Implementation eines Neuronalen Netzes auf Graphikkarten für Anwendungen in der Hochenergiephysik

Bachelorarbeit
zur Erlangung des Grades *Bachelor of Science*

an der
Hochschule Niederrhein
Fachbereich Elektrotechnik und Informatik
Studiengang *Informatik*

vorgelegt von
Michael Kämmerer
805467

Datum: 26. August 2013

Prüfer: Prof. Dr. Peer Ueberholz
Zweitprüfer: Dr. Sebastian Fleischmann

Eidesstattliche Erklärung

Schriftliche Abschlussarbeit

des Studenten: Michael Kämmerer
Matrikelnummer: 805467

Thema: *Implementation eines Neuronalen Netzes auf Graphikkarten für Anwendungen in der Hochenergiephysik*

Ich versichere durch meine Unterschrift, dass die vorliegende Abschlussarbeit ausschließlich von mir verfasst bzw. angefertigt wurde. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Betreuender Dozent: Prof. Dr. Peer Ueberholz

Krefeld, den *26. August 2013*

_____
(Michael Kämmerer)

Zusammenfassung

Diese Arbeit betrachtet die Implementierungen eines neuronalen Netzes mit dem Backpropagation-Algorithmus in einem Multilayer Perceptron. Dabei wird zwischen Optimierungen auf CPU und GPU eingegangen und größe der Dateneingabe differenziert und hinsichtlich der Performance verglichen. Als Ausgangspunkt dient eine Implementation aus dem TMVA-Frameworks. Es gibt kurze Einleitungen in das verwendete Framework, neuronale Netze, den verwendeten Lernalgorithmus und OpenCL.

Inhaltsverzeichnis

1 Einleitung

In der Schweiz in Genf liegt das CERN (Conseil Européen pour la Recherche Nucléaire), die europäische Organisation für Kernforschung. In 2012 gab es dort einen Durchbruch in der Hochenergiephysik. Es wurde erstmalig ein Teilchen gefunden, welches die Eigenschaften des Higgs-Bosons aufweist [2]. Derzeit wird untersucht, ob die beim Versuch gesammelten Daten zum Standardmodell konsistent sind.

Mit dem Large Hedron Collider (LHC) sollen theorethische Teilchen experimentell nachgewiesen werden. Im LHC werden Protonen in Paketen von 10^{11} Teilchen beschleunigt und zur Kollision gebracht. Dabei treten 40 Millionen Kollisionen pro Sekunde auf, die mit Partikeldetektoren beobachtet werden. Einer dieser Partikeldetektoren ist ATLAS.

Für jede Kollision erzeugt der Detektor einen Datensatz. Würde jeder dieser Datensätze gespeichert, ergäbe dies eine Datenrate von 70 TB pro Sekunde. Zur Bewältigung der hohen Datenrate kommen drei Filterebenen zum Einsatz. Die erste Ebene reduziert die Datenmenge mittels anwendungsspezifischer Hardware um einen Faktor von 1:400. Die verbleibenden 100.000 Datensätze pro Sekunde werden in einer zweiten Filterebene auf 3000 Datensätze pro SekundIn der dritten und damit letzten Ebene sind ca.e reduziert. Diese Ebene ist in Software implementiert und wird von ca. 500 Dualprozessoren berechnet. Die Filterung endet in einer dritten Ebene mit 1700 Prozessoren im Einsatz, welche die 3000 Datensätze auf 200 Datensätze pro Sekunde reduzieren. Die dritte Ebene ist ebenfalls als Software realisiert [3].

Im Laufe eines Jahres fallen so 3200 TB an Rohdaten an. Durch Weiterentwicklung der Verfahren zur Analyse der Daten wird die Rechenkapazität des LHC Computing Grid besser ausgenutzt. Beispielsweise wird durch Parallelisierung die gleichzeitige Verarbeitung größerer Datenmengen ermöglicht. Bei der Analyse wird unter anderem die Flugbahn der bei der Kollision entstandenen Teilchen rekonstruiert. Ein Teil dieses Verfahrens setzt ein neuronales Netz ein, welches in dieser Arbeit darauf untersucht wird, ob die Parallelisierung auf GPUs einen Performancegewinn bringt.

In Kapitel 2 werden theorethische und technische Informationen zum Verständnis der Arbeit vermittelt. Zuerst wird das konkrete Problem, das mit neuronalen Netzen gelöst wird, vorgestellt. Diese sind in TMVA implementiert, einem Framework, dass in 2.1 vorgestellt wird Dazu werden die einzelnen Schritt verständlich gemacht, die dort in einem Programmdurchlauf stattfinden. Im Anschluss daran werden in Kapitel 2.2 neuronale Netze in ihrem Aufbau und ihrer Funktionsweise erklärt. Neuronale Netze werden für den Einsatz mit einem Lernvorgang vorbereitet. Dieser Lernvorgang setzt den Backpropagation-Algorithmus ein und wird in Kapitel 2.3 erläutert. Zum leichteren Vergleich der Ergebnisse werden Histogramme und Performance-Kurven verwendet, die in

Kapitel 2.4 kurz erklärt werden. Zur Weiterentwicklung auf GPUs wird in Kapitel 2.5 OpenCL, ein Framework zur Entwicklung auf Grafikkarten, vorgestellt.

In Kapitel 3 wird zunächst eine eigene Implementation des neuronalen Netzes in C vorgestellt. Zusätzlich wird die C-Implementierung weiter zu beschleunigt, indem man BLAS, eine Bibliothek die auf Matrix- und Vektoroperationen optimiert ist, und OpenMP, eine API zur einfacheren Parallelisierung auf Prozessoren, verwendet. In Kapitel 4 wird daraufhin auf die Implementation auf der Grafikkarte eingegangen. Im Anschluss daran werden unter 5 beide Implementationen verglichen und in Kapitel 6 werden die Ergebnisse betrachtet und ein Fazit gezogen.

2 Grundlagen

Die Kollision einzelner Teilchen wird auch Event genannt. Die Rekonstruktion ihrer Flugbahn ist ein wichtiger Arbeitsschritt.

Innerhalb des ATLAS-Pixeldetektors werden die Positionen von geladenen Teilchen bestimmt. Der Detektor besteht aus drei Lagen, welche jeweils aus einem Zylinder und zwei Endstücken bestehen. Die Zylinder sind hauptsächlich mit $50\mu m \times 400\mu m$ großen Detektoren, auch Pixel genannt, bestückt. An diesen Pixeln werden die abgegebenen Ladungen der einzelnen Teilchen beim Durchfliegen gemessen.

Im Anschluss daran werden Pixel, die eine Ladung verzeichnet haben, zu sogenannten Clustern verbunden, wenn die Pixel eine gemeinsame Kante oder Ecke haben. Mit Interpolation über die Ladungen in den Pixeln über die Zeit kann man so die Position des Teilchens innerhalb eines Clusters bestimmen.

Es kann vorkommen, dass zwei oder mehr Teilchen ihre Ladung in nebeneinander liegenden Pixeln abgeben. Dadurch entstehen sogenannte shared Cluster. Um die Flugbahnen von Teilchen rekonstruieren zu können, muss man entscheiden, wie viele Teilchen jeweils durch einen Cluster geflogen sind. Dazu verwendet man als Eingangsparameter des neuronalen Netzes die Pixelladungen einer 7x7-Matrix und zusätzlicher anderer Daten, die hier nicht weiter erläutert werden, da sie nur einen geringen Teil der Eingangsdaten ausmachen.

Das Problem beim Bestimmen der Teilchenanzahl besteht darin, dass die einzelnen Ladungen der Teilchen übereinander liegen, wenn mehrere Teilchen bei einem Cluster beteiligt sind. Der Optimalfall wäre, dass genau ein Teilchen in einem Cluster liegt, da man so optimal die Flugbahnen rekonstruieren könnte. Die ist mehrheitlich der Fall, jedoch bilden Teilchen in einer erheblichen Zahl shared Cluster, wodurch die Positionsbestimmung aufwendiger wird.

Ein erster Schritt besteht nun darin zu entscheiden, ob ein, zwei oder drei Teilchen an einem Cluster beteiligt sind. Dazu wird ein neuronales Netz verwendet, welches mit computergenerierten Events trainiert wird und im Anschluss möglicherweise in der Lage ist zu entscheiden, ob ein einzelnes Teilchen vorliegt oder ob mehrere Teilchen durch die 7x7-Matrix geflogen sind.[1]

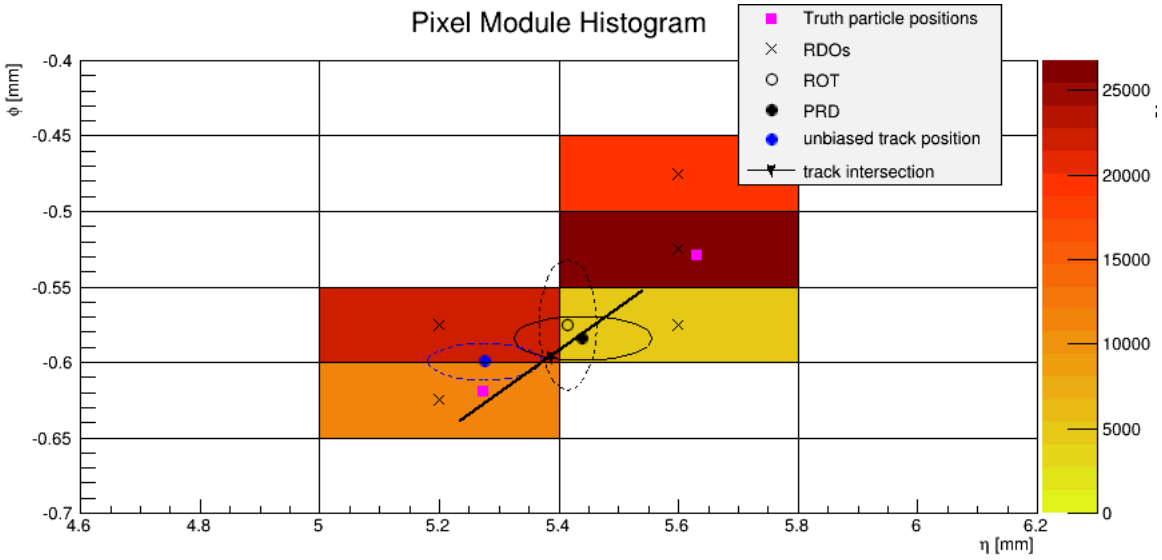

Abbildung 1: Beispielcluster. Die Farbe eines Pixels gibt seine Ladungsstärke an. Die pinken Vierecke markieren die Positionen der Teilchen.Im gezeigten Fall sind zwei Teilchen an dem Cluster beteiligt.

2.1 Toolkit for Multivariate Data Analysis

Das „Toolkit for Multivariate Data Analysis“,kurz TMVA, ist ein Teil von ROOT, ein Framework zur Datenanalyse, welches am CERN eingesetzt wird. Es arbeitet objektorientiert und bietet viele Funktionalitäten, um große Datenmengen effizient zu verarbeiten. Es verwendet dazu ein eigenes Datenformat, so genannte ROOT-Files und bietet viele Möglichkeiten, die darin enthalten Daten auszuwerten und visuell aufzubereiten. [7].

TMVA ist eines der in ROOT verwendeten Frameworks. Es bietet zehn verschiedene Funktionen an, um Daten zu analysieren. Dabei hat jede der Funktionen ihre Vor- und Nachteile, auf diese hier aber nicht weiter eingegangen wird. Es stellt auch das neuronale Netz zur Verfügung, welches in dieser Arbeit verwendet wird. TMVA hat bei der Benutzung einen festen Ablauf für das Einstellen und dem Übergeben von Parametern,

der bei jedem Makro gleich ist, jedoch sind die Details bei jeder ausgewählten Funktion anders. Dies wird durch den Einsatz des Fabrikmusters gewährleistet.

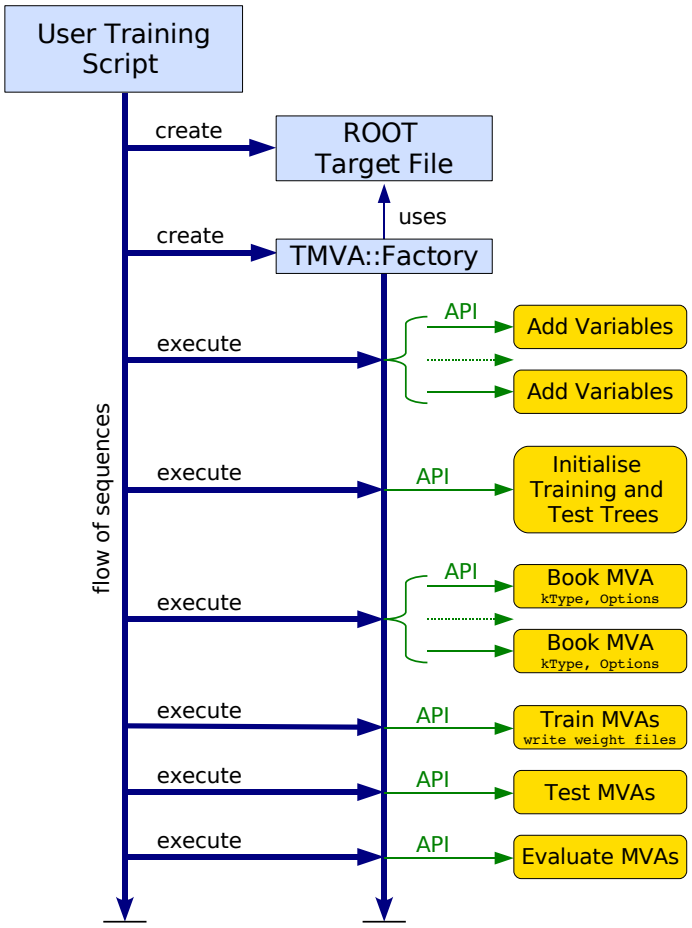

Abbildung 2: Ablauf von TMVA in Kurzform für alle nutzbaren Algorithmen[5]

Um den Ablauf von TMVA zu verstehen, schaue man sich schrittweise den Ablauf an. Hier wird jedoch nur das absolute Minimum der benötigten Funktionen für die Benutzung des neuronalen Netzes vorgestellt. Die ausführlichere Dokumentation kann man im TMVA User's Guide [5] nachschauen.

1. Fabrik erstellen

Im ersten Schritt für eine Analyse in TMVA wird ein Fabrikobjekt erstellt, dass für alle weiteren Schritte verwendet wird. Neben einem Namen für die Fabrik, um diese eindeutig identifizieren zu können, und einer Datei, in der die Ergebnisse gespeichert werden können, kann man auch eine Auswahl von zusätzlichen Optionen mit angeben. Diese werden als String angegeben, wobei die einzelnen Optionen mit „:“ getrennt werden und die Form „Option=Value“ haben. Der Funktionsaufruf hat die Syntax:

Listing 1: Erstellung einer Fabrik

```
TMVA::Factory* factory
= new TMVA::Factory( "<JobName>", outputFile, "<options>" );
```

2. Variablen und Trees bekannt machen

Innerhalb von ROOT werden Daten in einem bestimmten Format gespeichert. Dieses Format wird Tree genannt. Jeder Tree besteht aus einem Header, welcher aus einem Titel und einem Namen besteht. Desweiteren enthält der Tree eine Liste von Branches. Jeder Branch besteht aus einem Namen und einer Liste von Buffern, in denen die eigentlichen Daten gespeichert werden. Damit TMVA Daten aus den Branches einlesen kann, muss der Name des Trees der Fabrik bekannt sein. Dazu werden drei Funktionen benötigt. Die erste Funktion liest ein ROOT-File ein, die zweite erzeugt einen Pointer auf einen der Trees in der eingelesenen Datei. Dies ist nötig, da es in einem ROOT-File mehrere Trees geben kann. Mit der dritten Funktion übergibt man den Tree der Fabrik. Hierbei ist zu beachten, dass die Variable <TreeName>als Parameter für die Fabrik beliebig sein kann, solange dieser innerhalb der Fabrik einzigartig ist. Dieser muss beim zweiten Schritt mit den Namen des Trees in dem ROOT-File übereinstimmen, da TMVA diesen sonst nicht finden kann.

Listing 2: Tree einlesen

```
input = TFile::Open("<inputFile>");
TTree *tree= (TTree*)input->Get("<TreeName>");
factory->AddTree(tree,"<TreeName>);
```

Anschließend müssen der Fabrik die Variablen bekannt gegeben werden, die als Eingabe für das neuronale Netz dienen.

Listing 3: Möglichkeiten um Variablen bekannt zu machen

```
factory->AddVariable("<YourDescreteVar>", 'I');
factory->AddVariable("log(<YourFloatingVar>)", 'F');
factory->AddVariable("SumLabel := <YourVar1>+<YourVar2>", 'F');
factory->AddVariable("<YourVar3>", "Pretty Title", "Unit",'F');
```

Hierbei stehen mehrere Möglichkeiten zur Auswahl. Die erste Möglichkeit besteht darin, lediglich den Namen der Variablen und ihren Datentyp zu übergeben. F steht hierbei für Fließkommazahlen und I für ganze Zahlen. Bei der zweiten Funktion sieht man, dass es auch möglich ist, Rechnungen zu übergeben. Im dritten Aufruf kann man sehen, dass zudem Labels möglich sind. Diese sind als Kurzschreibweise des Terms zu verstehen und werden mit := gesetzt. Als letzte Möglichkeit kann man neben dem Variablennamen einen Variablentitel und eine zugehörige Maßeinheit angeben. Diese werden dann in Plots und der Bildschirmausgabe verwendet.

3. Trainings- und Testdaten vorbereiten

Als nächstes werden die Daten in einen Test- und einen Trainingsdatensatz geteilt. Dies geschieht, um nach dem Training unabhängig von den Trainingsdaten überprüfen zu können, ob das Netz die gewünschten Ergebnisse liefert und nicht etwa ausschliesslich die Trainingsdaten richtig erkennt.

Listing 4: Daten vorbereiten

```
TCut preselectionCut = "<Auswahl>";
factory->PrepareTrainingAndTestTree( preselectionCut,
        "<Optionen>" );
```

Der preselectionCut darf auch ein leerer String sein. Diese Variable dient dazu, auf alle Inputvariablen eine Vorauswahl zu treffen, wie z.B. dass nur Events genommen werden, bei denen nur ein Partikel verzeichnet ist. Als Optionen stehen zur Verfügung:

Option	default	vordefinierter Wert	Beschreibung
SplitMode	Random	Random, Alternate, Block	Methode, wie Traings- und Testdaten ausgesucht werden
SplitSeed	100	-	Seed für zufällige Eventmischung
NormMode	NumEvents	None, NumEvents, EqualNumEvents	Gesamt-Renormalisierung auf Event nach Event-Basis der Gewichte.

Tabelle 1: Optionen zum Vorbereiten der Trainings- und Testdaten, falls man eine Multi-Classification durchführen möchte. Weitere Optionen findet man unter [5] auf Seite 21.

4. Methode zur Analyse auswählen

Die Fabrik hat nun alle Daten um eine Analyse durchführen zu können. Nun werden eine oder mehrere Methoden zur Analyse gewählt, die nacheinander abgearbeitet werden. In dieser Arbeit wird nur auf die Optionen für neuronale Netze genauer eingegangen. Um eine Methode auszuwählen, gibt es folgende Funktion:

Listing 5: Analysemethode auswählen

```
factory->BookMethod( Types::kMLP, "MLP_ANN", "<options>" );
```

Als Optionen sind diese Parameter möglich:

Option	default	vordefinierter Wert	Beschreibung
NCycles	500	-	Anzahl Epochen
HiddenLayers	N,N-1	-	Architektur der versteckten Ebenen
NeuronType	sigmoid	linear, sigmoid, tanh, radial	Aktivierungsfunktion der Neuronen
TrainingMethod	BP	BP, GA, BFGS	Auswahl des Trainingsalgorithmus.
LearningRate	0.02	-	Lernrate des Netzes
DecayRate	0.01	-	Verfallsrate der Lernrate
BPMode	sequential	sequential, batch	Back-Propagation Lernmode
BatchSize	-1	-	Batchgröße im Batchmodus: Events/Batch. -1 für Batchgröße= alle Events

Tabelle 2: Optionen für das Neuronale Netz, hier sind nur die Optionen beschrieben, welche in dieser Arbeit benutzt wurden. Weitere Optionen stehen in [5] auf Seite 94.

5. Training, Test, Auswertung

Mit dem Befehl

Listing 6: Training aller Methoden

```
factory->TrainAllMethods();
```

beginnt die Fabrik das Training für alle ausgewählten Methoden. Die daraus resultierenden Gewichte werden im XML-Format in eine eigene Datei geschrieben. Im Anschluss kann man mit

Listing 7: Testen aller Methoden

```
factory->TestAllMethods();
```

den Testdatensatz über die trainierten Gewichte laufen lassen und erhält dann in einem ROOT-File, welches man der Fabrik beim Erstellen übergeben hat, die Resultate. Mit

Listing 8: Auswerten aller Methoden

```
factory->EvaluateAllMethods();
```

berechnet die Fabrik noch verschiedene Ergebnisse zu den Testresultaten wie z.B. die Korrelationskoeffizienten.

2.2 Neuronale Netze

Als Grundlage für künstliche neuronale Netze wird das Modell von Nervenzellen verwendet. Ein Neuron besteht aus einem Zellkörper, welcher mit Dendriten und einem Axon verbunden ist (siehe Abbildung 3). Dendriten haben die Aufgabe, die Signale anderer Nervenzellen zu empfangen und zusammenzuführen, um diese als Eingabe in die Nervenzelle weiter zu geben. Der Zellkörper interpretiert die Eingabe und schickt die daraus resultierende Ausgabe über sein Axon. Das Axon ist mit Synapsen an die Dendriten anderer Nervenzellen gekoppelt (Nicht abgebildet) [4].

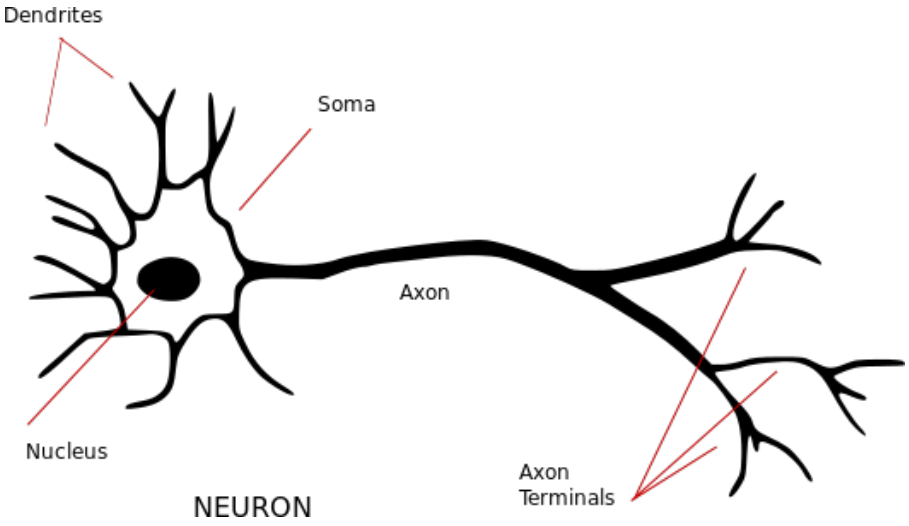

Abbildung 3: Neuron (Quelle: `http://commons.wikimedia.org/wiki/File:Neuron_-_annotated.svg`)

Naturwissenschaftlich werden Modelle entwickelt um Lernvorgänge im Gehirn zu beschreiben. Diese Modelle sind auch in der Informatik interessant. Alle Modelle haben dabei gemeinsam, dass sich über die Zeit hinweg eine hohe Fehlertoleranz und Robustheit gegen Störungen einstellt. Aus diesen Gründen werden künstliche neuronale Netze in der Informatik eingesetzt. Desweiteren ermöglichen künstliche neuronale Netze Dinge assoziativ zu speichern. Dies ist natürlich mit Nachteilen verbunden, so braucht das neuronale Netz einen hohen Zeitaufwand um etwas zu lernen. Zusätzlich gibt es keine Garantie dafür, dass überhaupt richtig gelernt wurde. Als Einsatzgebiet für neuronale Netze eignen sich Entscheidungsprobleme, Roboter, Mustererkennung, Spracherkennung und überall, wo man sich eine künstliche Intelligenz vorstellen kann [8].

Künstliche neuronale Netze bestehen aus Neuronen, die über Synapsen miteinander verbunden sind. Ein Neuron kann einen oder mehr Eingänge besitzen. Die Signale auf den

Eingängen werden in der Regel aufsummiert und ergeben einen Gesamtinput net_i. Dadurch erhält das Neuron einen Aktivitätswert a_i aus dem ein Ausgangswert o_i folgt. Dieser Ausgangswert kann nun entweder gemessen werden oder das Neuron ist über eine oder mehrere Synapsen, die eine gewichtete Verbindung $w_{i,j}$ darstellt, an ein weiteres Neuron gekoppelt (Abbildung 4) [8].

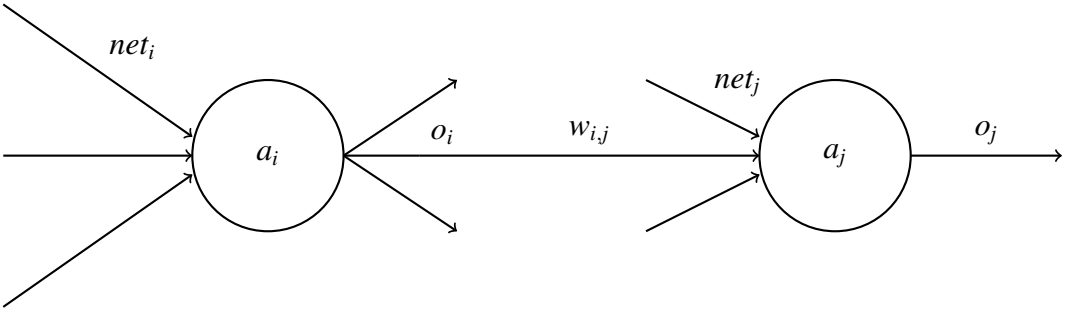

Abbildung 4: Zwei miteinander verbundene Neuronen in einem Neuronalen Netz

Zu diesem einfachen Modell des Aufbaus gibt es ein mathematisches Modell für die Funktionsweise von neuronalen Netzen. Dieses Modell kennzeichnet sich durch folgende Eigenschaften:

1. Jedes Neuron hat einen Aktivierungszustand $a_i(t)$ zum Zeitpunkt t.
2. Jedes Neuron hat eine Aktivierungsfunktion f_{act}. Diese gibt an, wie sich die Aktivierung in Abhängigkeit der alten Aktivierung $a_i(t)$, des Inputs net_i und eines Schwellwerts θ mit der Zeit ändert.
$$a_i(t+1) = f_{act}(a_i(t), net_i(t), \theta) \tag{1}$$
3. Jedes Neuron hat eine Ausgabefunktion f_{out}, die aus der Aktivierung den Output o berechnet.
$$o_i = f_{out}(a_i) \tag{2}$$
4. Es existiert ein Verbindungsnetzwerk, dass aus den Synapsen $w_{i,j}$ besteht.
5. Jedes Neuron hat eine Propagierungsfunktion, die aus den Ausgaben der anderen Neuronen die Netzeingabe berechnet. Diese lautet zumeist
$$net_j(t) = \sum o_i(t) w_{i,j}. \tag{3}$$
6. Eine Lernregel. In dieser Arbeit wird eine Lernregel verwendet, die eine vorgegebene Eingabe mit einer gewünschten Ausgabe vergleicht und daraus die Abweichung berechnet. Mit Hilfe der Abweichung wird dann die Stärke jeder einzelnen

Synapse neu berechnet. Dieser Vorgang wird beliebig oft wiederholt, sodass jedes Trainingmuster mehrmals präsentiert wurde.

Die Zustände des neuronalen Netzes werden solange geändert, bis ein stabiler Endzustand eintritt. Dieser muss jedoch nicht das gewünschte Ergebnis liefern, in diesem Fall hätte das neuronale Netz falsch gelernt.[8] Im Folgenden wird nur der Fall $a_i = f_{act}(net_i)$ und $o_i = a_i$ betrachtet. Zusammengefasst erhält man so:

$$o_i = f_{act}(net_i) \tag{4}$$

2.3 Backpropagation-Algorithmus

Damit das neuronale Netz seine Aufgabe erfüllen kann, muss es zunächst lernen. Dabei gibt es keine Einschränkung, wie gelernt werden kann. Es können neue Verbindungen oder Neuronen erstellt oder gelöscht werden, die Parameter der Neuronen angepasst werden oder die Stärke der Verbindungen geändert werden. Meistens wird die Modifikation der Verbindungsstärke gewählt, da dieses Lernverfahren am einfachsten ist. Auch das neuronale Netz, welches Grundlage für diese Arbeit ist, nutzt die Modifikation der Verbindungen zum Lernen. Desweiteren wird zwischen drei Lernverfahren unterschieden:

1. **Überwachtes Lernen:** Es existiert zu jedem Input ein gewünschter Output. Dieser wird mit dem tatsächlichen Output verglichen und das Netz angepasst.
2. **Bestärkendes Lernen:** Bei einem Input wird der Output auf richtig oder falsch überprüft. Mithilfe dieser Information wird das Netz angepasst.
3. **Unüberwachtes Lernen:** Das Netz organisiert sich selbstständig.

Am häufigsten wird das überwachte Lernen verwendet. Es existieren verschiedene Methoden um dieses zu realisieren. Da für diese Arbeit jedoch nur das Back-Propagation-Verfahren eine Rolle spielt, wird nur dieses Verfahren genauer erläutert [8].

Das Back-Propagation-Verfahren (BP) minimiert mithilfe eines Gradienten eine Fehlerfunktion. Diese wird meistens als Summe über die quadratische Abweichung des Outputs $o_{p,j}$ und des gewünschten Outputs $\hat{o}_{p,j}$ für ein Pattern(Event) p definiert.

$$E = \sum_{p}^{n} E_p \tag{5}$$

$$E_p = \frac{1}{2} \sum_{j}^{m} (o_{p,j} - \hat{o}_{p,j})^2 \tag{6}$$

Hierbei ist n die Anzahl der Pattern und m die Anzahl der Outputs. Mit Gleichung 7 versucht man die Gewichte entlang des Gradienten der Fehlerfunktion zu ändern, bis diese minimiert ist. Im optimalen Fall findet man damit das globale Minimum der Fehlerfunktion, meistens findet man jedoch nur ein lokales Minimum.

$$\Delta w_{ij} = -\eta \sum_{p}^{m} \frac{\partial E_p}{\partial w_{ij}} \tag{7}$$

η ist die Lernrate. Diese ist wichtig, um steuern zu können wie stark die Gewichte angepasst werden. Mit Hilfe der Kettenregel kann man die Formel umschreiben und auf zwei Faktoren reduzieren:

$$\Delta w_{ij} = \eta \sum_{p}^{m} -\frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{i_j}} \tag{8}$$

Der erste Faktor wird als Fehlersignal δ_{pj} bezeichnet.

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} \tag{9}$$

Bei der Berechnung von δ_{pj} muss die konkrete Aktivierungsfunktion der Zelle j beachtet werden.

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net_{pj}} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial f_{act}(net_{pj})}{\partial net_{pj}} \tag{10}$$

Jetzt unterscheidet man in welcher Ebene sich der Knoten j befindet:

1. j ist eine Ausgabezelle, dann gilt

$$-\frac{\partial E_p}{\partial o_{pj}} = (t_{pj} - o_{pj}). \tag{11}$$

   Für den Gesamtfehler gilt in diesem Fall

$$\delta_{pj} = f'_{act}(net_{pj})(t_{pj} - o_{pj}). \tag{12}$$

2. j ist eine verdeckte Zelle. Daraus ergibt sich, dass die Änderung der Fehlerfunktion von den Zwischenzellen k der darüber liegendenden Ebene abhängt [8].

$$-\frac{\partial E_p}{\partial o_{pj}} = -\sum_{k}^{m} \frac{\partial E_p}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial o_{pj}} = \sum_{k}^{m} \left(\delta_{pk} \frac{\partial}{\partial o_{pj}} \sum_{i}^{n} o_{pi} w_{ik} \right) = \sum_{k}^{m} \delta_{pk} w_{jk} \tag{13}$$

Der zweite Faktor ist o_{pi}.

$$\frac{\partial net_{pj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{i}^{m} o_{pi} w_{ij} = o_{pi} \tag{14}$$

Nun kann man mithilfe der beiden Faktoren die Änderung der Gewichte berechnen.

$$\Delta w_{ij} = \eta \sum_{p}^{m} o_{pi} \delta_{pj} \tag{15}$$

Um den Algorithmus besser zu verstehen schaue man sich das folgende Beispiel an.

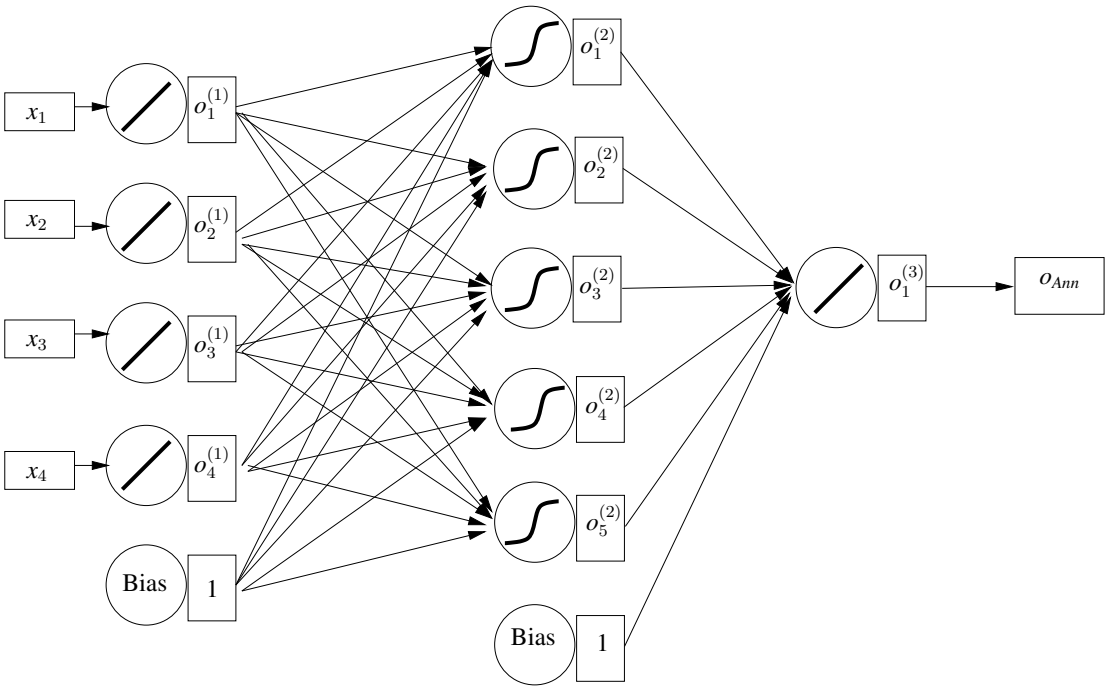

Abbildung 5: Neuronales Netz, wie es auch in TMVA verwendet wird

Das in Abbildung 5 gegebene Netz ist ein sogenanntes Multilayer Perceptron. Dies bedeutet, dass die Neuronen in mehreren Ebenen angeordnet sind und die Verbindungen zwischen den Neuronen jeweils an die nächsthöhere Ebene gekoppelt sind. Dabei werden die Ebenen in drei Ebenen unterteilt. Die erste Ebene ist hierbei die Eingabeebene und erhält die Eingabedaten. Die letzte Ebene ist die Ausgabeebene und gibt das Ergebnis aus. Die übrigen Ebenen dazwischen nennt man *versteckt*, da diese nicht direkt an Ein- und Ausgabe geknüpft sind.

Das Netz besteht aus einer Eingabeebene mit fünf Neuronen,wovon vier eine linearen Aktivierungsfunktion besitzen und ein Knoten als Bias dient. Hinzu kommt eine versteckte Ebene, welche als Aktivierungsfunktion den $\tanh$ benutzt und aus 5 Neuronen besteht mit einem sechsten Neuron als Bias. Dies ist ein Neuron, bei dem der Output immer 1 ist und keinen Input bekommt. Zur Vereinfachung werden die Bias-Knoten im Rest des Beispiels nicht weiter beachtet. Die Ausgabeebene besteht aus einem einzelnen Neuron, welches wieder eine lineare Aktivierungsfunktion besitzt. Daraus ergibt sich folgende Formel zum Berechnen des Ausgabeknotens:

$$o_{ANN} = \sum_{j=1}^{n_h} o_j^{(2)} w_{j1}^{(2)} = \sum_{j=1}^{n_h} \tanh\left(\sum_{i=1}^{n_{var}} x_i w_{ij}^{(1)}\right) \cdot w_{j1}^{(2)} \tag{16}$$

n_h ist die Anzahl der Knoten auf der versteckten Ebene und n_{var} ist die Anzahl der Knoten auf der Eingabeebene. $w_{ij}^{(1)}$ steht für die Gewichte zwischen den Neuronen i der Eingabeebene und den Neuronen j auf der versteckten Ebene. $w_{j1}^{(2)}$ ist das Gewicht zwischen den Neuronen j der versteckten Ebene und dem Ausgabeknoten. Als Propagiergierungsfunktion wurde das Produkt aus Output und Gewicht aufsummiert.

Während des Lernprozesses werden dem Netz N Trainingevents

$$x_p = (x_1, ..., x_{n_{var}})_p$$

$$p = 1, ..., N$$

gezeigt. Für jedes Trainingsevent p wird der Output o_{ANN} berechnet und mit dem gewünschen Output $\hat{o}_p = \{0, 1\}$ verglichen. Da in diesem Beispiel das Netz für eine Klassifikation mit zwei Klassen benutzt wird, beträgt der gewünschte Output 1 für Signal und 0 für kein Signal.

Die Fehlerfunktion E, die angibt, wie stark der Output mit dem gewünschen Output übereinstimmt ist definiert als (vgl. Gleichung 6)

$$E(x_1, ..., x_N|w) = \sum_{p=1}^{N} E_p(x_p|w) = \sum_{p=1}^{N} \frac{1}{2}(o_{ANN,p} - \hat{o}_p)^2 \tag{17}$$

Die Werte der Gewichte w, die die Fehlerfunktion minimieren wird mithilfe eines BP-Verfahrens berechnet. Man fängt mit einer zufälligen Gewichtsmenge $w^{(\rho)}$ an und berechnet diese neu, indem man sich eine kurze Distanz im w-Raum in Richtung $-\nabla_w E$ bewegt. Wenn man nun noch die Lernrate η hinzunimmt erhält man folgende Gleichung (siehe auch Gleichung 7):

$$w^{(\rho+1)} = w^{(\rho)} - \eta \nabla_w E \tag{18}$$

Die Gewichte, die mit der Ausgabeebene verbunden sind, werden neu berechnet mit

$$\Delta w_{j1}^{(2)} = -\eta \frac{\partial E_p}{\partial w_{j1}^{(2)}} \tag{19}$$

Mit Gleichung 16 eingesetzt ergibt sich daraus:

$$= -\eta (o_{ANN,p} - \hat{o}_p) \tanh \left(\sum_{j=1}^{n_{var}} x_i w_{j1}^{(1)} \right) \tag{20}$$

$$= -\eta (o_{ANN,p} - \hat{o}_p) o_{j,p}^{(2)} \tag{21}$$

und die Gewichte für die versteckte Ebene mit

$$\Delta w_{ji}^{(1)} = -\eta \frac{\partial E_p}{\partial w_{ji}^{(1)}} \tag{22}$$

$$= -\eta (o_{ANN,p} - \hat{o}_p) \left(1 - \tanh \left(\sum_{j=1}^{n_{var}} x_i w_{j1}^{(1)} \right)^2 \right) w_{j1}^{(2)} x_{i,p} \tag{23}$$

$$= -\eta (o_{ANN,p} - \hat{o}_p)(1 - o_{j,p}^{(2)^2}) w_{j1}^{(2)} x_{i,p}. \tag{24}$$

Werden die Gewichte bei jedem Event neu berechnet, spricht man von Online-Learning. Im Gegensatz dazu gibt es das Bulk-Learning bei dem man den Fehler einer Menge von Events aufsummiert, bevor man die Gewichte neu berechnet [5].

2.4 Qualität des Netzes

Um die trainierten neuronalen Netze miteinander vergleichen zu können, kann man sich Performance-Kurven erstellen. Diese sind bei einer einfachen Klassifikation eine Angabe, wie gut das Netz ein Signal vom Background trennen kann. Man kann hiermit angeben, wie gut das Netz einzelne Partikel von Partikelclustern bei größeren Netzen mit mehreren Ausgabeknoten trennt.

Eine einfachere Methode zum Vergleichen der Qualität des Netzes stellt die Nutzung von Histogrammen dar. Dazu wurde das TMVA von mir angepasst, sodass nach der Trainingsphase die Werte der Ausgabeneuronen und die Event-Klasse für jedes Event aus dem Testdatensatz in eine eigene Datei geschrieben wurden. Im Anschluss wird ein Ruby-Skript verwendet, dass aus den Daten ein Histogramm erzeugt und abspeichert.

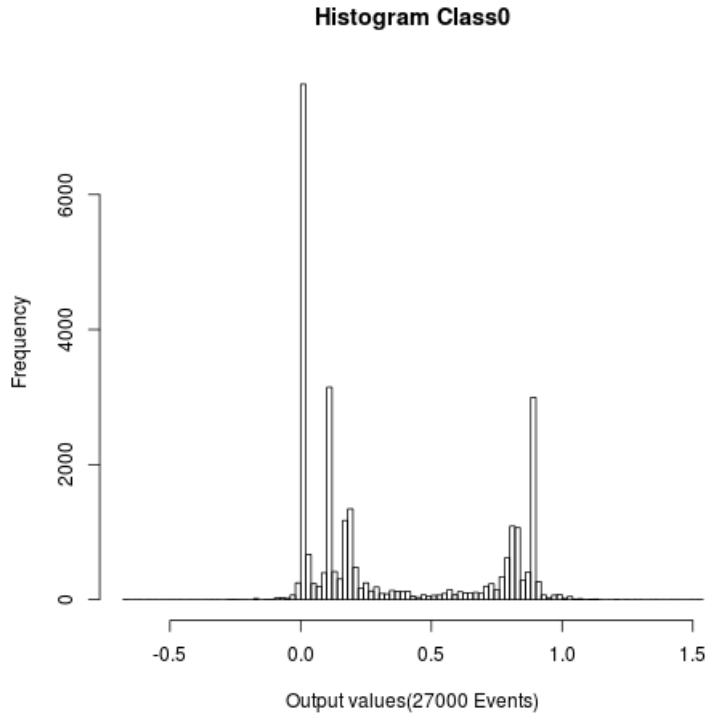

Abbildung 6: Beispielhistogramm

Durch die einfache Vergleichbarkeit der Histogramme kann man leicht sehen, ob das Netz richtig gelernt hat. In diesem Fall sollte das Histogramm der Klasse 0 an Ausgabeneuron 0 die meisten Werte in der Nähe von 1 haben. Das Beispiel in Abbildung 6 zeigt ein Histogramm über alle Ausgabeneuronen, weshalb es eine Häufung bei 0 und eine Häufung bei 1 gibt. Aus dieser Art von Histogramm lassen sich Kurven erstellen, mit deren Hilfe man Performance-Kurven berechnen kann. Dies wird im folgendenen erklärt:

Gegeben sind die Kurven für die erkannten Daten, welche man mit den Ausgabedaten eines TMVA-Skripts nach dem Evaluationsschritt berechnen kann, indem man verschiedene Funktionen von ROOT benutzt. Da die genaue Funktion des Skriptes an dieser Stelle nicht relevant ist, um die Performance-Kurven zu verstehen, wird darauf nicht näher eingegangen.

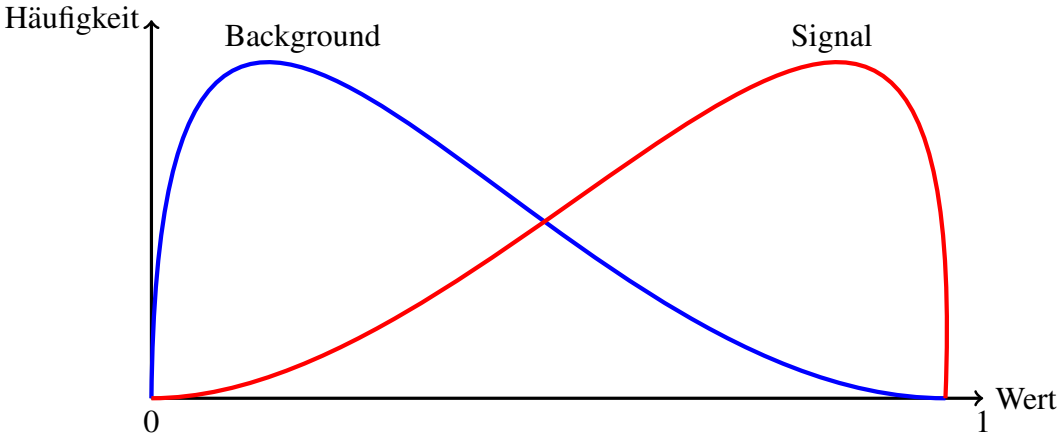

Abbildung 7: Beispielkurven für Signal und Background

Wenn man die Fläche der Kurven auf 1 normiert und sich einen beliebiebigen Punkt auf der X-Achse aussucht, kann man durch die Integrale der dadurch gesplitteten Kurven sehen, wie groß jeweils der Anteil des richtig erkannten Signals und des Backgrounds sind.

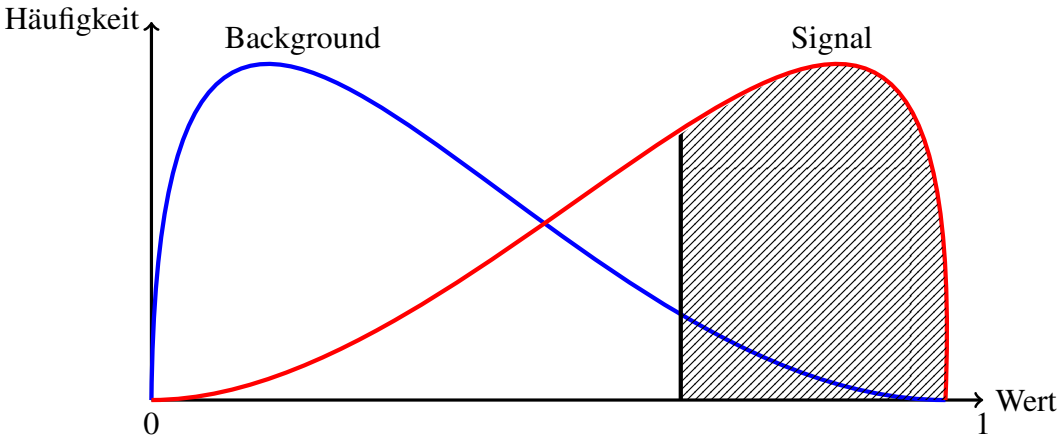

Abbildung 8: Integral (schraffiert) über Teile der beiden Kurven

Anhand der schraffierten Fläche kann man sehen, dass man ca. 20% Background und 80% Signal richtig erkennt. Diese „Messung“ kann man nun in eine Kurve umwandeln, die den richtig erkannten Anteil auf den Achsen angibt. Wenn man dies nun für alle Punkte durchführt, erhält man einen Graphen, der angibt wie viel man jeweils richtig

erkennt. Ein optimaler Verlauf nähert sich einem Rechteck, sodass sowohl Signal als auch Background zu 100% erkannt werden.

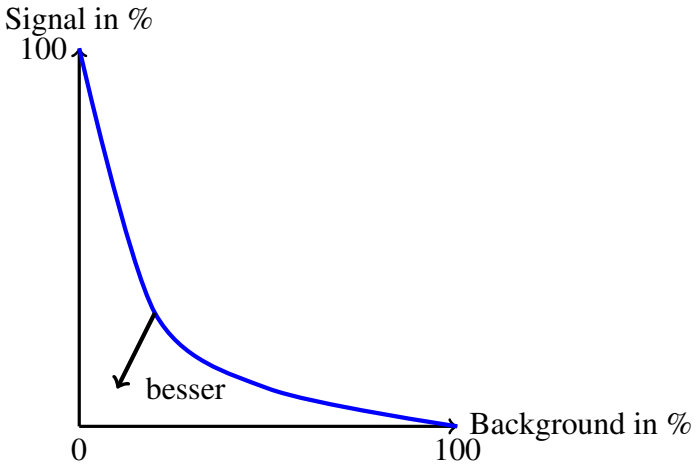

Abbildung 9: Performance-Kurve, aus der man erkennen kann, wie gut gelernt wurde

Je weiter die Blaue Linie in der linken unteren Ecke ist, desto besser wurden die Testergebnisse erkannt [6].

2.5 OpenCL

OpenCL ist ein Framework zur Programmierung auf CPUs, GPUs und anderen Prozessoren. Es stellt dabei einen offenen Standard dar, der von der Khronos-Group entwickelt, verwaltet und veröffentlich wird. Zum ersten mal wurde OpenCL im Jahre 2008 veröffentlicht und die ersten Produkte waren im Herbst 2009 verfügbar.

In OpenCL geschriebene Programme können auf einer großen Menge von Geräten laufen, z.B. kann das selbe Programm sowohl auf Smartphones als auch auf großen Supercomputern laufen. Diese Portabilität ist einer der Gründe für den Erfolg von OpenCL. Dies wird dadurch bewerkstelligt, dass die Hardware vom Programmierer explizit ausgesucht werden muss und diese nicht hinter Abstraktionen versteckt wird. Dies führt dazu, dass der Programmierer die volle Kontrolle darüber hat, wie der Programmfluss auf jedem einzelnen Gerät aussieht.

OpenCL ist dafür ausgelegt, Programme asynchron auszuführen und ermöglicht damit, auf den verschiedenen Geräten die Programme zu parallelisieren. Als Geräte zur Parallelisierung eignen sich Grafikkarten sehr gut, da diese viele Prozessorkerne haben. Ein gut parallelisierbares Programm, hat möglichst viele Teilabschnitte, die unabhängig

voneinander ausgeführt werden können ohne dass diese miteinander kommunizieren müssen. Ein einfaches Beispiel wäre eine Matrix-Matrix-Multiplikation. Hierbei kann die Berechnung der einzelnen Ergebnisse in der Endmatrix gleichzeitig stattfinden, da die einzelnen Ergebnisse nicht voneinander abhängig sind.

In OpenCL muss als erster Schritt um ein Programm auf einem Gerät auszuführen eine Plattform ausgewählt werden, da OpenCL mit einem Plattform-Modell arbeitet. Das Plattform-Modell besteht immer aus genau einem Host, welcher für die korrekte Abarbeitung der einzelnen OpenCL-Schritte, die Datenübertragung von und zu den benutzten Geräten und die Interaktion mit dem Benutzer zuständig ist. Der Host ist mit mindestens einem Gerät, in OpenCL als Device bezeichnet, verbunden. Jedes Device ist in Recheneinheiten (Compute Units) unterteilt, welche wiederrum in Prozesselemente (Process Elements oder PE) unterteilt sind. In den Prozesselementen werden die eigentlichen Operationen der Kernel ausgeführt.

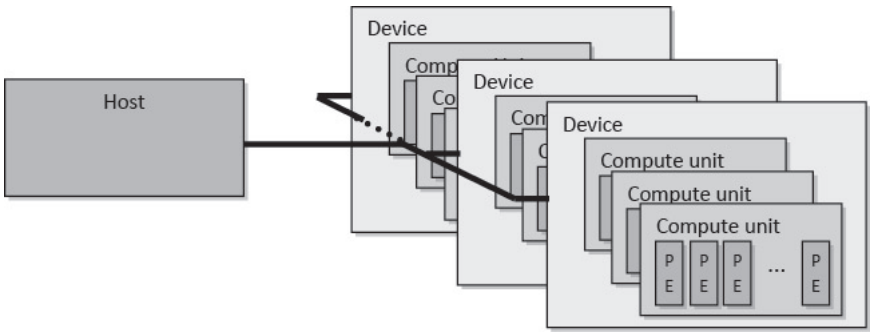

Abbildung 10: Plattformmodell von OpenCL. [15]

Der Programmierer kann verschiedene Informationen über die zur Verfügung stehenden Plattformen mit Hilfe von OpenCL abfragen und eine passende heraussuchen um diese zu nutzen.

Nachdem eine Plattform ausgesucht wurde, muss innerhalb der Plattform ein Device bestimmt werden, auf dem der Programmierer rechnen möchte. OpenCL bietet auch hier die Möglichkeit, verschiedene Informationen zu den einzelnen Devices abzufragen, dazu gehört z.B. wie viel Speicher auf dem Device zur Verfügung steht oder wie viele Compute Units vorhanden sind. Anhand der abgefragten Informationen kann man so das passende Device aussuchen.

Jetzt, wo eine Plattform und ein oder mehrere Devices ausgesucht wurden, kann man einen Context erstellen. Der Context definiert die Umgebung in denen ein oder mehrere Kernel definiert und ausgeführt werden. Da an dieser Stelle noch keine Kernel vorhanden sind, benötigt OpenCL nur die Plattform und die Devices. Später wird der Context erweitert und beinhaltet:

1. **Devices**: Alle OpenCL Devices die vom Host benutzt werden.
2. **Kernel**: Programmteile die auf den Devices ausgeführt werden.
3. **Programmobjekte**: Der Programmquellcode und die ausführbaren Programme, welche die Kernel implementieren.
4. **Speicherobjekte**: Objekte im Speicher auf die einzelne Instanzen der Kernel zugreifen können.

Als nächstes muss mit OpenCL eine Command-Queue erstellt werden. Dies ist eine Verbindung zwischen dem Host und einem Device, welches die Kommandos des Hosts an ein Device überträgt und die Abarbeitung der Kommandos verwaltet. Im einfachsten Fall werden die Kommandos in der Reihenfolge abgearbeitet, in der der Host diese überträgt. Mit Hilfe der Command-Queue kann der Host auch später abfragen, ob ein Befehl abgearbeitet wurde oder ein Fehler bei der Abarbeitung aufgetaucht ist.

Nach dem erstellen der Command-Queue wird der Speicher auf dem OpenCL-Device erzeugt und falls benötigt initialisiert. Der Speicher, der auf diese Art auf dem OpenCL-Device bereitgestellt wird, wird der globale Speicher genannt. OpenCL hat ein eigenes Speichermodell, welches zwischen fünf verschiedenen Speicherarten unterscheidet:

1. **Hostspeicher**: Auf diesen Speicherbereich kann nur der Host zugreifen.
2. **Globaler Speicher**: Auf diesen Speicher können alle Workitems aus jeder Workgroup lesend und schreibend zugreifen.
3. **Konstanter Speicher**: Auf diesen Speicher haben alle Workitems lesenden Zugriff. Der Speicher verändert sich während der Ausführung von Programmen nicht. Der Konstante Speicher ist ein Teil des globalen Speichers und wird vom Host allokiert und initialisiert.
4. **Lokaler Speicher**: Dieser Speicher wird von allen Workitems innerhalb einer Workgroup geteilt und kann von jedem Workitem schreiben und lesend genutzt werden.
5. **Privater Speicher**: Dieser Speicher ist für jeweils ein Workitem verfügbar.

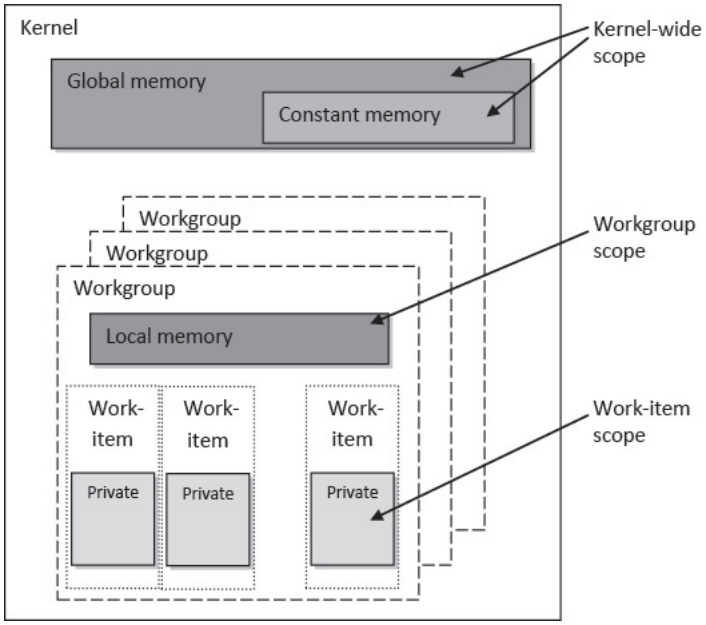

Abbildung 11: OpenCL-Speichermodell [15]

Die Begriffe Workgroup und Workitems werden später erläutert, wenn erklärt wird, wie Kernel ausgeführt werden. Momentan reicht es zu wissen, dass Workitems in Workgroups organisiert sind. Im Anschluss an die Speichererzeugung werden die Daten zwischen Host und Device übertragen.

Im nächsten Schritt werden die Kernel erzeugt. Dazu kann man diese entweder während das Programm läuft kompilieren oder vorkompiliert einlesen. Aus dem Jompiliervorgang entstehen Programmobjekte, aus aus denen man die einzelnen Kernel erzeugen kann.

Ein Kernel wird in OpenCL-C geschrieben. OpenCL-C stellt eine eigene Programmiersprache dar und basiert auf dem C99-Standard. Da OpenCL-C sich noch in der Entwicklung befindet existieren nicht alle Funktionalitäten die man aus der C-Programmierung kennt. Dazu zählen:

- Funktionszeiger
- Dynamische Arrays
- Rekursive Funktionsaufrufe
- Standardbibliotheken

Dadurch, dass es keine Standardbibliotheken in OpenCL-C gibt, ist auch keine dynamische Speicherallokierung möglich. Dies lässt sich jedoch, unter der Bedingung, dass

der Host vorher weiss wie viel Speicher benötigt wird, umgehen. Dazu kann der Host vor dem Kompilieren des Programmobjekts den Quellcode der Kernel um diese Informationen erweitern.

Nach der Erzeugung des Kernels muss der Host im Programm die Übergabeparameter des Kernels mit Hilfe von OpenCL setzen. Bevor der Kernel nun gestartet wird, ist es wichtig zu wissen, wie OpenCL einen Kernel ausführt.

Wenn ein Kernel gestartet wird, wird auf den ausgewählten Devices ein Threadgitter aufgespannt. Wieviele Dimensionen das Gitter hat, wird OpenCL beim Starten des Kernels mitgeteilt. Wie groß jeweils eine Dimension ist, bestimmt sich daraus, wie viele Threads das Device maximal gleichzeitig verarbeiten kann und welche Parameter der Programmierer beim starten des Kernels als lokale und globale Arbeitsgröße angegeben hat.

Die lokale Größe muss die globale Größe glatt teilen, da sich daraus die Anzahl der sogenannten Workgroups berechnet. Daraus erhält man ein Raum aus Workgroups, die sogenannte NDRange, in der jede Workgroup über eine Workgroup-ID eindeutig identifizert werden können. Jede Workgroup spannt einen eigenen Raum auf, der so groß ist wie die angegebenen lokalen Arbeitsgrößen. Threads innerhalb einer Workgroup werden als Workitems bezeichnet. Jedes Workitem kann dabei über eine global ID über alle Workgroups hinweg eindeutig identifiziert werden. Desweiteren hat jedes Workitem innerhalb seiner Workgroup eine eindeutige local ID.

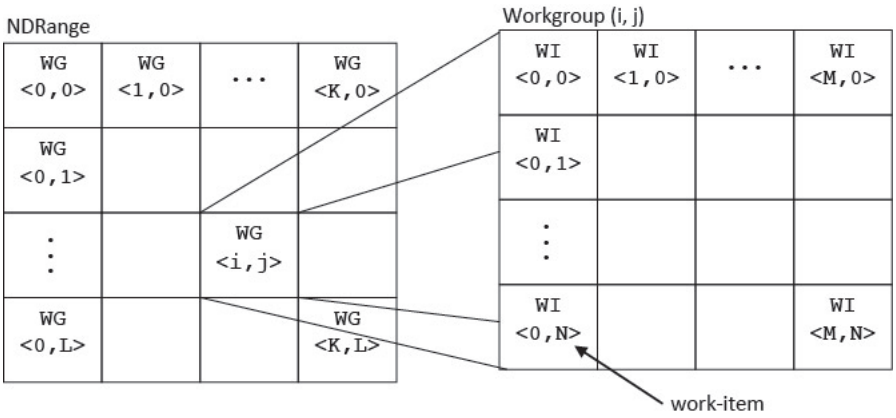

Abbildung 12: Organisation von Workgroups und Workitems [15]

Nachdem der Kernel gestartet wurde muss man warten, bis der Kernel abgearbeitet wurde. Dabei kann man entweder mit Hilfe von Events auf dem Host das Programm weiterlaufen lassen und zwischendurch überprüfen, ob der Kernel fertig ist oder man wartet darauf, dass die Command-Queue leer ist. Nun müssen nur noch mit OpenCL die

berechneten Daten vom Device wieder auf den Host zurückgelesen werden und die nicht mehr benötigten Objekte auf dem Host und dem Device wieder freigegeben werden. Die konkreten Parameter und die Namen der im Einzelnen benötigten OpenCL-Funktionen werden in [14] genauer erläutert.

3 Implementierung in C

In diesem Kapitel wird die Performanceverbesserung durch verschiedene Implementierungen in C bzw. C++ vorgestellt. (Siehe dazu auch [10]). Als Ausgangspunkt dient dazu die momentane Implementierung des neuronalen Netzes in TMVA. In TMVA wird durchgehend ein objektorientierter Programmieransatz verfolgt. Dies führt dazu, dass viele Funktionen nur wenig eigene Funktionalität aufweisen und hauptsächlich Daten über viele Funktionen durchgereicht werden, bis man letztendlich ein Ergebnis hat. Dies führt dazu, dass das Programm sehr langsam wird.

Um den Geschwindigkeitsgewinn messen zu können, wurden drei Datensätze verwendet. Der erste Datensatz ist klein und umfasst 8000 Events. Dieser wird bei der Installation von ROOT mit TMVA mitinstalliert. Der große Datensatz wurde von der Universität Wuppertal zur Verfügung gestellt und umfasst 191208 Events. Als dritter Datensatz wurde ein kleinerer Datensatz aus dem großen Datensatz generiert, da dieser beim Testen nicht soviel Zeit benötigt bei einem Programmdurchlauf. Zum Testen der Geschwindigkeit wurde ein Intel Core i7-2600S mit einer Taktrate von 2,8 GHz verwendet. Mehr Details über die einzelnen Datensätze gibt es in Tablle 3.

	Klein	Mittel	Groß
Eventanzahl	8000	12014	191208
Eingabeneuronen	4	60	62
Versteckte Ebenen	1	2	2
Anzahl versteckter Neuronen mit bias	10	26, 21	26, 21
Ausgabeneuronen	4	3	3
Anzahl Epochen	600	1000	1000

Tabelle 3: Eckdaten der Testdatensätze

Die fertige Implementierung des neuronalen Netzes ohne Funktionsaufrufe sieht folgendermaßen in Pseudocode aus:

Algorithmus 1: Algorithmus zum trainieren des neuronalen Netzes (forward propagation)

Eingabe : Events, η, Anzahl der Eingabevariablen pro Event, Anzahl der Epochen, Anzahl der Events, Synapsengewichte (zufällig vorinitialisiert), Neuronen pro Ebene, Anzahl der Ebenen, Bias (Hat die Ebene einen Bias-Knoten?), Zerfallsrate

Ausgabe : Synapsengewichte nach dem Lernen.

Beginn

Berechne Spätepochen als Anzahl der letzten 5% der Epochenzahl als Ganzzahl (siehe Algorithmus 2)

foreach *Epoche* **do**

foreach *Event* **do**

foreach *Neuron n auf Ausgabeebene* **do**

if *Neuronnummer* == *Eventklasse* **then**

Setze $\hat{o}_n = 1$

end

else

Setze $\hat{o}_n = 0$

end

end

Initialisiere die Eingabeebene mit den Eventwerten

foreach *Neuron auf Ebene l außer der Eingabeeben* **do**

for *Anzahl j der Neuronen auf der Ebene l* **do**

Setze $Neuron_{l,j} = 0$

for *Anzahl i der Neuronen auf Ebene* $l-1$ **do**

Berechne $Neuron_{l,j} = \sum_{k=1}^{i} Neuron_{l,k} \cdot w_{l,k,j}$

end

if *l nicht die Ausgabeebene ist* **then**

Berechne $Neuron_{l,j} = f_{act}(Neuron_{l,j})$

end

end

end

end

end

Ende

Algorithmus 2: Fortsetzug des Algorithmus zum Trainieren des neuronalen Netzes (backwards propagation)

begin
  foreach *Epoche* **do**
    foreach *Event* **do**
      Setze l als Nummer der letzten Ebene
      for *Anzahl j der Neuronen auf der letzten Ebene* **do**
        Berechne $\delta_{l,j} = Neurons_{l,j} - o_{t,j} \cdot Eventgewicht$
      end
      for *Jede Ebene l, von der letzten Ebene ausgehend, wobei die darunter liegende Ebene versteckt sein muss* **do**
        for *Anzahl j der Neuronen auf Ebene l* **do**
          Setze $sumdeltas = 0$; **for** *Anzahl k der Neuronen auf Ebene* $l+1$ **do**
            Berechne $sumeltas = \sum_{i=1}^{k} \delta_{l,i} \cdot w_{l,j,i}$
          end
          Berechne $\delta_{l,j} = f'_{act,l}(net_{l,j}) \cdot sumdeltas$
        end
      end
      for *Alle Ebenen l, außer der Ausgabeebene* **do**
        for *Anzahl j der Neuronen auf Ebene l+1 ohne den Bias-Knoten* **do**
          for *Anzahl i der Neuronen auf Ebene l* **do**
            Berechne: $w_{l,k,j} = \sum_{k=1}^{i} -\eta \cdot Neurons_{l,k} \cdot \delta_{l+1,j}$
          end
        end
      end
      if *Epoche ist Spätepoche* **then**
        Berechne: $\eta = \eta \cdot (1 - \sqrt{Zerfallsrate})$
      end
      else
        Berechne: $\eta = \eta \cdot (1 - Zerfallsrate)$
      end
    end
  end
end

Wenn man sich die Algorithmen anschaut, wird einem bereits in der ersten Zeile auffallen, dass man eine Unterscheidung zwischen normalen Epochen und Spätepochen macht. Am Ende jeder Epoche wird die Lernrate verkleinert um sich mögichst gut an ein lokales Minimum heranzutasten. Als kleine Optimierung werden hier die Spätepochen verwendet, um die Lernrate am Ende noch stärker zu verkleinern.

Desweiteren fält auf, dass jedes Event nicht nur aus den Eingabedaten für das neuronale Netz besteht, sondern auch eine Gewichtung hat und eine Klasse (siehe Algorithmus 2). Die Gewichtung der Events ist in den vorliegenden Testdatensätzen nicht wichtig, da diese dort alle den gleichen Wert haben, jedoch werden diese wichtig, wenn man Events mit bestimmten Daten zwar im Training drin haben möchte, jedoch verhindern möchte, dass diese einen zu großen Einfluss auf das Ergebnis haben.

Die Klassifizierung der Events geschieht innerhalb von TMVA und wird dazu genutzt, um jedem eingelesenen Tree eine Nummer zuzuweisen und bei der Multiklassifizierung ein Neuron auf der Ausgabeebene mit der gleichen Nummer zu erstellen, um darüber zu bestimmen, welches Ausgabeneuron ein Signal haben soll, wenn das entsprechende Event als Eingabe in das Netz gelesen wird.

Für die einfache Implementation des Batchmodus in C wurde die Anzahl der Ebenen auf vier beschränkt, da dies in den meisten Fällen ausreicht. Dadurch verliert man die Schleife über die Ebenen und den dazugehörigen Index bei den Variablen. Im Batchmodus wird das Bulk-Learning verwendet. In dieser Lernmethode werden die Fehler über einen Teil der Events aufsummiert und erst dann werden die Gewichte der Synapsen angepasst. Daraus ergibt sich, dass aus den Vektoren im Online-learning nun Matrizzen werden, die eine 2. Dimension mit einer größe entsprechend der Anzahl der Teilevents, der Batchgröße, haben.

Mein Betreuer hat weitere Versionen des Batchmodus mit OpenMP bzw. der BLAS-Bibliothekt geschrieben. Da beide Versionen einen Geschwindigkeitsgewinn bringen, wurden diese auch mit in die Testreihe aufgenommen. BLAS steht für Basic Linear Algebra Subprograms und ist eine Bibliothek, die Funktionen für einfache Vektor- und Matrixoperationen bereitstellt [12]. Da diese Bibliothek seit Jahren auf bessere Performance optimiert wird, verspricht diese einen Geschwindigkeitsgewinn gegenüber einer eigenen Implementation der Matrixmultiplikation.

OpenMP ist eine API, die mithilfe von Compilerdirektiven und Umgebungsvariablen eine Schnittstelle zum einfachen Erstellen und Verwalten von Threads auf CPUs bereit stellt. Da man dadurch alle Kerne der CPU ausnutzt und nicht nur einen, da die Matrixberechnungen im Batchmodus parallelisierbar sind, ergibt sich dadurch ein enormer Geschwindigkeitsgewinn gegenüber der einfachen Implementation.

Die Zeitmessungen und den daraus resultierenden Faktor für den Geschwindigkitsgewinn sind in den folgenden Tabellen zusammengefasst. Hierbei ist jede Tabelle für einen Datensatz. Der Zusatz „C“ bei den TMVA-Versionen steht für die eigene integrierte Implementierung.

Klein		
Version	Zeit	Speedup-Faktor
TMVA Original Online	30s	0
TMVA C Online	3,99s	7,52
TMVA Batch	28,7s	0
TMVA Batch C	4,18s	6,87
BLAS	2,51s	11,43
OpenMP	1,38s	20.79

Tabelle 4: Zeiten und Geschwindigkeitsgewinn des kleinen Datensatzes.

Mittel		
Version	Zeit	Speedup-Faktor
TMVA Original Online	847s	0
TMVA C Online	119s	7,11
TMVA Batch	791s	0
TMVA Batch C	117s	6,76
BLAS	47,75s	16,57
OpenMP	21,61s	36,6

Tabelle 5: Zeiten und Geschwindigkeitsgewinn des mittleren Datensatzes.

Groß		
Version	Zeit	Speedup-Faktor
TMVA Original Online	6840s	0
TMVA C Online	961s	7,12
TMVA Batch	6220s	0
TMVA Batch C	1070s	5,81
BLAS	506,86s	12,27
OpenMP	211,22s	29,44

Tabelle 6: Zeiten und Geschwindigkeitsgewinn des großen Datensatzes.

Den Tabellen kann man entnehmen, dass bereits die einfache C-Implementation ohne weitere Optimierungsschritte einen Geschwindigkeitsgewinn von Faktor 5-7 hat. Die einfache Batch-Implementation hat einen ähnlichen Geschwindigkeitsgewinn gegenüber der originalen TMVA-Version. Die Implementationen mit BLAS bzw. OpenMP

haben einen doppelten bis 6-fachen Geschwindigkeitsgewinn gegenüber der eigenen C-Implementierung.

Daraus folgt, dass man TMVA schon mit einfachen Mitteln sehr gut beschleunigen kann. In einer Umgebung, in der man das Programm als einziges auf der CPU laufen lässt, sollte die OpenMP-Version verwendet werden, falls jedoch noch andere Programme parallel gestartet sind und dementsprechend die anderen Kerne des Prozessors belegt sind, ist die BLAS-Implementation zu bevorzugen.

Neben dem Geschwindigkeitsgewinn ist es wichtig, dass die neuen Implementationen die gleichen Ergebnisse berechnen wie die ursprüngliche Implementation. Hierzu reicht es, die Histogramme der einzelnen Ausgabeneuronen pro Eventklasse in jedem Datensatz zu vergleichen.

Bei allen Versionen weichen ab dem mittleren Datensatz die Ergebnisse von der Ursprungsimplementation ab. Dazu wurden die Histogramme miteinander verglichen:

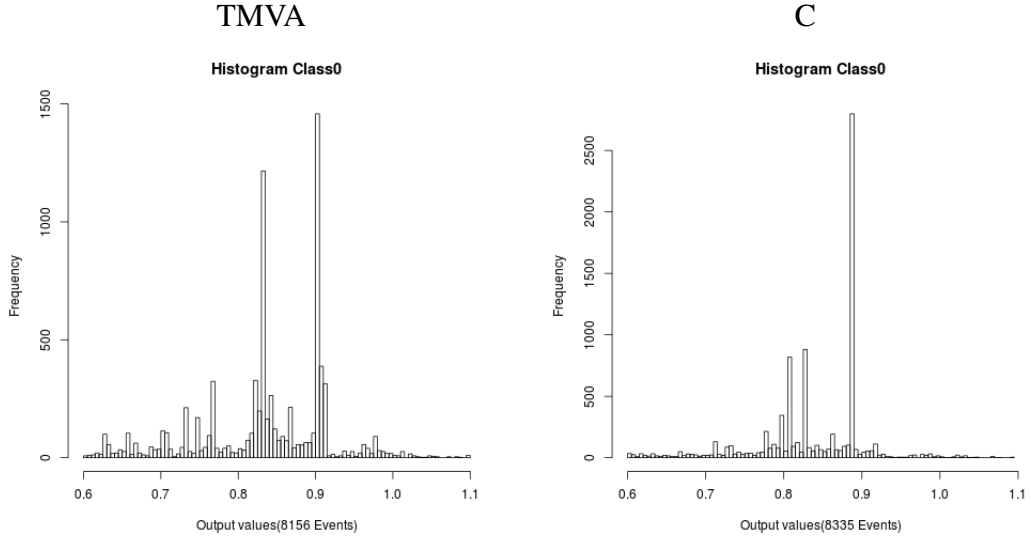

Abbildung 13: Histogramme der TMVA- und C-Implementierung des mittleren Datensatzes für Klasse 0 an Ausgabeneuron 0. Zur besseren Sichtbarkeit sind Werte unter 0,6 und über 1,1 herausgefiltert.

In Abbildung 13 ist zu erkennen, dass in beiden Implementierungen eine große Häufigeit in der Nähe von 1 ist. In beiden Fällen wurde also richtig gelernt, jedoch wurde in diesem Fall in der C-Implementation besser gelernt. Es werden Rundungsfehler als Ursache für diese Abweichungen vermutet, da bei einem gravierenden Fehler in der Implementation das Netz den Knoten komplett falsch lernen würde und die Ergebnisse nicht so nahe beieinander liegen würden.

Ein weiteres Indiz für Rundungsfehler ist die Tatsache, dass der Seed, der genutzt wird, um die Test- und Trainingsdaten in TMVA zu bestimmen, eine Auswirkung darauf hat, welche Implementation besser lernt.

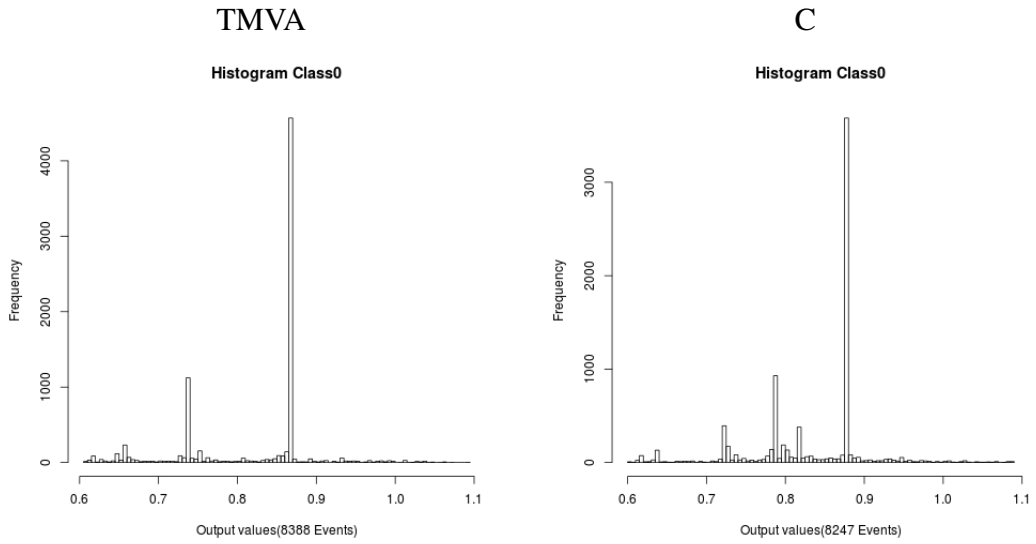

Abbildung 14: Histogramme der TMVA- und C-Implementierung des mittleren Datensatzes für Klasse 0 mit anderem Seed. Für Klasse 0 an Ausgabeneuron 0. Zur besseren Sichtbarkeit sind Werte unter 0,6 und über 1,1 herausgefiltert.

Auch bei diesen Histogrammen haben beide Implementationen richtig gelernt. In diesem Fall erkennt TMVA über 1000 Events mehr richtig als die C-Implementation und ist damit besser. Da die Ergebnisse je nach Aufteilung der Trainings- und Testdatensätze unterschiedlich gut sind bei beiden Implementationen, diese jedoch nicht weit auseinander liegen, kann man davon ausgehen, dass es sich hierbei um Rundungsfehler handelt und beide Versionen benutzt werden können um das neuronale Netz zu trainieren, da sich diese nicht sehr stark auswirken.

4 Implementierung in OpenCL

Bei der Implementierung in OpenCL wurden die gleichen Testdatensätze genutzt wie bei der C-Implementierung. Als Hardware für die Ausführung des Kernels wurde eine NVIDIA GeForce GTX 570 verwendet, die folgende Kenndaten hat:

Anzahl Kerne	480
Streamline Multiprozessoren	15
Kerne pro Prozessor	32
Lokaler Speicher	48 kb
Globaler Speicher	1280 MB
Speichertakt	1900 MHz
Prozessortakt	1484 MHz

Tabelle 7: Kenndaten der NVIDIA GeForce GTX 570

Es wurde das Batch-Learning mit Hilfe von OpenCL implementiert, wobei die Forward- und Back-Propagation über ein Batch auf einen Kernel ausgelagert wurde. Im Hauptprogramm wurde darauf geachtet, dass die Initialisierung und die Übertragung der Startwerte für den Algorithmus außerhalb der Epochenschleife und der Schleife über die Anzahl der Batches liegt, um die Datenübertragung zwischen Host und Grafikkarte möglichst gering zu halten, da so eine höhere Programmgeschwindigkeit erreicht werden kann.

Die einzigen Daten, die öfter übertragen werden, sind für jedes Batch die Nummer des Start- und des Endevents, als auch die neu berechnete Lernrate, da diese am Ende jeder Epoche neu berechnet wird. In der Schleife über die Anzahl der Batches wird für jedes Batch der Kernel einmal gestartet und am Ende der Epoche wird auf das Beenden der Kernel gewartet. Nachdem alle Epochen berechnet wurden, werden die Synapsenwerte von der Grafikkarte wieder ins Hauptprogramm gelesen. Der Algorithmus des Kernels sieht folgendermaßen aus:

Algorithmus 3: Kernel der OpenCL-Implemenation

Eingabe : Neuronenwerte der Eingabeebene, Alle Synapsen, Eventgewichte, Zielwerte für Back-Propagation, Lernrate
Ausgabe : Trainierte Synapsen

```
1  begin
2      Lege lokalen Speicher für alle benötigten Zwischenwerte an (Neuronen, Deltas...),
       wobei Dimension Eventanzahl auf Blockgröße beschränkt wird.
3      Setze nEv= Globale ID in Dimension 0
4      Setze j = Globale ID in Dimension 1
5      Setze Block_ID = Lokale ID in Dimension 0
6      Berechne: Batchgroeße = Eventnummer des letzten Events - Eventnummer des
       ersten Events
7      Berechne: Anzahl der Blöcke = Batchgröße / Blockgröße als Ganzzahl
8      if Batchgröße%Blockgröße !=0 then
9          Anzahl der Blöcke +1
10     end
11     Initialisiere lokale Synapsen mit 0
12     foreach Block do
13         Berechne Forward-Propagation
14         Berechne Deltas
15         foreach Event in Batch do
16             if nEv == Batchevent then
17                 Berechne Synapsenwerte
18             end
19             lokale Barriere
20         end
21         lokale Barriere
22         Addiere Blockgröße auf nEv
23     end
24     if Globale ID in Dimension ist 0 then
25         Addiere lokale Synapsenwerte auf globale Synapsenwerte
26     end
27 end
```

Der Kernel teilt ein Batch in kleinere Blöcke auf, sodass jeder Thread einen Teil des Batches berechnet. Der lokale Speicher ist kleiner als der globale Speicher und kann nicht alle Events gleichzeitig speichern, deshalb werden immer nur die Daten des gerade zu berechnenden Blockes von jedem Thread gespeichert. Der lokale Speicher wird genutzt, da dieser geringere Zugriffszeiten hat als der globale Speicher und somit der Kernel schneller in der Abarbeitung des Programmes ist.

Die Schleifenkonstruktion von Zeile 15-20 ist wichtig, damit die Threads in der richtigen Reihenfolge die lokalen Synapsenwerte aufaddieren. Durch die lokale Barriere am Ende der Schleife wird dafür gesorgt, dass die Threads nacheinander auf den Speicher zugreifen, da immer nur maximal ein Thread die Bedingung innerhalb der Schleife erfüllt. Diese Operation wird Reduktion genannt und ist bei Summen die von mehreren Threads gebildet werden wichtig, da ansonsten z.B. Thread 0 und Thread 1 den gleichen Wert aus dem Speicher lesen, ihren jeweiligen Wert aufaddieren und zurückschreiben. Dies würde zu einem falschen Ergebnis führen, da Thread 1 das Ergebnis der Addition aus Thread 0 benötigt.

Die gleiche Technik wird in Zeile 24 angewandt, um die lokalen Synapsenwerte auf die globalen Synapsenwerte zu addieren. Auch hier ist die Reihenfolge der Speicherzugriffe wichtig, jedoch schreibt hier nicht jeder Thread seinen eigenen Wert in den globalen Speicher, sondern es wird genau ein Thread ausgewählt, welcher dann seriell alle globalen Synapsen ändert.

Groß	
Version	Zeit
TMVA Batch	6220s
OpenCL	18611s
Mittel	
TMVA Batch	791s
OpenCL	1806s
Klein	
TMVA Batch	28,7s
OpenCL	58s

Tabelle 8: Zeiten der verschiedenen Datensätze in TMVA und OpenCL

Bei den Zeiten in Tabelle 8 sieht man, dass die OpenCL-Implementation mehr als doppelt soviel Zeit benötigt wie die Originalimplementation. Dies kann verschiedene Gründe haben.

NVIDIA empfiehlt, dass die Arbeitsgrößen immer ein Vielfaches von 32 sind[16]. Dies ist hier nicht der Fall, da die Blockgröße und damit auch die Anzahl der Threads in Dimension 0 nur 8 beträgt. Dies liegt daran, dass Dimension 1 für die Größe der verschiedenen Ebenen des neuronalen Netzes verwendet wird. Der große Datensatz hat 62 Eingabeknoten, woraus sich eine Dimensionsgröße von 64 ergibt.

Die Reduktionsschritte um die Synapsen korrekt zu berechnen werden seriell ausgeführt. Die einzelnen Kerne der Grafikkarte sind nicht besonders leistungsstark, wodurch die serielle Abarbeitung des Programmabschnitts langsam ist. Es gibt jedoch

Möglichkeiten die Reduktion in einer logarithmischen Anzahl von Schritten zu berechnen, was die Ausführungszeit wieder beschleunigt (siehe dazu [17]).

Da für jedes Batch der Anfangs- und Endwert des Batches und die Lernrate auf die Grafikkarte übertragen werden müssen, wird auch hier das Programm verlangsamt, da jede Datenübertragung Zeit kostet und man diese nach Möglichkeit vermeiden sollte. Dies könnte man lösen, indem man die Schleife über die einzelnen Batches und die Schleife über die Epochen aus dem Hauptprogramm auch noch in den Kernel überträgt. So könnte dieser alle benötigten Daten selber berechnen und müsste im Hauptprogramm nur einmal aufgerufen werden.

5 Vergleich

Nachdem beide Implementationen erläutert wurden, werden diese in diesem Kapitel hinsichtlich der Geschwindigkeit miteinander verglichen. Da die OpenCL-Implementation nur für den Batch-Modus implementiert wurde, wird hier nur auf die Batch-Implementationen eingegangen.

Klein		
Version	Zeit	Speedup-Faktor
TMVA Batch	28,7s	0
TMVA Batch C	4,18s	6,87
BLAS	2,51s	11,43
OpenMP	1,38s	20.79
OpenCL	58s	0.51

Tabelle 9: Zeiten und Geschwindigkeitsgewinn des kleinen Datensatzes.

Mittel		
Version	Zeit	Speedup-Faktor
TMVA Batch	791s	0
TMVA Batch C	117s	6,76
BLAS	47,75s	16,57
OpenMP	21,61s	36,6
OpenCL	1806s	0.46

Tabelle 10: Zeiten und Geschwindigkeitsgewinn des mittleren Datensatzes.

Groß		
Version	Zeit	Speedup-Faktor
TMVA Batch	6220s	0
TMVA Batch C	1070s	5,81
BLAS	506,86s	12,27
OpenMP	211,22s	29,44
OpenCL	18611s	0.36

Tabelle 11: Zeiten und Geschwindigkeitsgewinn des großen Datensatzes.

Die OpenCL-Implementation schneidet deutlich am schlechtesten ab, da diese nur halb so schnell ist wie die Originalimplementation. Damit ist diese für den produktiven Einsatz nicht geeignet. Gründe weshalb diese so langsam sein könnte werden in Kapitel 4 genannt. In Edinburgh wurde bereits das selbe Problem untersucht und mit CUDA eine Version implementiert, die schneller ist als die Originalversion, jedoch wurden in dieser Arbeit die BIAS-Knoten nicht berücksichtigt[9]. Dies deutet jedoch daraufhin, dass auch in OpenCL eine schnelle Version möglich ist.

Bei den Implementationen auf der CPU schneidet die Implementation mit dem Einsatz von OpenMP am besten ab. Dies liefert einen weiteren Hinweis darauf, dass das Problem prinzipiell gut zu parallelisieren ist. Sollten also mehrere Kerne auf dem Prozessor frei sein, ist die OpenMP-Implementation gegenüber allen anderen Versionen zu bevorzugen.

Sollte man nur einen Kern zur Verfügung haben, ist die Implementierung mit BLAS um bis zu einem Faktor von 16 schneller als die Orignal-Implementation.

Am langsamsten ist die einfache Implementierung in C, welche immer noch um einen Faktor 5-7 schneller ist als die TMVA-Version. Im Vergleich schneiden die neuen CPU-Implementierungen gut ab, während in die OpenCL-Version auf der Grafikkarte noch viel Arbeit investiert werden muss bis diese mithalten kann.

6 Fazit

Das Ziel der Arbeit war die Beschleunigung eines neuronalen Netzes innerhalb des Frameworks TMVA. Dazu wurden verschiedene Implementationen für die CPU in C geschrieben. Eine Version war reiner C-Code, eine weitere Implementation nutzt die BLAS-Bibliothek und eine dritte Implementation benutzt OpenMP zur weiteren Beschleunigung des Netzes. Alle drei Versionen sind schneller als die ursprüngliche Version, wobei die OpenMP-Implementation am schnellsten ist.

Desweiteren wurde eine Version mit OpenCL realisiert, um diese auf der GPU laufen zu lassen. Diese Version ist langsamer als die Original-Implementation. Dies kann an einer falsch ausgenutzten Grafikkarte liegen, da NVIDIA als Arbeitsgrößen ein vielfaches von 32 empfiehlt und dies nicht gegeben ist. Desweiteren gibt es im Kernel mehrere Stellen wo der Programmfluss nicht parallelisiert ist, sondern die Threads gezwungen werden seriell auf Speicherbereiche zuzugreifen, damit das Ergebnis richtig ist. Zusätzlich werden für jedes Batch im Hauptprogramm noch Daten zwischen Host und Grafikkarte übertragen.

Die Übertragung der Daten pro Schleifendurchlauf ließe sich lösen, indem man die Schleife über die Batches und über die Epochen zusätzlich in den Kernel aufnimmt. Die seriellen Abschnitte im Kernel lassen sich insofern parallelisieren, dass die Reduktion in einer logarithmischen Anzahl von Schritten gelöst wird und dadurch schneller wäre als die momentane Implementation. Das Problem der schlechten Grafikkartennutzung ließe sich insofern lösen, als dass man keine Netze erlaubt, die auf einer der Ebenen mehr als 32 Knoten haben, da man so die Arbeitsgrößen auf jeweils 32 einstellen kann, und damit die Karte wie von NVIDIA empfohlen ausgenutzt wird.

Literaturverzeichnis

[1] K. Prokofiev ,K. E. Selbach: *Neural network based cluster reconstruction in the ATLAS pixel detector*, Journal of Physics: Conference Series 396,022040, 2012

[2] G. Aad et al.:*Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC*, Physics Letters B, Volume 716, Issue 1, Seite 1–29, 17 September 2012

[3] ATLAS-Group: *ATLAS*, `http://www.atlas.ch/fact-sheets-view.html`, 2013

[4] EHaseler: *Nervenzelle*, `http://de.wikipedia.org/w/index.php?title=Nervenzelle&stableid=119646035` , 16:56, 17. Jun. 2013

[5] A. Hoecker, P. Speckmayer, J. Stelzer, J. Therhaag, E. von Toerne, H. Voss: *TMVA Users Guide*, `http://tmva.sourceforge.net/docu/TMVAUsersGuide.pdf`, 2009

[6] Se'taan: *Receiver Operating Characteristic*, `http://de.wikipedia.org/w/index.php?title=Receiver_Operating_Characteristic&stableid=119283837`, 22:17, 6. Jun. 2013

[7] Fons Rademakers et al.:*About Root*, `http://root.cern.ch/drupal/content/about`, 1995-2012

[8] David E. Rumelhart, James L. McClelland: *Parallel Distributed Processing Explorations in the microstrucutre of Cognition Volume 1: Foundations*, A Bradford Book, 1987

[9] A.Hoecker: *Acceleration of multivariate analysis techniques in TMVA using GPUs*, Journal of Physics: Conference Series 396,022055, 2012

[10] Vincent Vanhoucke et al.: *Improving the speed of neural networks on CPUs*, Deep Learning and Unsupervised Feature Learning Workshop, NIPS, 2011

[11] Honghoon Jang et al.: *Neural Network Implementation using CUDA and OpenMP*, Digital Image Computing: Techniques and Applications, Seite 155-161, IEEE, 2008

[12] Atlas*BLAS* `http://math-atlas.sourceforge.net/`, 2013

[13] OpenMP ARB Corporation: *OpenMP*, `http://openmp.org/openmp-faq.html`, 2013

[14] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, D. Ginsburg: *OpenCL Programming Guide*, Addison-Wesley Longman, 2011

[15] B. R. Gaster, L. Howes D. R. Kaeli, P. Mistry, D. Schaa: *Heterogeneous Computing with OpenCL*, Elsevier Science & Technology, 2011

[16] NVIDIA, *NVIDIA OpenCL Best Practices Guide* , `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`, 2009

[17] AMD, *OpenCL™ Optimization Case Study: Simple Reductions*, `http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/`, 2010

Abbildungsverzeichnis