0.0;

      // Durchgehen aller input_layer-Zellen
      for (int k=0; k < 7; k++)
      {
        for (int l=0; l < SAMPLERATE; l++)
        {
          z += weights[i][j][k][l] * input_layer[k][l];
        }
      }

//Wenn Wert groesser als Schwelle dann Output auf 1, sonst 0
      if (status == 0)
      {
        if (z >= barrier)
          output_layer[i][j] = 1;
        else
          output_layer[i][j] = 0;
      }
      else
        output_layer[i][j] = pow(z, CONTRAST);
    }
  }
}
```

B.4. Quellcode zum Trainieren des SLP

```
trainieren()
// Verbessert die Gewichte
{
  // Bestimmt die Zelle, die getroffen wurde
  int output_X = (int) (hit.x + MCP_RADIUS) /
      OUTPUT_LAYER_CELL_WIDTH;
  int output_Y = (int) (hit.y + MCP_RADIUS) /
      OUTPUT_LAYER_CELL_WIDTH;

  // Berechnet den Erwartungswert
  calculateOutput(0);

  // Verbessern der Gewichte
  for (int i=0; i < OUTPUT_LAYER_CELL_COUNT; i++)
  {
```

```
      for (int j=0; j < OUTPUT_LAYER_CELL_COUNT; j++)
      {
        if (i == output_X && j == output_Y)
        // diese Zelle wurde getroffen
        {
          if (output_layer[i][j] < 1)
          // Zelle wurde getroffen, aber nicht vorhergesagt
          // -> Gewichte erhoehen
          {
            for (int k=0; k < 7; k++)
            {
              for (int l=0; l < SAMPLERATE; l++)
              {
                // Erhoehen des Gewichts (Maximum 1)
                weights[i][j][k][l] = __min(1,
          weights[i][j][k][l] + gain() * input_layer[k][l]);
              }
            }
          }
        }
        else
        // Zelle nicht getroffen
        {
          if (output_layer[i][j] > 0)
          // Zelle hat gefeuert, wurde nicht getroffen
          // -> verkleinern
          {
            for (int k=0; k < 7; k++)
            {
              for (int l=0; l < SAMPLERATE; l++)
              {
                // Erniedrigen des Gewichts (Minimum 0)
                weights[i][j][k][l] = __max(0,
          weights[i][j][k][l] - gain() * input_layer[k][l]);
              }
            }
          }
        }
      }
    }
}
```