Fachhochschule Aachen
Campus Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Technomathematik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Design und Anwendung eines skalierbaren parallelen künstlichen neuronalen Netz

Masterarbeit

von

Matthias Richerzhagen
Matrikelnummer: 855 423

Jülich, Oktober 2016

Eigenständigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Ort, Datum	Matthias Richerzhagen

Diese Arbeit wurde betreut von:

1. Prüfer:	Prof. Dr. rer. nat. Bodo Kraft
2. Prüfer:	Prof. Dr.-Ing. Morris Riedel

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Um Informationen über die Beschaffenheit eines Gebietes zu erhalten, werden beim „Remote Sensing“ [1] Sensoren an Satelliten und Flugzeuge eingesetzt, um aus großer Entfernung Messungen über die Erdoberfläche zu erhalten. Hierbei kommen Sensoren zum Einsatz, welche zum Beispiel die elektromagnetische Abstrahlung der Erdoberfläche in unterschiedlichen Frequenzbereichen messen.

Um die Beschaffenheit eines Gebietes mit Remote Sensing zu bestimmen, muss anhand von diesen Messwerten zum Beispiel entschieden werden, um welche Art von Bodenbedeckung es sich an einer Stelle in einem Gebiet handelt und die verschiedenen Klassen von Bodenbedeckungen in einem Gebiet voneinander unterschieden werden.

Da der abgedeckte Bereich eines Sensors und die dadurch gemessenen Daten sehr groß sind, ist es nicht praktikabel einen Menschen die Klassifizierung vornehmen zu lassen. Des Weiteren bestehen die Daten aus sehr vielen verschiedenen Kanälen von Messwerten, sodass diese, wenn überhaupt, nur von einem Spezialisten interpretiert werden könnten.

Um dennoch in der Lage zu sein, die durch Remote Sensing gewonnenen Daten auswerten zu können und zusätzliche Informationen zu gewinnen, müssen die einzelnen Datenpunkte automatisch klassifiziert werden. Hierzu kommen, statt einer manuellen Auswertung durch einen Menschen, Ansätze aus dem Machine Learning zum Einsatz. Die Remote Sensing Daten aus der Anwendung werden in die sogenannten Trainingsdaten und Testdaten im Machine Learning unterteilt.

Um einen Machine Learning Ansatz zu verwenden muss zunächst ein Modell trainiert werden. Hierzu kommen Computer Algorithmen zum Einsatz. Diese Machine Learning Algorithmen werden in zwei Unterkategorien unterteilt: Das überwachte und das unüberwachte Lernen (engl. supervised und unsupervised learning).

Beim unüberwachten Lernen versucht ein Algorithmus ohne zusätzliche Information in den Daten eine Struktur zu erkennen. Hierzu gehört zum Beispiel das Clustering, bei dem versucht wird, nur anhand der Messwerte ähnliche Datenpunkte zu identifizieren und diese in Cluster zu gruppieren. Hierzu kann zum Beispiel der k-Means[2] oder der Density-Based Spatial Clustering of Applications with Noise (DBSCAN)-Algorithmus[3] verwendet werden.

Im Gegensatz zum unüberwachtem, werden beim überwachtem Lernen dem Algorithmus zusätzliche Informationen über einen Datenpunkt bereitgestellt. Ein Klassifizierungs-Algorithmus aus dem überwachtem Lernen bekommt zusätzlich zu den Datenpunkten auch deren Klassenzugehörigkeit mitgeteilt. Es wird dann versucht anhand dieser Informationen ein Modell zu erzeugen, das eine Generalisierung ermöglicht, welche auch auf unbekannte Datenpunkte angewandt werden kann, um deren Klassenzugehörigkeit möglichst genau zu bestimmen.

Auch zur Klassifizierung stehen unterschiedliche Algorithmen zur Verfügung. Im Rahmen dieser Arbeit wird ein künstliches neuronales Netz, in Form eines mehrschichtigen Perzeptrons (engl. Multilayer Perzeptron, MLP) [4] verwendet.

Ein Perzeptron[5] stellt ein einfaches Lernmodell dar, bei dem künstliche Neuronen aus einer Eingabe eine entsprechende Ausgabe erzeugen. Die künstlichen Neuronen beinhalten Gewichtungen für jeden Wert einer Eingabe und werten mit einer gewichteten Summe eine Schwellwertfunktion aus, um ihre Aktivierung zu bestimmen. Die künstlichen Neuronen sind dabei der Funktion von Nervenzellen im Gehirn des Menschen und anderen Lebewesen nachempfunden.

Mit einem Algorithmus zum Anpassen der Gewichtungen der Neuronenverbindungen kann das Perzeptron dann mit einem Trainingsdatensatz trainiert werden. Die Gewichtungen der künstlichen Neuronen werden dabei schrittweise dahingehend verändert, dass das Perzeptron für eine Eingabe die gewünschte Ausgabe erzeugt.

Für ein mehrschichtiges Perzeptron werden mehrere Perzeptren hintereinander geschaltet, um ein Modell zu erhalten, welches in der Lage ist noch kompliziertere Strukturen als ein einfaches Perzeptron zu erlernen. Das Training eines MLPs verläuft ähnlich, wie das eines Perzeptrons. Es kann mit dem sogenannten Backpropagation Algorithmus [4] trainiert werden.

Da die Datenmengen im Remote Sensing sehr groß sind, können diese zunehmend nicht mehr ohne die Unterstützung eines Supercomputers verarbeitet werden. Um diese großen Rechenkapazitäten nutzen zu können, werden allerdings dementsprechend angepasste parallele Algorithmen benötigt. Des Weiteren ist das Trainieren eines Modells oft mit längeren Trainingszeiten verbunden und so ist das Ziel der Parallelisierung auf Supercomputern auch eine schnellere Ausführungszeit zu erreichen. Daher soll ein paralleles und skalierbares Multilayer Perzeptron entwickelt werden.

Der technologische Fortschritt bei Grafikkarten (engl. graphics processing unit, GPU) hat dazu geführt, dass diese immer effizienter und billiger geworden sind. Zusätzlich lassen sich diese nun mit der „General Purpose Computation on GPU“ (GPGPU)-Technik auch für beliebige Berechnungen einsetzten und können so, durch ihre große Anzahl an Rechenkernen, sehr schnell viele Berechnungen gleichzeitig durchführen. So haben GPUs Einzug in den Bereich des Supercomputing genommen. Wegen ihrer guten Effizienz wird in dieser Arbeit für den parallelen Algorithmus auf GPUs gesetzt.

Diese Arbeit ist wie folgt strukturiert. Zunächst werden in Kapitel 2 die Grundlagen von Remote Sensing, Parallel Computing im Allgemeinen, sowie künstlichen neuronalen Netzen erläutert. Anhand dessen wird die Problemstellung dargestellt.

In Kapitel 3 werden verwandte Arbeiten vorgestellt. Basierend darauf wird in Kapitel 4 ein Design für eine Parallelisierung des Backpropagation-Algorithmus entwickelt und in Kapitel 5 einige Details einer solchen Implementierung dargestellt. Zuletzt werden in Kapitel 6 die Ergebnisse dargestellt und evaluiert sowie eine Aussicht auf eventuelle weitere zukünftige Aufgaben gegeben. In Kapitel 7 wird die Arbeit noch einmal zusammengefasst.

2 Grundlagen

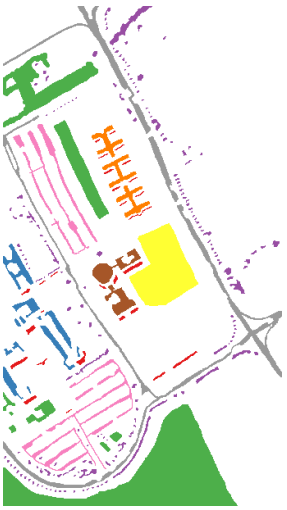
2.1 Remote Sensing

Remote Sensing ist ein Verfahren zur Gewinnung von Informationen über die Erdoberfläche, bei dem Sensoren an Flugzeugen oder Satelliten zum Einsatz kommen. Diese Sensoren messen zum Beispiel die elektromagnetische Abstrahlung der Oberfläche in einem sehr breiten Frequenzspektrum.

Da die unterschiedlichen Materialien auf der Erdoberfläche eine unterschiedliche Abstrahlung besitzen, können anhand dieser Messungen Rückschlüsse über die Beschaffenheit der Erdoberfläche gezogen werden [1].

Für Anwendungen im Bereich des Remote Sensing, wie Städteplanung, Forstwirtschaft, Katastrophenschutz aber auch militärische Anwendungen spielt Klassifizierung häufig eine wichtige Rolle, denn es sollen zum Beispiel Informationen über die Bodenbedeckung in Städten oder die Größe, Art und Bestand von Wäldern gewonnen werden. Hierzu müssen die verschiedenen Typen von Bodenbedeckung, beziehungsweise die Arten von Wäldern voneinander unterschieden werden. Ein Beispiel für einen Remote Sensing Datensatz ist in Abb. 2.1 gegeben.

(a) Farbbild



(b) Klassenlabels

Nr.	Klasse	Anzahl
1	Asphalt	6631
2	Meadow	18649
3	Gravel	2099
4	Trees	3064
5	Metal sheets	1345
6	Bare soil	5029
7	Bitumen	1330
8	Self-blocking bricks	3682
9	Shadows	947

(c) Klassenzuordnungen

Abbildung 2.1: Beispiel für Remote Sensing: Ein Luftbild der Pavia University mit verschiedenen Arten von Bodenbedeckung. Der vollständige Datensatz enthält 9 Klassen, 103 Frequenzbänder und besitzt eine geometrische Auflösung von 1,3 m. [6]

Zur Klassifizierung können Algorithmen aus dem Machine Learning verwendet werden. Es gibt viele verschiedene Algorithmen, diese Arbeit beschränkt sich jedoch auf den Einsatz von künstlichen neuronalen Netzen, beziehungsweise mehrschichtigen Perzeptrons.

Durch die große Anzahl an Sensoren und die Größe des Gebietes, welches abgetastet wird, entstehen beim Remote Sensing sehr große Datenmengen. Diese können zunehmend nicht mehr mit seriellen Algorithmen verarbeitet werden, sodass eine parallele Verarbeitung benötigt wird. Zusätzlich kann durch eine parallele Verarbeitung der Prozess der Klassifikation deutlich beschleunigt werden. Es muss jedoch auch mit einer parallelen Verarbeitung sichergestellt werden, dass die Klassifizierungsgenauigkeit, also der Prozentsatz der richtig klassifizierten Datenpunkte im Datensatz, beibehalten wird.

2.1.1 Klassifizierung

Klassifizierung bedeutet, einzelne Datenpunkte zu vordefinierten Klassen zuzuordnen. Diese Entscheidung wird anhand der Messdaten eines Datenpunktes getroffen. Diese Messdaten werden auch als „Features“ oder Merkmale bezeichnet. Ein Datenpunkt, welcher mehrere Features besitzt, kann dann als „Feature-Vektor“ interpretiert werden.

Die Daten aus dem Remote Sensing entsprechen dann Bildern, bei denen jedes Pixel einen Datenpunkt darstellt. Im Gegensatz zu normalen Farbbildern, welche üblicherweise nur aus den drei Farbkanälen Rot, Blau und Grün bestehen, besitzen die Remote Sensing Bilder mehr als 3 Kanäle für verschiedenste Wellenlängen. Aufgrund dieser hohen Dimensionalität ist es schwierig diese zweidimensional für das menschliche Auge darzustellen.

Um die Klassifizierung zu verbessern existieren Techniken, welche zum Beispiel die Anzahl der Features reduzieren können oder aber künstliche Features zu einem Datenpunkt hinzufügen. Diese beinhalten zum Beispiel Informationen aus geographisch benachbarten Datenpunkten, denn die Umgebung eines Datenpunktes kann sehr hilfreich sein, um seine Klassenzugehörigkeit zu bestimmen.

Der Feature-Vektor entspricht einem Ortsvektor in einem mehrdimensionalen Raum, dem „Feature Space“, welcher so viele Dimensionen besitzt, wie Features vorhanden sind. Um eine Klassifizierung durchführen zu können wird eine Trennlinie (im zweidimensionalen), beziehungsweise eine Trennebene oder auch Trennoberfläche (im mehrdimensionalen, engl. decision boundary) gesucht, welche die Klassen im Feature Space perfekt oder annähernd voneinander separiert.

Wenn diese Trennebene bekannt ist, muss für einen unklassifizierten Datenpunkt lediglich bestimmt werden, auf welcher Seite der Trennebene er sich befindet, um eine mögliche Klassenzugehörigkeit zu bestimmen. Eine Klassifizierung ist nur möglich, wenn eine solche Trennebene existiert und es möglich ist aus den gemessenen Features eine Klassenzugehörigkeit abzuleiten.

Algorithmen aus dem Machine Learning versuchen auf verschiedene Arten solch eine Trennebene zu bestimmen. Diese Suche und Anpassung an die vorhandenen Daten wird auch als Trainingsprozess, beziehungsweise „Training“ bezeichnet.

Für viele Anwendungsfälle ist es nicht ausreichend eine lineare Trennebene zu bestimmen, denn manche Probleme sind nicht „linear separierbar“. Es werden dementsprechend nichtlineare decision boundarys benötigt. In der Praxis sind Lösungen etabliert, welche wegen der häufig nicht vorhandenen linearen separabilität auch in einem gewissen Maße Fehler zulassen, um die allgemeine Generalisierung der Lösung zu verbessern und somit ein robusteres Modell zu erhalten.

2.1.2 Training und Testdaten

Für die Verwendung eines supervised Learning Algorithmus wird „ground truth“ oder „labeled data“ benötigt. Hierbei handelt es sich um Datenpunkte für die bereits eine Klassifizierung bekannt ist, welche als korrekt angenommen wird. Diese wird durch manuelle Auswertungen von Spezialisten oder sogar Bodenbegutachtungen eines gemessenen Gebietes bestimmt. Dies ist unter Umständen sehr aufwändig und kann deshalb nicht für das gesamte Gebiet durchgeführt werden.

Um einen Machine Learning Algorithmus einzusetzen zu können muss sichergestellt werden, dass der Algorithmus auch korrekte Ergebnisse errechnet. Damit dies sichergestellt ist, muss neben dem Training des Machine Learning Algorithmus auch immer ein Testen stattfinden. Hierzu wird ein zweiter Datensatz benötigt, für den genau wie für den Trainingsdatensatz schon die Klassenlabels für alle Datenpunkte bekannt sind.

Somit können nicht alle groud truth Daten nur für das Training verwendet werden. Ein Großteil der ground truth Daten muss zum Testen des Modells zurückgehalten werden.

Für den Testdatensatz wird mit dem Modell eine Vorhersage durchgeführt und mit den vorhandenen Klassenzugehörigkeiten überprüft, ob das gelernte Modell auch für ungesehene Daten gute, beziehungsweise akzeptable Ergebnisse produziert, also der Prozentsatz der richtig klassifizierten Daten für den Anwendungsfall hoch genug ist. Dieser Testdatensatz darf allerdings nur zum Testen des Ergebnisses verwendet werden, da sonst die Ergebnisse verfälscht würden.

2.2 Parallel Computing

Um große Datenvolumen zu verarbeiten und große Rechenpower nutzen zu können, wird Parallel Computing verwendet. Anstatt ein Problem auf nur einem Prozessorkern mit nur einem Prozess zu verarbeiten werden mehrere Prozesse auf mehreren Kernen parallel eingesetzt. Dabei werden nicht mehrere Kerne auf einem herkömmlichen Laptop verwendet, sondern Supercomputer. Ein Supercomputer besitzt dann im Vergleich zu einem Laptop nicht vier, sondern mehrere hunderttausend Prozessoren.

Um mehrere Prozessorkerne gleichzeitig effizient nutzen zu können muss der Algorithmus parallelisierbar sein. Das ist er, wenn sich einzelne Arbeitsschritte des Algorithmus in voneinander unabhängige Teilprobleme unterteilen lassen. Diese werden dann parallel bearbeitet. Hierzu werden häufig die zu verarbeitenden Daten auf die einzelnen Prozesse aufgeteilt, sodass jeder Prozess nur einen Teil der Daten besitzt.

Für die Kommunikation zwischen den Prozessen wird ein Kommunikationsnetzwerk benötigt. Desto weniger Kommunikation zwischen den Prozessen stattfinden muss, desto

besser funktioniert die Parallelisierung. Nicht jeder Algorithmus lässt sich parallelisieren und bei vielen müssen Anpassungen gemacht werden um eine Parallelisierung zu ermöglichen.

Große Parallelrechner, welche sehr viele Prozessorkerne besitzen und viele Prozesse gleichzeitig ausführen können werden als Supercomputer bezeichnet.

Parallelrechner werden in zwei Kategorien eingeteilt: solche mit gemeinsamem Speicher(engl. shared memory) und solche mit verteiltem Speicher(engl. distributed memory).

2.2.1 Speedup

Um die Güte einer Parallelisierung zu bestimmen wird der sogenannte „Speedup“ benutzt. Nach dem Amdahlschen Gesetz[7] ergibt sich der Speedup aus dem Quotienten aus der schnellstmöglichen Ausführungszeit mit einem seriellen Algorithmus und der Ausführungszeit einer gleichwertigen parallelen Ausführung.

$$\text{Speedup}(p) = \frac{\text{serielle Ausführungszeit}}{\text{parallele Ausführungszeit mit } p \text{ Prozessen}}$$

Mit einer steigenden Prozessoranzahl sollte bei einer guten Parallelisierung dann auch der Speedup ansteigen. Von einem optimalen oder linearen Speedup spricht man, wenn der Speedup der Anzahl der verwendeten Prozesse entspricht.

Das Amdahlsche Gesetz geht davon aus, dass die Problemgröße konstant bleibt. Dies hat als Resultat, dass selbst wenn nur kleinste Teile des Programms nicht parallelisierbar sind ein optimaler Speedup mit vielen Prozessen nicht erreicht werden kann und damit der Speedup durch die seriellen Teile einer parallelen Anwendung begrenzt ist.

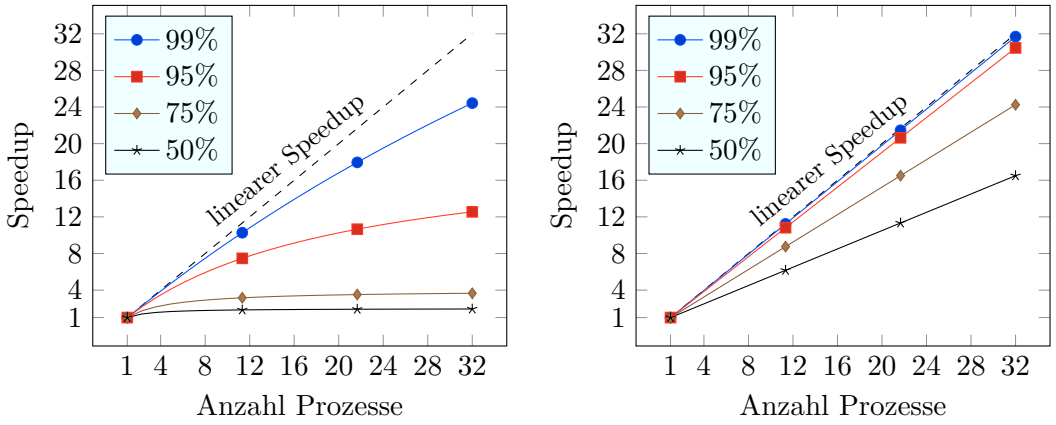
Als Reaktion wurde Gustafsons Gesetz[8] vorgestellt, welches zeigt, dass wenn die Problemgröße mit der Prozessoranzahl steigt, sehr wohl ein Speedup erreicht werden kann. In Abb. 2.2a und 2.2b auf der nächsten Seite ist der Speedup für verschiedene Grade der Parallelisierung mit Amdahls und Gustafsons Gesetz dargestellt.

Speedup nach Amdahl wird als „strong scaling“ und Speedup nach Gustafson als „weak scaling“ bezeichnet. Beide Metriken sind nach wie vor im Parallel Computing relevant.

2.2.2 Gemeinsamer Speicher

Bei Parallelrechnern mit gemeinsamem Speicher können alle Prozessorkerne mit einem einheitlichen Adressbereich auf den gemeinsamen Speicher zugreifen. Da heutzutage ein Großteil der Heim-Computersysteme bereits mit mehreren Prozessorkernen ausgestattet ist, welche auf einen gemeinsamen Arbeitsspeicher zugreifen, sind diese Systeme bereits kleine Parallelrechner.

Ein gemeinsamer Speicher hat den Vorteil, dass Daten nicht zwischen verschiedenen Prozessen hin und her kopiert werden müssen. Ab einer bestimmten Anzahl an Prozessen ist es jedoch kaum noch möglich die Speicherzugriffszeit aller Prozesse auf den gesamten Speicher klein zu halten, da der Synchronisationsaufwand überhand nimmt.



(a) Speedup nach Amdahls Gesetz bei gleichbleibender Problemgröße.

(b) Speedup nach Gustafsons Gesetz bei steigender Problemgröße.

Abbildung 2.2: Speedup in Abhängigkeit davon, wie viel Prozent der Laufzeit sich parallelisieren lassen.

Als Programmiermodell kann zum Beispiel OpenMP[9] verwendet werden. Es ermöglicht den Source Code des Programms mit bestimmten Anotationen zu versehen, die festlegen, welche Bereiche eines Programms auf welche Art und Weise parallel verarbeitet werden sollen. OpenMP verwendet anstatt Prozessen etwas leichtgewichtigere Threads.

2.2.3 Verteilter Speicher

Bei Systemen mit verteiltem Speicher können die Prozesse nur auf ihren eigenen Speicherbereich zugreifen und es existiert kein gemeinsamer Adressbereich. Benötigt ein Prozess die Daten eines anderen Prozesses müssen Nachrichten ausgetauscht werden.

Als Programmiermodell kommt hier der Message Passing Interface (MPI)-Standard[10] zum Einsatz, welcher eine Schnittstelle zum Versenden und Empfangen von Nachrichten zwischen Prozessen bereitstellt.

Es werden einfache Punkt zu Punkt, aber auch kompliziertere globale Kommunikationen wie Reduktionen und Broadcasts unterstützt. Im Gegensatz zu Systemen mit gemeinsamen Speicher muss hier auch sichergestellt werden, dass die Daten geschickt auf die Prozesse aufgeteilt werden.

2.2.4 Hybride Systeme

Ein Großteil der heutigen Superrechner sind hybride Systeme. Ein Supercomputer besteht dann aus mehreren Nodes, welche jeweils mehrere Prozessoren sowie einen lokalen Speicher besitzen. Innerhalb eines Nodes kann der gemeinsame Speicher mit mehreren Prozessoren genutzt werden. Über die Node-Grenzen hinweg ist der Speicher jedoch verteilt, sodass Kommunikation mit MPI stattfinden muss. Dies ist in Abb. 2.3 auf der nächsten Seite dargestellt.

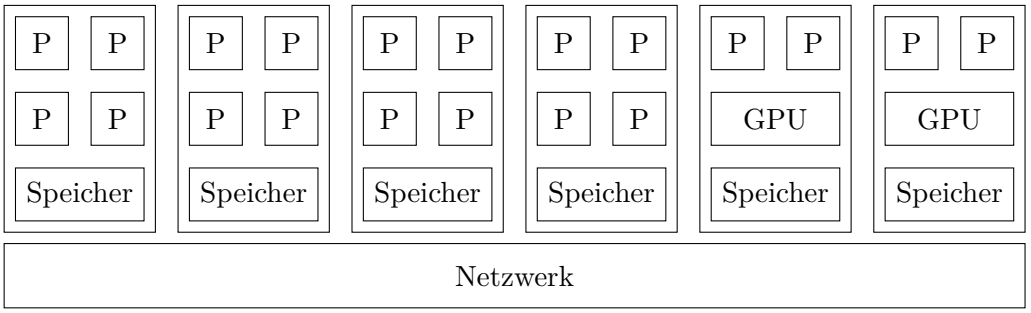


Abbildung 2.3: Schematische Darstellung eines hybriden Supercomputers. Verschiedene Nodes, welche aus mehreren Prozessoren, eigenem Speicher und eventuell einer Grafikkarte bestehen, sind über ein Netzwerk miteinander verbunden.

2.2.5 Grafikkarten und Beschleuniger-Prozessoren

Neben den hybriden Systemen, welche nur aus CPUs bestehen, existieren auch solche, die sogenannte Koprozessoren besitzen, welche die CPU bei bestimmten Berechnungen unterstützen können.

Als Koprozessor kommen zum Beispiel GPUs wie eine NVIDIA Tesla oder aber spezielle Beschleuniger-Prozessoren wie ein Intel Xeon Phi zum Einsatz. Durch den Einsatz von Koprozessoren kann eine herausragende Effizienz erreicht werden: Im November 2015 waren alle Supercomputer auf den ersten 40 Plätzen der Green500 Topliste, welche die effizientesten Supercomputer auf der ganzen Welt auflistet, mit Koprozessoren in Form von GPUs oder Beschleuniger-Prozessoren ausgestattet [11].

Eine GPU besteht neben ihrem eigenen Speicher aus sehr vielen kleinen Prozessoren, sogenannten Shader-Einheiten, welche durch die sehr schnelle Anbindung an den Speicher der GPU und deren sehr große Anzahl ein sehr hochgradiges paralleles Rechnen ermöglichen. Um diese Shader Einheiten zu verwenden muss zunächst ein entsprechendes Programm, ein sogenannter Kernel, für die entsprechende GPU kompiliert und anschließend auf diese kopiert werden. Dieser GPU Kernel kann dabei nur auf den Speicherbereich der GPU zugreifen. Das bedeutet, dass alle Eingabedaten auf die GPU kopiert werden müssen und auch die Ausgabedaten wieder von der GPU in den Hauptspeicher der CPU kopiert werden müssen. Ein ständiges hin und her kopieren von Daten ist sehr ineffizient und sollte deshalb vermieden werden. Die Architektur einer GPU und die Verbindung zur CPU ist in Abb. 2.4 auf der nächsten Seite dargestellt.

Für die Programmierung von Koprozessoren stehen verschiedene Frameworks zur Verfügung, welche größtenteils von der verwendeten Hardware abhängen. Für NVIDIA Grafikkarten, welche sehr weit verbreitet sind, kommt zum Beispiel das CUDA Framework zum Einsatz. Mit der Open Computing Language (OpenCL) [12] wurde versucht eine einheitliche Schnittstelle für Koprozessoren zu entwickeln, welche unabhängig von der verwendeten Hardware ist. Dementsprechend ist diese jedoch eingeschränkter als die hardwarespezifischen Schnittstellen.

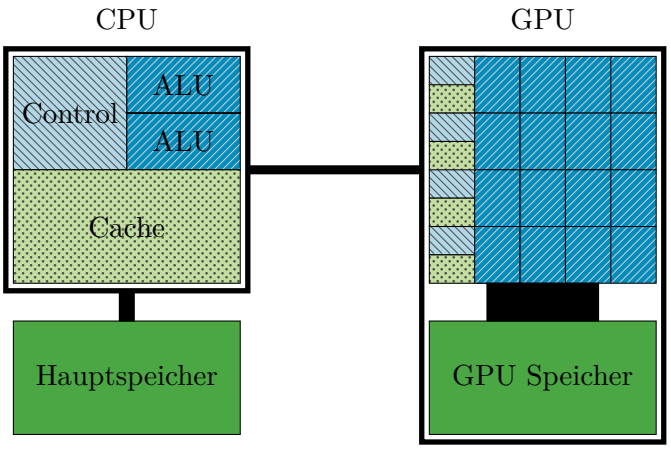


Abbildung 2.4: GPU Architektur und Verbindung mit der CPU.

2.3 Künstliche neuronale Netze

Künstliche neuronale Netze (KNN, englisch: Artificial Neural Network, ANN, [13]) gehören zu den Lernmodellen, die zum maschinellen Lernen verwendet werden. Bei KNNs wird versucht die Struktur des Gehirns, beziehungsweise eines neuronalen Netzes auf dem Computer nachzubilden, um einen lernfähigen Algorithmus zu erhalten. KNNs sind in der Lage nichtlineare Funktionen zu approximieren.

Im Folgenden Abschnitt 2.3.1 wird ein kurzer Einblick in die Funktionsweise eines natürlichen neuronalen Netzes gegeben. In Abschnitt 2.3.2 wird die Funktionsweise eines künstlichen neuronalen Netzes anhand eines Perzeptrons beschrieben und dann in Abschnitt 2.3.3 auf ein mehrschichtiges Perzeptron erweitert.

2.3.1 Natürliche neuronale Netze

Neuronale Netze sind die Kommunikationsform der Nervenzellen/Neuronen in Lebewesen. Ein natürliches neuronales Netz kann aus bis zu mehreren Millionen bis Milliarden, miteinander verknüpften, Neuronen bestehen. Jedes einzelne Neuron erhält von den anderen, mit ihm verbundenen Neuronen, Reize. Übersteigen die Reize ein bestimmtes Aktivierungspotential löst das Neuron seinerseits einen Reiz aus.

Die Stärke des Reizes, die ein Neuron von einem anderen empfängt, hängt von der Entfernung des Verbindungspunktes zum Zellkern ab. Jedes verbundene Neuron kann also einen unterschiedlich starken Einfluss auf die Aktivierung haben.

Manche der Neuronen, die untereinander das Netz bilden, besitzen Verbindungen zu Nervenzellen wie dem Tastsinn und dem Sehnerv, aber auch den Muskeln um Bewegungen zu steuern. Mit diesen, im Folgenden Eingabe- und Ausgabeneuronen genannt, kann das neuronale Netz seine Umgebung wahrnehmen und auf sie reagieren.

Mit künstlichen neuronalen Netzen wird versucht solch ein neuronales Netz im Computer nachzubilden. Die Schwierigkeit hierbei besteht unter anderem darin, die Verbin-

dungen im künstlichen neuronalen Netz so zu bilden, dass auf einen äußeren Einfluss die gewünschte Reaktion des Netzes erfolgt.

2.3.2 Perzeptron

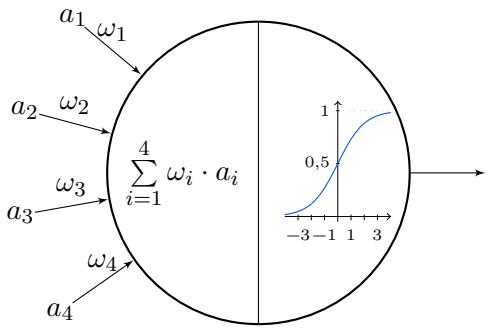
Ein Perzeptron[5] ist ein sehr einfaches Lernmodell, welches die Grundform eines künstlichen neuronalen Netzes darstellt. Es besteht aus einer vorgegebenen Anzahl an Eingabe- und Ausgabeneuronen. Jedes Eingabeneuron ist mit jedem Ausgabeneuron direkt verbunden und allen Verbindungen wird eine Gewichtung zugewiesen. Eine fehlende Verbindung entspricht dann einer Gewichtung von 0.

Die Aktivierungen der Einabeneuronen werden durch den Datenpunkt vorgegeben. Jedes Eingabeneuron ist für ein Feature verantwortlich.

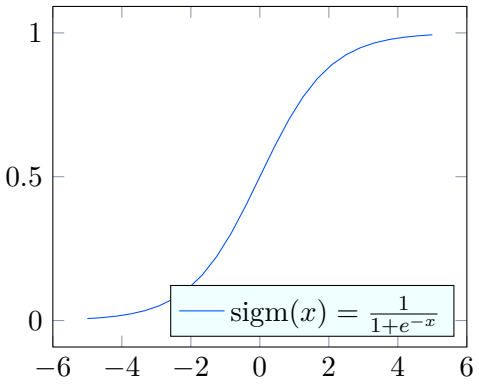
Um die Reaktion des Perzeptrons auf eine Eingabe zu erhalten, bildet ein Ausgabeneuronen eine gewichtete Summe über alle Aktivierungen der Eingangsneuronen und wertet mit dieser Summe eine Aktivierungsfunktion aus. Der Wert dieser Funktion an der entsprechenden Stelle entspricht dann der Aktivierungsstärke des Ausgabeneurons. Abb. 2.5a stellt dies dar.

Als Aktivierungsfunktion wird meistens die Sigmoidfunktion $\text{sigm}(x) = \frac{1}{1+e^{-x}}$ verwendet, da sie sehr positive Eigenschaften bezüglich ihrer Ableitung besitzt und nur einen Wertebereich zwischen 0 und 1 besitzt. Diese Funktion ist in Abb. 2.5b dargestellt.

Diese Aktivierungsfunktion wird zusätzlich um einen Schwellwert verschoben. Dieser lässt sich auch als ein zusätzliches Eingabeneuron auffassen, welches stets aktiviert ist.

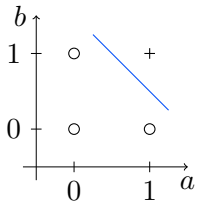


(a) Schematische Darstellung eines künstlichen Neurons. Ein Neuron summiert die ankommenden Aktivierungen gewichtet auf und wendet eine Schwellwert-Funktion auf die gewichtete Summe an, um die eigene Aktivierung zu bestimmen.



(b) Verlauf der Sigmoid Funktion.

Abbildung 2.5: Ein künstliches Neuron und die Sigmoid Aktivierungsfunktion

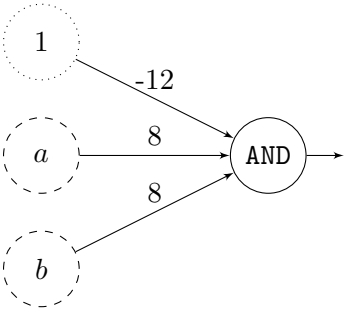


(a) `AND` lässt sich mit einer einzelnen Trennlinie lösen.

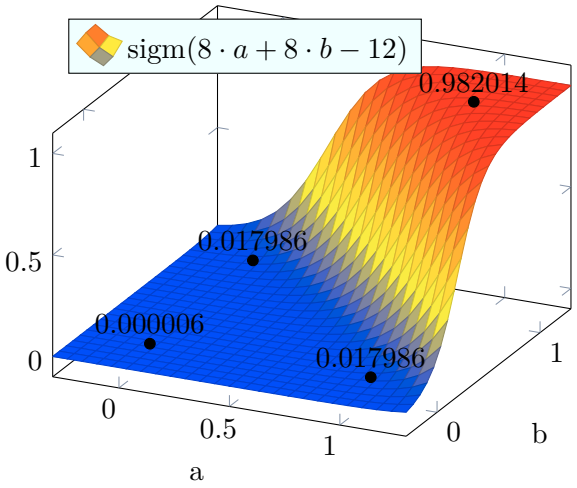
a	b	a `AND` b
0	0	0
0	1	0
1	0	0
1	1	1

(b) Wahrheitswerttabelle für `AND`

Abbildung 2.6: Das logische „Und“ (`AND`).



(a) Perzeptron mit Gewichtungen für zwei Eingabe- und ein Schwellwert-Neuron.



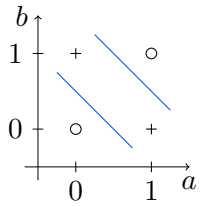
(b) Ausgabe des neuronalen Netzes aus a. Das logische „Und“ wird sehr gut angenähert.

Abbildung 2.7: Perzeptron, welches das logische „Und“ darstellt.

Abb. 2.6 stellt das logische „Und“-Problem als Diagramm und Wahrheitswerttabelle dar. Dieses wird von dem Perzeptron in Abb. 2.7 angenähert und die Ausgabe entspricht der Funktion $\text{sigm}(8 \cdot a + 8 \cdot b - 12) = \text{sigm}(8 \cdot (a + b - 1{,}5))$. Für dieses Beispiel wurden die Gewichtungen 8 für beide Eingabeneuronen a und b, sowie der Schwellwert -12 von Hand bestimmt. Der Faktor 8 wurde empirisch gewählt um eine ausreichende Steigung an der Trennstelle zu erreichen.

Solch ein einfaches Perzeptron kann eine logische Funktion jedoch niemals exakt erfüllen, da die Werte 0 und 1 von der Aktivierungsfunktion nur asymptotisch angenähert, aber nicht angenommen werden.

Aufgrund der Beschaffenheit des Perzeptrons ist es allerdings nur sehr eingeschränkt nutzbar. Das logische „exklusive Oder“ (`XOR`) lässt sich zum Beispiel nicht mit einem einzelnen Perzeptron darstellen, da es ein nicht linear separierbares Problem darstellt und somit eine einzelne Trennlinie nicht ausreicht, um die verschiedenen Ausgaben voneinander zu trennen.



(a) XOR lässt sich nicht durch eine einzige Trennlinie lösen.

a	b	a XOR b	a NAND b	a OR b
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	0	0	1

(b) Wahrheitswerttabelle für die logischen Verknüpfungen XOR, NAND und OR.

Abbildung 2.8: Das „exklusive Oder“ (XOR).

Ein XOR lässt sich jedoch durch den logischen Ausdruck (a NAND b) AND (a OR b) darstellen, was soviel bedeutet wie „Nicht beide aber mindestens eines“. Für diesen Ausdruck werden lediglich NAND, AND und OR-Operatoren benötigt, welche alle mit Perzeptren abgebildet werden können. In Abb. 2.8 ist das XOR-Problem mit einem Diagramm und einer Wahrheitswerttabelle dargestellt.

2.3.3 Mehrschichtiges Perzeptron

Es hat sich gezeigt, dass sich durch den Einsatz von verschachtelten Perzeptrons komplexere Funktionen abbilden lassen. Dies wird als mehrschichtiges Perzeptron (englisch: Multilayer Perceptron, MLP) [4] bezeichnet. Hierbei werden die Eingabe-Neuronen nicht, wie beim einfachen Perzeptron, direkt mit den Ausgabeneuronen verbunden, sondern es wird zwischen Eingabe- und Ausgabeschicht eine oder mehrere weitere Neuronenschichten eingefügt, welche als „Hidden Layer“ bezeichnet werden. Jedes Neuron enthält alle Aktivierungen der vorherigen Schicht. Abb. 2.9 auf der nächsten Seite stellt ein solches MLP dar.

Erst die Neuronen der letzten versteckten Schicht sind dann mit den Ausgabeneuronen in der Ausgabeschicht verbunden. Während die Anzahl der Eingabe- und Ausgabeneuronen oft vom zu lösenden Problem abhängt, aber durch Feature Engineering vorbereitet werden kann, ist die Anzahl der versteckten Neuronen eine Stellgröße, welche vorgegeben und an das zu lösende Problem angepasst werden muss.

Mit einem mehrschichtigen Perzeptron lässt sich auch das „exklusive Oder“ (XOR) abbilden, welches sich durch ein einfaches Perzeptron nicht abbilden lässt. Durch das Einfügen von zwei versteckten Neuronen kann eine logische Schaltung nachgebaut werden, welche ein XOR mit einem negierten „Und“ (NAND), einem „Oder“ (OR), sowie einem normalen AND implementiert.

Ein solches Netz und die dazugehörige Ausgabe, welche der Funktion 2.1 entspricht, ist in Abb. 2.10a und 2.10b auf der nächsten Seite dargestellt.

$$\text{sigm}\Big(8 \cdot \big(\text{sigm}\left(-8 \cdot (a + b - 1.5)\right) + \text{sigm}\left(8 \cdot (a + b - 0.5)\right) - 1.5\big)\Big) \tag{2.1}$$

Die Gewichtungen des Netzes wurden, wie zuvor beim AND-Perzeptron, von Hand gewählt. Es lässt sich erkennen, dass zum Darstellen des NAND lediglich die Vorzeichen der

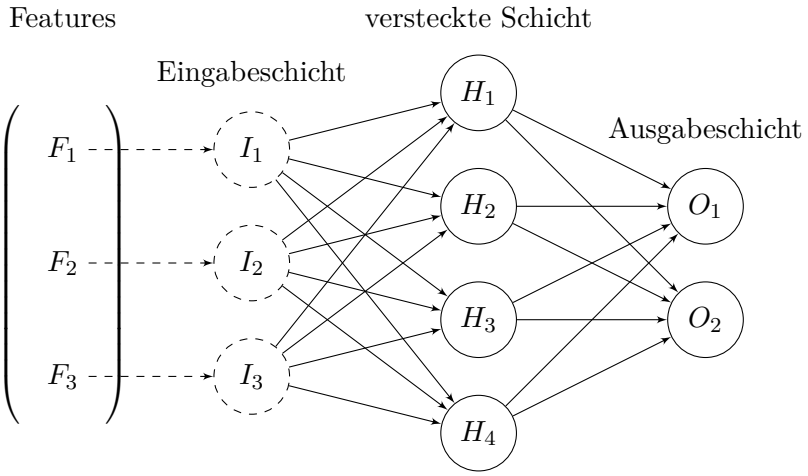
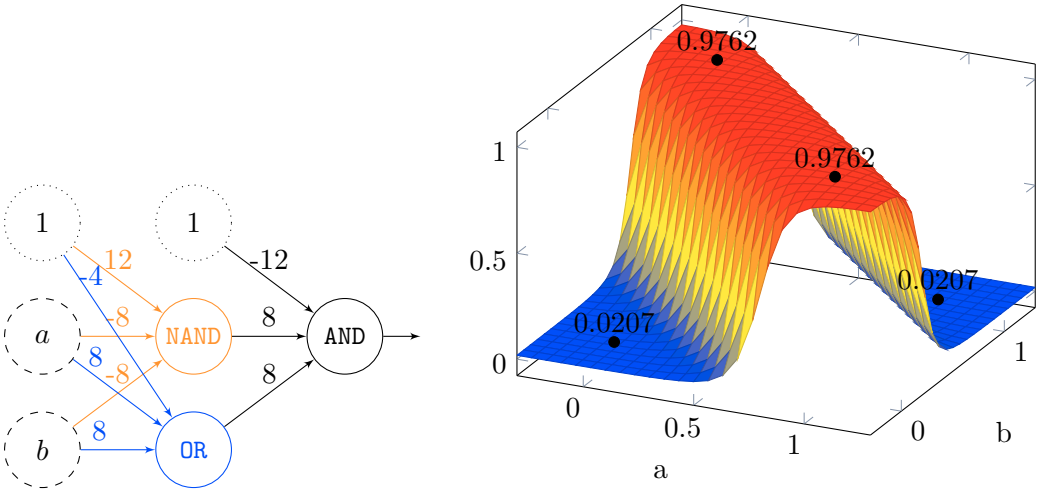


Abbildung 2.9: Schematische Darstellung eines künstlichen neuronalen Netzes mit drei Eingabe-, vier versteckten und zwei Ausgabe-Neuronen. Jede Verbindung zwischen zwei Neuronen ist mit einer Gewichtung ω versehen.



(a) MLP welches ein `XOR` mit `NAND`, `OR` und `AND` abbildet.

(b) Graph der Funktion 2.1, welche dem MLP aus a entspricht.

Abbildung 2.10: „exklusives Oder“ als mehrschichtiges Perzeptron

Gewichte und des Schwellwertes eines AND-Perzeptrons gedreht wurden. Das OR konnte durch Verschieben der Trennebene des AND in Richtung Koordinatensystem-Ursprung, also durch Anpassen des Schwellwertes, erzeugt werden. Auch hier ließ sich die Steigung an den Trennstellen durch einen Faktor steuern.

2.3.4 Klassifizierung mit KNNs

Da mit einem einzelnen Ausgabeneuron nur der Status „aktiviert“ und „deaktiviert“ unterschieden wird, kann ein einzelnes Ausgabeneuron nur für eine binäre Klassifizierung mit zwei Klassen verwendet werden. Existieren mehr als zwei Klassen müssen also mehrere Ausgabeneuronen verwendet werden, um in der Ausgabeschicht die einzelnen Klassen voneinander zu unterscheiden. Hierzu wird das sogenannte „One hot encoding“ verwendet.

Dies bedeutet, dass für jede Klasse des Klassifizierungsproblems ein eigenes Ausgabeneuron erzeugt wird, welches die Klasse repräsentiert. Das KNN wird dann so trainiert, dass für einen Datenpunkt nur das Ausgabeneuron aktiviert sein soll, welches die Klasse des Datenpunktes repräsentiert. Zur Auswertung wird dann das Ausgabeneuron mit der stärksten Aktivierung bestimmt.

2.3.5 Backpropagation Algorithmus

Für echte Datensätze lassen sich die Gewichtungen nicht von Hand bestimmen, sondern müssen von einem Algorithmus erarbeitet werden. Dies wird als „Training“ des MLP bezeichnet. Anschließend muss das resultierende MLP getestet werden, um sicherzustellen, dass es auch für Datenpunkte, die nicht Teil des Trainings waren, sinnvolle Ausgaben erzeugt. Der Algorithmus sollte also eine Lösung liefern, die auch gut auf ungesehene Daten generalisiert und nicht nur genau an die Trainingsdaten angepasst ist (engl. overfitting).

Einer der wichtigsten Algorithmen zum Training eines MLP ist der Backpropagation Algorithmus. Bei diesem handelt es sich um einen Algorithmus aus dem Bereich des überwachten Lernens (engl. supervised learning). Zum Anwenden eines supervised learning Algorithmus muss ein Datensatz zur Verfügung stehen, für den die gewünschte Ausgabe bereits bekannt ist. Solch ein Datensatz wird auch als „ground truth“ oder „labeled data“ bezeichnet.

Mit solch einem Datensatz kann schrittweise ein MLP mit dem Backpropagation Algorithmus trainiert werden. Das Training besteht aus mehreren Phasen, die wiederholt werden. Diese sind in Abb. 2.11 auf der nächsten Seite dargestellt und werden nun erläutert.

Zunächst wird der Datensatz in Batches aufgeteilt. Die Anzahl der Batches, beziehungsweise die Größe eines Batches, ist hierbei eine Stellgröße, welche die Geschwindigkeit und Qualität des Trainings sowohl positiv als auch negativ beeinflussen kann und abhängig vom Datensatz gewählt werden muss.

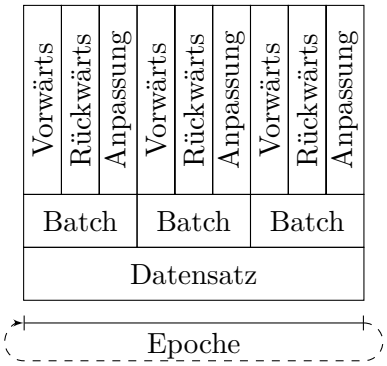


Abbildung 2.11: Ablauf des Training eines künstlichen neuronalen Netz. Der Datensatz wird in Batches unterteilt, mit denen dann der Backpropagation-Algorithmus durchgeführt wird. Dies wird über mehrere Epochen wiederholt.

Mit solch einem Batch, welcher aus einem oder mehreren Datenpunkten aus dem Datensatz besteht, wird dann zunächst eine „Forward Propagation“ durchgeführt, also die Ausgabe des Netzes für diese Datenpunkte im Batch bestimmt.

Anschließend folgt die „Backpropagation“. Hierbei wird, beginnend bei der Ausgabeschicht, die Abweichung der Ausgabe von der Gewünschten bestimmt. Anhand dieser lässt sich durch den Gradienten der Aktivierungsfunktion eine Gewichtsänderung und die gewünschte Ausgabe für die nächste Ebene bestimmen. Hierbei wird implizit der Gradient Descent Algorithmus angewendet [4]. Dann werden die berechneten Anpassungen vorgenommen.

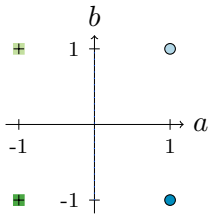
In einer Epoche wird dieser Prozess einmal mit allen Batches durchgeführt. Da sich die Gewichte immer nur schrittweise verbessern, besteht das Training aus mehreren Epochen. Diese werden solange wiederholt, bis eine gewünschte Fehlerschranke unterschritten oder eine maximale Anzahl an Epochen überschritten wird.

Genau wie der Gradient Descent Algorithmus kann auch der Backpropagation Algorithmus zu einem lokalen Minimum konvergieren. In Abb. 2.12 auf der nächsten Seite ist die Fehlerfunktion 2.2 eines Perzeptrons für verschiedene Gewichtungen dargestellt. Zusätzlich wird die Gewichtsänderung des Perzeptrons mit zwei Eingabeneuronen beim Backpropagation Algorithmus über 10 000 Epochen hinweg für verschiedene Startwerte abgebildet. Für manche Startwerte wird nur ein lokales aber nicht das globale Minimum gefunden, sodass das Perzeptron ein falsches Ergebnis für manche der Datenpunkte liefert.

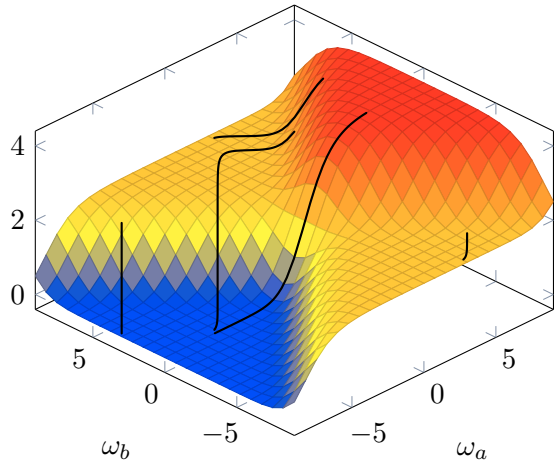
Die Fehlerfunktion 2.2 enthält alle Trainings Datenpunkte und soll vom Backpropagation Algorithmus durch Anpassen der Gewichte ω_a und ω_b minimiert werden.

$$\begin{aligned}(\text{sigm}(1 \cdot \omega_a + 1 \cdot \omega_b) - 0)^2 + (\text{sigm}(1 \cdot \omega_a - 1 \cdot \omega_b) - 0)^2 + \\ (\text{sigm}(-1 \cdot \omega_a + 1 \cdot \omega_b) - 1)^2 + (\text{sigm}(-1 \cdot \omega_a - 1 \cdot \omega_b) - 1)^2 \quad (2.2)\end{aligned}$$

a	b	Ziel
1	1	0
1	-1	0
-1	1	1
-1	-1	1



(a) Tabelle und Diagramm für das Beispiel: Das Perzeptron soll eine 0 für Datenpunkte rechts der Y-Achse und eine 1 für Datenpunkte links der Y-Achse erzeugen.



(b) Die Fehlerfunktion (2.2) und die Entwicklungen der Gewichte ω_a und ω_b beim Backpropagation-Algorithmus mit verschiedenen Startwerten über jeweils 10 000 Epochen.

Abbildung 2.12: Backpropagation-Beispiel mit einem Perzeptron mit zwei Eingabeneuronen und verschiedenen Startwerten. Der Algorithmus kann zu einem lokalen Minima/Plateau konvergieren.

2.4 Problemstellung

Es wurden mehrere Probleme identifiziert, für die im Rahmen dieser Arbeit Lösungen erarbeitet werden sollen.

- Eine serielle Implementierung eines Algorithmus ist nicht mehr ausreichend, um Datensätze wie aus dem Remote Sensing effizient verarbeiten zu können.
- Durch die Verwendung von größeren KNNs und Datensätzen nimmt die Ausführungszeit zu. Es kann nötig sein ein Problem schneller zu lösen.
- Es wird also ein paralleler und skalierbarer Algorithmus benötigt.
- Es wird eine frei verfügbare Implementierung benötigt.
- Die Klassifizierungsgenauigkeit eines Algorithmus muss beibehalten werden und darf nicht durch eine Parallelisierung eingeschränkt werden.

2.5 Zusammenfassung

Mit Remote Sensing existiert ein Verfahren zur Informationsgewinnung über die Erdoberfläche, bei dem die Erdabstrahlung von Flugzeugen oder Satelliten gemessen wird. Um die gewonnene Daten auswerten zu können müssen die einzelnen Pixel eines Remote Sensing Datensatzes klassifiziert werden. Hierzu wird Machine Learning verwendet, welches Modelle zur automatischen Klassifizierung bereitstellt. Diese Modelle werden anhand einer Vorgabe mit einem Algorithmus trainiert um das Modell dann auf unbekannte Datensätze anwenden zu können.

Um einen Algorithmus schneller oder auf größeren Datenmengen ausführen zu können kann Parallel Computing verwendet werden. Hierbei rechnen mehrere Prozesse, die auf einem Supercomputer ausgeführt werden, gemeinsam an der Lösung, sodass Teile des Algorithmus parallel ausgeführt werden und die Ausführung somit schneller beendet ist. Zusätzliche Grafikkarten und Beschleuniger-Prozessoren sind ein gutes Mittel um bestimmte Berechnungen noch stärker zu beschleunigen.

Mit künstlichen neuronalen Netzen wird versucht die Struktur im Gehirn von Lebewesen nachzubilden und so ein Modell zu erhalten, welches dann mit einem Algorithmus trainiert werden kann.

In dieser Arbeit soll eine parallele Implementierung eines künstlichen neuronalen Netzes entwickelt und angewendet werden.

3 Verwandte Arbeiten

Machine Learning Algorithmen werden häufig für die Auswertung von Daten im Remote Sensing eingesetzt. Auch mit künstlichen neuronalen Netzen konnten in diesem und anderen Anwendungsfeldern bereits gute Ergebnisse erzielt werden [14]. Es existieren jedoch kaum frei verfügbare parallele Implementierungen.

In Tabelle 3.1 auf der nächsten Seite sind einige Implementierungen von künstlichen neuronalen Netzen aufgelistet. Es existieren zwar sowohl freie als auch kommerzielle Implementierungen, nur wenige unterstützen jedoch eine Parallelisierung oder die Verwendung von GPUs.

Die vielversprechendsten Ansätze für eine freie Implementierung eines parallelen MLP sind Spark MLib und MXNet.

Die Spark MLib ist eine Bibliothek, welche Implementierungen für Machine Learning Algorithmen bereit stellt, welche auf Spark basieren. Spark ist eine Bibliothek für die Verarbeitung von großen Datenmengen auf Cluster Systemen. Für Spark MLib existiert keine Dokumentation, welche die Art der Parallelisierung des MLP beschreibt, aus dem Quellcode ergab sich jedoch, dass eine Parallelisierung durch duplizieren des MLP und einer parallelen Berechnung für verschiedene Datenpunkte implementiert wurde.

Bei Spark wird im Gegensatz zum „High Performance Computing“ (HPC) auf einen „High Throughput Computing“ (HTC)-Ansatz gesetzt. Das bedeutet, dass anstelle von einem hochperformanten und stark vernetzten Supercomputer eine lose verbundene Menge von zum Teil heterogenen Rechenknoten zum Einsatz kommt. Hierzu wird der MapReduce-Ansatz verwendet. Die Daten liegen geteilt vor und es wird auf die verteilten Daten eine Map-Funktion angewendet. Anschließend werden die von der Map-Funktion erzeugten Daten mit einer Reduktion zusammengefasst. Sowohl die Map- als auch die Reduce-Funktion wird hierbei parallel ausgeführt, aber nur für die Reduktion wird eine Kommunikation benötigt.

Von Haus aus bringt Spark keine Unterstützung von GPUs mit, aber es existieren Experimente um auch mit Spark GPU Ressourcen nutzen zu können. Diese sind jedoch noch nicht in den Hauptentwicklungszweig zurückgeflossen.

Mit MXNet soll eine flexible und effiziente Bibliothek für Deep Learning geschaffen werden, welche in verschiedensten Umgebungen genutzt werden kann. Es werden sowohl CPUs, als auch GPUs unterstützt. Des Weiteren soll die Bibliothek auf verschiedensten Rechenumgebungen wie Clustern, Servern und Desktop-PCs aber auch Smartphones genutzt werden können.

MXNet befindet sich in aktiver Entwicklung. Es können mehrere Prozesse gestartet werden, welche dann über normale Socket Verbindungen untereinander kommunizieren. MXNet besitzt zwar insofern eine MPI Unterstützung, dass mit MPI mehrere Prozesse gestartet werden können, die Kommunikation profitiert jedoch nicht von einem auf MPI

Name	Beschreibung	Parallelisierung
scikit-learn[15]	Freie Machine Learning Bibliothek für Python, welche auch ein neuronales Netz mit Stochastic Gradient Descent implementiert. Parallelisierung oder GPUs werden nicht unterstützt.	Nein
Spark MLib[16]	Machine Learning Bibliothek für Spark. Implementiert einen MLP-Klassifier. Unterstützt von Haus aus nur CPUs.	Ja
Apache Mahout[17]	Machine Learning Bibliothek für Hadoop, Spark, H2O und Flink. Implementiert einen MLP-Klassifier, jedoch ohne Parallelisierung.	Nein
Matlab[18]	Die „Neural Network Toolbox“ enthält eine MLP Implementierung. Mit der „Parallel Computing Toolbox“ kann eine Parallelisierung mit CPUs und Nvidia GPUs durchgeführt werden. Beide Toolboxen kosten jeweils bis zu 1000 €/Lizenz.	Ja
FANN[19]	Die (freie) Fast Artificial Neural Network-Bibliothek für C und C++. Keine Unterstützung für Parallelisierung oder GPUs.	Nein
MXNet[20]	Freie Bibliothek für Deep Learning, welche auch für MLPs verwendet werden kann. Parallel zu dieser Arbeit entwickelt. Unterstützt Parallelisierung mit mehreren CPUs und GPUs aber ohne MPI-Kommunikation. Schnittstellen für diverse Programmiersprachen wie Python und R.	Ja

Tabelle 3.1: Verschiedene Softwarebibliotheken, welche künstliche neuronale Netze implementieren, mit Schwerpunkt der Analyse auf Parallelisierungsansätzen.

optimierten Netzwerk und den MPI-Kommunikations-Routinen, da nur normale Socket Verbindungen genutzt werden.

3.1 Generelle Lösungsansätze einer Parallelisierung

Es kommen mehrere verschiedene Parallelisierungs-Strategien in Frage, die im Folgenden dargestellt werden.

3.1.1 Neuronen aufteilen

Ein häufig verwendeter Ansatz zum Parallelisieren des Backpropagation Algorithmus ist das Aufteilen der Neuronen auf die einzelnen Prozesse. Wenn sichergestellt ist, dass alle Prozesse eine ausreichend große Anzahl an Neuronen aus allen Schichten erhalten, können alle Prozesse die Aktivierungen und Gewichtsänderungen für ihre Neuronen gleichzeitig berechnen. Hierzu benötigt jeder Prozess jedoch die Aktivierungen aller Neuronen aus der vorherigen Schicht, welche zu großen Teilen nur auf anderen Prozessen bekannt sind. Es wird also Kommunikation benötigt. Dies ist in Abb. 3.1 dargestellt.

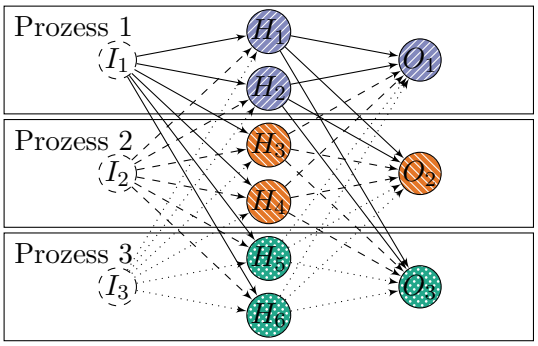


Abbildung 3.1: Aufteilung der einzelnen Neuronen auf verschiedene Prozesse. Da die Pfeile die Prozessorgrenzen kreuzen wird Kommunikation benötigt.

Dieser Parallelisierungsansatz eignet sich hervorragend für Parallelsysteme mit einem gemeinsamen Speicher, da ansonsten eine Kommunikation von jedem Prozess mit allen anderen durchgeführt werden muss um die benötigten Aktivierungen zu erhalten.

Dieser Ansatz wird zum Beispiel in [21] angewendet. Hier wird die serielle Berechnung der Skalarprodukte durch Aufrufe der parallelen Bibliothek PETSc ersetzt. Es konnte so jedoch, aufgrund der benötigten Kommunikation, kein Speedup erreicht werden.

In [22] wurde das Problem der massiven Kommunikation zwischen den Prozessen dadurch vermieden, dass die Architektur des Netzwerkes vereinfacht und eingeschränkt wurde. Anstatt jedes Neuron mit jedem anderen aus der vorherigen Schicht zu verbinden, wird das MLP in Streifen geteilt, welche nur sehr lose über die Randneuronen miteinander verbunden sind. Das MLP ist dann nur innerhalb dieser Streifen vollständig vernetzt. Dies wird in Abb. 3.2 auf der nächsten Seite dargestellt.

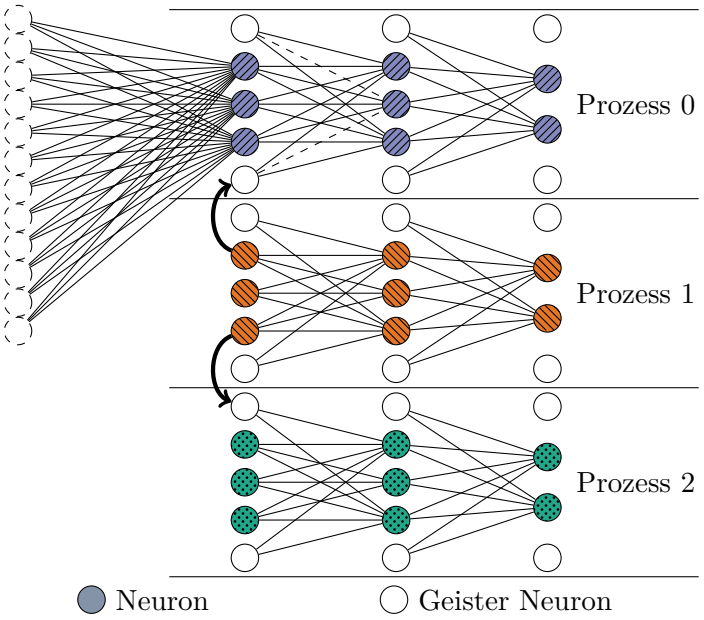


Abbildung 3.2: Parallelisierungsansatz aus [22, Abbildung 5]. Nicht jedes Neuron ist mit jedem anderem aus der vorherigen Schicht verbunden.

In der Abb. 3.2 aus [22] sind die Geisterneuronen der ersten Schicht nicht mit den mittleren Neuronen der zweiten Schicht eines Prozesses verbunden. Hierbei handelt es sich vermutlich um einen Fehler der Darstellung, denn die Geisterneuronen der ersten Schicht sollten mit allen Neuronen der nächsten Schicht verbunden sein, wie dies auch für die Verbindungen zwischen der zweiten und dritten Schicht abgebildet ist. Die fehlenden Verbindungen im Orginalbild sind für Prozess 0 mit gestrichelten Linien angedeutet.

Diese Art der Aufteilung bringt jedoch den Nachteil mit sich, dass die Anzahl der Prozesse die Netzstruktur massiv beeinflusst. Ein solches Netz kann immer nur mit der gleichen oder einer vielfachen der ursprünglichen Prozessanzahl effizient genutzt werden. Außerdem wird mit wachsender Anzahl an Prozessen die Konnektivität des Netzes immer weiter eingeschränkt, sodass die Konvergenz nicht mehr unbedingt sichergestellt ist. In dieser Arbeit werden dementsprechend nur vollständig vernetzte MLPs betrachtet.

Für die Verwendung von mehreren GPUs ist der Ansatz, die Neuronen einer Schicht auf mehrere Prozesse zu verteilen unbrauchbar, da während der Berechnung eines Datenpunktes mehrere Kommunikationen stattfinden und somit die Daten aus dem Speicher der GPU heraus kopiert werden müssten. Des Weiteren kann sich die gleichmäßige Aufteilung der Neuronen auf die Prozesse als schwierig herausstellen, wenn zum Beispiel nur wenige Ausgabeneuronen in der Ausgabeschicht vorhanden sind.

3.1.2 Duplizieren des Netzwerkes

Anstatt das Netzwerk auf mehrere Prozesse aufzuteilen, kann es auch auf allen Prozessen zur Verfügung stehen. Nur die benötigten Trainingsdaten werden in Blöcken auf die Prozesse aufgeteilt, sodass jeder Prozess das komplette Netz, aber nur einen Teil der Daten besitzt. Der Ansatz ist vergleichbar mit dem Cascade SVM Verfahren[23] für Support Vector Maschinen[24], einer anderen Klassifizierungsmethode aus dem Machine Learning.

Das hat den Vorteil, dass jeder Prozess mit seiner eigenen GPU eine komplette Forward- und Backward-Propagation durchführen kann, ohne dass die Berechnungen durch Kommunikation unterbrochen werden müssen. Anschließend müssen nur die Gewichtsänderungen zwischen den Prozessen synchronisiert werden.

Dieser Ansatz wurde auch in [25] gewählt und dort als „Pattern Parallel Training“ bezeichnet. Für die Implementierung von [25] wurde MPI und die Fast Artificial Neural Network (FANN)-Bibliothek[19] verwendet, welche eine einfache und schnelle Schnittstelle und Implementierung für die Verwendung eines KNN darstellt. Für diesen Parallelisierungsansatz musste FANN jedoch erweitert werden, um Zugriff auf die benötigten internen Datenstrukturen zu erhalten.

In [14] werden zwei verschiedene Ansätze zur Parallelisierung eines MLP erfolgreich für Remote Sensing eingesetzt. Es wird ein „Exemplarer“-Parlallelisierungsansatz, der dem „Pattern Parallel“ Ansatz aus [25] entspricht, mit einem „hybriden“ Ansatz verglichen.

Für den hybriden Ansatz werden lediglich die Neuronen einer einzelnen versteckten Schicht so auf die Prozesse aufgeteilt, dass jeder Prozess sowohl die Gewichtungen für die ankommenden, als auch die ausgehenden Verbindungen seiner eigenen versteckten Neuronen besitzt. Die Eingabe- und Ausgabeneuronen sind allerdings auf allen Prozessen vorhanden.

Abb. 3.3 auf der nächsten Seite stellt diese Parallelisierung dar. In dieser Abbildung aus [14] werden jedoch auch die Neuronen der versteckten Schicht als Stapel von vier Neuronen dargestellt. Dies ist insofern ein Fehler in der Darstellung, da diese Neuronen nur auf einem Prozess existieren und somit die Abbildung als Stapel nicht gerechtfertigt ist.

Diese Aufteilung ermöglicht es die Kommunikation zwischen den Prozessen zu reduzieren. So muss zwischen den Prozessen nur eine einzige Kommunikation für die Berechnung eines Datenpunktes stattfinden, um die gewichteten Summen der Ausgabeneuronen zu aggregieren. Dies lässt sich mit einer einzigen `MPI_Allreduce`-Operation erreichen.

Dieser Ansatz bringt jedoch das Problem mit sich, dass nur MLPs mit einer einzigen versteckten Schicht verwendet werden können. Es hat sich jedoch herausgestellt, dass für manche Anwendungsfälle mit dem sogenannten „Deep Learning“, bei dem KNNs mit sehr vielen Schichten verwendet werden, viel bessere Ergebnisse erreicht werden können. Dementsprechend soll in dieser Arbeit auf die Einschränkung, dass für die Parallelisierung nur eine einzelne versteckte Schicht verwendet werden kann, verzichtet werden.

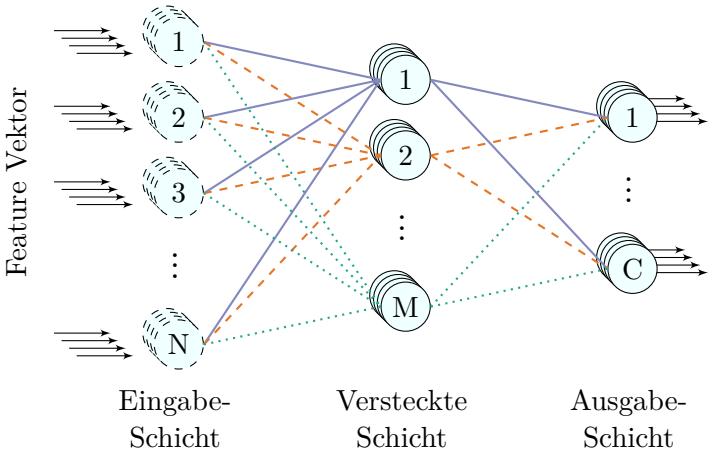


Abbildung 3.3: Hybride Partitionierung aus [14, Abbildung 8.4]. Die Eingabe und Ausgabeschichten sind auf allen Prozessen gemeinsam. Die versteckten Neuronen sind auf die Prozesse verteilt (durchgezogene, gestrichelte und gepunktete Linien bezeichnen die Gewichtungen, die zu den drei Prozessen gehören).

4 Design der parallelen Backpropagation

Durch ihre schnelle Speicheranbindung und die Möglichkeit viele Berechnungen auf einem gemeinsamen Speicherbereich parallel auszuführen sind GPUs sehr gut für die benötigten Berechnungen von MLPs geeignet. Mit GPUs allein kann häufig schon eine gravierende Beschleunigung im Vergleich zu einer CPU Implementierung erreicht werden. Deshalb sollen in dieser Arbeit GPUs eingesetzt werden.

Es existieren viele verschiedene Arten von GPUs, deshalb ist es ratsam eine Bibliothek zu verwenden, welche möglichst plattformunabhängiges Programmieren ermöglicht, jedoch nicht zu große Einschnitte bei der Geschwindigkeit mit sich bringt. Um mehrere GPUs gemeinsam verwenden zu können muss zusätzlich eine Datenverteilung und Parallelisierungsstrategie entwickelt werden.

Zunächst wird in Abschnitt 4.1 die Datenverteilung auf die GPUs erläutert. In Abschnitt 4.2 wird die Aufteilung der Daten in Batches beschrieben. Anschließend beschreibt Abschnitte 4.3 und 4.4 die Durchführung der Forward- und Backpropagation. Zusätzlich werden in Abschnitt 4.5 Erweiterungen des Backpropagation-Algorithmus vorgestellt. Abschnitt 4.6 beschreibt die Synchronisierung zwischen den Prozessen.

4.1 Datenverteilung

Aufgrund der Ergebnisse aus Kapitel 3 wurde eine Art des „Pattern Parallel Training“ aus [25] und dem „Exemplaren“ Ansatz aus [14] gewählt. Das bedeutet, dass jeder Prozess eine vollständige lokale Kopie des MLP, jedoch nur einen Teilbereich der Daten besitzt.

Die Trainingsdaten bestehen aus einer Matrix, welche die einzelnen Datenpunkte des Datensatzes in Spalten enthält und einem Vektor, welcher für jeden Datenpunkt in der Matrix die Klassenzugehörigkeit durch einen ganzzahligen Wert festlegt.

Für jede Schicht, außer der Eingabeschicht, muss eine $I \times N$ Gewichtsmatrix und ein Spaltenvektor mit N Zeilen abgelegt werden, wobei N der Anzahl Neuronen in dieser Schicht und I der Anzahl der Neuronen in der vorherigen Schicht entspricht. Die Gewichtungen in diesen Matrizen sollen schrittweise durch den Backpropagation-Algorithmus an die Trainingsdaten angepasst werden.

Um ein paralleles Einlesen der Daten zu ermöglichen, sollten die Trainingsdaten, sowie das MLP in einem Dateiformat abgelegt werden, welches dies unterstützt. Hierzu kann zum Beispiel das HDF5 Format[26] verwendet werden.

Alle Prozesse laden die komplette Beschreibung des MLP aus der Datei. Die Trainingsdaten werden möglichst gleichmäßig auf die Prozesse aufgeteilt, sodass jeder Prozess nur einen Teilbereich der Daten einlesen muss. Dies kann sowohl durch ein round-robin, aber

auch eine blockweise Aufteilung der Daten geschehen. Bei einer blockweisen Aufteilung sollten die Daten nicht sortiert vorliegen, da sonst eventuell ein Prozess nur Trainingsdaten einer Klasse besitzt, was einen Einfluss auf das Training haben kann.

Desto mehr Prozesse vorhanden sind, desto weniger Daten muss ein einzelner Prozess im Speicher halten. Bei größeren Datenmengen kann eine parallele Datenaufteilung erforderlich sein, da die Daten zu groß für den Speicher eines einzelnen Rechners sind. Die Aufteilung ist in Abb. 4.1 dargestellt.

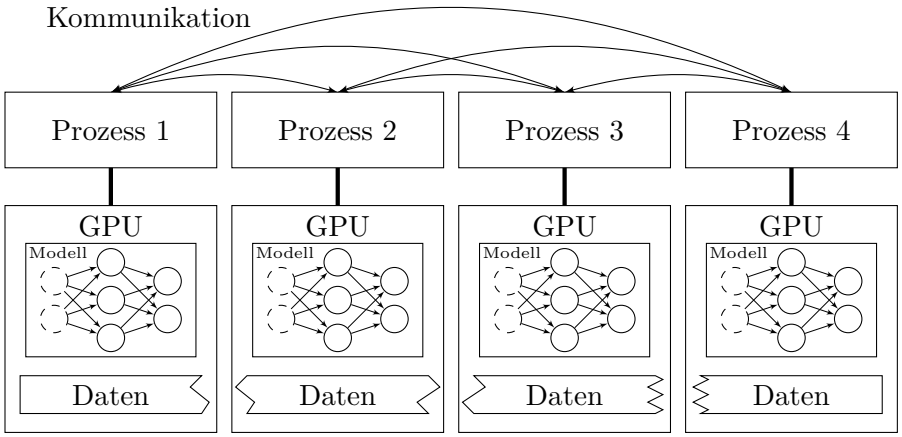


Abbildung 4.1: Die Datenverteilung. Jeder Prozess ist für eine GPU verantwortlich und besitzt eine vollständige Kopie des MLP sowie einen Teil der Trainingsdaten.

4.2 Auswählen der Daten für einen Batch

Um nicht ständig Steuersignale an die GPU zu senden, können mehrere Datenpunkte zu einem Batch zusammengefasst werden. Diese Daten werden dann in einer gemeinsamen Forward- und Backpropagation verwendet. Wie viele Datenpunkte in einem Batch gemeinsam propagiert werden sollen muss vorgegeben werden.

Hierbei ist es möglich die Daten sequentiell, anhand ihrer Reihenfolge im Datensatz, zu Batches zusammenzufassen.

Um ein besseres Ergebnis des KNNs zu erhalten ist es manchmal ratsam die einzelnen Datenpunkte in einer zufälligen Reihenfolge zu trainieren, um sicherzustellen, dass nicht die Reihenfolge der Datenpunkte und die Zusammensetzung der Batches einen Einfluss auf das Training hat.

Werden alle vorhandenen Datenpunkte in einem einzelnen Batch berechnet wird dies als „Batch Gradient Descent“ bezeichnet. Verwendet man lediglich einen einzelnen Datenpunkt ist dieses das „Stochastic Gradient Descent“-Verfahren. Beliebige Zwischengrößen werden als „Minibatch“ bezeichnet.

4.3 Forward Propagation

Für die Forward Propagation eines Datenpunktes kann eine Matrix-Vektor-Multiplikation der Gewichtsmatrix einer Schicht mit dem Datenvektor durchgeführt werden. Sollen mehrere Datenpunkte in einem Batch verarbeitet werden, können diese zu einer Matrix zusammengefasst und mit einer Matrix-Matrix Multiplikation gemeinsam berechnet werden.

Für eine Schicht, welche aus N Neuronen besteht und I Aktivierungen aus der vorherigen Schicht erhält, berechnet sich die Aktivierung x_n für das n-te Neuron mit den Gewichtungen $\omega_{\cdot,n}$ und Schwellwert θ_n als

$$x_n = \text{sigm}\left(\sum_{i=1}^{I} \omega_{i,n} \cdot v_i + \theta_n\right) = \text{sigm}\left(\langle \vec{\boldsymbol{\omega_n}}, \vec{\boldsymbol{v}} \rangle + \theta_n\right) = \text{sigm}\left(\vec{\boldsymbol{\omega_n}}^{\intercal} * \vec{\boldsymbol{v}} + \theta_n\right).$$

$\vec{\boldsymbol{\omega_n}}$ ist der Spaltenvektor, welche alle Gewichtungen der ankommenden Verbindungen für das n-te Neuron der aktuellen Schicht enthält. Der Spaltenvektor $\vec{\boldsymbol{v}}$, beziehungsweise v_1 bis v_I, enthalten die Aktivierungen der vorherigen Schicht.

Der $*$ Operator wird hier verwendet um ein Matrix-Matrix-Produkt zwischen zwei Vektoren oder Matrizen zu kennzeichnen, während $\langle \cdot, \cdot \rangle$ ein Skalarprodukt und $\cdot$ eine Multiplikation mit einem Skalar darstellt.

Die Gewichtungen eines Neuron werden in Spalten der Matrix Ω abgelegt. GPU Bibliotheken verwenden eine „column-major“ Speicherordnung. So liegen die Spalten der Matrix und somit die Gewichtungen eines Neuron direkt hintereinander im Speicher. Das bedeutet, dass die Gewichtungen beim Schreiben in eine Datei nicht neu geordnet werden müssen.

Fasst man alle $\vec{\boldsymbol{\omega_n}}$ zu Spalten einer Matrix Ω zusammen, erhält man eine $I \times N$ Matrix und kann alle gewichteten Summen einer Schicht in einem einzelnen Schritt mit $\Omega^{\intercal} * \vec{\boldsymbol{v}}$ berechnen.

Durch Verwendung der Gewichtsmatrix Ω und des Schwellwertvektor $\vec{\boldsymbol{\theta}}$ ergeben sich alle Aktivierungen der Neuronen einer Schicht als

$$\vec{\boldsymbol{x}} = \text{sigm}\left(\Omega^{\intercal} * \vec{\boldsymbol{v}} + \vec{\boldsymbol{\theta}}\right).$$

Die Aktivierungsfunktion sigm wird hierbei nicht auf einen Skalar, sondern auf jedes Element des Spaltenvektors, der sich aus $\Omega^{\intercal} * \vec{\boldsymbol{v}} + \vec{\boldsymbol{\theta}}$ ergibt, einzeln angewendet.

Wird anstelle des Spaltenvektors $\vec{\boldsymbol{v}}$ eine Matrix verwendet, welche in mehreren Spalten die verschiedene Aktivierungen der vorherigen Schicht eines Batches enthält, ergibt sich auch als Ergebnis der $\Omega^{\intercal}$-Matrix-Multiplikation eine Matrix, welche in jeder Spalte die Aktivierungen der Neuronen für die entsprechende Spalte in der Eingabe-Matrix enthält. Für die weitere Berechnung muss dann der Schwellwert-Spaltenvektor $\vec{\boldsymbol{\theta}}$ entsprechend auf alle Spalten der Ergebnismatrix von $\Omega^{\intercal} * M$ addiert werden. Da eine Matrix-Vektor-Addition nicht sinnvoll definiert ist muss aus dem Vektor zunächst eine Matrix erstellt werden, welche den Schwellwert-Spaltenvektor in jeder Spalte enthält.

4.4 Backpropagation

Die Durchführung des Backpropagation Algorithmus entspricht einem Gradient Descent, welcher die Fehlerfunktion $E = \frac{1}{2}\sum_{i=1}^{N}(t_i - o_i)^2$ minimiert. t_i ist hierbei die gewünschte Aktivierung in der Ausgabeschicht und o_i die Aktivierung, welche von dem MLP mit den aktuellen Gewichten erzeugt wird. Mit einer „Learning Rate“ η kann die Schrittweite des Gradient Descent Algorithmus gesteuert werden.

Für den Gradient Descent Algorithmus muss die Fehlerfunktion differenzierbar sein, was auf die Differenzierbarkeit der Aktivierungsfunktion zurückgeführt werden kann. Es können verschiedene Aktivierungsfunktionen verwendet werden.

Die Sigmoid Aktivierungsfunktion hat den Vorteil, dass sich die Ableitungen sehr einfach berechnen lassen:

$$\begin{aligned}
\operatorname{sigm}(t) &= \frac{1}{e^{-t}+1} \cdot \frac{e^t}{e^t} = \frac{e^t}{1+e^t} \\
\operatorname{sigm}'(t) &= \frac{e^t \cdot (1+e^t) - e^t \cdot e^t}{(1+e^t)^2} \\
&= \frac{e^t}{1+e^t} \cdot \frac{(1+e^t) - e^t}{1+e^t} \\
&= \frac{e^t}{1+e^t} \cdot \left(1 - \frac{e^t}{1+e^t}\right) \\
&= \operatorname{sigm}(t) \cdot (1 - \operatorname{sigm}(t))
\end{aligned}$$

Somit kann das Aktivierungsergebnis aus der Forward Propagation in der Backpropagation wiederverwendet werden, um den Wert der Ableitung zu bestimmen. Dies funktioniert auch für die tanh-Aktivierungsfunktion mit $\tanh'(t) = 1 - \tanh^2(t)$.

Für ein Neuron n in der Ausgabeschicht, berechnen sich die Gewichtsänderungen $\Delta\omega_{\cdot,n}$, wie in Formel 4.1 dargestellt, aus den Aktivierungen der vorherigen Schicht $\vec{\boldsymbol{v}}$, sowie der Differenz $t_n - o_n$ zwischen der gewünschten Ausgabeaktivierung dieses Neurons t_n und o_n, der tatsächlichen Aktivierung welche von dem Neuron mit den aktuellen Gewichten erzeugt wird.

$$\Delta\omega_{i,n} = \eta \cdot \delta_n \cdot v_i \qquad \delta_n = \operatorname{sigm}'\left(\langle \vec{\boldsymbol{\omega}}_{\boldsymbol{n}}, \vec{\boldsymbol{v}} \rangle + \theta_n\right) \cdot \overbrace{(t_n - o_n)}^{e_n} \tag{4.1}$$

Für ein Neuron in einer versteckten Schicht ist die Berechnung ein wenig komplizierter, da für diese Neuronen die Abweichung von der gewünschten Aktivierung nicht direkt berechnet werden kann, sondern aus den $\hat{\delta}_i$'s, sowie Gewichtungen $\omega_{n,j}$ von den Neuronen der in Vorwärtsrichtung nachfolgenden Schicht berechnet werden müssen:

$$\Delta\omega_{i,n} = \eta \cdot \delta_n \cdot v_i \qquad \delta_n = \operatorname{sigm}'\left(\langle \vec{\boldsymbol{\omega}}_{\boldsymbol{n}}, \vec{\boldsymbol{v}} \rangle + \theta_n\right) \cdot \overbrace{\sum_{j=1}^{M} \hat{\delta}_j \cdot \omega_{n,j}}^{e_n}$$

$$\Delta\theta_n = \eta \cdot \delta_n$$

Es werden also die δ_i Werte aus der vorherigen Backpropagation, sowie Gewichte aus einer anderen Schicht benötigt. Durch diese Zurückpropagierung des Fehlers erhält der Algorithmus seinen Namen. Um die Berechnung unabhängig davon zu machen, ob eine Schicht eine versteckte oder die Ausgabeschicht ist, kann die Abweichung durch e_n ersetzt werden, welches sich dann aus der Abweichung $t_n - o_n$ oder der vorherigen Backpropagation berechnet.

Auch die Backpropagation lässt sich mit Matrixmultiplikationen für alle Neuronen einer Schicht und mehrere Datenpunkte gleichzeitig durchführen.

$$\Delta\Omega = \eta \cdot \vec{\boldsymbol{v}} * \vec{\boldsymbol{\delta}}^{\mathsf{T}} \qquad \vec{\boldsymbol{\delta}} = \vec{\boldsymbol{e}} \odot \operatorname{sigm}'\left(\Omega^{\mathsf{T}} * \vec{\boldsymbol{v}} + \vec{\boldsymbol{\theta}}\right)$$
$$\Delta\vec{\boldsymbol{\theta}} = \eta \cdot \vec{\boldsymbol{\delta}}$$

Der $\odot$-Operator bezeichnet hier eine elementweise skalare Multiplikation zweier Vektoren. Durch die Multiplikation des Spaltenvektors $\vec{\boldsymbol{v}}$, welcher I Zeilen besitzt, und des Zeilenvektors $\vec{\boldsymbol{\delta}}^{\mathsf{T}}$ mit N Spalten ist $\Delta\Omega$ eine $I \times N$ Matrix und entspricht somit den Dimensionen von Ω.

Für die Berechnung der Ausgabeschicht wird $\vec{\boldsymbol{e}}$ auf $\vec{\boldsymbol{t}} - \vec{\boldsymbol{o}}$ gesetzt und dann von jeder Schicht mit $\vec{\boldsymbol{e}} = \Omega * \vec{\boldsymbol{\delta}}$ neu berechnet.

Wurde anstelle des Vektors $\vec{\boldsymbol{v}}$ eine Matrix M bestehend aus mehreren Datenpunkten eines Batches propagiert, dann sind auch $\vec{\boldsymbol{\delta}}$ und $\vec{\boldsymbol{e}}$ Matrizen. Durch das Matrix-Matrix-Produkt werden die einzelnen Gewichtsänderungen aller Datenpunkte implizit aufaddiert.

$$\underbrace{\left(\begin{array}{c:c:c} \vec{\boldsymbol{v}}_1 & \cdots & \vec{\boldsymbol{v}}_b \end{array}\right)}_{M} * \underbrace{\left(\begin{array}{c} \vec{\boldsymbol{\delta}}_1^{\mathsf{T}} \\ \hdashline \vdots \\ \hdashline \vec{\boldsymbol{\delta}}_b^{\mathsf{T}} \end{array}\right)}_{\text{Transponierte Matrix aus } \vec{\boldsymbol{\delta_1}},\ldots,\vec{\boldsymbol{\delta_b}}} = \vec{\boldsymbol{v}}_1 * \vec{\boldsymbol{\delta}}_1^{\mathsf{T}} + \cdots + \vec{\boldsymbol{v}}_b * \vec{\boldsymbol{\delta}}_b^{\mathsf{T}}$$

Lediglich der Schwellwertänderungs-Vektor $\Delta\vec{\boldsymbol{\theta}}$ muss gesondert durch Zeilensummen der Matrix $\eta \cdot \vec{\boldsymbol{\delta}}$ bestimmt werden.

4.5 Backpropagation Erweiterungen

Um die Ergebnisse des Backpropagation Algorithmus zu verbessern existieren mehrere Strategien, von denen zwei nun im Folgenden vorgestellt werden.

4.5.1 Weight Decay

Um eine bessere Generalisierung zu erreichen wird ein Strafterm eingeführt, welcher große Gewichtungen bestraft und reduziert. Dies führt dazu, dass ein „overfitting“ des MLP an die Daten vermieden wird. [27]

Hierzu muss lediglich die Gewichtsänderung um einen zusätzlichen Term erweitert werden.

$$\Delta\omega_{i,n} = \eta \cdot \delta_n \cdot v_i + \lambda \cdot \omega_{i,n}$$

Somit wird ein zusätzlicher Parameter λ eingeführt, welcher die Stärke des „Weight Decay“ steuert. Dieser Parameter muss in Abhängigkeit von der Learningrate und der Batchgröße gewählt werden.

4.5.2 Momentum

Eine weitere Erweiterung des Backpropagation Algorithmus, welcher eine schnellere Konvergenz, eine Verringerung von Oszilation und die Vermeidung von lokalen Minimas verspricht ist das sogenannte „Momentum“. Diese Verbesserung wird so auch für den Gradient Descent Algorithmus verwendet.

Anstatt nur die Richtung des Gradienten an der aktuellen Stelle in Betracht zu ziehen, wird zusätzlich auch die vorherige Gewichtsänderung zu einem Anteil auf die Richtung des nächsten Iterationsschrittes addiert. Bildlich wird dies als ein Ball auf der Fehleroberfläche beschrieben, welcher eine gewisse Beschleunigung besitzt und sich nicht nur anhand der Krümmung der Oberfläche bewegt, sondern auch seine aktuelle Bewegungsrichtung aufgrund von vorhandener Beschleunigung beibehält.

Sind die Richtungen von zwei aufeinander folgenden Iterationen entgegengesetzt führt dies zu einer impliziten Reduzierung der Schrittweite. Wenn jedoch die beiden Richtungen identisch sind, wird die Schrittweite implizit vergrößert. Sowohl die Learningrate, als auch die Batchgröße haben einen Einfluss auf die Wahl des Momentum-Parameters. Ein Gradient Descent mit und ohne Momentum ist in Abb. 4.2 für ein Beispiel abgebildet.

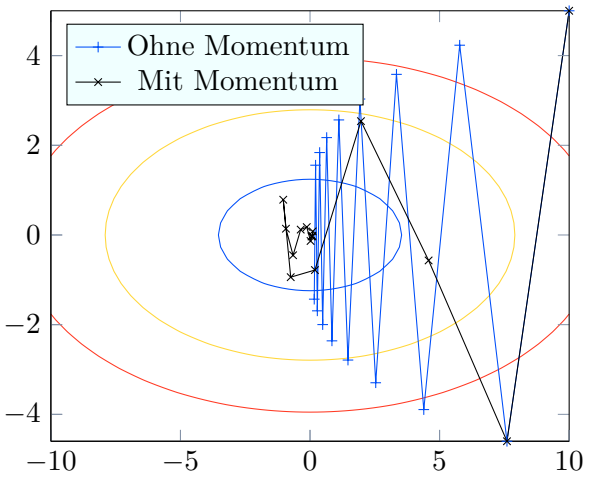


Abbildung 4.2: Jeweils 15 Gradient Descent Iterationen mit und ohne Momentum zum Finden des Minimums der Funktion $z = 8 \cdot x^2 + 56 \cdot y^2$. Es wurde eine Learningrate von 0,015 und ein Momentum von 0,5 verwendet. Durch das Momentum wurde das Minimum mit deutlich weniger Iterationen erreicht.

4.6 Synchronisierung

Für die Synchronisierung der duplizierten MLPs kommen verschiedene Strategien in Frage, welche nun dargestellt werden.

Da mehrere GPUs verwendet werden sollen, welche keinen gemeinsamen Speicherbereich besitzen bietet sich eine Synchronisierung mit MPI an. Diese sollte generell so selten wie möglich durchgeführt werden, da die Parallelisierung ineffizienter wird, desto häufiger Kommunikation stattfinden muss. Für die Verwendung von GPUs ist dies besonders wichtig, da ein zusätzlicher Overhead durch die erhöhte Zugriffszeit auf die Daten auf der GPU entstehen kann. Dieser Overhead kann bei der Verwendung von einer Nvidia GPU mit CUDA und einer CUDA-Aware MPI Implementierung reduziert werden.

Ein Beispiel für sehr ineffiziente Synchronisierung ist in Abb. 4.3 dargestellt.

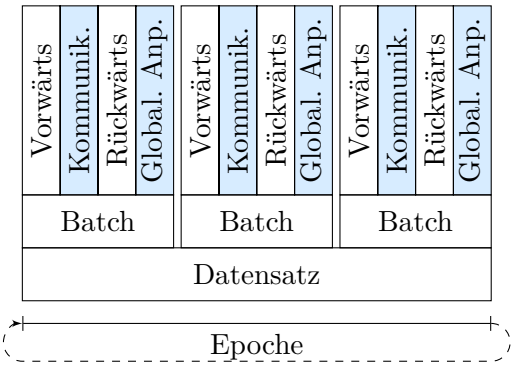


Abbildung 4.3: Kommunikation nach jeder Vorwärts und Rückwärtspropagierung. Dies sollte vermieden werden, da sonst die Berechnung immer wieder von Kommunikation unterbrochen wird.

4.6.1 Synchronisierung nach einem Batch

Die Gewichtsänderungen für einen Batch werden zunächst nur berechnet und noch nicht angewandt, da die Änderungen vorher zwischen allen Prozessen synchronisiert werden sollen. Dies ist in Abb. 4.4 auf der nächsten Seite dargestellt.

Um die Gewichtsänderungen zwischen allen Prozessen zu synchronisieren müssen die $\Delta\Omega$s und $\Delta\vec{\boldsymbol{\theta}}$s aller Prozesse für jede Schicht des MLP aufaddiert werden. Anschließend soll das Ergebnis auf allen Prozessen zur Verfügung stehen. MPI stellt für diesen Anwendungsfall die `MPI_Allreduce`-Funktion zu Verfügung, welche es erlaubt die Daten aller Prozesse mit einem Reduktions-Operator zu verbinden und die verarbeiteten Daten anschließend auf allen Prozessen zur Verfügung zu haben.

Somit berechnen alle Prozesse gemeinsam einen einzelnen Batch, der aus mehreren Datenpunkten besteht und erhalten das gleiche Ergebnis, welches auch ein einzelner Prozess berechnen würde, wenn er die gleichen Datenpunkte zu einem Batch zusammenfasst.

Durch die Aufteilung der Datenpunkte auf die Prozesse kann es zwar trotzdem dazu kommen, dass ein einzelner Prozess faktisch andere Daten zu einem Batch zusammen-

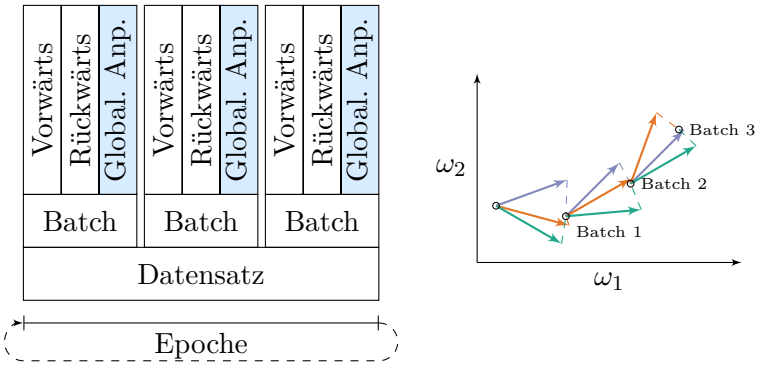


Abbildung 4.4: Synchronisierung nach Batch. Nach jedem Batch wird die Anpassung gemeinsam von allen Prozessen durchgeführt.

fasst, als mehrere Prozesse dies tun, jedoch ist trotzdem sichergestellt, dass die Parallelisierungsstrategie keinen Einfluss auf die Lernfähigkeit des MLP hat, denn die exakt gleichen Berechnungen könnten mit der richtigen Datenreihenfolge auch mit einem einzelnen Prozess so stattfinden.

4.6.2 Synchronisierung nach einer Epoche

Eine Alternative zur Synchronisierung nach jedem Batch ist die Synchronisierung nach jedem Abschließen aller Batches eines Datensatzes, also einer Epoche. Alle Prozesse berechnen die Gewichts- und Schwellwertänderungen für alle ihre Batches der lokal vorhandenen Daten in einer Epoche und wenden diese auf ihre lokale Kopie des MLP an. Somit sind nach der Backpropagation des ersten Batch bis zum Ende der Epoche alle Prozesse auf einem unterschiedlichen Stand und trainieren ihre lokalen MLPs mit unterschiedlichen Datenpunkten. Dies ist in Abb. 4.5 dargestellt.

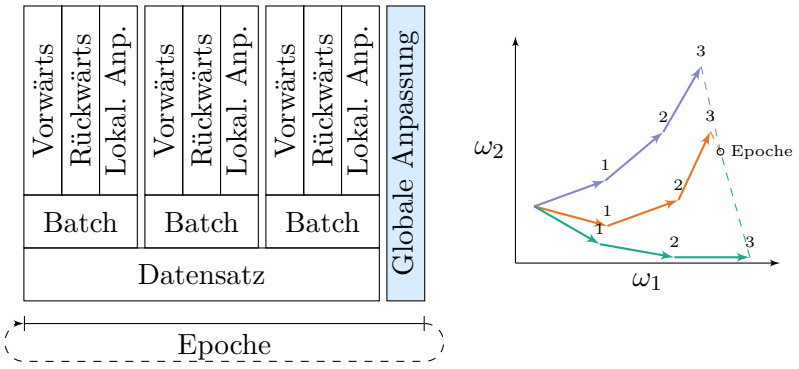


Abbildung 4.5: Synchronisierung nach Epoche. Nur am Ende jeder Epoche findet eine Synchronisierung statt. Jeder Prozess wendet die Änderungen aus den Batches vorher lokal an.

Um die unterschiedlichen MLPs der Prozesse wieder zusammenzuführen und ein gemeinsames Netz zu erhalten, berechnen alle Prozesse gemeinsam die Durchschnitte aller Gewichtungen und verwenden diese als neue Anfangsgewichtungen für die nächste Epoche.

Dies hat den Vorteil, dass nur eine Kommunikation für jede Epoche stattfinden muss und sich so auch kleinere lokale Batches verwenden lassen. Jedoch kann diese Wahl der Parallelisierung einen Einfluss auf die Konvergenz des Algorithmus haben. Die Anzahl der verwendeten Prozesse hat so einen Einfluss auf das Ergebnis des Algorithmus.

4.7 Zusammenfassung

GPUs sind durch ihren hohen Grad an Parallelisierung auf einem gemeinsamen Speicher sehr gut geeignet für die Berechnung von MLPs. Für eine skalierbare Parallelisierung, welche GPUs verwendet, darf die Berechnung nicht durch wiederholte Kommunikationen gestört werden, da das Kopieren der Daten aus dem Speicher der GPU die Kommunikation zusätzlich ineffizient macht.

Durch geschickte Matrix-Matrix Multiplikationen lassen sich mehrere Datenpunkte gemeinsam als ein „Batch“ propagieren. So muss die GPU nicht für jeden einzelnen Datenpunkt angesteuert werden und die Berechnung kann effizienter durchgeführt werden, da die Kommunikationszeit im Verhältnis zur Rechenzeit der GPU reduziert wird.

Das Duplizieren des Mehrschichtigen Perzeptrons auf mehrere GPUs ermöglicht es, dass die Daten für eine vollständige Forward- und Backpropagation auf der GPU verweilen und die Berechnung nicht durch Kommunikation unterbrochen werden muss.

Für die Parallelisierung kommen zwei verschiedene Arten der Synchronisierung in Frage:

- Durch das Synchronisieren nach jedem Batch verhält sich der parallelisierte Algorithmus wie eine serielle Implementierung.
- Bei der Synchronisierung nach jeder Epoche findet weniger Kommunikation statt, die Parallelisierung hat jedoch einen Einfluss auf das Ergebnis des Algorithmus.

5 Implementierung

Die ArrayFire[28] Bibliothek stellt eine einfache Schnittstelle bereit, die es ermöglicht Programme zu schreiben, welche ohne Änderungen am Programmcode vorzunehmen sowohl auf der CPU, als auch auf NVIDIA GPUs mit CUDA oder AMD GPUs und Beschleunigerkarten mit OpenCL[12] ausgeführt werden können. Dies ermöglicht es, das Programm plattformunabhängig einzusetzen.

Die Eigenheiten der GPU Programmierung für die verschiedenen Backends – CPU, CUDA oder OpenCL – werden so durch die Bibliothek erledigt. Es stehen Implementierungen von ArrayFire für C, C++, Python und Rust zur Verfügung. Für diese Arbeit wurde die C++ Schnittstelle verwendet.

Der Kern der ArrayFire C++ Schnittstelle ist die `array`-Klasse, welche eine Matrix darstellt, die auf einem der drei Backends abgelegt wurde. Mit einem Objekt der Klasse können dann viele verschiedene Operationen und Funktionen wie Addition, Subtraktion, elementweise Multiplikation und insbesondere Matrix Multiplikationen durchgeführt werden. Die Berechnungen werden dann automatisch auf dem gewünschten Backend ausgeführt, was zu einer großen Beschleunigung im Vergleich zu reinem CPU Code führen kann.

ArrayFire verwendet hierfür einen Syntax-Baum mit einem Just in Time(JIT)-Compiler, welcher die benötigten GPU Kernel erzeugt und mehrere Operatoren zu einem einzigen Kernel zusammenfassen kann um die Effizienz zu erhöhen [29]. Außerdem besitzt ArrayFire ein automatisches Speichermanagement, welches nicht mehr benötigten Speicherbereich auf der GPU automatisch wieder frei gibt.

In diesem Kapitel werden nun wesentliche Implementierungsdetails genauer dargestellt.

5.1 Erzeugen und Durchmischen der Batches

Wie in Abschnitt 4.2 auf Seite 26 dargestellt, soll nicht jeder Datenpunkt einzeln, sondern mehrere Datenpunkte als ein gemeinsamer Batch auf der GPU verarbeitet werden. Um die einzelnen Batches, welche von dem Algorithmus trainiert werden sollen, auf der GPU zu unterteilen wird die Möglichkeit von ArrayFire verwendet einen Unterabschnitt eines `af::array`-Objektes als eigenes Objekt zu erhalten. Hierbei wird der Trainingsdatensatz als große Matrix verwendet, aus der eine Untermatrix extrahiert wird, welche den Batch darstellt.

Mit `array_object(af::span, af::seq(startindex, lastindex))` lassen sich die Spalten von `startindex` bis `lastindex` aus dem `array_object` als eigenes `af::array` ansprechen, welches dann für alle weiteren Berechnungen verwendet werden kann. Die

`af::span`-Konstante ist hierbei ein besonderer Wert, welcher angibt, dass eine komplette Dimension des `af::array`-Objektes, hier also alle Zeilen, verwendet werden sollen. Die `af::seq`-Funktion erzeugt eine Sequenz beginnend bei dem angegebenen Startwert bis zum angegebenen Endwert (inklusive).

Um die Datenpunkte in einer zufälligen Reihenfolge zu Trainieren musste ein Trick angewendet werden, da ArrayFire noch keine vorgefertigte Funktion zum Erzeugen einer zufälligen Liste von Zahlen im Bereich 0 bis N besitzt: Die `af::sort`-Funktion erlaubt es den Inhalt eines `af::array`-Objekt zu sortieren und bietet zusätzlich die Möglichkeit eine Liste der sortierten Indizes zu erhalten. [30]

Um also eine zufällige Reihenfolge von Indizes von 0 bis N zu erhalten wird ein mit $N + 1$ zufälligen Zahlen gefülltes `af::array`-Objekt sortiert. Die von der `af::sort`-Funktion erzeugten Indizes von 0 bis N, welche die sortierte Reihenfolge der zufälligen Daten darstellen, sind dann in einer zufälligen Reihenfolge. Listing 5.1 stellt den Code dar, der dies implementiert.

```
1  af::array shuffled_idx, sorted_randomizer;
2  //Vor jeder Epoche:
3  //af::randu(N) erzeugt einen Vektor mit N gleichverteilten Zufallszahlen.
4  //Diese werden mit af::sort sortiert. sorted_randomizer enthält die sortierten
5  //Daten und shuffled_idx die durchmischten Indizes.
6  af::sort(sorted_randomizer, shuffled_idx, af::randu(N));
7  //Für jeden Batch:
8  //Hole die Indizes für einen Batch aus shuffled_idx
9  af::array batchidx = shuffled_idx(af::seq(startindex, lastindex));
10 //Und mit diesem Batch-Index-Array wird dann das Batch-af::array erzeugt.
11 af::array batchsamples = data_array(af::span, batchidx);
```

Listing 5.1: Durchmischen der Datenpunkte im Datensatz mit ArrayFire.

5.2 Forward Propagation

In Abschnitt 4.3 auf Seite 27 wurde erläutert, wie ein Batch aus mehreren Datenpunkten mit einer einzelnen Matrix-Matrix-Multiplikation vorwärts propagiert werden kann. Mit ArrayFire lässt sich diese Forward Propagation wie in Listing 5.2 auf der nächsten Seite dargestellt implementieren: Mit der ArrayFire Funktion `af::matmulTN` wird eine Matrix-Multiplikation durchgeführt, bei welcher der linke Operand implizit transponiert wird. Diese implizite Transponierung ermöglicht es die Berechnung auf die Orientierung der Daten im Speicher hin zu optimieren.

Die `af::tile`-Funktion erzeugt aus dem Schwellwert-Spaltenvektor eine Matrix, sodass der Schwellwert auf alle Spalten der Matrix mit den gewichteten Summen gleichermaßen aufaddiert werden kann. Zuletzt wird die Aktivierungsfunktion der Neuronen auf alle Elemente dieser Matrix angewandt.

So kann sowohl für einzelne als auch mehrere, als Matrix gebündelte, Datenpunkte die Forward Propagation einer Schicht berechnet werden. Für die weiteren Schichten wird anstelle des Datenpunktes die Ausgabe der vorherigen Schicht verwendet und mit

anderen Gewichten und Schwellwerten die gleiche Berechnung durchgeführt, bis die Ausgabeschicht erreicht wird.

```
1 // matmul(transpose(IxN), (Ixb))  =
2 // matmul(         (NxI), (Ixb))  = (Nxb)
3 af::array sumOfWeightedInputs = af::matmulTN(weights, input);
4 // Nxb += tile(Nx1, 1, b) = Nxb
5 sumOfWeightedInputs += af::tile(bias, 1, sumOfWeightedInputs.dims(1));
6 return activation(sumOfWeightedInputs);
```

Listing 5.2: Forward Propagation einer Schicht mit den Gewichtungen `weights` und den Schwellwerten `bias`. `input` enthält b verschiedene Aktivierungen der vorherigen Schicht in Spalten, beziehungsweise verschiedene Datenpunkte. Als Ergebnis ergibt sich eine $N \times b$ Matrix, welche für jedes der N Neuronen b Aktivierungen für die Datenpunkte des Batches enthält.

5.3 Backpropagation

In Abschnitt 4.4 auf Seite 28 wurde die Durchführung der Backpropagation, also der Berechnung der Gewichts- und Schwellwertänderungen, für mehrere Datenpunkte in einem Batch beschrieben. In Listing 5.3 ist der Code für die Berechnung dieser Gewichts- und Schwellwertänderungen einer Schicht mit ArrayFire dargestellt. Die Berechnungen der Gewichtsänderungen lassen sich direkt aus den Formeln übertragen.

Für die Berechnung der Schwellwertänderungen wird die `af::sum`-Funktion von ArrayFire verwendet, um die `delta`-Matrix zeilenweise aufzuaddieren und so alle Schwellwertänderungen für alle Datenpunkte im Batch zu vereinen. Wenn die Schwellwertänderungen ein Teil der Gewichtsmatrix wären, entspräche das einer Zeile von Einsen in der `input`-Matrix im vorherigen `matmulNT`-Aufruf.

```
1 // scalar_mult((Nxb), (Nxb)) = (Nxb)
2 af::array delta = error * activation_deriv(/*...*/);
3 // matmul((Ixb), transpose(Nxb)) = matmul((Ixb), (bxN)) = (IxN)
4 this->weights_update += matmulNT(input, delta);
5 // (Nx1) += sum((Nxb), 1) = Nx1
6 this->bias_update += af::sum(delta, 1);
7 this->update_count += input.dims(1);
8 // matmul((IxN), (Nxb)) = (Ixb);
9 return af::matmul(this->weights, delta);
```

Listing 5.3: Backpropagation Code zum Berechnen der Gewichts- und Schwellwertänderung einer Schicht. Erhält in `input` die Aktivierungen der vorherigen Schicht und in `error` die Abweichung vom gewünschten Wert $\vec{e}$.

5.4 Kommunikation

Für die Kommunikation wird MPI mit ArrayFire verwendet. Um `af::array`-Objekte zu versenden wurden Wrapper-Funktionen für die benötigten MPI Kommunikationen geschrieben, welche anstatt eines Pointers auf einen Speicherbereich Referenzen auf die von ArrayFire bereitgestellten `af::array`-Objekte als Argumente erwarten.

Liegen die Daten eines `af::array`-Objektes auf dem CPU Backend kann mithilfe der `device()`-Funktion der Pointer auf den Speicherbereich der Daten im Hauptspeicher ermittelt werden. Dieser Pointer kann dann direkt an MPI übergeben werden und die Daten so ohne eine Kopie direkt von Ort und Stelle versendet werden.

Liegen die Daten eines `af::array`-Objektes jedoch auf einem GPU-Backend, müssen diese gegebenenfalls von der GPU in den Hauptspeicher kopiert werden, um sie mit MPI versenden zu können. ArrayFire stellt zum Kopieren des Inhaltes eines `af::array`-Objektes in den Hauptspeicher der CPU die `host()`-Funktion zur Verfügung. Aus dem Speicher der CPU können die Daten dann mit den normalen MPI Aufrufen kommuniziert werden. Anschließend müssen die eventuell von anderen Prozessen empfangenen Daten mit einer weiteren Kopieranweisung wieder zurück in den Speicher der GPU kopiert werden.

Dieser Vorgang des wiederholten Kopierens macht die Kommunikation zusätzlich ineffizienter, wenn eine GPU verwendet wird. Um diese Ineffizienz zu vermeiden wurde CUDA-Aware MPI[31] verwendet.

CUDA-Aware MPI ermöglicht es, einen Pointer, welcher auf einen CUDA-Speicherbereich auf der GPU zeigt direkt an eine MPI Funktion zu übergeben. Die CUDA-Aware MPI Implementierung vermeidet dann das vollständige Kopieren der Daten in den Hauptspeicher und erhöht so die Effizienz der Kommunikation. Die Daten können direkt aus dem Speicher der Grafikkarte an die Netzwerkkarte weitergeleitet werden. Es muss kein Zwischenspeicher angelegt werden um die Daten zu versenden.

Um an den benötigten CUDA-Pointer zu gelangen, kann wieder die `device()`-Funktion von ArrayFire verwendet werden, welche für ein `af::array`-Objekte auf einem CUDA-Backend den internen CUDA-Pointer zurückgibt. Die CUDA-Aware MPI Implementierung erkennt automatisch anhand des Adressbereiches ob es sich bei einem Pointer um einen normalen CPU- oder einen CUDA-Pointer handelt, sodass die gleichen MPI Funktionen verwendet werden können.

Nachdem die `device()`-Funktion verwendet wurde, muss nach Verwendung des Speichers auf dem gleichen `af::array`-Objekt die `unlock()`-Funktion aufgerufen werden, um den Pointer wieder für das automatische Speichermanagement von ArrayFire verfügbar zu machen.

Um CUDA-Aware MPI mit ArrayFire verwenden zu können war es zusätzlich nötig vor jedem MPI Aufruf mit einem CUDA Pointer die `af::sync()`-Funktion aufzurufen. Diese stellt sicher, dass alle Berechnungen auf der GPU abgeschlossen sind, bevor die Ausführung des Programms fortgesetzt wird. Ansonsten kam es dazu, dass mit CUDA-Aware MPI falsche oder unfertige Daten versendet und somit unkorrekte Ergebnisse berechnet wurden [32].

Da nur die OpenMPI Implementierung von CUDA-Aware MPI, nicht jedoch MVAPICH2, eine Funktion zum Prüfen der Verfügbarkeit von CUDA-Aware MPI besitzt, ist in der aktuellen Laufzeitumgebung eines Programms nicht immer ersichtlich, ob CUDA-Aware MPI überhaupt zur Verfügung steht. Deshalb wurde ein `--cudampi` Kommandozeilenargument hinzugefügt, mit dem die Verwendung aktiviert werden kann.

Unterstützt eine MPI Implementierung kein CUDA-Aware MPI kommt es bei einem MPI-Aufruf mit einem CUDA Pointer zu einem Speicherzugriffsfehler, da der CUDA Pointer von der MPI Implementierung als Pointer in den Hauptspeicher der CPU interpretiert wird. Dieser Pointer liegt dann jedoch nicht im zugreifbaren Speicherbereich und das Programm wird mit einer Fehlermeldung beendet.

Wird das `--cudampi` Argument übergeben wird für die `af::array` Objekte, welche auf einem CUDA Backend liegen, die gleiche Kommunikation durchgeführt, wie für welche, die auf einem CPU Backend liegen. Es wird lediglich der `device()`-Pointer an die MPI Funktion übergeben und anschließend wieder `unlock()` aufgerufen und es finden keine zusätzlichen Kopieroperationen statt. Hierzu wurde die `can_use_device_pointer()`-Hilfsmethode erweitert, welche von den MPI-Wrapper Funktionen aufgerufen wird.

5.5 JuML Integration

Um eine freie Implementierung zu Verfügung zu stellen wurde die Implementierung, welche in dieser Arbeit entstanden ist, in die JuML-Bibliothek integriert. Hierbei handelt es sich um eine sich in Entwicklung befindende Bibliothek, welche auch auf ArrayFire basiert.

Ziel der Bibliothek ist es verschiedene skalierbare parallelisierte Machine Learning Algorithmen mit einer einheitlichen Schnittstelle bereit zu stellen, sodass diese auf eine einfache Weise auf einen Datensatz angewandt, verglichen und verwendet werden können. Hierzu stellt JuML sowohl eine C++ Schnittstelle, als auch ein Python Modul bereit.

Damit die Integration erfolgreich ist, wurden die von JuML bereitgestellten Datenstrukturen und Interfaces verwendet und implementiert.

5.6 Zusammenfassung

ArrayFire ist eine Bibliothek, welche eine einfache und Plattform-unabhängige Schnittstelle für Grafikkarten bereitstellt. Mit ArrayFire konnte die Forward- und Backpropagation implementiert werden, sodass sie auf mehreren GPUs ausgeführt wird und hierbei mehrere Datenpunkte gleichzeitig als ein Batch propagiert werden.

Für die Kommunikation der Daten wurde MPI verwendet. Steht kein CUDA-Aware MPI zur Verfügung, müssen die Daten zunächst aus dem GPU Speicher in den Hauptspeicher der CPU kopiert werden. Dies macht Kommunikation zusätzlich ineffizient. Ist CUDA-Aware MPI verfügbar kann die Kommunikation von Daten auf der GPU mit MPI effizienter stattfinden, da nicht die kompletten Daten in dem Hauptspeicher der CPU kopiert werden müssen.

Diese Optimierung ist jedoch nicht plattformunabhängig einsetzbar, da CUDA-Aware MPI nur mit NVIDIA GPUs verwendet werden kann. Das Zielsystem welches für die Implementierung in dieser Arbeit verwendet wurde besitzt jedoch NVIDIA GPUs, sodass CUDA Aware MPI hier in Betracht gezogen werden konnte.

Die in dieser Arbeit entstandene Implementierung wird als Teil der JuML Bibliothek frei verfügbar gemacht.

6 Evaluation

Im Folgenden wird in Abschnitt 6.1 die Versuchsdurchführung mit der JUBE-Software beschrieben.

In Abschnitt 6.2 wird die verwendete Versuchsumgebung und in Abschnitt 6.3 die verwendeten Datensätze vorgestellt. Abschnitt 6.4 beschreibt dann die Ergebnisse mit einem Datensatz im Detail. Zuletzt wird in Abschnitt 6.5 eine Bewertung der Ergebnisse und ein Ausblick gegeben.

6.1 Durchführung der Versuche

Zum automatischen Durchführen der Versuche wurde das JUBE Benchmarking Environment[33] verwendet. Die Software des Jülich Supercomputing Centers des Forschungszentrums Jülich ermöglicht es die Ausführung eines Programms mit verschiedenen Konfigurationen zu automatisieren.

Hierzu werden in einer XML-Datei die möglichen Parameter des Programms angegeben, welche dann durch Variablenersetzung zu einem Programmaufruf zusammengefasst werden. JUBE erzeugt automatisch alle möglichen Parameterkombinationen und es ist sogar möglich Parameter in Abhängigkeit von anderen Parametern zu setzen. Ein Beispiel für eine JUBE Konfigurationsdatei mit mehreren Parametern ist in Listing 6.1 auf der nächsten Seite dargestellt.

Mit kleinen, in die XML Datei integrierten, Python Skripten lassen sich Parameter an Bedingungen knüpfen und Parameterwerte können zum Beispiel aus anderen Parametern berechnet werden. So lässt sich zum Beispiel verhindern, dass wenn nur ein Prozess verwendet wird sowohl das normale MPI, als auch CUDA-Aware MPI verwendet wird, denn ein einzelner Prozess benötigt keine Kommunikation und dementsprechend müssen nicht beide Verfahren ausprobiert werden. Die Parameterdefinition für solch einen Parameter ist beispielhaft in Listing 6.2 auf der nächsten Seite dargestellt.

Es stehen noch weitere Funktionen, wie das Kopieren von Dateien und das anschließende Ersetzen von Text zur Verfügung. So können die benötigten Job-Skripte automatisch erzeugt und anschließend mit Hilfe des Batchsystem auf dem Supercomputer ausgeführt werden.

Außerdem ist es möglich die Ergebnisdateien mit Regulären Ausdrücken zu durchsuchen. Mit den gefundenen Daten kann dann automatisch eine Tabelle erzeugt werden, welche die Suchergebnisse und Parameter für alle Ausführungen enthält. So kann zum Beispiel die erreichte Klassifizierungsgenauigkeit, sowie die Laufzeit einer Parameterkombination aus der Ausgabedatei ermittelt werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <benchmark name="parameterspace" outpath="bench_run">
    <comment>A parameterspace creation example</comment>

    <!-- Configuration -->
    <parameterset name="param_set">
      <!-- Create a parameterspace out of two template parameter -->
      <parameter name="number" type="int">1,2,4</parameter>
      <parameter name="text" separator=";">Hello;World</parameter>
    </parameterset>

    <!-- Operation -->
    <step name="say_hello">
      <use>param_set</use> <!-- use existing parameterset -->
      <do>echo "$text $number"</do> <!-- shell command -->
    </step>
  </benchmark>
</jube>
```

Listing 6.1: JUBE Beispiel `parameterspace.xml`. `echo` wird sechsmal mit allen Kombinationen aus 1, 2 und 4 mit „Hello“ und „World“ aufgerufen.

```
<parameter name="ntasks" type="int">1,4,8,12</parameter>
<parameter name="cudampiarg" mode="python">
  "--cudampi," if ${ntasks} > 1 else ""
</parameter>
<!--        ↑ Dieses Komma sorgt dafür, dass das Argument auch weggelassen wird,
              wenn ${ntasks} > 1 ist. -->
```

Listing 6.2: JUBE cudampi Python Parameter. Nur wenn `${ntasks}` größer als 1 ist, wird das Programm einmal mit dem „`--cudampi`“ Argument und einmal ohne ausgeführt. Die `${ntasks}`-Variable wird vor der Ausführung des Skriptes von JUBE ersetzt. Resultiert in $3 \cdot 2 + 1 = 7$ Ausführungen: Für `ntasks`=4, 8 und 12 einmal mit „`--cudampi`“ und einmal ohne. Für `ntasks` $= 1$ wird nur eine Ausführung ohne erzeugt.

JUBE wurde dazu verwendet die Ausführung mit mehreren Prozessen und unterschiedlichen Konfigurationen automatisch durchzuführen und anschließend die Laufzeit und Klassifizierungssgenauigkeit aus der Ausgabedatei zu lesen.

6.2 Versuchsumgebung

Als Versuchsumgebung kam der JURECA-Supercomputer[34] des Jülich Supercomputing Centers im Forschungszentrum Jülich zum Einsatz. Bei dem System handelt es sich um einen hybriden Supercomputer, welcher aus insgesamt 1872 Nodes besteht, welche durch ein InfiniBand-Netzwerk miteinander verbunden sind. Jeder Node ist hierbei mit zwei Intel Xeon E5-2680 v3 Haswell CPUs mit jeweils 12 CPU Kernen bestückt.

Von den Nodes besitzen 1605 jeweils 128 GB Speicher. Für Anwendungen mit größeren Speicheranforderungen stehen zusätzlich 128 Nodes mit 256 GB und 64 Nodes mit 512 GB Speicher zur Verfügung.

Weitere 75 Nodes sind zusätzlich mit zwei NVIDIA K80 GPUs ausgestattet, welche jeweils 4992 CUDA Kerne und 24 GB Speicher besitzen. Diese GPUs sind in zwei logische GPUs aufgeteilt, sodass diese auf jedem Node als vier voneinander unabhängige GPUs verwendet werden müssen.

Für die Versuche in dieser Arbeit wurden Jobs mit einem, vier, acht und zwölf Prozessen mit jeweils einer GPU pro Prozess ausgeführt. Somit konnten die Versuche mit einem, zwei und drei Nodes durchgeführt werden.

Das System besitzt eine Peak Performance von 1,8 Petaflop/s bei Verwendung der CPUs. Durch die GPUs erhöht sich die Geschwindigkeit um zusätzlich 0,44 Petaflop/s.

Der JURECA-Supercomputer stieg im November 2015 auf Platz 49 in die Top500 Liste[35, 36] ein, welche die stärksten Supercomputer der Welt auflistet. Im Juni 2016 rutschte JURECA auf Platz 57 ab. Zusätzlich ist JURECA mit 1719,27 Megaflops/W im November 2015 auf Platz 112[37] und im Juni 2016 auf Platz 125[38] der Green500 Liste auf der die effizientesten Supercomputer gelistet werden.

6.3 Remote Sensing Anwendung

Es wurden drei verschiedene Datensätze aus dem Anwendungsgebiet Remote Sensing verwendet um zu überprüfen, ob diese mit der Implementierung erfolgreich verwendet werden können. Die drei Datensätze unterscheiden sich in der Anzahl der Dimensionen, der Anzahl der Datenpunkte und auch der Anzahl der vorhandenen Klassen im Datensatz. In Tabelle 6.1 auf der nächsten Seite ist eine Übersicht gegeben.

Der kleinste verwendete Datensatz[40, 39], welcher 16 verschiedene Klassen beinhaltet und nur aus 543 Trainings- und 53 586 Testdatenpunkten besteht wurde vom AVIRIS Sensor[45] über dem Kalifornischen „Salinas Valley“ aufgenommen und besitzt ursprünglich 224 Kanäle/Features, von denen jedoch 20 wasserabsorbierende Kanäle entfernt wurden. Hierbei handelt es sich um einen Rohdatensatz bei dem direkt die Sensordaten verwendet werden. Diese wurden lediglich auf ein Intervall von 0 bis 1 normiert.

Datensatz	Anzahl Datenpunkte			Anzahl	
	Training	Test	Insgesamt	Features	Klassen
`salinas_raw` [39, 40]	543	53 586	54 129	204	16
`indian_processed` [41, 42]	33 396	300 555	333 951	30	52
„Rome“ [43, 44]	77 542	697 859	775 401	55	9

Tabelle 6.1: Die drei Datensätze im Überblick.

Die Evaluation zeigt, dass mit diesem Datensatz und der Implementierung gute Klassifizierungsgenauigkeiten von bis zu 90 % erreicht werden, jedoch ist die Parameterwahl für diesen Datensatz aufgrund der sehr kleinen Anzahl an Datenpunkten sehr eingeschränkt. Es kann zum Beispiel nur mit sehr kleinen lokalen Batchgrößen von höchstens 543 bei einem einzelnen Prozess gearbeitet werden. Werden mehr Prozesse verwendet sinkt die größtmögliche lokale Batchgröße durch die Datenaufteilung auf die Prozesse noch weiter ab.

Somit ist es nicht möglich mit diesem Datensatz die Eigenschaften und den Einfluss der Parallelisierung und der Batchgröße darzustellen. Die kleine Anzahl an Datenpunkten führt zusätzlich dazu, dass nur ein sehr kleiner Speedup erreicht werden kann.

Bei dem verwendeten mittelgroßen Datensatz[41, `indian_processed_...`], der sich aus 33 396 Trainings- und 300 555 Testdatenpunkten zusammensetzt und 52 Klassen beinhaltet handelt es sich um einen vorprozessierten Datensatz [42]. Hier wurde die Anzahl der Features von ursprünglichen 200 auf 30 deutlich reduziert, wobei Feature Engineering Aspekte vom Autor des Datensatzes angewendet wurden, um eine bessere Informationsdichte zu erhalten.

Dieser Datensatz stellte sich jedoch als problematisch heraus. Es konnte für diesen Datensatz kein Modell erzeugt werden, welches eine Klassifizierungsgenauigkeit von mehr als 75 % besaß. Auch mit anderen Ansätzen konnten mit diesem Datensatz keine herausragenden Genauigkeiten erreicht werden. In [42] wurde mit einer Support Vector Machine eine maximale Klassifizierungsgenauigkeit von 77,02 % erreicht.

Dies lässt sich zum einen damit erklären, dass die Datenpunkte sehr ungleich auf die sehr große Anzahl an Klassen verteilt sind, was einen Einfluss auf das Training hat. So sind für die kleinste Klasse in dem Testdatensatz nur 16 Datenpunkte vorhanden, während für die größte Klasse 6356 Datenpunkte zur Verfügung stehen. Auch eine Reduzierung der Klassen auf die 10 größten, welche immer noch 67,32 % des Datensatzes ausmachen, brachte keine Verbesserung. Des Weiteren ist dies ein sehr anspruchsvoller Datensatz, da er eine sehr hohe Anzahl an sogenannten „Mixed Pixels“ beinhaltet, also Datenpunkte, welche eigentlich zu mehreren Klassen gehören. Außerdem sind die Datenpunkte in sehr genaue Klassen unterteilt worden. So sollen in diesem Datensatz nicht nur Sojabohnenfelder von Maisfeldern, Flüssen, Seen und anderen Bodenbedeckungen unterschieden werden, sondern zusätzlich auch noch verschiedene Anbautechniken und Zustände der Soja- und Maisfelder erkannt werden. Dies macht den Datensatz sehr anspruchsvoll.

Klasse	Anzahl Datenpunkte	
	Training	Test
Buildings	18 126	163 129
Blocks	10 982	98 834
Roads	16 353	147 176
Light Train	1606	14 454
Vegetation	6962	62 655
Trees	9088	81 792
Bare Soil	8127	73 144
Soil	1506	13 551
Tower	4792	43 124
Insgesamt	77 542	697 859

Tabelle 6.2: Rome Datensatz bestehend aus neun Klassen, 55 Features und insgesamt 775 401 Datenpunkten. In [43] stehen das Training und Testset als `sdap_area_all_training`, beziehungsweise `sdap_area_all_test` zur Verfügung. [44]

Bei dem größten Datensatz[43, `sdap_area_all_...`] mit 77 542 Trainings- und 697 859 Testdatenpunkten handelt es sich um eine Aufnahme der Stadt Rom mit dem QuickBird Satelliten. Es werden neun verschiedene Klassen von Bodenbedeckung unterschieden, welche in Tabelle 6.2 aufgelistet sind. Für diesen Datensatz wurden mit dem „Self Dual Attribute Profile“ (SDAP) [44] 55 Features erzeugt.

Der komplette Datensatz mit Training- und Testset ist so groß, dass es nicht möglich ist, den kompletten Datensatz zusätzlich zu den Gewichtungen des MLP in den Speicher einer einzelnen GPU-Unterteilung der verwendeten Versuchsumgebung zu laden. Um dennoch eine Laufzeit mit einer einzelnen GPU-Unterteilung bestimmen zu können wurde der Testdatensatz in zwei Teile halbiert.

Dies hat den weiteren Vorteil, dass zwei unabhängige Testsets für die Parameterbestimmung des Netzes und Algorithmus und das anschließende Training und Evaluierung der Parallelisierung zur Verfügung stehen.

Im folgenden Abschnitt werden die Ergebnisse anhand des Rome Datensatzes vorgestellt, weil er sich gut eignet die Parallelisierungseffekte darzustellen und die Ziele der Arbeit besser zu evaluieren.

6.4 Diskussion der Ergebnisse

Es hat sich herausgestellt, dass Mehrschichtige Perzeptrons sehr schwierig zu trainieren sind, da nicht nur die Anzahl der Neuron-Gewichtungen, welche vom Algorithmus optimiert werden sollen, sehr groß ist, sondern auch die Anzahl an Parametern, die eingestellt werden müssen, damit der Algorithmus eine ausreichend gute Lösung findet.

Somit benötigen künstliche neuronale Netze sehr viel Optimierungszeit und Arbeit, um im Remote Sensing oder anderen Anwendungsgebieten eingesetzt werden zu können. Fokus dieser Arbeit soll jedoch nicht die perfekte Optimierung eines künstlichen neuronalen Netzes für einen einzelnen Anwendungsfall sein, sondern die generelle Bereitstellung eines parallelen und skalierbaren Algorithmus für die Verwendung auf Supercomputern. Die letztendliche finale Optimierung, sowie das Feature Engineering wird von Domainwissenschaftlern durchgeführt.

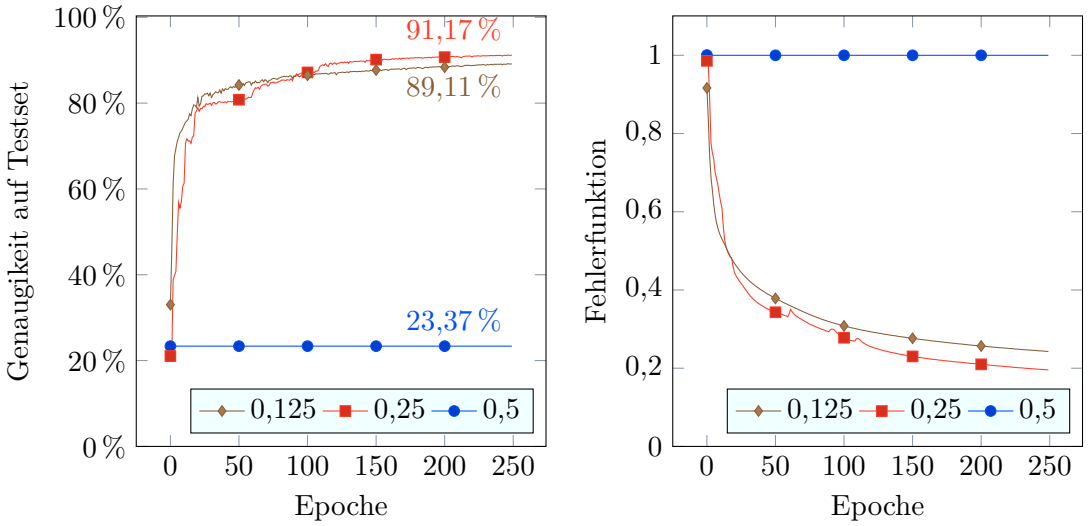
Zusätzlich zu der großen Anzahl an Parametern bestehen auch Abhängigkeiten zwischen den Parametern, sodass der optimale Wert für einen Parameter von der Wahl der anderen Parameter abhängen kann. Das bedeutet, dass es nicht ausreicht für jeden Parameter einzeln einen guten Wert zu suchen, sondern es muss insgesamt eine gute Parameterkombination gefunden werden.

Die Metrik der Genauigkeit des Modells ist zusätzlich davon abhängig für wie viele Epochen das Modell trainiert wurde und neben den Abhängigkeiten unter den Parametern können auch zeitliche Abhängigkeiten der Parameter existieren. Das bedeutet, dass ein Parameterwert zwar für die ersten Epochen des Algorithmus gute Ergebnisse liefern kann, dann jedoch für spätere Epochen zu klein oder groß gewählt ist und angepasst werden muss. Somit ist es schwierig die Wahl der Parameterkombination anhand einer bestimmten Anzahl an Epochen zu bewerten, da nicht sichergestellt ist, dass die Parameterwahl auch für weitere Epochen gut ist.

Manche Parameter haben nicht nur einen Einfluss auf die Konvergenzgeschwindigkeit, bzw. Genauigkeit des resultierenden Modells, sondern auch auf die Ausführungszeit und beeinflussen somit den Speedup. Eine Auflistung der unterschiedlichen Parameter und deren Einflüsse ist in Tabelle 6.3 gegeben und wird im Folgenden noch genauer erläutert.

Parameter	Art	Hat Einfluss auf	
		Genauigkeit	Speedup
Anzahl Epochen	Ganzzahl	✓	✗
Learning-Rate	Zahl	✓	✗
Weight-Decay	Zahl	✓	✗
Momentum	Zahl	✓	✗
Durchmischen der Batches	Ja/Nein	✓	✗
Synchronisation: Epoche/Batch	Ja/Nein	✓	✓
Anzahl Neuronen und Schichten	Ganzzahlen	✓	✓
Batchgröße	Ganzzahl	✓	✓
Größe des Datensatz	Ganzzahl	✓	✓
Anzahl Prozesse	Ganzzahl	✓[1]	✓
CUDA-Aware MPI	Ja/Nein	✗	✓

Tabelle 6.3: Die verschiedenen Parameter, die einen Einfluss auf die Genauigkeit und den Speedup haben. [1]Nur bei Synchronisation nach Epoche.



(a) Die Entwicklung der Klassifizierungsgenauigkeit auf dem Testset.

(b) Der Verlauf des durchschnittlichen Fehlers pro Datenpunkt.

Abbildung 6.1: Die Konvergenz des Algorithmus mit verschiedenen Learningraten und einem Momentum von 0,5.

6.4.1 Learning-Rate, Weight-Decay und Momentum

Die drei Parameter, welche einen großen Einfluss auf die Konvergenzgeschwindigkeit des Algorithmus haben sind die Learning-Rate, das Weight-Decay und das Momentum. Diese Parameter haben jedoch keinen Einfluss auf die Ausführungszeit einer Epoche. Werden sie jedoch falsch gewählt müssen eventuell viel mehr Epochen berechnet werden um ein gleich gutes Ergebnis zu erzielen, als dies mit einer anderen Parameterwahl der Fall gewesen wäre. Für diese Parameter müssen Zahlen angegeben werden, die üblicherweise zwischen 0 und 1 liegen.

Da die Parameter nicht ganzzahlig sind ist es praktisch nicht möglich den perfekten Parameterwert zu bestimmen, da es durch die Nachkommastellen möglich ist einen Wert beliebig nahe anzunähern.

Wird die **Learningrate** zu groß gewählt kommt es dazu, dass der Fehler oszilliert und kein Minimum gefunden werden kann. Eine zu kleine Wahl führt jedoch dazu, dass die Konvergenzgeschwindigkeit des Algorithmus so stark abnimmt, dass kein Ergebnis in einer akzeptablen Rechenzeit erreicht werden kann.

Der **Momentum**-Parameter fügt dem Gradient-Descent Algorithmus der Backpropagation einen Beschleunigungsterm hinzu. Für diesen Parameter gelten ähnliche Bedingungen wie für die Learningrate. Ein zu hoher Wert führte dazu, dass kein Minimum gefunden werden konnte. Im großen und ganzen war es jedoch mit dem Momentum Parameter möglich die Konvergenz des Algorithmus zu beschleunigen.

In Abb. 6.1 ist der Verlauf der Klassifizierungsgenauigkeit und der Fehlerfunktion eines MLP mit 1000 versteckten Neuronen, einem Momentum von 0,5, einer Batchgröße

von 100 und verschiedenen Learningrates für den Rome-Datensatz dargestellt. Bei einer zu großen Learningrate findet keine Konvergenz statt. Obwohl eine Learningrate von 0,125 in den ersten Epochen zunächst bessere Ergebnisse erzielt, wird die maximale Genauigkeit und der minimale Fehler letztendlich mit einer Learningrate von 0,5 erreicht. Es ist nicht ersichtlich, ob eine eventuell bessere Learningrate im Bereich $[0.125, 0.25]$ oder $[0.25, 0.5]$ zu suchen ist.

Der **Weight-Decay**-Parameter soll verhindern, dass ein overfitting durch das neuronale Netz stattfindet. Die Verwendung eines solchen Parameters wird auch als „Regularisierung“ bezeichnet. Nach jeder Iteration des Backpropagation Algorithmus wird jede Gewichtung um einen Anteil reduziert. Dies führte jedoch dazu, dass auch schon für sehr kleine Parameter-Werte die Genauigkeit des resultierenden Modells sowohl auf dem Testset, als auch auf dem Trainingset massiv eingeschränkt wurde.

Selbst für einen sehr kleinen Parameter-Wert von 10^{-6} konnte nach der gleichen Anzahl an Epochen nicht annähernd die Genauigkeit erreicht werden, welche ohne die Verwendung von Weight-Decay erreicht wurde. Symptome für overfitting, wie das Abfallen der Genauigkeit auf dem Testset bei gleichzeitigem Ansteigen der Genauigkeit auf dem Trainingsset wurden auch ohne Weight-Decay nicht beobachtet.

6.4.2 Batchgröße und Größe des Datensatzes

Die Batchgröße hat einen großen Einfluss auf die Konvergenzgeschwindigkeit sowie den Speedup. Desto kleiner die Batchgröße gewählt wird, desto mehr Batches müssen für eine vollständige Epoche berechnet werden.

Der Overhead, welcher durch die Vor- und Nachbereitung der Berechnung eines Batches entsteht, sorgt so dafür, dass es effizienter ist wenige große als sehr viele kleine Batches zu berechnen. Dieser Effekt wird noch verstärkt, wenn nach jeder Berechnung eines Batches eine Kommunikation mit anderen Prozessen stattfinden muss.

In Abb. 6.2 ist dieser Overhead beispielhaft für einen linearen Anstieg der Berechnungszeit bei Vergrößerung eines Batches und einer konstanten Vor- und Nachbearbeitungszeit dargestellt.

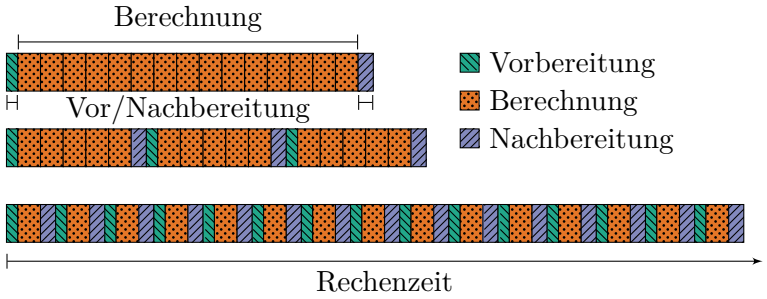
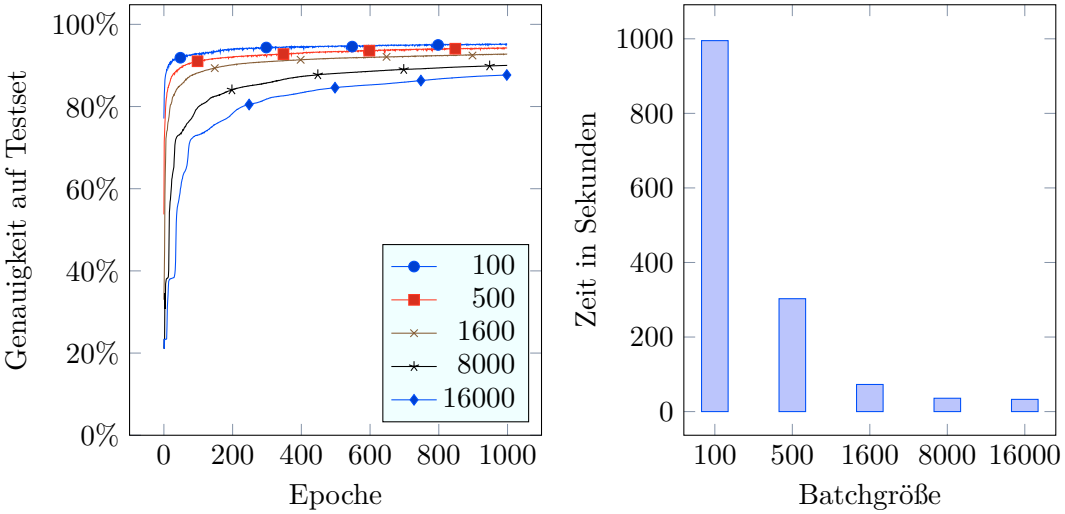


Abbildung 6.2: Beispiel für die Entwicklung der Rechenzeit mit verschiedenen Batchgrößen. Durch die benötigte Vor- und Nachbearbeitung ist es effizienter große Batches zu berechnen.

Die Batchgröße ist nach oben durch die Anzahl der Datenpunkte in dem verwendeten Datensatz beschränkt, sodass für kleinere Datensätze auch nur eine vergleichsweise kleine Batchgröße gewählt werden kann. Für größere Datensätze können dann größere Batchgrößen gewählt werden oder es müssen für eine Epoche mehr Batches berechnet werden, als für einen kleineren Datensatz bei gleicher Batchgröße. Somit hat die Größe des Datensatzes und die Wahl der Batchgröße einen großen Einfluss auf die Laufzeit und damit auch auf den Speedup.

In Abb. 6.3 ist die Entwicklung der Klassifizierungsgenauigkeit und die Rechendauer für ein Beispiel mit verschiedenen Batchgrößen zunächst ohne Parallelisierung dargestellt, um den reinen Einfluss der Batchgröße auf die Klassifizierungsgenauigkeit darzustellen.



(a) Entwicklung der Klassifizierungsgenauigkeit auf dem Testset. Desto höher die Batchgröße, desto langsamer lernt das MLP.

(b) Rechendauer für verschiedene Batchgrößen. Desto kleiner die Batchgröße, desto länger dauert die Berechnung der Epochen

Abbildung 6.3: Entwicklung der Genauigkeit und Rechendauer für ein Beispiel mit verschiedenen Batchgrößen über eine feste Anzahl von 1000 Epochen und mit identischen Parametern.

Desto größer die Batchgröße gewählt wird, desto schlechter ist die Klassifizierungsgenauigkeit nach der gleichen Anzahl an Epochen. Betrachtet man die Lernentwicklung genauer kann man feststellen, dass der Algorithmus zunächst lernt alle Datenpunkte der Klasse mit den meisten Datenpunkten zuzuordnen und erst anschließend diese Klassifizierung verfeinert und beginnt die Datenpunkte zwischen der größten und zweitgrößten Klasse zu unterscheiden.

Größere Klassen dominieren die Lernentwicklung des MLP und dieser Effekt wird durch eine große Batchgröße verstärkt, da desto größer ein Batch ist, desto mehr Daten-

punkte der größeren Klassen sind in dem Batch enthalten und verdrängen die kleineren Klassen mit weniger Datenpunkten. Dies wird auch als „sampling bias“ bezeichnet.

Um den Speedup mit verschiedenen Batchgrößen zu vergleichen muss die Art der Synchronisierung in Betracht gezogen werden.

6.4.3 Anzahl der Neuronen und Schichten und CUDA-Aware MPI

Da für jedes Neuron mehrere Gewichtungen und ein Schwellwert bestimmt werden muss, legt die Anzahl der Neuronen und Schichten des MLP die Anzahl der Unbekannten fest, welche vom Backpropagation-Algorithmus optimiert werden müssen. Diese Anzahl beeinflusst somit die Rechenzeit, aber auch die Konvergenzgeschwindigkeit und die Genauigkeit des resultierenden Modells. Eine größere Anzahl an Neuronen führt nicht nur dazu, dass die Berechnungsdauer einer Epoche zunimmt, sondern auch dazu, dass mehr Epochen benötigt werden, um gute Gewichtungen für alle Neuronen zu finden.

Sind nicht genug Neuronen vorhanden kann es sein, dass das MLP nicht in der Lage ist die Daten ausreichend zu approximieren und somit nicht die gewünschte Ausgabe erzeugt. Zu viele Neuronen können jedoch overfitting begünstigen. Somit wird eine möglichst kleine Anzahl an Neuronen bevorzugt.

Ist die Anzahl an Neuronen klein führt dies jedoch durch die verringerte Rechenzeit dazu, dass eine Parallelisierung gegebenenfalls nicht effizient durchgeführt werden kann, da der Kommunikationsaufwand größer wird, als der benötigte Rechenaufwand. Zusätzlich sinkt jedoch der Kommunikationsaufwand für kleine Netze, da insgesamt weniger Daten ausgetauscht werden müssen.

CUDA-Aware MPI

In Abb. 6.4 auf der nächsten Seite ist der Speedup für zwei Ausführungen mit identischer Paramterwahl, jedoch einmal mit und einmal ohne CUDA-Aware MPI dargestellt. Es hat sich herausgestellt, dass ohne CUDA-Aware MPI kein Speedup erreicht werden kann.

Nachdem das in Abschnitt 5.4 auf Seite 38 beschriebene Problem, bei der Verwendung von ArrayFire und CUDA-Aware MPI, behoben wurde, konnte kein Einfluss von CUDA-Aware MPI auf die Klassifizierungsgenauigkeit mehr festgestellt werden. Die resultierenden Modelle sind identisch.

6.4.4 Art der Synchronisierung und die Genauigkeit

Die Synchronisierung nach jeder Epoche ermöglicht es, lokal kleinere Batchgrößen für eine Backpropagation zu verwenden. So lässt sich die verlangsamte Konvergenz des Algorithmus bei großen Batchgrößen vermeiden.

Wie in Abschnitt 6.4.2 erläutert führt eine kleine Batchgröße auch zu einer erhöhten Anzahl an Batches pro Epoche. Eine kleine Batchgröße würde bei einer Synchronisierung nach jedem Batch deshalb zu einem großen Kommunikationsaufwand führen. Dies wird durch die Synchronisierung nach jeder Epoche zusätzlich vermieden. Wird jedoch sowieso nur ein einzelner Batch pro Epoche berechnet sind die Ergebnisse der beiden

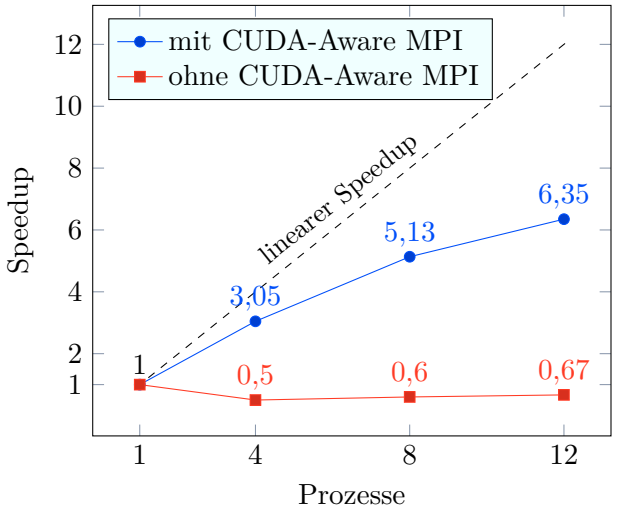


Abbildung 6.4: Speedup mit und ohne CUDA-Aware MPI. Es wurden identische Parameter verwendet.

Synchronisierungsarten identisch. Nichtsdestotrotz führt die Synchronisierung nach jeder Epoche dazu, dass das Ergebnis des Algorithmus zusätzlich von der Anzahl der verwendeten Prozesse abhängt.

Im Folgenden werden verschiedene Resultate mit dem Rome-Datensatz für ein kleines MLP von 50 versteckten Neuronen und ein größeres MLP mit 1000 Neuronen vorgestellt, welche für die gleiche Anzahl von 250 Epochen mit den beiden verschiedenen Arten der Synchronisierung trainiert wurden. Hierbei wurde immer CUDA-Aware MPI verwendet.

Um für beide Netzgrößen eine gute Konvergenz zu erreichen mussten für die beiden verschiedenen Netze zwei verschiedene Learningrates verwendet werden. Für das kleinere MLP mit 50 versteckten Neuronen konnte eine Learningrate von 2,0 verwendet werden. Das größere MLP mit 1000 versteckten Neuronen konvergierte jedoch nicht mit einer Learningrate größer als 0,5. Für dieses Netz wurde eine Learningrate von 0,25 verwendet. Zusätzlich wurde für das Training der beiden Netze ein Momentum Parameter von 0,5 gewählt.

Diese Parameter wurden jeweils für eine serielle Ausführung und einer Batchgröße von 100 auf ausreichende Konvergenz innerhalb von 250 Epochen geprüft. Diese Anzahl an Epochen führte für die gewählten Parameter zu einem guten Ergebnis, die erreichten Klassifizierungsgenauigkeiten lassen sich jedoch durch die Berechnung weiterer Epochen noch verbessern.

Beim Vergleichen des Speedup ist zu beachten, dass aufgrund der Architektur der Versuchsumgebung immer zwei Prozesse eine hardwaremäßig vorhandene GPU gemeinsam verwenden, sodass bei 12 Prozessen nur 6 echte GPUs zur Verfügung stehen. Dies kann einen negativen Einfluss auf den Speedup haben.

Es wird nun zunächst auf die Synchronisierung nach jedem Batch eingegangen und anschließend die Ergebnisse mit der Synchronisierung nach jeder Epoche verglichen.

Synchronisierung nach einem Batch

Abb. 6.5 auf der nächsten Seite stellt den erreichten Speedup bei Verwendung der Synchronisation nach jedem Batch mit verschiedenen Batchgrößen dar. Die Batchgröße gibt hier die Anzahl an Datenpunkten an, welche von allen Prozessen gemeinsam in einem Batch berechnet wird. Es lässt sich erkennen, dass mit dem kleinen MLP kein Speedup erreicht wird, da durch die geringe Anzahl an versteckten Neuronen die Berechnung des Netzes so schnell durchgeführt werden kann, dass eine Parallelisierung nicht effizient ist.

Für das MLP mit 1000 versteckten Neuronen kann zwar ein Speedup erreicht werden, dieser ist jedoch nur marginal. Der Speedup verhält sich hier proportional zur verwendeten Batchgröße und steigt mit einer Erhöhung der Batchgröße an. Dies liegt daran, dass bei einer kleinen Batchgröße sehr viele Batches pro Epoche gerechnet werden müssen und somit auch sehr viel Kommunikation für die Berechnung einer einzigen Epoche stattfindet. Bei einer Synchronisation nach jedem Batch wird somit die Ausführungszeit durch Kommunikation dominiert.

Wie in Abb. 6.6 auf der nächsten Seite dargestellt nimmt jedoch mit einer Erhöhung der Batchsize die erreichte Genauigkeit ab, sodass diese nicht beliebig erhöht werden kann.

Mit dem kleineren MLP mit 50 versteckten Neuronen konnte so zwar kein Speedup erreicht werden, die Klassifizierungsgenauigkeiten auf dem Testset sind jedoch mit 90,63 %, 85,04 % und 80,53 % ausreichend hoch, sodass eine Klassifizierung durchgeführt werden könnte. Für das größere MLP mit 1000 versteckten Neuronen wurden mit diesen Parametern keine guten Klassifizierungsgenauigkeiten erreicht.

Wie erwartet wurde die Klassifizierungsgenauigkeit bei der Synchronisation nach einem Batch nicht oder nur kaum durch die Anzahl der verwendeten Prozesse beeinflusst. Ein Unterschied zwischen den Genauigkeiten bei verschiedener Anzahl an Prozessen besteht nur bei dem größeren MLP und einer Batchgröße von 1600. Dieser Ausreißer lässt sich durch den Einfluss der Zusammensetzung der Batches erklären.

Da bei einer parallelen Ausführung die Datenpunkte gleichmäßig auf die Prozesse aufgeteilt sind und ein Batch aus der gleichen Anzahl Datenpunkte von jedem Prozess zusammengesetzt wird sind manche Zusammensetzungen von Datenpunkten zu einem Batch nicht möglich. Dies kann u einem anderen Lernergebnis führen. Durch Durchmischen der Datenpunke konnte dieser Effekt verkleinert werden.

Synchronisierung nach einer Epoche

Für die Synchronisation nach jeder Epoche wurden auch ein kleines und ein großes MLP mit 50 und 1000 versteckten Neuronen und verschiedenen Batchgrößen getestet. Hierbei werden nicht wie bei der Synchronisation nach jedem Batch die globalen, sondern lokale Batchgrößen verglichen, da dies die Batchgröße ist, welche zum Anpassen der Gewichtungen lokal verwendet wird. Somit reduziert sich die Anzahl der Batches pro Epoche, denn durch die Aufteilung der Daten auf die Prozesse stehen weniger Daten pro Prozess zur Verfügung, desto mehr Prozesse verwendet werden.

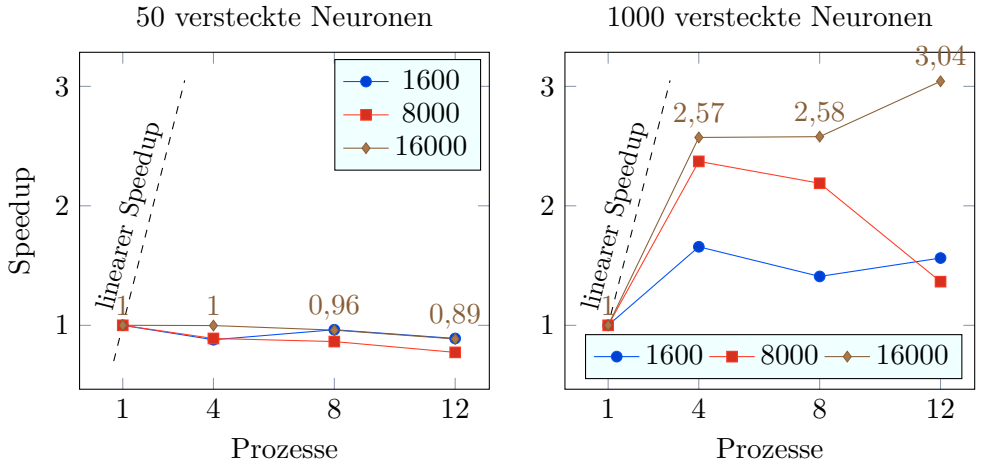


Abbildung 6.5: Speedup mit Synchronisation nach Batch für ein MLP mit 50 und eines mit 1000 versteckten Neuronen und verschiedene globalen Batchgrößen.

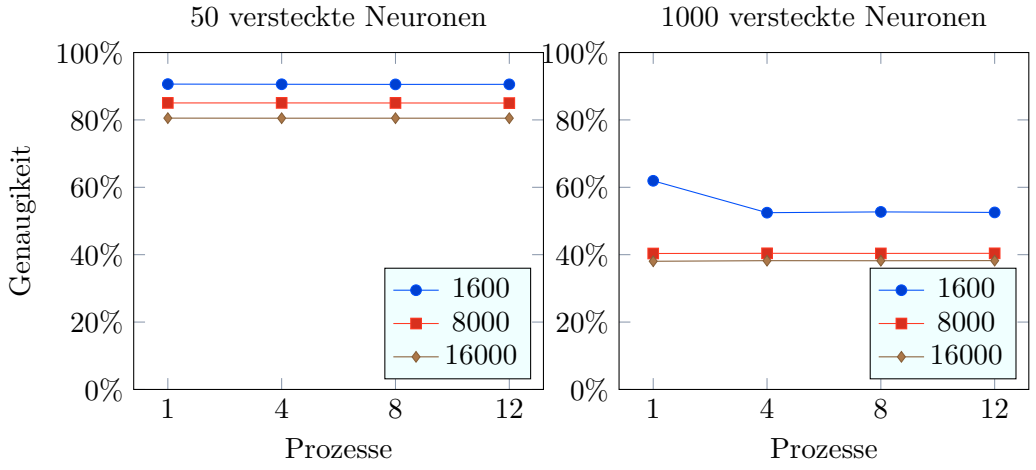


Abbildung 6.6: Klassifizierungsgenauigkeit mit Synchronisation nach Batch für ein MLP mit 50 und eines mit 1000 versteckten Neuronen und verschiedenen globalen Batchgrößen.

Wie in Abb. 6.7 auf der nächsten Seite zu erkennen ließ sich mit der Synchronisation nach jeder Epoche und verschiedenen Batchgrößen sowohl für das große, als auch das kleine MLP ein sehr viel besserer Speedup erreichen als dies mit der Synchronisation nach jedem Batch möglich war. Dies ist hauptsächlich durch den stark verminderten Kommunikationsaufwand pro Epoche zu erklären, da nur eine einzige Synchronisation stattfindet. Im Gegensatz zur Synchronisation nach einem Batch bringt eine Verkleinerung der Batchgröße sogar eine Verbesserung des Speedups mit sich. Hierbei wurden generell kleinere Batchgrößen als mit der Synchronisierung nach jedem Batch verwendet.

In Abb. 6.8 auf der nächsten Seite ist die erreichte Klassifizierungsgenauigkeit dargestellt. Mit der Synchronisierung nach einer Epoche konnte im Gegensatz zur Synchronisierung nach jedem Batch auch bei dem größeren MLP mit 1000 versteckten Neuronen eine gute Klassifizierungsgenauigkeit erreicht werden. Insbesondere ist zu erkennen, dass die Parameterwahl der kleinsten verwendeten Batchgröße, sowohl die besten Genauigkeitsergebnisse lieferte, als auch den besten Speedup ermöglichte.

Wie zuvor in Abschnitt 6.4.2 auf Seite 48 bereits dargestellt wird die Berechnungszeit größer, desto kleiner die Batchgröße gewählt wird. Diese ansteigende Berechnungszeit bei Verkleinerung der Batchgröße führt hier zu einem guten Speedup, da eine Parallelisierung umso effizienter ist, desto mehr Arbeit zwischen zwei Synchronisierungsvorgängen parallel erledigt werden kann. Eine noch stärkere Verkleinerung der Batchgröße könnte den Speedup noch verstärken, würde jedoch dazu führen, dass die Rechenzeit insgesamt sehr stark zunimmt.

Es hat sich jedoch auch bestätigt, dass die Anzahl der verwendeten Prozesse einen Einfluss auf die Klassifizierungsgenauigkeit des erzeugten Modells hat. Für das kleinere MLP ist dies zwar noch zu vernachlässigen, das größere ist jedoch zumindest bei einer größeren lokalen Batchgröße stark eingeschränkt. Dies muss bei der Verwendung beachtet werden.

6.5 Bewertung und Ausblick

Es konnte gezeigt werden, dass mit MLPs Klassifizierungsprobleme im Remote Sensing gelöst werden können. Dies ist jedoch mit der Schwierigkeit verbunden, dass vorher gute Parameter für das MLP und den Backpropagation Algorithmus bestimmt werden müssen. Hierbei hat sich herausgestellt, das bereits kleine MLPs mit wenigen versteckten Neuronen ausreichen können um ein Modell mit einer guten Klassifizierungsgenauigkeit zu erhalten.

Die Optimierung der Parameter nimmt für Wissenschaftler aus den Anwendungsgebieten wie Remote Sensing sehr viel Zeit ein, ist jedoch nicht Kernbestandteil dieser Arbeit. Vielmehr sollte ein Design und eine Implementierung eines parallelen künstlichen neuronalen Netzes geschaffen werden, sodass den Wissenschaftlern aus den Anwendungsgebieten ein schnelleres Arbeiten durch Parallelisierung und den Einsatz von Supercomputern ermöglicht wird.

Mit der entwickelten Parallelisierungsstrategie konnte ein Speedup erreicht werden, jedoch ist nicht immer sichergestellt, dass die Parallelisierung keinen negativen Einfluss

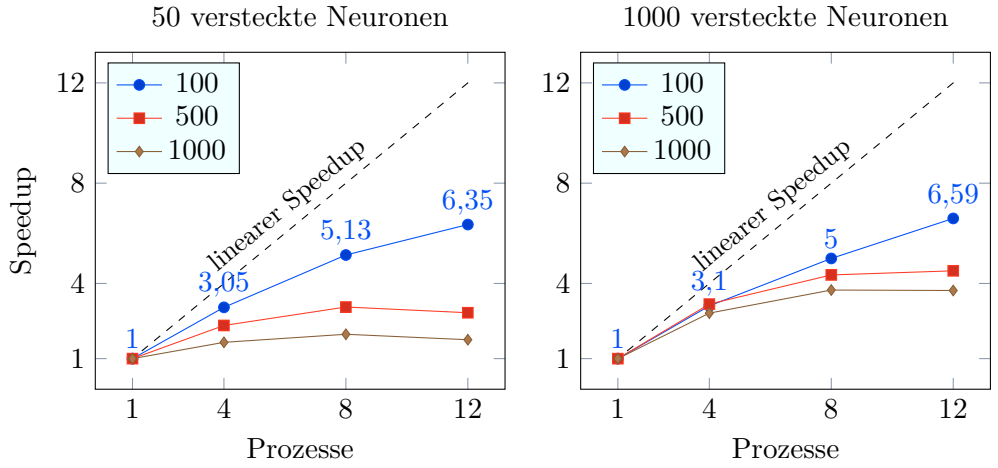


Abbildung 6.7: Speedup mit Synchronisation nach Epoche für ein MLP mit 50 und eines mit 1000 versteckten Neuronen und verschiedenen lokalen Batchgrößen.

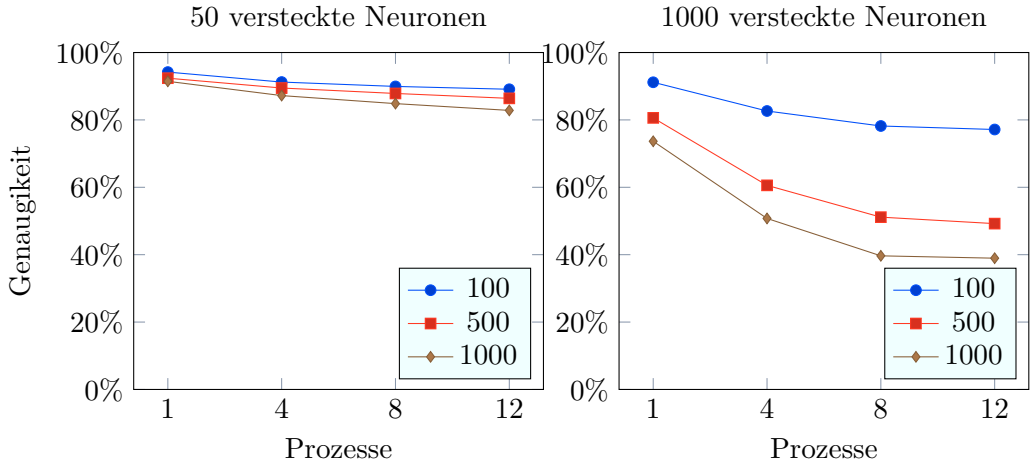


Abbildung 6.8: Klassifizierungsgenauigkeit mit Synchronisation nach Epoche für ein MLP mit 50 und eines mit 1000 versteckten Neuronen und verschiedenen lokalen Batchgrößen.

auf die Qualität des resultierenden Modells hat. Der erreichte Speedup ist zusätzlich stark von der Parameterwahl der Batchgröße sowie der Größe des verwendeten MLP abhängig. Diese Parameter beeinflussen neben dem Speedup auch die erreichte Klassifizierungsgenauigkeit des Modells.

Des Weiteren hat sich herausgestellt, dass die Parameterwahl des Backpropagation Algorithmus einen starken Einfluss auf die Konvergenz hat und diese Parameter abhängig von anderen Parametern ausreichend gut gewählt werden müssen um überhaupt eine Konvergenz zu erreichen. Hinzu kommt die Schwierigkeit, dass sich diese Parameter gegenseitig beeinflussen. Aufgrund der großen Anzahl an Parametern ist es jedoch kaum möglich verschiedene Parameterwerte auszuprobieren, da bereits bei nur 3 Parametern und nur 3 verschiedenen Werten pro Parameter 27 verschiedene Modelle trainiert werden müssten um alle Parameterkombinationen bewerten zu können. Diese Anzahl steigt sehr schnell sehr stark an, insbesondere, wenn die verschiedenen Modelle zusätzlich mit mehreren Parallelisierungsstrategien und Prozessoranzahlen getestet werden müssen.

In Zukunft wäre es von Vorteil eine Möglichkeit zu finden die Anzahl der Parameter, welche eingestellt werden müssen, zu reduzieren. Dies könnte zum Beispiel für die Learningrate durch eine automatische Schrittweitensteuerung erreicht werden. Alternativ könnte anstelle des Backpropagation Algorithmus, welcher auf Gradient Descent basiert, auch ein anderer Optimierungsalgorithmus implementiert werden, der diese große Anzahl an Parametern vermeidet.

Durch die Implementierung eines auf den Lernalgorithmus angepassten GPU Kernels, anstelle der ArrayFire-Methoden, wäre es vermutlich möglich die Berechnung auf der GPU noch effizienter zu gestalten, insbesondere könnten so mehrere sehr kleine Batches innerhalb eines einzigen GPU Kernels berechnet werden, sodass der Aufwand des Ansteuerns der GPU für jeden Batch reduziert wird. Hierbei müsste dann jedoch auf die Plattformunabhängigkeit der ArrayFire Lösung verzichtet werden.

Die durchgeführten Evaluationen und Analysen zeigten auch, dass die Anzahl der Datenpunkte einer Klasse im Datensatz einen starken Einfluss darauf hat, wie schnell diese Klasse von dem MLP erlernt wird. Dies könnte dazu führen, dass das MLP für Datenpunkte großer Klassen zu stark angepasst wird, also ein overfitting besteht, während die Struktur von unterrepräsentierten Klassen noch nicht erlernt ist. Dem könnte durch ein Gewichten der Datenpunkte anhand der Klassengröße oder ein zufälliges Auswählen von gleich vielen Datenpunkten jeder Klasse entgegen gewirkt werden.

In dieser Arbeit wurde für die Ausgabeschicht die Sigmoid-Aktivierungsfunktion verwendet. Eine weitere mögliche Verbesserung wäre die Implementierung der Softmax-Aktivierungsfunktion. Diese ist besonders gut geeignet für MLPs, welche mehrere Klassen klassifizieren, da sie sicherstellt, dass die Summe der Aktivierungen der Neuronen in der Ausgabeschicht 1 ergibt. So können die Aktivierungen der Ausgabeneuronen als Wahrscheinlichkeiten aufgefasst werden.

Anstatt ein einzelnes MLP für alle Klassen gleichzeitig zu trainieren wäre es auch möglich mehrere MLPs gleichzeitig nebeneinander zu verwenden. Diese könnten dann zum Beispiel mit zwei verschiedenen Klassen trainiert werden, sodass ein einzelnes MLP nur die Entscheidung treffen muss, ob ein Datenpunkt zu der einen oder einer anderen Klasse gehört. Zum Bestimmen der Klassenzugehörigkeit müssten dann jedoch mehrere

MLPs für einen Datenpunkt ausgewertet werden und alle Ergebnisse zusammengefasst werden.

7 Zusammenfassung

Im Rahmen dieser Arbeit sollte ein skalierbares und paralleles künstliches neuronales Netz entwickelt und angewendet werden. Hierzu wurde die Modellierung eines KNN als Mehrschichtiges Perzeptron gewählt.

Eine skalierbare und parallele Implementierung wird benötigt, da die Trainingszeiten von künstlichen neuronalen Netzen, insbesondere bei größeren Datenmengen, sehr groß werden können und eine serielle Implementierung nicht mehr in der Lage ist die Ergebnisse in einer annehmbaren Zeit zu berechnen.

Hierbei sollte sichergestellt sein, dass die Parallelisierung keinen zu starken negativen Einfluss auf die Qualität des resultierenden Modells hat.

Da GPUs sehr gut für die benötigten Berechnungen beim Trainieren eines MLPs geeignet sind, wurden zwei Parallelisierungsstrategien für die Verwendung mehrerer GPUs entwickelt. Die beiden Strategien unterscheiden sich darin, wie und wann eine Synchronisation zwischen den Prozessen, welche die GPUs steuern, stattfindet.

Für die Implementierung wurde die ArrayFire Bibliothek verwendet, welche eine einfache und einheitliche Schnittstelle für die Verwendung von verschiedenen Typen von GPUs und Beschleunigerkarten bereitstellt. Die Implementierung wurde in die sich in Entwicklung befindende JuML Bibliothek integriert, welche frei verfügbare Implementierungen für Machine Learning Algorithmen bereitstellen soll. Somit steht ist die Implementierung frei zur Verfügung. Es kamen Techniken wie CUDA-Aware MPI zum Einsatz um die Kommunikation zwischen den Prozessen zu optimieren. CUDA-Aware MPI ist jedoch nicht plattformunabhängig einsetzbar, führte jedoch zu einer massiven Verbesserung des Speedups.

Die in dieser Arbeit erstellte Implementierung wurde auf Datensätze aus dem Anwendungsgebiet Remote Sensing angewendet. Hierbei konnte gezeigt werden, dass der verwendete Algorithmus und die entwickelte Parallelisierungsstrategie für Datensätze aus diesem Anwendungsgebiet einsetzbar und geeignet ist. Es konnten sowohl gute Klassifizierungsgenauigkeiten als auch ein guter Speedup erreicht werden.

Sowohl der Speedup, als auch die Genauigkeit des Modells sind jedoch stark von den gewählten Parametern des Algorithmus abhängig und müssen individuell für jedes Problem einzeln bestimmt werden. Dies stellt eine große Schwierigkeit dar, da aufgrund der hohen Parameteranzahl ein Ausprobieren von allen möglichen Kombinationen nicht möglich ist. Diese Schwierigkeit sollte in Zukunft durch eine Reduktion der Anzahl der Parameter oder anderen Techniken vermieden werden.

Es konnten mit diesem Ansatz sowohl mit als auch ohne Parallelisierung gute Modelle erzeugt werden. Bei der Parallelisierungsstrategie, die den besseren Speedup liefert, hat die Anzahl der verwendeten Prozesse jedoch einen Einfluss auf die Klassifizierungsgenauigkeit des resultierenden Modells.

Literatur

[1] Siamak Khorram u. a. *Remote Sensing*. 1. Aufl. Springer-Verlag New York, 2012. DOI: 10.1007/978-1-4614-3103-9.

[2] M. A. Wong J. A. Hartigan. "Algorithm AS 136: A K-Means Clustering Algorithm". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), S. 100–108. ISSN: 00359254, 14679876. URL: http://www.jstor.org/stable/2346830.

[3] Martin Ester u. a. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. Hrsg. von Evangelos Simoudis, Jiawei Han und Usama M. Fayyad. AAAI Press, 1996, S. 226–231.

[4] D. E. Rumelhart, G. E. Hinton und R. J. Williams. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition". In: Hrsg. von David E. Rumelhart, James L. McClelland und CORPORATE PDP Research Group. Bd. 1. Cambridge, MA, USA: MIT Press, 1986. Kap. Learning Internal Representations by Error Propagation, S. 318–362. ISBN: 0-262-68053-X. URL: http://dl.acm.org/citation.cfm?id=104279.104293.

[5] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), S. 65–386.

[6] *Pavia University scene*. URL: http://www.ehu.eus/ccwintco/index.php?title=Hyperspectral_Remote_Sensing_Scenes#Pavia_University_scene.

[7] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, S. 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560.

[8] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (Mai 1988), S. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: http://doi.acm.org/10.1145/42411.42415.

[9] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. URL: http://openmp.org/.

[10] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.1*. 4. Juni 2015. URL: http://www.mpi-forum.org (besucht am 03. 08. 2016).

[11] CompuGreen. *The Green500 List - November 2015*. URL: http://green500.org/lists/green201511 (besucht am 03. 08. 2016).

[12] John E. Stone, David Gohara und Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *IEEE Des. Test* 12.3 (Mai 2010), S. 66–73. ISSN: 0740-7475. DOI: 10.1109/MCSE.2010.69. URL: http://dx.doi.org/10.1109/MCSE.2010.69.

[13] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. New York, NY, USA: Cambridge University Press, 2003. ISBN: 0521642981. URL: http://www.inference.phy.cam.ac.uk/itprnn/book.pdf.

[14] Javier Plaza u. a. "Parallel Classification of Hyperspectral Images Using Neural Networks". In: *Computational Intelligence for Remote Sensing*. Hrsg. von Manuel Graña und Richard J. Duro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 193–216. ISBN: 978-3-540-79353-3. DOI: 10.1007/978-3-540-79353-3_8. URL: http://dx.doi.org/10.1007/978-3-540-79353-3_8.

[15] *Neural network models (supervised) - Multi-layer Perceptron*. scikit-learn 0.18dev0 documentation. URL: http://scikit-learn.org/dev/modules/neural_networks_supervised.html (besucht am 25. 08. 2016).

[16] *Classification and regression - Multilayer perceptron classifier*. Spark 2.0.0 MLib Documentation. URL: https://spark.apache.org/docs/latest/ml-classification-regression.html#multilayer-perceptron-classifier (besucht am 25. 08. 2016).

[17] Apache Mahout. *Multilayer Perceptron*. URL: https://mahout.apache.org/users/classification/mlp.html.

[18] *Neural Network Toolbox*. MATLAB. URL: http://de.mathworks.com/products/neural-network/ (besucht am 25. 08. 2016).

[19] S. Nissen. *Implementation of a Fast Artificial Neural Network Library (fann)*. Techn. Ber. Department of Computer Science University of Copenhagen (DIKU), 2003. URL: http://fann.sf.net.

[20] *MXNet - Flexible and Efficient Library for Deep Learning*. URL: http://mxnet.io/ (besucht am 25. 08. 2016).

[21] Ian Wesley-Smith und Dr. Gabrielle Allen. "A Parallel Artificial Neural Network Implementation". In: *Proceedings of The National Conference On Undergraduate Research (NCUR) 2006*. The University of North Carolina at Asheville, 2006. URL: http://cactuscode.org/media/news/ncur20/Cactus_WesleySmith06.pdf (besucht am 16. 06. 2016).

[22] Lyle N. Long und Ankur Gupta. "Scalable Massively Parallel Artificial Neural Networks". In: *Journal of Aerospace Computing, Information, and Communication* 5.1 (Jan. 2008). URL: http://www.personal.psu.edu/lnl/papers/aiaa20057168.pdf (besucht am 16. 06. 2016).

[23] Hans P. Graf u. a. "Parallel Support Vector Machines: The Cascade SVM". In: *Advances in Neural Information Processing Systems 17*. Hrsg. von L. K. Saul, Y. Weiss und L. Bottou. MIT Press, 2005, S. 521–528. URL: http://papers.nips.cc/paper/2608-parallel-support-vector-machines-the-cascade-svm.pdf.

[24] Bernhard E. Boser, Isabelle M. Guyon und Vladimir N. Vapnik. “A Training Algorithm for Optimal Margin Classifiers”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT ’92. Pittsburgh, Pennsylvania, USA: ACM, 1992, S. 144–152. ISBN: 0-89791-497-X. DOI: 10.1145/130385.130401. URL: http://doi.acm.org/10.1145/130385.130401.

[25] George Dahl, Alan McAvinney und Tia Newhall. “Parallelizing Neural Network Training for Cluster Systems”. In: *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*. PDCN ’08. Innsbruck, Austria: ACTA Press, 2008, S. 220–225. ISBN: 978-0-88986-714-7. URL: http://dl.acm.org/citation.cfm?id=1722252.1722292.

[26] The HDF Group. *Hierarchical Data Format, version 5*. 1997-NNNN. URL: http://www.hdfgroup.org/HDF5/.

[27] Anders Krogh und John A. Hertz. “A Simple Weight Decay Can Improve Generalization”. In: *Advances in Neural Information Processing Systems 4*. Hrsg. von J. E. Moody, S. J. Hanson und R. P. Lippmann. Morgan-Kaufmann, 1992, S. 950–957. URL: http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf.

[28] Pavan Yalamanchili u. a. *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*. Atlanta, 2015. URL: https://github.com/arrayfire/arrayfire.

[29] *Performance of ArrayFire JIT Code Generation*. 18. Feb. 2015. URL: http://arrayfire.com/performance-of-arrayfire-jit-code-generation/ (besucht am 22. 06. 2016).

[30] *ArrayFire - Implement Shuffle*. Kommentar zu Issue von shehzan10. URL: https://github.com/arrayfire/arrayfire/issues/1072#issuecomment-214001295.

[31] Jiri Kraus. *An Introduction to CUDA-Aware MPI*. 13. März 2013. URL: https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/ (besucht am 30. 06. 2016).

[32] *CUDA-Aware MPI with `device<>()` pointer needs `af::sync()`*. Issue auf GitHub. 21. Juli 2016. URL: https://github.com/arrayfire/arrayfire/issues/1510 (besucht am 21. 07. 2016).

[33] Jülich Supercomputing Centre of Forschungszentrum Jülich. *JUBE Benchmarking Environment*. 3. März 2016. URL: http://www.fz-juelich.de/jsc/jube (besucht am 25. 07. 2016).

[34] Forschungszentrum Jülich. *JURECA*. URL: http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html (besucht am 16. 06. 2016).

[35] TOP500.org. *Top500 List - November 2015*. URL: http://www.top500.org/list/2015/11/ (besucht am 16. 06. 2016).

[36] Top500.org. *JURECA*. URL: http://www.top500.org/system/178718 (besucht am 16. 06. 2016).

[37] CompuGreen. *The Green500 List - November 2015 von 101 bis 200*. URL: http://green500.org/lists/green201511&green500from=101&green500to=200 (besucht am 16. 06. 2016).

[38] CompuGreen. *The Green500 List - Juni 2016 von 101 bis 200*. URL: http://green500.org/lists/green201606&green500from=101&green500to=200 (besucht am 16. 06. 2016).

[39] *Salinas*. URL: http://hdl.handle.net/11304/47a86ef2-497a-11e4-81ac-dcbd1b51435e.

[40] *Salinas scene*. URL: http://www.ehu.eus/ccwintco/index.php?title=Hyperspectral_Remote_Sensing_Scenes#Salinas_scene.

[41] *Indian pines: raw and processed*. 4. Feb. 2015. URL: http://hdl.handle.net/11304/7e8eec8e-ad61-11e4-ac7e-860aa0063d1f.

[42] Gabriele Cavallaro u. a. "On understanding big data impacts in remotely sensed image classification using support vector machine methods". In: *IEEE journal of selected topics in applied earth observations and remote sensing* 8.10 (2015), S. 4634–4646.

[43] *Rome data set OK*. Mai 2014. URL: http://hdl.handle.net/11304/4615928c-e1a5-11e3-8cd7-14feb57d12b9 (besucht am 07. 09. 2016).

[44] Gabriele Cavallaro u. a. "Smart data analytics methods for remote sensing applications". In: *2014 IEEE Geoscience and Remote Sensing Symposium*. IGARSS 2014 - 2014 IEEE International Geoscience und Remote Sensing Symposium, Quebec City (Canada), 13 Jul 2014 - 18 Jul 2014. IEEE, 13. Juli 2014, S. 1405–1408. DOI: 10.1109/IGARSS.2014.6946698. URL: https://juser.fz-juelich.de/record/172735.

[45] *AVIRIS - Airborne Visible / Infrared Imaging Spectrometer*. URL: http://aviris.jpl.nasa.gov/ (besucht am 24. 08. 2016).