

Frankfurt University of Applied Sciences

– Fachbereich 2: Informatik und Ingenieurwissenschaften –

Nebenläufige Algorithmen im maschinellen Lernen: Analyse, Implementierung und vergleichende Untersuchungen zur Parallelisierung einer Bibliothek für künstliche neuronale Netze

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt am 21. Mai 2024 von
Luca Andrea John Vinciguerra

Matrikelnummer: 1296334

Referent : Prof. Dr. Thomas Gabel
Korreferent : Prof. Dr. Christian Baun

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Frankfurt, 21. Mai 2024



**Luca Andrea John
Vinciguerra**

Inhaltsverzeichnis

1	Einleitung	2
1.1	Aufgabenstellung	3
1.2	Gliederung	4
2	Grundlagen	5
2.1	Grundprinzipien von neuronalen Netzen	5
2.2	Einführung in die Parallelisierung	7
2.2.1	Vor- und Nachteile von Parallelisierung	8
2.2.2	Parallelisierung in vorwärtsgerichteten Netzen	9
2.2.3	Implementierung von Parallelisierungstechniken	10
2.2.4	Auswirkungen auf die Leistungsfähigkeit	11
3	Implementierung der Parallelisierung	12
3.1	Erläuterung des n++-Simulatorkerns	12
3.2	Bestehender Code	14
3.3	Datensatz	16
3.4	Voraussetzungen	16
3.4.1	Entfernung von geteilten Speicherzugriffen	17
3.4.2	Verlagerung der zu parallelisierenden Routine	18
3.5	Vorstellung der Implementierung	19
3.5.1	Verwendung von threadsicheren Funktionen	20
3.5.2	Entfernung globaler Variablen	22
3.5.3	Verwendung eines Threadpools	23
4	Experimentelle Untersuchungen	26
4.1	Testumgebung	26
4.2	Getestete Hardware	27

4.3	Benchmark Script	29
4.4	Ergebnisse	31
4.5	Auswertung	38
5	Fazit und Ausblick	40
A	Programmcode	41
	Abbildungsverzeichnis	50
	Literatur	52

Kurzfassung

todo: Die Kurzfassung muss noch geschrieben werden

Kapitel 1

Einleitung

Die Informatik schöpft oft Inspiration aus der Natur, sei es durch die Nachahmung tierischer Bewegungen bei Robotern, der Organisation von Multiagentensystemen in vogelähnlichen Schwärmen oder der Anwendung evolutionärer Algorithmen zur Simulation natürlicher Prozesse. Doch eines der faszinierendsten Phänomene der Natur ist das menschliche Gehirn und seine Fähigkeit, aus Erfahrung zu lernen. Dieses komplexe Organ beschäftigt Wissenschaftler seit langem, und die Suche nach Möglichkeiten, seine Lernfähigkeit zu simulieren, hat zu bedeutenden Fortschritten geführt.

Ein zentrales Element im Gehirn ist das Neuron, auch Nervenzelle genannt, das als Grundbaustein für die Informationsverarbeitung dient. Das Menschliche Gehirn besitzt circa 10 Milliarden Neuronen. Jedes dieser Neuronen besteht aus einem Zellkörper, mehreren Dendriten und einem Axon. Die Dendriten empfangen elektrische Signale von davor geschalteten Neuronen und fungieren somit als Eingangsebene für Informationen. Diese eingehenden Signale werden zum Zellkörper weitergeleitet, wo sie aufsummiert werden. Wird ein bestimmter Schwellenwert überschritten, leitet das Neuron das elektrische Potentiale über das Axon weiter, welches es elektrochemisch an nachgeschaltete Neuronen über deren Dendriten weiterleitet. Das Axon agiert somit als Ausgangsebene für Informationen des Neurons. Durch diese Kettenreaktion können im Gehirn somit komplexe Sachverhalte verarbeitet werden [1].

Inspiziert von diesem biologischen Vorbild entwickelte Frank Rosenblatt 1958 das Mo-

dell des Perzeptrons - ein künstliches Neuron, welches die Grundlage für die Entwicklung heutiger künstlicher neuronaler Netzwerke darstellt [2]. Diese Netzwerke können mithilfe von Lernalgorithmen trainiert werden, um vielfältige Probleme zu bewältigen, welche mit konventionellen Computeralgorithmen, wenn überhaupt, nur schwer zu lösen sind. Mittlerweile tragen künstliche neuronale Netzwerke mitunter in verschiedensten Branchen zu der Realisierung von Software in vielfältigen Anwendungsgebieten bei.

Aufgrund der großen Menge an Daten und benötigten Rechenleistung für die Darstellung von neuronalen Netzwerken in Computern ist die Frage der Skalierbarkeit und Effizienz der neuronalen Netzwerke von entscheidender Bedeutung. Insbesondere die Verarbeitung großer Datenmengen erfordert effiziente Algorithmen und Techniken zur Parallelisierung, um den gegebenen Anforderungen beispielsweise im Bezug auf Latenz gerecht zu werden. In dieser Arbeit wird daher die Parallelisierung von neuronalen Netzen thematisiert und untersucht, wie diese Techniken die Leistung und Effizienz beeinflussen.

1.1 Aufgabenstellung

In Anbetracht der stagnierenden Entwicklung der Taktrate aufgrund des Annäherns an das physikalische Limit konnten in den letzten Jahren keine großen Verbesserungen in der Einkernleistung erzielt werden. Deshalb setzten Prozessorhersteller weit verbreitet auf Mehrkernprozessoren, um Leistungssteigerungen zu ermöglichen [3]. Angesichts der möglichen Leistungssteigerung durch effizientes Nutzen aller verfügbaren Kerne ist es von besonderem Interesse, die Leistung der bestehenden n++-Bibliothek für maschinelles Lernen durch Parallelisierung zu verbessern [4]. Die n++-Bibliothek ist in C++ implementiert, weshalb die Parallelisierung mithilfe von Threads realisiert werden soll. Eine zentrale Herausforderung besteht darin, geeignete Stellen in der Bibliothek als auch in Anwendungsprogrammen zu identifizieren, die von der Parallelisierung profitieren könnten. Hierbei werden vorhandene Vorarbeiten [5] und Implementierungen als Vergleich herangezogen und gegebenenfalls optimiert.

Diese Arbeit zielt darauf ab, die potenziell erzielten Leistungsverbesserungen durch die Parallelisierung zu untersuchen und zu quantifizieren. Durch die Implementierung der

Parallelisierung und die anschließende Ausführung von Algorithmen der Vorarbeit kann der Effekt der Parallelisierung auf die Leistung von Programmen, welche die n++-Bibliothek verwenden, evaluiert werden. Zudem werden die Auswirkungen verschiedener Parameter und Konfigurationen im Zusammenhang mit der Parallelisierung analysiert, um ein umfassendes Verständnis der Leistungsverbesserung durch Parallelisierung zu erlangen.

Insgesamt strebt diese Arbeit danach, nicht nur die technische Umsetzung der Parallelisierung zu präsentieren, sondern auch deren Auswirkungen auf die Leistungsfähigkeit der n++-Bibliothek für maschinelles Lernen zu untersuchen und zu bewerten.

1.2 Gliederung

Nach einer Einführung in die Thematik und die Vorgehensweise der Arbeit im ersten Kapitel, folgt im zweiten Kapitel die Vermittlung der notwendigen Grundlagen. Dazu gehören die Grundprinzipien neuronaler Netze sowie die Grundlagen der Parallelisierung. Das dritte Kapitel beschreibt die Implementierung der Parallelisierung im n++-Simulator. Nach der Erläuterung des n++-Simulatorkerns und der Analyse des bestehenden Codes werden die Voraussetzungen für die Parallelisierung geschaffen und die Implementierung detailliert vorgestellt. Im vierten Kapitel werden die experimentellen Untersuchungen zur Evaluation der Parallelisierungstechniken präsentiert. Die Testumgebung, die Testmethodik und die Ergebnisse der experimentellen Untersuchungen werden ausführlich beschrieben und analysiert. Abschließend fasst das Fazit die wichtigsten Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Forschungsarbeiten.

Kapitel 2

Grundlagen

Dieses Kapitel legt die Grundlagen für künstliche neuronale Netze dar, indem es detailliert auf ihre Struktur und Funktionsweise eingeht, mit Fokus auf vorwärtsgerichtete Netze. Dabei wird ein umfassender Überblick über die potenziellen Anwendungsbereiche von künstlichen neuronalen Netzen gegeben.

Des Weiteren wird die Thematik der Parallelisierung sowohl im allgemeinen Kontext als auch speziell im Zusammenhang mit neuronalen Netzen erläutert. Es werden die Vor- und Nachteile dieser Technik beleuchtet und die gängigsten Methoden zur Parallelisierung von Berechnungen in neuronalen Netzen werden ausführlich diskutiert. Die Nutzung von Grafikprozessoren (GPUs) für parallele Berechnungen wird dabei angesprochen, jedoch liegt der Fokus auf den grundlegenden Prinzipien der Parallelisierung von neuronalen Netzen und deren Implementierung mittels Thread- und Prozessparallelisierung.

2.1 Grundprinzipien von neuronalen Netzen

Neuronale Netze sind ein wesentlicher Bestandteil des maschinellen Lernens und der künstlichen Intelligenz. Sie sind inspiriert von der Funktionsweise des menschlichen Gehirns und bestehen aus einer Ansammlung miteinander verbundener Knoten, die genau wie beim menschlichen Gehirn als Neuronen bezeichnet werden [6].

Ein mehrlagiges neuronales Netz ist in verschiedene Schichten organisiert, die je-

weils spezifische Funktionen erfüllen. Die erste Schicht wird oft als Eingangsschicht bezeichnet und empfängt die Rohdaten oder Merkmale, die dem Netz präsentiert werden. Diese Daten werden dann durch das Netz weitergeleitet, wobei jede Schicht eine spezifische Transformation durchführt. Zwischen der Eingangsschicht und der Ausgangsschicht können mehrere versteckte Schichten vorhanden sein. Diese versteckten Schichten sind entscheidend für die Fähigkeit des Netzes, komplexe Muster zu lernen und abstrakte Merkmale zu extrahieren. Jede Schicht lernt auf unterschiedlichen Abstraktionsebenen und trägt zur schrittweisen Verbesserung der Leistung des Netzes bei.

Die Funktionsweise eines neuronalen Netzes lässt sich allgemein in zwei Hauptphasen unterteilen: Vorwärtspropagierung und Rückwärtspropagierung. Während der Vorwärtspropagierung fließen die Eingabedaten vorwärts durch das Netz, beginnend mit den Neuronen der Eingabeschicht, welche die Rohdaten empfangen, und endend mit den Neuronen der Ausgabeschicht, welche die Vorhersagen oder Klassifikationen des Netzes abbilden. Jedes Neuron ist mit anderen Neuronen verbunden, und diese Verbindungen sind mit Gewichten versehen, die die Stärke der Verbindung zwischen den Neuronen darstellen [6].

Während der Vorwärtspropagierung durchläuft jede Eingabe eine Reihe von Schichten im Netz, wobei jede Schicht aus einer bestimmten Anzahl von Neuronen besteht. Jedes Neuron einer Schicht erhält Eingaben von den Neuronen der vorherigen Schicht, multipliziert diese Eingaben mit den entsprechenden Gewichten und summiert sie, wie in Gleichung 2.1 dargestellt. Anschließend wird wie in Gleichung 2.2 definiert eine Aktivierungsfunktion auf die gewichtete Summe angewendet, um die Ausgabe des Neurons zu berechnen, die dann an die Neuronen der nächsten Schicht weitergeleitet wird.

$$Eingabe = w_0b + \sum_{i=1}^n w_i x_i \quad (2.1)$$

$$Ausgabe = f_a(Eingabe) \quad (2.2)$$

w_0 : zugehöriges Gewicht; b : Schwellenwert; w_i : zugehörige Gewichte; x_i : Eingabewerte

f_a : Aktivierungsfunktion

Die Ausgangsschicht liefert schließlich die Ergebnisse der Netzberechnungen, sei es in Form einer Klassifikation, Regression oder einer anderen Art der Informationsverarbeitung, je nach den Anforderungen der spezifischen Anwendung. Durch die Strukturierung des Netzes in Schichten und die Festlegung spezifischer Funktionen für jede Schicht kann das neuronale Netz effizient Informationen verarbeiten und komplexe Probleme lösen.

Die Rückwärtspropagierung ist der Prozess, bei dem das Netz lernt, indem es seine Gewichte entsprechend der Fehler zwischen den tatsächlichen und den vorhergesagten Ausgaben anpasst. Dies geschieht durch die Berechnung von Gradienten zum Beispiel mit Hilfe des Backpropagation-Algorithmus und die Anpassung der Gewichte mithilfe eines Optimierungsalgorithmus wie dem Gradientenabstiegsverfahren.

add citation

2.2 Einführung in die Parallelisierung

Die Parallelisierung stellt einen zentralen Ansatz dar, um die Leistungsfähigkeit von Computersystemen zu steigern, insbesondere angesichts der Tatsache, dass moderne CPUs und GPUs über eine wachsende Anzahl von Kernen verfügen. Kernpunkt der Parallelisierung ist die simultane und unabhängige Ausführung von Aufgaben oder Berechnungen, anstatt einer sequenziellen Abfolge. Dieser Ansatz findet breite Anwendung in verschiedenen Bereichen wie High-Performance-Computing (HPC), Datenverarbeitung, Simulationen, künstlicher Intelligenz und weiteren [7].

Eine Vielzahl von Herangehensweisen zur Parallelisierung existiert, die abhängig von der Problemstellung und der verfügbaren Hardware eingesetzt werden können. Die Task-Parallelisierung zielt darauf ab, Aufgaben auf mehrere Prozessoren oder Kerne zu verteilen. Insbesondere für Anwendungen mit vielen simultan auszuführenden Aufgaben wie parallele Suchalgorithmen oder Simulationen von physikalischen Systemen eignet sich diese Art der Parallelisierung besonders [7].

Ein weiterer Ansatz ist die Datenparallelisierung, bei der ein Problem in kleinere Teile zerlegt wird, die jeweils auf unterschiedlichen Datensätzen operieren. Dieser Ansatz ist besonders effektiv für Anwendungen, die eine simultane Verarbeitung großer Datenmengen erfordern, wie beispielsweise Bildverarbeitung oder maschinelles Lernen [7].

Es ist jedoch zu betonen, dass nicht alle Probleme gleichermaßen für eine Parallelisierung geeignet sind. Manche Probleme beinhalten intrinsische Abhängigkeiten oder Sequenzialität, die eine effektive Parallelisierung erschweren oder gar unmöglich machen.

2.2.1 Vor- und Nachteile von Parallelisierung

Die Parallelisierung bietet eine Vielzahl von Vorteilen, die zur Leistungssteigerung von Computersystemen beitragen. Einer der offensichtlichsten Vorteile ist die Verbesserung der Ausführungsgeschwindigkeit von Programmen und Berechnungen durch die gleichzeitige Ausführung von Aufgaben oder die Verarbeitung von Daten auf mehreren Prozessoren oder Kernen. Diese beschleunigte Ausführung ist insbesondere bei rechenintensiven Anwendungen von Vorteil [7].

Ein zusätzlicher Vorteil der Parallelisierung liegt in ihrer Skalierbarkeit, da Aufgaben oder Daten auf mehrere Ressourcen aufgeteilt werden können, um Systeme leichter an wachsende Anforderungen anzupassen. Dies ermöglicht es, die Leistungsfähigkeit von Systemen flexibel zu erweitern, ohne dass eine komplette Neuentwicklung erforderlich ist [7]. Des Weiteren kann die Parallelisierung die Auslastung von Ressourcen optimieren und Engpässe reduzieren, indem sie Prozessoren oder andere Hardware-Ressourcen effizient nutzt. Dies trägt dazu bei, die Gesamtleistung des Systems zu verbessern. Eine effiziente Nutzung der verfügbaren Hardware ist nicht nur im Bezug der Systemleistung vorteilhaft zu bewerten, sondern ermöglicht es auch, mehr Arbeit auf weniger Systemen auszuführen, da das volle Leistungspotenzial aller Systeme ausgenutzt wird [7]. So werden auch Kosten gesenkt.

Trotz dieser Vorteile gibt es auch einige Nachteile und Herausforderungen bei der Implementierung von Parallelisierung. Ein wichtiger Aspekt sind die erhöhten Anforderungen an die Programmierung und die Algorithmuskonzeption, da die Entwicklung paralleler Algorithmen und die Verwaltung paralleler Prozesse spezifisches Fachwissen erfordern. Darüber hinaus können Probleme wie Datenabhängigkeiten, Wettlaufsituationen und Synchronisationskonflikte auftreten, die die Entwicklung und Fehlerbehebung erschweren. Parallele Implementierungen sind fast ausschließlich komplexer als ihre sequenziellen Pendants. Es gibt einige Ansätze, die Parallelisierung dem Programmierer ge-

genüber transparent zu gestalten [8], jedoch stellten sich diese Bemühungen größtenteils als erfolglos heraus [9], da sie oft unvorhersehbare Verhaltensweisen mit sich bringen und das Debugging erschweren.

Ein weiterer Nachteil ist die potenzielle Zunahme des Energieverbrauchs, insbesondere wenn die Parallelisierung nicht effizient implementiert ist. Dies ist besonders relevant in Umgebungen, in denen Energieeffizienz ein wichtiges Anliegen ist, wie beispielsweise in mobilen Geräten oder Rechenzentren.

2.2.2 Parallelisierung in vorwärtsgerichteten Netzen

Für die parallele Ausführung des Trainings mehrerer Netze unabhängig voneinander spielt die Thread- und Prozessparallelisierung eine bedeutende Rolle. Diese Techniken bieten Mechanismen, um das Training der Netze auf mehrere Threads oder Prozesse aufzuteilen, was die Effizienz und Geschwindigkeit des Trainings verbessern kann [7].

Thread-Parallelisierung bezieht sich auf die Aufteilung des Trainingsprozesses eines Netzes in mehrere Threads, die gleichzeitig auf einem einzigen Prozessorkern oder auf mehreren Kernen eines Mehrkernprozessors ausgeführt werden können. In diesem Szenario ermöglicht die Thread-Parallelisierung das gleichzeitige Training mehrerer Netze, wobei jeder Thread sich auf das Training eines bestimmten Netzes konzentriert. Dies kann die Gesamttrainingsdauer reduzieren und die Auslastung der verfügbaren Prozessorressourcen optimieren [7].

Prozessparallelisierung hingegen umfasst die Aufteilung des Trainingsprozesses in mehrere unabhängige Prozesse, die auf verschiedenen Prozessorkernen oder sogar auf verschiedenen physikalischen Maschinen ausgeführt werden können. Bei der Prozessparallelisierung werden die Trainingsvorgänge mehrerer Netze auf separaten Prozessen ausgeführt, was eine hochgradig parallele Verarbeitung und Skalierbarkeit über mehrere Computerknoten hinweg ermöglicht. Die Kommunikation zwischen den Prozessen kann über verschiedene Mechanismen wie Sockets, Messaging-Systeme oder gemeinsam genutzte Speicherbereiche erfolgen [7].

Die Parallelisierung des Trainings und Testens von neuronalen Netzen mittels Grafik-

prozessoren (GPUs) hat sich als eine zentrale Methode zur Steigerung der Effizienz und Geschwindigkeit in der modernen maschinellen Lernforschung etabliert. GPUs sind besonders gut für die Verarbeitung der massiven parallelen Berechnungen geeignet, die beim Training großer neuronaler Netze erforderlich sind [10]. GPUs enthalten eine große Anzahl von Kernen, die für das gleichzeitige Ausführen vieler Operationen konzipiert sind. Dies steht im Gegensatz zu den wenigen, dafür aber leistungstärkeren Kernen einer CPU. Bei neuronalen Netzen bestehen die Berechnungen häufig aus Matrix-Multiplikationen und anderen linear-algebraischen Operationen, die sich gut parallelisieren lassen [10]. Auf diese Parallelisierungstechnik wird in dieser Arbeit jedoch nicht weiter eingegangen, da der Fokus auf der Thread-Parallelisierung liegt.

2.2.3 Implementierung von Parallelisierungstechniken

Datenparallelismus ist eine der am häufigsten verwendeten Techniken zur Parallelisierung neuronaler Netze. Der Datensatz wird in kleinere Datensätze aufgeteilt, die gleichzeitig auf verschiedenen Recheneinheiten verarbeitet werden. Jeder Thread führt die gleichen Operationen auf unterschiedlichen Datenpartitionen durch, und die Ergebnisse werden anschließend kombiniert [11].

Beim Modellparallelismus wird das neuronale Netz selbst in verschiedene Teile aufgeteilt, die gleichzeitig auf verschiedenen Recheneinheiten ausgeführt werden. Diese Parallelisierungstechnik wird zum Beispiel häufig verwendet, wenn das Modell zu groß ist, um auf einer einzelnen GPU oder einem einzelnen Knoten zu laufen. Jede Recheneinheit berechnet einen Teil des Modells, und die Ergebnisse werden zur Berechnung der nächsten Schicht kombiniert.

Für die Implementierung von neuronalen Netzen kommen heutzutage häufig Bibliotheken wie PyTorch oder TensorFlow zum Einsatz. Diese bieten Implementierungen von vielen Algorithmen für neuronale Netze, die von den angesprochenen Parallelisierungstechniken Gebrauch machen, was die Implementierung von parallelisierten und performanten neuronalen Netzen deutlich vereinfacht hat [11]. Diese Bibliotheken nehmen dem Entwickler viel Arbeit ab, erlauben jedoch keine genaue Kontrolle über die Parallelisierung.

2.2.4 Auswirkungen auf die Leistungsfähigkeit

Ein wesentlicher Vorteil besteht darin, dass durch die parallele Ausführung mehrerer Netze (mit verschiedenen Seeds) gleichzeitig eine Vielzahl von Trainingsdurchläufen durchgeführt werden kann. Dies ermöglicht es, eine breite Palette von Modellen zu trainieren und verschiedene hyperparameterabhängige Variationen zu erkunden, um letztendlich das optimale Modell zu identifizieren. Durch die gleichzeitige Ausführung dieser Trainingsläufe können Entwickler Zeit sparen und schneller zu aussagekräftigen Ergebnissen gelangen.

Des Weiteren bietet die parallele Ausführung die Möglichkeit, Inferenzoperationen gleichzeitig durchzuführen. Mehrere Eingaben können gleichzeitig an duplizierte Netze weitergeleitet werden, um eine simultane Auswertung zu ermöglichen. Dies beschleunigt nicht nur den Inferenzprozess erheblich, sondern ermöglicht auch eine effizientere Nutzung der verfügbaren Hardwareressourcen.

Ein weiterer Vorteil besteht in der verbesserten Skalierbarkeit der Anwendung. Durch die Nutzung von Thread- oder Prozessparallelisierung kann die Anwendung problemlos auf mehreren Rechenknoten oder sogar in Cloud-Umgebungen skaliert werden. Dies ermöglicht es, die Trainings- und Inferenzkapazitäten je nach Bedarf flexibel anzupassen und die Gesamtperformance der Anwendung zu optimieren.

Texte o
dentlich
verbin-
den

Kapitel 3

Implementierung der Parallelisierung

In diesem Kapitel wird zunächst die n++-Bibliothek und der bestehende Code vorgestellt, als auch auf den verwendeten Datensatz eingegangen. Anschließend werden die konzeptionellen Voraussetzungen erläutert und daraufhin wird die Implementierung detailliert vorgestellt. Dabei wird auch die Struktur der neuen Implementierung mit der der Vorarbeit verglichen.

3.1 Erläuterung des n++-Simulatorkerns

N++ ist ein Simulator für neuronale Netze, der als Forschungsprojekt an der Universität Karlsruhe entwickelt wurde. Die Software ermöglicht die Simulation mehrerer neuronaler Netze und strebt danach, dem Anwender eine einfache Erweiterung der Grundfunktionen sowie eine benutzerfreundliche Schnittstelle für Anwendungsprogramme bereitzustellen [4].

Die Bibliothek n++ ist in C++ verfasst. Da sie seit über 20 Jahren besteht, verwendet sie größtenteils keine modernen C++-Features, unter anderem auch um die Kompatibilität mit C beizubehalten, da der Kern des Simulators auf dieser älteren Sprachversion

aufbaut. Oft werden im Quellcode Funktionen der Standardbibliothek von C denen von C++ vorgezogen.

n++ ermöglicht es dem Benutzer, die Topologie des neuronalen Netzes zu spezifizieren und es an die spezifischen Anforderungen anzupassen. Hierbei können Parameter wie die Anzahl der Schichten, die Größe der Schichten sowie die Dimensionen der Ein- und Ausgabeschichten festgelegt werden [12]. Nach der Konfiguration des Netzes können Eingabemuster durch Vorwärtspropagation propagiert werden. Die resultierenden Ausgaben können abgerufen werden, und optional kann durch eine Rückführung des Fehlervektors ein Lernprozess des Netzes durchgeführt werden, wobei die Gewichte automatisch angepasst werden. Generierte Netze können in Dateien gespeichert werden, um sie zu einem späteren Zeitpunkt wiederzuverwenden, insbesondere für reproduzierbare Experimente.

In Codeausschnitt 3.1, welcher eine vereinfachte Form des Beispielnetzes aus der n++-Dokumentation darstellt [12], wird ein Beispielnetz mit drei Schichten erstellt. Die Eingabeschicht hat dabei zwei Parameter, die Ausgabeschicht drei, und die versteckte Schicht hat vier Parameter. Es ist ersichtlich, dass die n++-Bibliothek das einfache Austauschen von Updatefunktionen unterstützt. In den Zeilen 16 und 17 wird die Updatefunktion dynamisch auf Backpropagation gesetzt, was ohne großen Aufwand möglich ist.

```
1  #include "n++.h"
2
3  #define INPUTS 2
4  #define OUTPUTS 3
5  #define LAYERS 3
6
7  int main() {
8      Net net;
9      // Schichten des Netzes erstellen und miteinander verbinden
10     int layerNodes[LAYERS] = {INPUTS, 4, OUTPUTS};
11     net.create_layers(LAYERS, layerNodes);
12     net.connect_layers();
13     // Gewichte mit Zufallszahlen zwischen 0 und 0,5 initialisieren
14     net.init_weights(0, 0.5);
15     // Updatefunktion auf Rückwärtspropagation setzen
16     float uparams[5] = {0.1, 0.9, 0, 0, 0};
17     net.set_update_f(BP, uparams);
18 }
```

Abbildung 3.1: Vereinfachte Form des Beispielnetzes aus der n++-Dokumentation welche ein Netz mit 3 Schichten erstellt

3.2 Bestehender Code

Als bestehender Code wird ein Experiment aus der Bachelorarbeit von Artur Brening [5] betrachtet. In dieser Bachelorarbeit wurde der Gradientenabstieg zum Trainieren neuronaler Netze implementiert. Dabei wurden viele Experimente mit verschiedenen Datensätzen und Lernmethoden zum Vergleich implementiert. Eines dieser Experimente wird als Grundlage für diese Bachelorarbeit übernommen.

Ausgewählt wurde das Experiment “MAGIC_BP”, welches Rückwärtspropagierung nutzt. In dem Experiment wird der Magic Datensatz verwendet, auf welchen im Abschnitt 3.3 eingegangen wird. Es wird unter Einbindung der n++-Bibliothek ein neuronales Netz erstellt und zunächst ein festgelegter Teil des Datensatzes als Trainingsdatensatz genommen. Jede dieser Trainingsdaten wird zunächst vorwärts durch das neuronale Netz propagiert. Anschließend findet eine Fehlerbestimmung der resultierenden Ausgabe des Netzes statt, und der Fehlervektor rückwärtspropagiert. Nachdem alle Datensätze aus

dem Trainingsdatensatz vor- und rückwärtspropagiert wurden, definiert das Programm den verbleibende Teil des Datensatzes als Testdatensatz. Genau wie die Trainingsdatensätze wird jeder Datensatz zuerst vor- und dann rückwärts propagiert. Nach jedem Datensatz wird hier jedoch geprüft, ob die Klassifizierung des Netzes korrekt ist, um einen Überblick über den Anteil korrekt klassifizierter Datensätze zu schaffen. Dieser Prozess wird für 10000 Epochen, also 10000 mal, über den gesamten Datensatz ausgeführt um das Netz lernen zu lassen.

Der beschriebene Ablauf läuft sequenziell für 10 verschiedene Netze mit jeweils verschiedenen Startseed ab, um das Verhalten mit verschiedenen Zufallszahlen zu überprüfen. Nachdem alle 10 Netze trainiert und getestet wurden, werden die Testergebnisse analysiert und in eine Datei geschrieben, um sie weiter verwerten zu können.

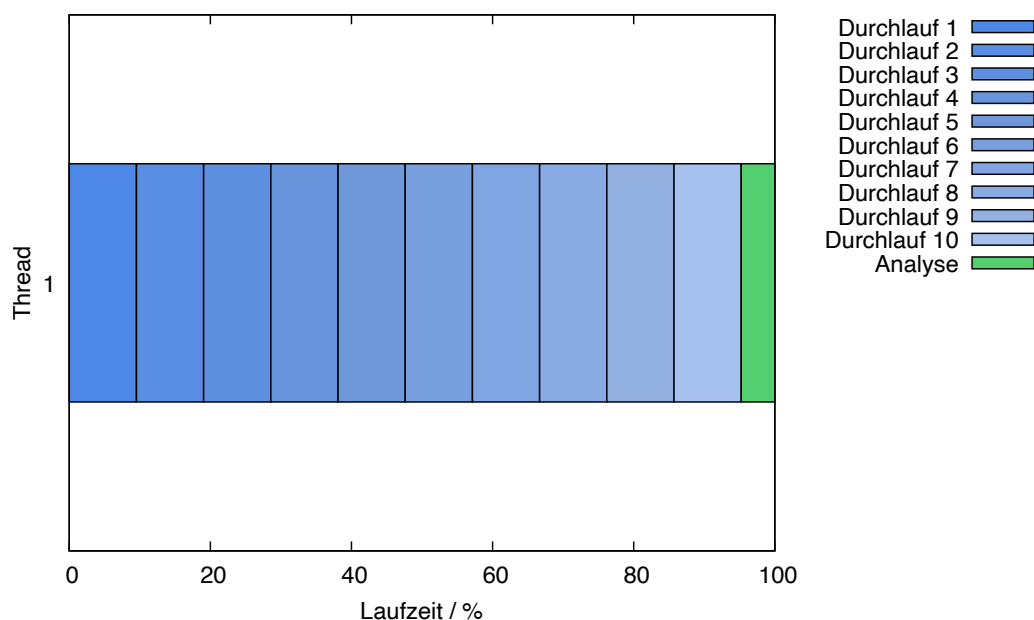


Abbildung 3.2: Schematischer Ablauf des bestehenden Programms. In jedem Durchlauf wird ein neuronales Netz mit 10000 Epochen trainiert und anschließend getestet. In der Analyse werden die Ergebnisse aller Netze zusammengetragen und in eine Datei geschrieben.

Wie in Abbildung 3.2 ersichtlich laufen alle Netze, auf der Grafik in blau dargestellt, auf einem Thread. Anschließend wird die Analyse, in grün dargestellt, ausgeführt. In dieser Arbeit werden die Trainings- und Testdurchläufe parallelisiert, was bedeutet,

dass potenziell 10 Netze gleichzeitig trainiert und getestet werden können. Dies könnte möglicherweise zu großen Leistungsverbesserungen führen, da das Training und Testen der Netze den Großteil der Laufzeit einnimmt, und die Analyse nur wenige Sekunden in Anspruch nimmt.

3.3 Datensatz

Der verwendete Datensatz MAGIC Gamma Telescope aus dem UCI Machine Learning Repository ist eine bedeutende Ressource für die Forschung im Bereich der Gamma-Teleskopie. Er enthält eine Vielzahl von Beobachtungen, die von Gammastrahlen-Teleskopen gemacht wurden. Jede Beobachtung wird durch eine Reihe von Merkmalen beschrieben, die aus den gemessenen Eigenschaften der Gammastrahlen stammen. Das Hauptziel bei der Verwendung dieses Datensatzes ist die Klassifizierung von Beobachtungen in verschiedene Kategorien oder Klassen. Durch die Anwendung von Klassifizierungsalgorithmen können Muster und Zusammenhänge in den Daten identifiziert werden, was wiederum dazu beitragen kann, das Verständnis der Gammastrahlenphänomene im Universum zu vertiefen [13]. Das UCI Machine Learning Repository ist eine bekannte Datenbank, die eine Vielzahl von Datensätzen für die Forschung und Entwicklung im Bereich des maschinellen Lernens bereitstellt. Die Daten sind kostenfrei verfügbar und können für verschiedene Zwecke verwendet werden. Der Datensatz enthält insgesamt 19020 Beobachtungen [13].

3.4 Voraussetzungen

Für eine erfolgreiche Parallelisierung des Anwendercodes ist eine umfassende Analyse und Modifikation desselben unerlässlich. Dieser Abschnitt diskutiert die grundlegenden Voraussetzungen, die vor der Implementierung von Parallelisierungsstrategien berücksichtigt werden müssen. In erster Linie erfordert die Parallelisierung die Identifizierung und Beseitigung von Abhängigkeiten innerhalb des Algorithmus sowie die Anpassung der Implementierung, um die Effizienz und Skalierbarkeit auf mehreren Prozessoren oder Rechenkernen zu gewährleisten [14].

3.4.1 Entfernung von geteilten Speicherzugriffen

Geteilte Speicherzugriffe, auch Shared Memory genannt, häufig realisiert durch globale Variablen im Quellcode, ermöglichen es verschiedenen Teilen eines Programms, auf dieselben Daten zuzugreifen. Diese Abhängigkeit ermöglicht es dem Programmierer, komplexe Algorithmen simpel zu implementieren. Während dies in einer sequenziellen Umgebung einigermaßen gut funktionieren kann, können Probleme auftreten, wenn versucht wird, solche Konstrukte in einem parallelen Kontext zu verwenden.

Bei der Parallelisierung eines Programms ist es entscheidend, dass verschiedene Threads oder Prozesse unabhängig voneinander arbeiten können, um eine effiziente und sichere Ausführung zu gewährleisten. Globale Variablen führen jedoch schnell zu potenziellen Konflikten, da mehrere Threads gleichzeitig auf denselben Speicherbereich zugreifen können. Dies kann zu Wettlaufsituationen, inkonsistenten Zuständen und anderen unerwarteten Verhaltensweisen führen, die die Zuverlässigkeit und Korrektheit des Programms beeinträchtigen [15].

Um dieses Problem zu lösen, ist es notwendig, die Abhängigkeit von geteilten Speicherzugriffen so weit wie möglich zu reduzieren. Dies erfolgt durch die Umstrukturierung des Quellcodes, um den Einsatz globaler Variablen zu minimieren oder ganz zu eliminieren. Statt globaler Variablen können lokale Variablen verwendet werden, die nur innerhalb bestimmter Funktionsbereiche gültig sind und somit den Zugriff auf den Speicher einschränken. Diese Vorgehensweise kann zusätzlich Vorteile in Bezug auf Speicherlecks und Speicherbedarf mit sich bringen, da die Variablen nur für die Zeit ihrer Verwendung im Hauptspeicher vorhanden sind. Darüber hinaus können Datenstrukturen wie Klassen oder Strukturen verwendet werden, um Daten zu kapseln und den Zugriff über klar definierte Schnittstellen zu ermöglichen [15].

Bei der Entfernung von geteilten Speicherzugriffen ist es wichtig, auch geeignete Synchronisationsmechanismen einzuführen, um kritische Abschnitte des Codes zu schützen. Dies kann die Verwendung von Mutexen, Semaphoren oder anderen Mechanismen umfassen, um sicherzustellen, dass nur ein Thread gleichzeitig auf bestimmte Ressourcen zugreifen kann, und so potenzielle Wettlaufbedingungen zu vermeiden. Die Verwendung

von Mutexen sollte jedoch nicht ohne Vorbehalt in Erwägung gezogen werden, da sie weitere Abhängigkeiten schafft, welche die Leistungsgewinne durch mehrere Threads wieder negieren könnten. Ist dies der Fall, sollte über eine größere Umstrukturierung der Architektur nachgedacht werden, um das Programm mit Parallelisierung kompatibel zu machen [15].

3.4.2 Verlagerung der zu parallelisierenden Routine

Um eine effektive Parallelisierung zu erreichen, ist es von entscheidender Bedeutung, den spezifischen Teil des Codes zu identifizieren, der für die parallele Ausführung geeignet ist. Dieser Prozess erfordert eine sorgfältige Analyse des Quellcodes, um Bereiche zu lokalisieren, die unabhängig voneinander ausgeführt werden können und keine oder nur minimale Abhängigkeiten zu anderen Teilen des Programms aufweisen. Solche Bereiche können typischerweise Schleifen oder Abschnitte sein, die große Mengen von Daten verarbeiten, ohne auf Zwischenergebnisse anderer Bereiche angewiesen zu sein. Gegebenenfalls kann es auch sinnvoll sein, mit einem Profiler die Laufzeit des Programms zu analysieren, um zutreffende Teile zu identifizieren [14].

Nachdem der geeignete Bereich identifiziert wurde, ist es notwendig, ihn aus dem Hauptcode auszulagern und in eine separate Routine oder Funktion zu überführen. Diese ausgelagerte Routine sollte autonom arbeiten können, ohne auf globale Variablen oder gemeinsam genutzte Ressourcen außerhalb ihres Bereichs zuzugreifen. Durch diese Isolierung können potenzielle Konflikte vermieden und die Parallelisierung erleichtert werden [14].

Es ist essenziell sicherzustellen, dass die ausgelagerte Routine keine Abhängigkeiten zu anderen Teilen des Codes hat, um eine effiziente Parallelisierung zu ermöglichen. Hierbei müssen gegebenenfalls erforderliche Parameter übergeben und Rückgabewerte behandelt werden, um eine reibungslose Interaktion mit dem Rest des Programms zu gewährleisten [14].

Die Verlagerung der zu parallelisierenden Routine ist mitunter der wichtigste Schritt bei der Implementierung von Parallelisierungsstrategien und bildet die Grundlage für eine

effiziente und robuste parallele Ausführung des Programms. Durch die Identifizierung und Isolierung geeigneter Bereiche können potenzielle Engpässe reduziert und die Leistung des Programms optimiert werden.

Unter Umständen ist es nicht ohne Weiteres möglich, einfach einen bestimmten Teil des bestehenden Quellcodes auszulagern und diesen zu parallelisieren. Ist dies der Fall, so muss der Quellcode allgemein konzeptionell umstrukturiert werden. In der Praxis ist dies einer der größten Faktoren, die zur Komplexität von Parallelisierung beitragen.

3.5 Vorstellung der Implementierung

In dieser Sektion wird die Implementierung vorgestellt, wobei einige Aspekte im Anschluss genauer beleuchtet werden. Einige Code-Segmente werden im Vergleich mit der vorherigen Implementierung dargestellt, um die Veränderungen zu veranschaulichen.

Die Herangehensweise der Implementierung war es, zunächst eigenständige Code-Segmente in Funktionen auszulagern, um die Verantwortung einzelner Funktionen klar zu definieren, und große Funktionen mit viel Code herunterzubrechen. Dies ermöglicht es, einfacher zu identifizieren, welche Segmente sich möglicherweise für eine parallele Durchführung eignen. Für die Verlagerung von Code in eigene Funktionen ist es erforderlich, Übergabewerte und Rückgabewerte dieser Funktionen zu definieren, um weiterhin auf die Daten zugreifen zu können. Dabei wurde bereits darauf geachtet, alle neuen Funktionen ohne Seiteneffekte zu implementieren, was bedeutet, dass sie keine externen Zustände verändern und für gegebene Übergabewerte immer die gleichen Rückgabewerte erzeugen. Dies ist für die Strukturierung des Codes für die Parallelisierung von großem Vorteil.

Bei der Analyse stellte sich recht schnell heraus, dass sich der Abschnitt, in welchem ein neuronales Netz zunächst trainiert und dann getestet wird, am besten für die Parallelisierung eignet. Dieser Abschnitt wurde in eine Funktion namens „runSeed“ ausgelagert. Die Funktion bekommt den auszuführenden Seed und den gesamten Datensatz, der bei Programmstart einmalig eingelesen, in Zeilen aufgeteilt und deserialisiert wird, übergeben. Dies unterscheidet sich von der bestehenden Implementierung, in welcher der Datensatz

bei jedem Durchlauf mehrfach zerteilt und eingelesen wird, worauf in der nächsten Sektion 3.5.1 genau eingegangen wird. Dies hat auch Leistungsimplikationen, da der überarbeitete Ansatz nicht nur threadsicher, sondern auch performanter ist.

3.5.1 Verwendung von threadsicheren Funktionen

Besonders zur Manipulation von Zeichenketten macht sowohl die n++-Bibliothek als auch der Code des Experimentes von Herrn Brening viel von klassischen Funktionen aus der C-Standardbibliothek Gebrauch. Beispielsweise wird wie in Abbildung 3.5.1 zu sehen in der n++-Bibliothek die Funktion `strtok` verwendet, welche eine Zeichenkette tokenisieren kann, um ein neuronales Netz aus einer Datei zu laden und zu deserialisieren.

```
611 int Net::load_net( char filename[] ) {  
612     ...  
613     else if (strcmp(line,"topology",7)==0){  
614         value=strtok(line, " \t\n"); /* skip first token (== topology)*/  
615         for(i=0,value=strtok( NULL, " \t" );(value!=NULL)&&(i<MAX_LAYERS);  
616             value=strtok( NULL, " \t\n" ),i++){  
617             ...  
        }
```

Abbildung 3.3: Ausschnitt aus dem bestehenden Code der N++ Bibliothek, in welchem viel Gebrauch von der `strtok`-Funktion gemacht wird.

Werden mehrere Netze gleichzeitig geladen, kann die Verwendung von `strtok` in diesem Kontext jedoch zu Problemen und Konflikten führen, da die `strtok`-Funktion intern den Zustand der Zeichenkettenzerteilung speichert, um bei folgenden Aufrufen den nächsten Teil der Zeichenkette zurückzugeben. Wenn die Funktion also an mehreren Stellen gleichzeitig aufgerufen wird, entstehen inkonsistente Ergebnisse. Für diese Fälle existiert in der Standardbibliothek zum Beispiel die Funktion `strtok_r`, welche ähnlich wie `strtok` funktioniert, den internen Zustand jedoch in einem Pointer speichert, und es somit ermöglicht in jedem Thread einen separaten Pointer zu benutzen [16]. Der Code musste in diesem Fall leicht umstrukturiert werden, danach ist er aber threadsicher. Das Ergebnis ist in Abbildung 3.5.1 ersichtlich.


```
611 int Net::load_net( char filename[] ) {  
612     ...  
613     char *saveptr; // Speichert den Zustand der Tokenisierung der Zeichenkette  
614     ...  
615     else if (strncmp(line, "topology", 7) == 0) {  
616         strtok_r(line, " \t\n", &saveptr); /* skip first token (== topology)*/  
617         for (i = 0, value = strtok_r(nullptr, " \t", &saveptr);  
618             (value != nullptr) && (i < MAX_LAYERS);  
619             value = strtok_r(nullptr, " \t\n", &saveptr), i++) {  
620             ...  
        }
```

Abbildung 3.4: Ausschnitt aus dem überarbeiteten Code der N++ Bibliothek, in welchem die strtok-Funktion durch die strtok_r-Funktion ersetzt wurde.

Diese Verwendung von strtok tritt sehr häufig in dem n++-Quellcode auf. Auch in dem Code von Herrn Brening wird strtok verwendet, um den Datensatz einzulesen, und die relevanten Parameter zu extrahieren. Jeder dieser Funktionsaufrufe wurde durch threadsichere Alternativen wie strtok_r ersetzt. Weitere häufige verwendete Beispiele für thread-unsichere Funktionen der Standardbibliothek, sind asctime, ctime und localtime, aber auch die Zufallsgeneratorfunktionen wie rand und srand. Wichtig zu erwähnen ist jedoch, dass je nach Anwendungsfall die threadsicheren Funktionen wie strtok_r nicht verfügbar sein könnten, zum Beispiel, wenn man die Portabilität für POSIX-Systeme vor 2001 gewährleisten möchte [16].

Um das Nachvollziehen des Programmablaufs zu vereinfachen, verfügte die bestehende Implementierung über viele Konsolenausgaben welche mit cout und dem «-Operator von C++ realisiert worden sind. Diese Konsolenausgaben sind nicht threadsicher, da die Konkatenation von Zeichenketten bei gleichzeitigen Aufrufen die Daten verschiedener Threads sofort ausgegeben werden, bevor sie konkateniert wurden. Als Lösung wurde in der parallelisierten Implementierung die printf Funktion genutzt, welche die Zeichenkette zuerst formatiert und konkateniert, bevor sie in der Konsole ausgegeben wird. Diese Änderung führt jedoch auch zu geringen Leistungsunterschieden, da die Funktionsweise beider Methoden nicht vergleichbar ist.

3.5.2 Entfernung globaler Variablen

In der bestehenden Implementierung von Herrn Brening, von der ein Teil in Abbildung 3.5 zu sehen ist, werden die meisten Variablen global, also außerhalb von Funktionen und Klassen, definiert. Die Verwendung von globalen Variablen ermöglicht eine subjektiv empfundene höhere Lesbarkeit und Verständlichkeit des Quellcodes, da alle verwendeten Variablen zu Beginn der Datei definiert sind und somit an einer Stelle eingesehen werden können. Des Weiteren ermöglichen globale Variablen eine Reduzierung von Parameterübergaben in Funktionen, da auf sie von allen Funktionen direkt zugegriffen werden kann.

```
24 Net* net;
25 string* trainingData;
26 string* testingData;
27 int dataSetCount;
28 int seed;
29
30 int topology[LAYERS];
31 float uparams[10];
32 FTYPE *in_vec, *out_vec, *tar_vec;
```

Abbildung 3.5: Teil der globalen Variablendefinition in der bestehenden Implementierung von Herr Brening

Die globalen Variablen werden für die Datensätze, das neuronale Netz und die Propagierung von Daten verwendet. Im Kontext der Parallelisierung entstehen durch diese Verwendung von globalen Variablen jedoch Konflikte, welche ein Beibehalten dieser Struktur unmöglich machen. So entstehen beispielsweise geteilte Speicherzugriffe wie bereits in Sektion 3.4.1 angesprochen, da nun mehrere Netze zeitgleich laufen sollen. So müsste es beispielsweise entweder eine Variable für jedes Netz geben, oder man schützt die Zugriffe beispielsweise mit einem Mutex, was die Parallelisierung jedoch unwirksam machen würde. Deshalb wurde sich entschieden, die globalen Variablen zu entfernen, und erst an der Stelle zu definieren, wo sie verwendet werden. Das neuronale Netz und die Vektoren werden in Abbildung 3.6 beispielsweise erst innerhalb eines Threads definiert, sodass jeder

Thread und jeder Ablauf über separate Netze verfügt, was die Konflikte beseitigt.

```
242 static void runSeed(const int seed, const vector<DataRow> &trainingset,  
243     const vector<DataRow> &testingset) {  
244     ...  
245     const auto net = new Net();  
246     ...  
247     const auto inVec = new float[net->topo_data.in_count];  
248     const auto outVec = new float[net->topo_data.out_count];  
249     const auto tarVec = new float[net->topo_data.out_count];
```

Abbildung 3.6: Verlagerung der globalen Variablen zu lokalen Variablen in der parallelen Implementierung.

Die globalen Variablen, welche die Ergebnisse der Experimente speichert, mussten nicht entfernt werden, da bereits 10, also eine für jeden Seed bestehen. Der Code wurde vereinfacht, da dem Array stattdessen eine weitere Dimension hinzugefügt wurde, die spezifiziert, von welchem seed die Ergebnisse stammen. So wurde der Code lesbarer, und es gibt weiterhin getrennte Speichersegmente für verschiedene Threads.

3.5.3 Verwendung eines Threadpools

Ein Threadpool ist ein Entwurfsmuster, das häufig in der parallelen Programmierung verwendet wird, um die Verteilung von Aufgaben auf eine gewisse Anzahl an Threads zu vereinfachen. Ein Threadpool besteht aus einer festgelegten Anzahl von vorab initialisierten Threads, die darauf warten, Aufgaben auszuführen. Sobald eine Aufgabe an den Threadpool übergeben wird, wird sie von einem verfügbaren Thread ausgeführt, während andere Threads weiterhin bereitstehen, um zusätzliche Aufgaben zu übernehmen [17]. Dieses Konzept trägt dazu bei, die Kosten für die Erstellung und Zerstörung von Threads zu minimieren und die Systemressourcen effizient zu nutzen. Die Funktionsweise eines Threadpools kann wie folgt beschrieben werden:

1. Initialisierung: Ein genaue Anzahl von Threads wird beim Start des Programms erstellt und in Bereitschaft gehalten.

2. Aufgabenzuweisung: Aufgaben werden einer Warteschlange hinzugefügt, die vom Threadpool verwaltet wird.
3. Aufgabenausführung: Verfügbare Threads nehmen Aufgaben aus der Warteschlange und führen sie aus.
4. Rückführung: Nach der Ausführung einer Aufgabe kehrt der Thread in den Pool zurück und wird für neue Aufgaben bereitgestellt.
5. Terminierung: Stehen keine weiteren Aufgaben an und wird eine Terminierungsfunktion aufgerufen, fährt der Threadpool geordnet herunter, indem laufende Aufgaben abgeschlossen und anschließend die Threads beendet werden. Die Terminierung kann blockierend oder nicht-blockierend implementiert sein, abhängig von den Anforderungen der Anwendung.

Ein Threadpool ermöglicht eine kontrollierte Anzahl von gleichzeitig ausgeführten Threads, wodurch Probleme wie übermäßige Thread-Erstellung und Ressourcenkonflikte vermieden werden. Durch die Begrenzung der Anzahl aktiver Threads können die Systemressourcen effizienter genutzt und die Stabilität des Systems gewährleistet werden. Stehen beispielsweise nur 2 Kerne zur Verfügung, ist es in vielen Fällen effizienter, die Aufgaben von 2 Threads abarbeiten zu lassen, anstatt 50 Threads gleichzeitig zu starten. Zudem wird die Verwaltung von Threads zentralisiert und vereinfacht, da der Threadpool die Lebenszyklen der Threads und die Aufgabenzuweisung übernimmt.

Da die C++-Standardbibliothek nicht über einen ThreadPool verfügt, wurde eine Implementierung aus dem Internet verwendet [17]. Der verwendete ThreadPool ist simpel aufgebaut, und in einer einzigen C++-Header-Datei implementiert. Abbildung 3.7 zeigt ein Beispielprogramm, welches den Threadpool verwendet.

```
1  #include "threadpool/threadpool.h"
2
3  int main() {
4      // Threadpool mit 2 Threads erstellen
5      ThreadPool *threadpool = new ThreadPool(2);
6      // 50 Beispielaufgaben erstellen und der Warteschlange hinzufügen
7      for (int i = 1; i <= 50; i++) {
8          threadpool->enqueue([i] { // Abhängigkeiten müssen spezifiziert werden
9              printf("Aufgabe %d\n", i);
10          });
11      }
12      // Aufruf des Destruktors terminiert Threads nach Abschluss aller Aufgaben
13      delete threadpool;
14      return 0;
15  }
```

Abbildung 3.7: Beispielhafter C++-Code zur Verwendung eines Threadpools: Ein Threadpool mit zwei Threads wird erstellt, 50 Aufgaben werden der Warteschlange hinzugefügt und anschließend wird der Threadpool blockierend terminiert.

In dieser Arbeit wird der Threadpool verwendet, um das Trainieren und Testen der einzelnen neuronalen Netze auf mehrere Kerne zu verteilen. Es handelt sich folglich immer um 10 Aufgaben, welche der Warteschlange des Threadpools hinzugefügt werden. Die genaue Verteilung der Aufgaben auf die verfügbaren Threads wird im nächsten Kapitel erläutert.

Kapitel 4

Experimentelle Untersuchungen

Um die Wirksamkeit der Parallelisierung der N++-Bibliothek in C++ zu bewerten, wird ein umfassender Benchmark-Test durchgeführt. Dieser Test umfasst verschiedene Kombinationen von Threads, Anzahl an Datensätzen und verschiedenen Computern mit verschiedenen Prozessorarchitekturen, um die Auswirkungen der Implementierung auf die Leistung der Bibliothek unter unterschiedlichen Bedingungen zu untersuchen.

4.1 Testumgebung

Das C++ Programm wurde unter Einbindung der N++-Bibliothek auf dem jeweiligen System selbst kompiliert. Dabei wurde die Optimierungsstufe O2 verwendet, welche eine für Produktionssoftware gängige Optimierungsstufe ist. Die O2 Optimierungsstufe wendet fast jede Compileroptimierung an, die die Compiler zu bieten haben. Dabei werden lediglich als sehr unsicher eingestufte Optimierungen ausgelassen. Auf Linux wurde der GCC Compiler (Version 12.2.0-14) und auf MacOS der Clang Compiler (Version 1500.3.9.4) verwendet, um native Binärdateien für die spezifische Prozessorarchitektur zu kompilieren. Das heißt, das Programm wurde nicht unter Emulation, sondern vollständig nativ ausgeführt.

Die Tests wird mit unterschiedlichen Thread-Anzahlen ausgeführt, darunter 10, 8, 6, 4 und 2 gleichzeitig laufenden Threads, um den Einfluss der Parallelisierung auf die Ausführungsgeschwindigkeit zu untersuchen. Zusätzlich wird das Programm auch mit einem einzelnen Thread ausgeführt, um einen Vergleich mit der vorausgegangenen Implementierung herstellen zu können. Für jede Thread-Anzahl werden außerdem verschiedene Größen an Datensätzen getestet. Die Größe der Datensätze wird über die Anzahl an Partitionen spezifiziert. Eine Partition bedeutet dabei, dass die gesamte Datenmenge verwendet wird, wohingegen 4 Partitionen bedeuten, dass nur ein Viertel, also 25% der Datenmenge verwendet werden. Das Programm wird in diesem Test mit 1, 2, 4 und 8 Partitionen getestet, wobei es auf dem langsamsten Computer nur auf 4 und 8 beschränkt wurde.

Für jede Kombination von Threads und Partitionen werden mindestens 5 Testläufe durchgeführt, um robuste Durchschnittswerte zu erhalten und Schwankungen zu minimieren. Gemessen wird die benötigte Zeit für den gesamten Programmdurchlauf in Sekunden. Vor jedem Durchlauf werden die Testgeräte auf einen neutralen Zustand zurückgesetzt, um faire Vergleichsbedingungen sicherzustellen. Dies wird gewährleistet, indem gleiche Seeds für die Zufallszahlgeneratoren verwendet werden, und sichergestellt wird, dass keine anderen Programme laufen. Das Ergebnis jedes Programmdurchlaufs wird in eine Datei geschrieben, um vergleichen zu können, ob mit verschiedenen Anzahlen von Threads die gleichen Ergebnisse berechnet werden.

Nach Abschluss der Testläufe werden die erzielten Ergebnisse automatisch analysiert und Durchschnittswerte für jede Kombination von Threads und Partitionen berechnet. Diese Durchschnittswerte dienen dazu, die möglichen Schwankungen der Testabläufe auszugleichen, und ein neutraleres Ergebnis zu liefern.

4.2 Getestete Hardware

Als erstes Testsystem kommt ein MacBook Pro 16-Zoll Laptop mit dem M1 Pro Prozessor zum Einsatz. Der M1 Pro wurde 2021 vorgestellt und ist eine hochskalierte Version des M1 Prozessors. Er verfügt über insgesamt 12 CPU-Kerne, darunter 8 Hochleistungs-

und 4 Effizienzkerne [18]. Die Hochleistungskerne sind für rechenintensive Aufgaben konzipiert, während die Effizienzkerne für weniger anspruchsvolle Aufgaben und eine Reduzierung des Energieverbrauchs optimiert sind. In Bezug auf die Leistung bietet der M1 Pro Prozessor eine sehr beeindruckende Single-Core-Leistung und ist das performanteste der getesteten Systeme. Die Hauptspeichergröße beträgt 16 GB, welche mit einer Speicherbandbreite von 200 GBit/s [18] angebunden sind. Auch die Cache-Größe des Prozessors ist sehr hoch, was die Latenz bei Speicherzugriffen zusätzlich vermindern kann. Verwendet wurde MacOS Sonoma 14.5 mit dem Darwin Kernel Version 23.5.0 und Clang Version 1500.3.9.4.

Als Desktopprozessor wurde der AMD Ryzen 5 3600XT getestet, welcher ein Prozessor aus der Ryzen 3000-Serie von AMD ist, der im Jahr 2020 eingeführt wurde. Er verfügt über insgesamt 6 CPU-Kerne und 12 Threads auf Basis der Zen 2-Architektur, welche auch im Servermarkt verbreitet ist. Es handelt sich um einen Desktopprozessor mit einer maximalen Leistungsaufnahme von 95 Watt [19]. Der Prozessor unterstützt DDR4-RAM mit regulären Geschwindigkeiten von bis zu 3200 MHz [19]. Die genaue Speicherbandbreite und Größe hängt von der verwendeten RAM-Konfiguration ab, da Desktop Prozessoren modular in verschiedene Systeme eingebaut werden können. In Bezug auf den Cache verfügt der Ryzen 5 3600XT über 32KB L1-Cache, 512KB L2-Cache und 32MB L3-Cache [19]. In der Konfiguration des Testcomputers sind 16GB DDR4 Speicher mit 3600 MHz verbaut, und der Prozessor wird mit einer Wasserkühlung gekühlt, was für eine gleichmäßige und robuste Kühlung sorgt. Als Betriebssystem kam Fedora 40 mit dem Linux Kernel der Version 6.8 zum Einsatz.

Um auch ein System mit begrenzter Leistung zu testen wurde die Implementierung auch auf einem Raspberry Pi 3 getestet, welcher ein Single-Board-Computer ist, der von der Raspberry Pi Foundation entwickelt wurde. Ein Single-Board-Computer ist eine vollständige Computerplatine, die alle erforderlichen Komponenten wie Prozessor, Speicher, Ein- und Ausgabean Anschlüsse und Stromversorgung auf einer einzigen Platine vereint. Der Raspberry Pi 3 wurde 2016 veröffentlicht und basiert auf einem ARM Cortex-A53 Quad-Core-Prozessor mit einer Taktfrequenz von 1,2 GHz und 1 GB LPDDR2-RAM [20]. Der Raspberry Pi 3 ist aufgrund seiner geringen Größe, seines geringen Strom-

verbrauchs und seiner vielfältigen Einsatzmöglichkeiten beliebt. Er wird häufig in Bildungsprojekten, Hobbyprojekten, Heimautomatisierungssystemen und als kostengünstige Entwicklungsumgebung für Softwareentwickler eingesetzt. Trotz seiner begrenzten Leistungsfähigkeit erfüllt der Raspberry Pi 3 wichtige Funktionen in verschiedenen Anwendungsbereichen aufgrund seiner Kompaktheit und seines erschwinglichen Preises. Aufgrund der begrenzten Leistung und der damit verbundenen Laufzeiten konnte der Testablauf nicht vollständig durchgeführt werden. Es wurden maximal 4 Threads getestet, und die Partitionen wurden auf 8 und 4 begrenzt. Die Linux Kernel Version 6.6 kam auf Raspberry Pi OS Lite 12 in der 64-Bit Version zum Einsatz.

Um die Skalierbarkeit und das Verhalten der Algorithmen in unterschiedlichsten Umgebungen zu beleuchten, wurden die Benchmarks zusätzlich auf einem VPS-Server des Anbieters Contabo durchgeführt. Diese Wahl bot gleich zweifache Vorteile: Zum einen ermöglichte sie die Evaluierung der Algorithmenleistung in einer Cloud-Umgebung, die in der heutigen Anwendungslandschaft immer relevanter wird, zum anderen bietet sie einen modernen performanten Prozessor, welche jedoch aufgrund der Virtualisierung auf 2 Kerne beschränkt ist. Der Server verfügt über einen AMD EPYC 7282 Prozessor, der speziell für rechenintensive Aufgaben in Serverumgebungen entwickelt wurde. Mit seinen 16 Kernen und 32 Threads bietet er eine beachtliche Rechenleistung. Der Basistakt liegt bei 2,8 GHz und der Cache ist 64 MB groß [21]. Von diesem leistungsstarken Prozessor stehen dem VPS-Server lediglich 2 Kerne und 4 GB Arbeitsspeicher zur Verfügung. Diese Einschränkung sollte Aufschluss darüber geben, wie effizient die Algorithmen mit limitierten Ressourcen umgehen können und wie sie in skalierbaren Umgebungen performen. Der verwendete Linux-Kernel war die Version 6.1.0-21 auf Debian 12.

4.3 Benchmark Script

Aus 6 verschiedenen Anzahlen an Threads, 4 verschiedenen Größen der Datensätze und 5 Durchläufen pro Kombination ergeben sich 120 einzelne Tests, die pro System ausgeführt werden müssen. Um diese Arbeit zu erleichtern und einen reproduzierbaren Testprozess zu ermöglichen, wurde ein Skript geschrieben, welches alle Tests nacheinander automatisch

ausführt.

Das Skript misst die benötigten Laufzeiten der Durchläufe und schreibt nach jedem Durchlauf die benötigte Zeit in eine Datei. Zusätzlich werden die Zeit und Informationen zu jedem Durchlauf auch in eine CSV-Datei geschrieben, um das Auswerten der Benchmarks auf einem System durch nur eine einzige Datei zu ermöglichen.

Als Parameter ist es möglich, die maximale Anzahl an Threads festzulegen. So ist es beispielsweise auf einem Raspberry Pi sinnvoll, das Programm nur mit maximal 4 Threads zu testen, da er nur über 4 Kerne verfügt.

Das Skript ist simpel und wurde in Bash geschrieben, was die Portabilität zwischen Linux und MacOS gewährleistet. Dabei wurde jedoch das Programm bc verwendet, welches auf den meisten Linux-Distributionen standardmäßig nicht vorinstalliert ist.

```
1  #!/bin/bash
2  ATTEMPTS=5
3  PARTITIONS=(8 4 2 1) # Verschiedene Partitionen zum Testen
4  THREADS=(10 8 6 4 2 1) # Verschiedene Thread Anzahlen zum Testen
5  AVAILABLE_THREADS=10 # Maximal verfügbare Anzahl an Threads für aktuelles System
6
7  function run_attempt {
8      # Hier wird ein Durchlauf durchgeführt und getestet
9  }
10
11 # Test für jede mögliche Kombination durchführen
12 for partition in ${PARTITIONS[@]};do
13     for threads in ${THREADS[@]};do
14         if ((threads <= AVAILABLE_THREADS));then
15             for ((i = 1; i <= ATTEMPTS; i++)); do
16                 run_attempt "$threads" "$partition" "$i"
17             done
18         fi
19     done
20 done
```

Abbildung 4.1: Vereinfacht dargestelltes Benchmark Skript. Es testet die benötigte Laufzeit der Implementierung mit verschiedenen Thread-Anzahlen und Datensatzgrößen.

4.4 Ergebnisse

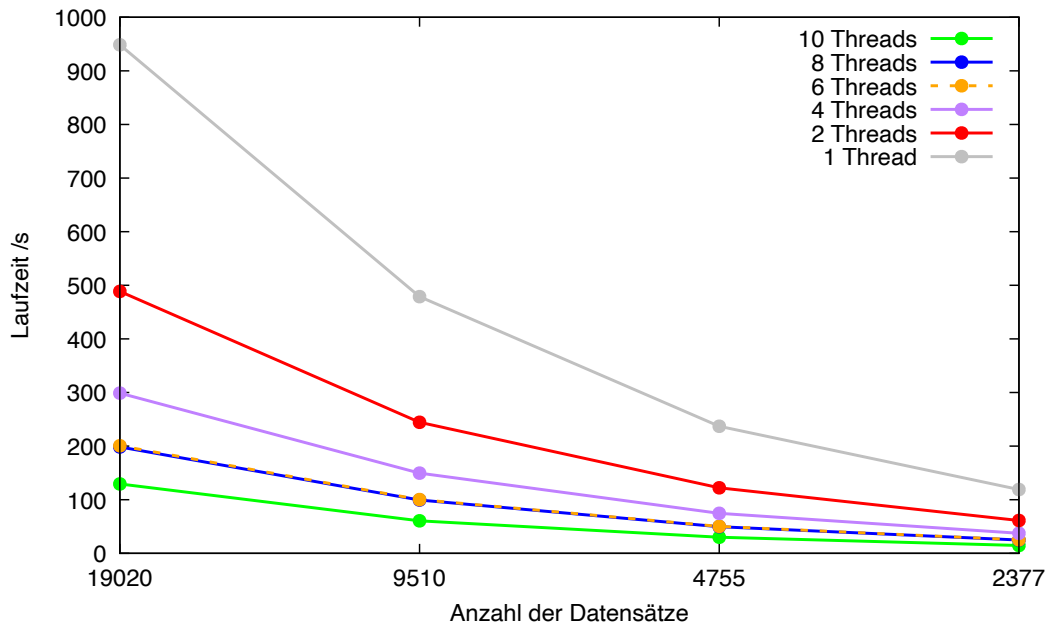


Abbildung 4.2: Leistungstest auf Apple M1 Pro: Einfluss von Thread-Anzahlen auf Laufzeit bei variierenden Datensatzgrößen, welche den Mittelwert von 5 Durchläufen darstellt. Die Standardabweichung der Laufzeit ist in Tabelle 4.1 einzusehen.

Die Analyse der Benchmark-Ergebnisse auf dem M1 Pro-Prozessor aus Abbildung 4.2 liefert wertvolle Einblicke in die Effektivität von Parallelisierung des Programms. Durch Tests mit 10, 8, 6, 4, 2 und 1 Threads konnte eine größtenteils lineare Skalierung der Leistung beobachtet werden, wobei 10 Threads annähernd eine zehnfache Beschleunigung im Vergleich zu einem einzelnen Thread erzielten. Dies ist besonders bemerkenswert, da nur 8 Hochleistungskerne zur Verfügung stehen, und die verbleibenden 2 Threads gezwungenermaßen auf Effizienzkerneln laufen müssen.

Interessanterweise zeigt sich jedoch eine bemerkenswerte Konvergenz der Leistungskurven bei 6 und 8 Threads. Dies resultiert aus der Verteilung der parallelen Aufgaben auf die verfügbaren Threads, welche in den Abbildungen 4.3 und 4.4 schematisch dargestellt ist. Mit 6 Threads werden zunächst 6 Aufgaben bearbeitet, während 4 Aufgaben verbleiben. Die verbleibenden 4 Aufgaben erfordern ungefähr dieselbe Ausführungszeit wie die 2 zusätzlichen Aufgaben bei Verwendung von 8 Threads, was in den Abbildungen

an der gleichen Breite der Durchläufe erkennbar ist. Diese Konvergenz erklärt die nahezu identischen Leistungskurven bei 6 und 8 Threads. Bei 6 Threads könnten zwar insgesamt 6 Aufgaben bearbeitet werden, jedoch sind beim zweiten Durchlauf 2 Threads im Leerlauf, während die verbleibenden 4 Threads genutzt werden. Bei 8 Threads bleiben im zweiten Durchlauf sogar 6 Threads inaktiv. Diese Effizienzunterschiede führen zu einer vergleichbaren Ausführungszeit für die verbleibenden Aufgaben bei 6 und 8 Threads.

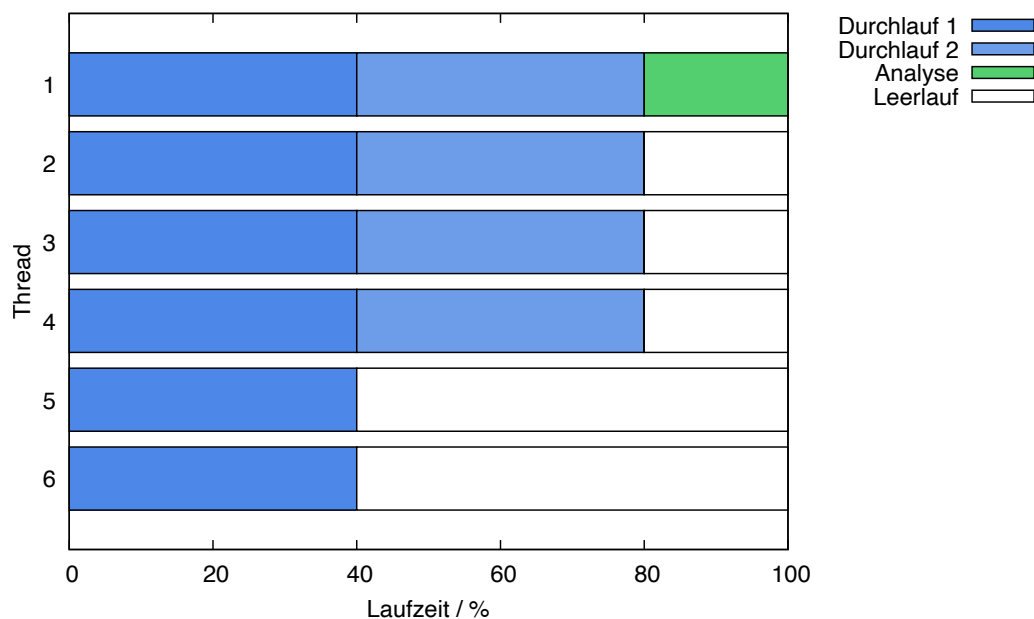


Abbildung 4.3: Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 6 Threads

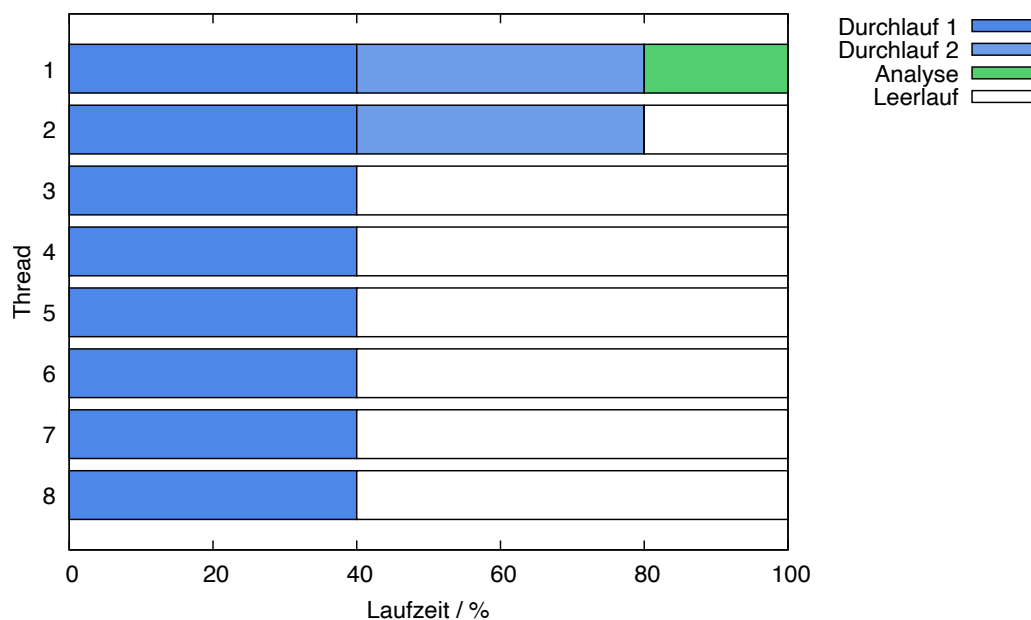


Abbildung 4.4: Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 8 Threads

Da für den Analyseschritt im Programmablauf alle 10 Ergebnisse benötigt werden, ist es algorithmisch nicht möglich, die Analyse vor Abschluss aller 10 Durchläufe auszuführen, um die Kerne, die im Leerlauf sind, zu nutzen. Es kann sich also nur eine gleichmäßige und effiziente Nutzung der Threads ergeben, wenn eine ganzzahlige Fraktion der gesamten Aufgaben als Threads zur Verfügung stehen. Bei 10 Aufgaben bedeutet das, dass 10, 5, 2 und 1 Thread ohne signifikanten Leerlauf der Threads voll ausgenutzt werden können, wenn man die Analyse auslässt.

Die Standardabweichung der Ergebnisse des Apple M1 Pro, welche in Tabelle 4.1 ablesbar ist, erweist sich insgesamt als ziemlich gering, was für eine zuverlässige Verteilung der Threads von dem Betriebssystem auf die verschiedenen Kerne spricht. Zu beobachten ist, dass die Standardabweichung selbstverständlich bei Tests, welche mehr Zeit beanspruchen höher ist, aber auch, dass bei diesem System die Standardabweichung bei 4 Threads verhältnismäßig leicht höher erscheint. Dies könnte eventuell an dem Scheduler des Kernels liegen, welcher dynamisch entscheidet, auf welchen Kernen Aufgaben laufen sollen. Ein weiterer Faktor könnte die heterogene Architektur mit den Hochleistungs- und Effizienzkernen sein.

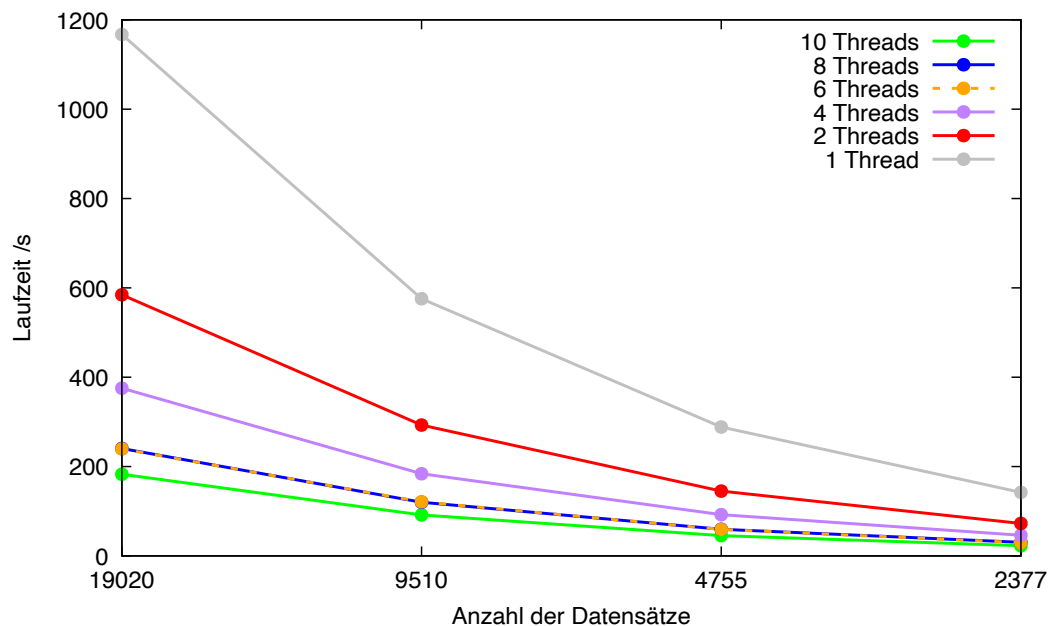


Abbildung 4.5: Leistungstest auf AMD Ryzen 5 3600XT: Einfluss von Thread-Anzahlen auf Laufzeit bei variierenden Datensatzgrößen, welche den Mittelwert von 5 Durchläufen darstellt. Die Standardabweichung der Laufzeit ist in Tabelle 4.1 einzusehen.

Die angesprochenen Beobachtungen bestätigen sich beim zweiten Testsystem, dem AMD Ryzen 5 3600XT weitgehend bestätigen. Genau wie beim Apple M1 Pro lässt sich eine erfolgreiche Skalierung der Leistung bei höheren Thread-Zahlen feststellen, was in Abbildung 4.5 veranschaulicht ist. Dieses System ist auch sehr leistungsstark, und kann von Mehrkernanwendungen stark profitieren. Insgesamt liegen die benötigten Laufzeiten circa 20% über denen des Apple M1 Pro. Auch die zuvor erwähnte Konvergenz bei 6 und 8 verwendeten Threads ist zu erkennen.

Besonders zu beobachten ist, dass der Leistungssprung der Laufzeiten von 6 auf 10 Threads bei diesem System geringer ist als bei dem Apple M1 Pro. Dies ist zu erwarten, da der AMD Ryzen 5 3600XT nur über 6 Kerne verfügt, welche zwar mithilfe von Hyper-Threading 12 physische Threads verfügbar stellen [19], jedoch nicht mit der Leistung von 12 Prozessorkernen vergleichbar sind. Bei 10 laufenden Threads müssen sich somit 4 der 6 vorhandenen Prozessorkerne mit 2 Aufgaben gleichzeitig beschäftigen.

Die Standardabweichung der Testergebnisse ist im Allgemeinen auf dem gleichen Niveau mit denen des vorher angesprochenen Systems. Deutliche Unterschiede zwischen

Linux und MacOS konnten nicht beobachtet werden und für das getestete Programm scheint es keine großen Unterschiede zwischen dem Linux-Kernel und dem Darwin-Kernel von MacOS zu geben.

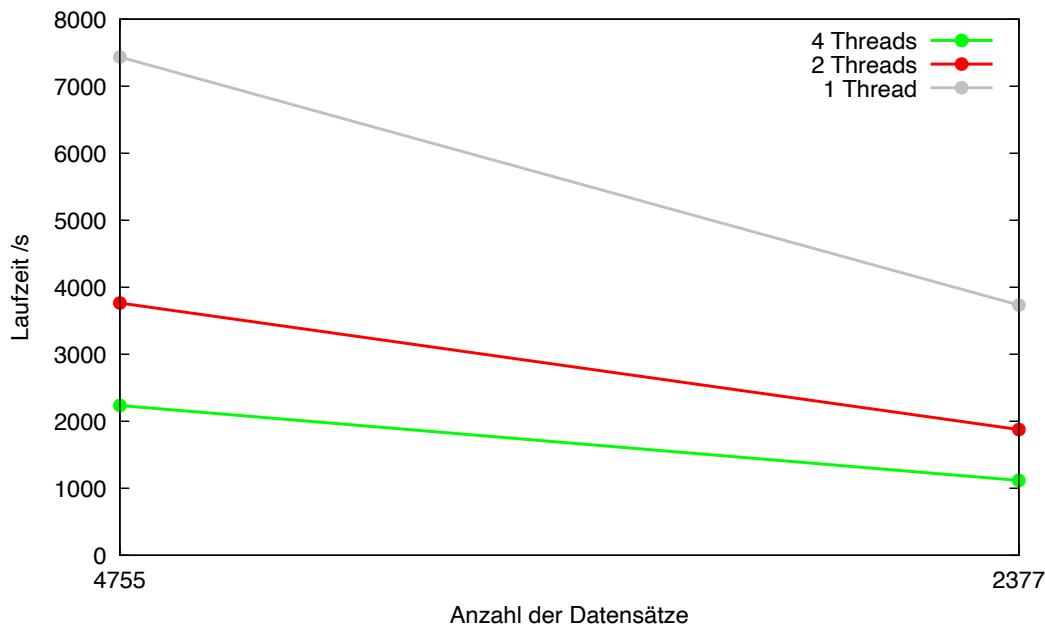


Abbildung 4.6: Leistungstest auf Raspberry Pi 3: Einfluss von Thread-Anzahlen auf Verarbeitungszeit bei variierenden Datensatzgrößen

Der Raspberry Pi stellt das langsamste getestete System dar. Wie bereits in Sektion 4.2 erläutert und in Abbildung 4.6 konnten deshalb aufgrund der zeitlichen Begrenzung nicht alle Tests durchgeführt werden. Im Vergleich zum Apple M1 Pro benötigt der Raspberry Pi bei 2 Threads mit 4755 Datensätzen beispielsweise nahezu 26-mal länger. Es ist zu sehen, dass auch der Raspberry Pi von der Parallelisierung profitiert, was gerade bei einem so langsamen System absolut gesehen enorme Unterschiede macht. So verbessert sich die Laufzeit bei 4755 Datensätzen von 7436 Sekunden (02h 03m 56s) bei einem verwendeten Thread auf 2235 Sekunden (37m 15s) unter der Verwendung von 4 Threads, was bedeutet, dass 1 Stunde und 26 Minuten an Laufzeit eingespart werden können.

Im Gegensatz zu dem MacBook mit macOS und dem Ryzen-System mit Fedora Linux lief auf dem getesteten Raspberry Pi keine grafische Oberfläche, welche eventuell Ressourcen während des Tests in Anspruch nehmen könnte. Wirklich in den Ergebnissen durch eine deutlich geringere Standardabweichung im Vergleich zur Laufzeit bemerkbar

machte sich das jedoch nicht, weshalb davon auszugehen ist, dass das Testen auf einem System mit grafischer Oberfläche auf diesen Systemen keine signifikanten Leistungseinbußen bedeutet. Da aber sowohl die vorhandene als auch die überarbeitete Implementierung weniger als 100 Megabyte Hauptspeicher benötigen, und sichergestellt wurde, dass während des Tests keine anderen Programme im Hintergrund laufen, war auch mit keinen großen Problemen diesbezüglich zu rechnen.

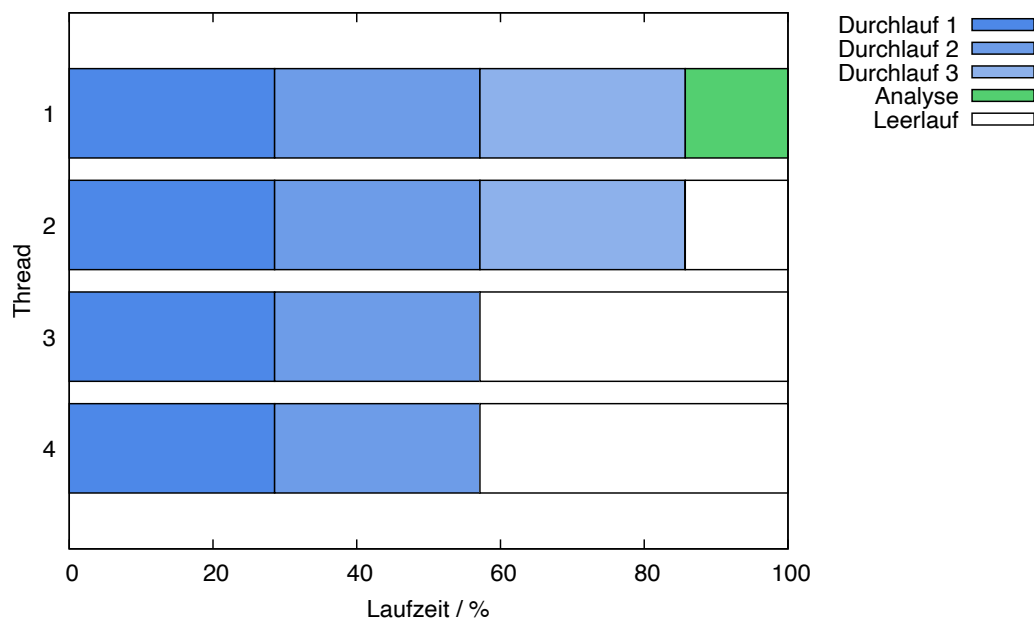


Abbildung 4.7: Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 8 Threads

Die Verteilung der 10 neuronalen Netze auf die 4 vorhandenen Kerne bei einem Programmablauf mit 4 Threads ist in Abbildung 4.7 dargestellt. Für die ersten 8 Durchläufe können die 4 Kerne des Raspberry Pi voll ausgenutzt werden, während danach nur noch maximal 2 von dem Programm ausgelastet werden. Da das Programm auf dem Raspberry Pi nicht mit 6 und 8 Threads getestet wurde ist die zuvor angesprochene Konvergenz nicht ersichtlich.

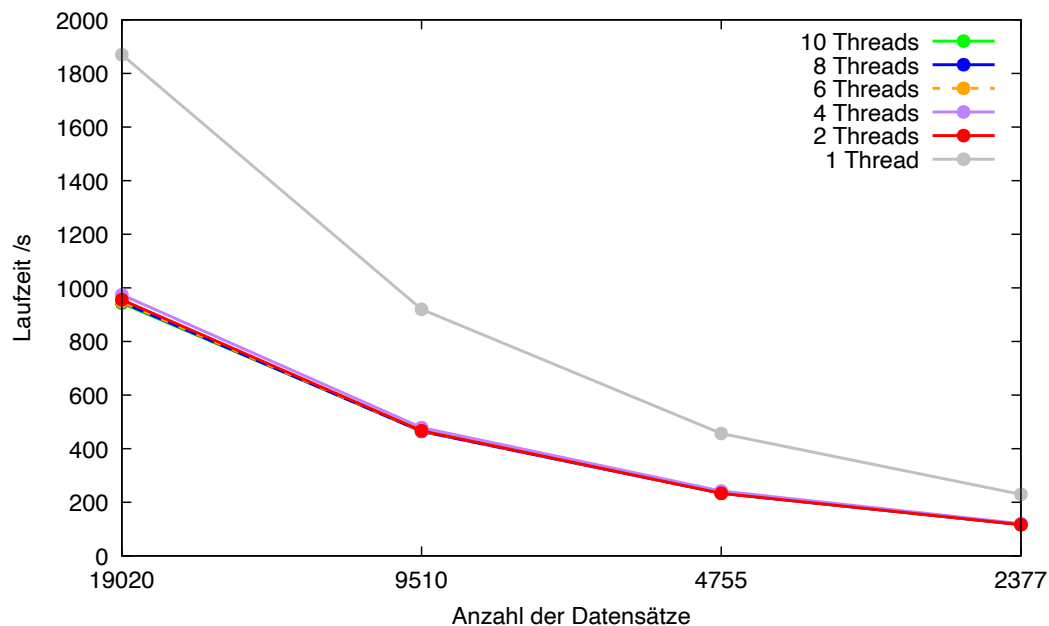


Abbildung 4.8: Leistungstest auf AMD EPYC 7282 mit 2 verfügbaren Kernen: Einfluss von Thread-Anzahlen auf Verarbeitungszeit bei variierenden Datensatzgrößen

Als letztes System wurde der virtuelle Server mit einem AMD EPYC 7282 Prozessor getestet. Es zeigt sich in Abbildung 4.8, dass er unter Verwendung eines Threads ungefähr halb so schnell wie der Apple M1 Pro ist. Dem virtuellen Server stehen nur 2 Kerne zur Verfügung, was sich deutlich in den Ergebnissen widerspiegelt. In den Tests mit 2 Threads zeigten sich genau wie bei den anderen getesteten Systemen deutliche Leistungsverbesserungen. Daran ist erkennbar, dass das Programm auch in Serverumgebungen genau so von Parallelisierung profitiert, wie zum Beispiel der AMD Ryzen 5 3600XT. Dies ist zu erwarten, da der AMD EPYC Prozessor die gleiche Mikroarchitektur wie der AMD Ryzen Prozessor namens Zen 2 [21] verwendet. Der Serverprozessor ist in diesem Fall eine hochskalierte Version des für Privatkunden handelsüblichen Ryzen Prozessors. Abgesehen von der Anzahl der vorhandenen Kerne unterscheidet sich der AMD Epyc Prozessor auch durch seine deutlich geringere Taktraten vom AMD Ryzen, welche es ermöglichen, mehr Kerne bei gleicher verfügbarer Kühlleistung zu verbauen. Server haben normalerweise deutlich mehr Kerne als Prozessoren für Privatanwender.

Deutlich erkennbar ist, dass dieses System nicht von mehr als 2 Threads profitiert. Die Leistungskurven von 2, 4, 6, 8 und 10 Threads liegen nahezu exakt übereinander, da

die beiden vorhandenen Kerne von 2 Threads bereits voll ausgelastet werden.

4.5 Auswertung

Die durchgeführten Benchmarks auf den verschiedenen Testsystemen demonstrieren eindeutig die Leistungsverbesserungen durch die Parallelisierung der Anwendung und der Änderungen in der n++-Bibliothek. In allen getesteten Konfigurationen zeigte sich eine signifikante Reduktion der Laufzeit bei der Nutzung von mehr als einem Thread, was die Effektivität der parallelisierten Implementierung unterstreicht. Bis auf die Ausnahme der gleichbleibenden Laufzeiten bei 6 und 8 Threads, wie bereits im vorigen Abschnitt 4.4 erwähnt, lässt sich eine effektive Reduktion der Laufzeiten bei der Verwendung von mehr Threads ablesen.

Die getesteten Systeme umfassten sowohl verschiedene Prozessorarchitekturen als auch Betriebssysteme, um die Generalisierbarkeit der Ergebnisse sicherzustellen. In jedem Szenario wurde ein konsistenter Leistungszuwachs verzeichnet, sobald mehrere Threads zur Ausführung der Berechnungen herangezogen wurden, wie unten in Tabelle 4.1 ablesbar.

Zusammenfassend lässt sich festhalten, dass die parallelisierte Implementierung der Vorarbeit erfolgreich war und die gesetzten Ziele in Bezug auf die Laufzeitoptimierung erreicht wurden. Die durchgeführten Tests bestätigen die Annahme, dass die Nutzung von Multithreading in der Lage ist, die Performance signifikant zu verbessern, wodurch die Effizienz und Praktikabilität der Bibliothek für reale Anwendungen gesteigert wird.

Tabelle 4.1: Benchmark-Ergebnisse für alle getesteten Systeme. Die benötigte Laufzeit t ist ein Mittelwert aus jeweils mindestens 5 Durchläufen. Die Standardabweichung dieser Durchläufe ist mit σ gekennzeichnet.

Threads	Datenmenge	Ryzen 3600XT		Apple M1 Pro		Raspberry Pi 3		VPS Server	
		t	σ	t	σ	t	σ	t	σ
1	19020	1167.2	4.3	948.6	2.5	-	-	1870.1	7.1
1	9510	575.6	8.3	478.8	9.7	-	-	919.9	1.5
1	4755	288.5	1.0	237.0	0.0	7436.2	19.6	456.7	1.3
1	2377	142.2	2.2	119.0	1.2	3734.3	36.3	229.6	0.9
2	19020	584.5	0.7	488.6	0.9	-	-	955.7	4.1
2	9510	292.9	0.0	244.4	0.5	-	-	466.9	2.7
2	4755	145.1	0.3	122.2	0.4	3764.5	33.9	233.3	1.8
2	2377	72.6	0.2	61.2	0.4	1876.1	12.0	116.6	0.7
4	19020	375.4	3.2	298.8	0.4	-	-	974.2	19.0
4	9510	183.9	7.3	149.6	0.5	-	-	478.6	3.1
4	4755	92.3	2.4	74.6	0.5	2235.2	6.1	242.0	2.1
4	2377	46.2	1.4	37.4	0.5	1117.0	2.0	120.3	1.2
6	19020	239.5	0.4	200.6	0.5	-	-	947.8	5.0
6	9510	121.0	0.3	100.2	0.4	-	-	465.8	2.6
6	4755	59.3	0.0	50.2	0.4	-	-	234.4	2.4
6	2377	29.7	0.1	25.2	0.4	-	-	117.5	1.6
8	19020	240.4	1.4	198.4	0.5	-	-	946.6	5.2
8	9510	120.1	0.5	99.4	0.5	-	-	464.3	1.8
8	4755	59.6	0.1	49.6	0.5	-	-	234.0	0.8
8	2377	30.4	1.3	24.8	0.4	-	-	116.8	0.8
10	19020	182.8	0.4	129.4	10.2	-	-	942.4	4.8
10	9510	91.7	0.9	60.6	3.6	-	-	466.6	1.3
10	4755	45.4	0.1	30.0	2.2	-	-	234.3	0.8
10	2377	22.8	0.1	14.6	0.5	-	-	117.1	0.6

Kapitel 5

Fazit und Ausblick

Der Code der Vorarbeit wurde parallelisiert und die n++-Bibliothek wurde threadsicher umgestaltet. Durch ausführliche Leistungstests konnte die Effizienz der Parallelisierung hinsichtlich der benötigten Laufzeit untersucht werden. Eine deutliche Verbesserung der benötigten Laufzeiten konnte erzielt und aufgezeigt werden. Es wurde bestätigt, dass sich die erzielten Verbesserungen nicht lediglich auf ein System beschränken, sondern auf mehreren Prozessorarchitekturen und Betriebssystemen realisiert werden.

Der implementierte Ansatz ermöglicht bei einem Programmablauf mit 8 Threads keine Laufzeitverbesserungen gegenüber einem Ablauf mit 6 Threads. Darüber hinaus sind bei Verwendung von über 10 Threads keine Verbesserungen zu erwarten. Diese Limitierung ist den nur maximal 10 parallel laufenden Aufgaben geschuldet. Um diese aufzuheben, bedarf es einer vollständigen Neukonzeption der Parallelisierungsstrategie wie zum Beispiel dem Modellparallelismus.

Ein weiterer Ansatz wäre es, die n++-Bibliothek um weitere Funktionen zur Parallelisierung zu erweitern. Dies könnte zum Beispiel die Parallelisierung von internen Co-deabschnitten oder die Bereitstellung von Hilfsfunktionen zur Vereinfachung paralleler Implementierungen für Anwender umfassen.

Anhang A

Programmcode

```
1  #include <string>
2  #include <sstream>
3  #include <vector>
4  #include <fstream>
5  #include <algorithm>
6  #include <random>
7
8  #include "npp/npp.h"
9  #include "threadpool/threadpool.h"
10
11 using namespace std;
12
13 #define DATASET_FILENAME "magic_normalized.txt"
14 #define DATASET_INITIAL_SHUFFLE_SEED 13267
15 #define SEEDS 10
16 #define EPOCHS 10000
17
18 #define PERCENT_TRAINING 0.8
19 #define PERCENT_TESTING (1.0 - PERCENT_TRAINING)
20 #define INPUTS 10
21 #define OUTPUTS 2
22 #define HIDDEN 9
23 #define LAYERS 4
24
25 // Results are stored here
26 double results[SEEDS + 2][EPOCHS + 1][3];
27 double combinedMSE[SEEDS + 2][EPOCHS + 1];
28 double combinedMSETesting[SEEDS + 2][EPOCHS + 1];
```

```

29 double combinedCorrects[SEEDS + 2][EPOCHS + 1];
30
31 /**
32  * Represents a row in a dataset.
33  * Every row has a number of floats which matches the INPUTS of the neural network.
34  * Furthermore the classification is stored as a char.
35  */
36 typedef struct {
37     float input[INPUTS];
38     char classification;
39 } DataRow;
40
41 /**
42  * Initializes the given neural net with the predefined topology.
43  * @param net Net instance to be initialized
44  * @param randomSeed seed used by the random number generator
45  */
46 void createNeuralNet(Net *net, int seed) {
47     // Create neural network topology
48     int topology[LAYERS];
49     topology[0] = INPUTS;
50     topology[1] = HIDDEN;
51     topology[2] = HIDDEN;
52     topology[3] = OUTPUTS;
53     // Create the layers and connect the weights
54     net->create_layers(LAYERS, topology);
55     net->connect_layers();
56     net->set_seed(seed);
57     net->init_weights(0, 0.5);
58     // Initialize activation functions
59     float uparams[10];
60     uparams[0] = 0.001;
61     uparams[1] = 0.95;
62     uparams[2] = 0;
63     net->set_layer_act_f(1, LOGISTIC);
64     net->set_layer_act_f(2, LOGISTIC);
65     net->set_layer_act_f(3, LOGISTIC);
66     net->set_update_f(BP, uparams);
67 }
68
69 /**
70  * Attempts to parse a DataRow from the given string.
71  * @param line string to be parsed
72  * @param row row that will be used to store the parsed dataset row
73  * @return true if the string could be parsed, false otherwise
74  */

```

```

75 bool parseDatasetRow(const string &line, DataRow *row) {
76     // Create a string stream to parse the line
77     stringstream ss(line);
78     string substr;
79     // Parse the first floats separated by ';'
80     for (float & i : row->input) {
81         if (!getline(ss, substr, ';') || !(istringstream(substr) >> i)) {
82             return false; // If conversion fails or delimiter is missing, return false
83         }
84     }
85     // Parse the last character (classification)
86     if (!getline(ss, substr, ';') || substr.size() != 1) {
87         return false;
88     }
89     row->classification = substr[0];
90     return true;
91 }
92
93 /**
94  * Checks whether the file with the given filename exists.
95  * @param name relative file path and name
96  * @return true if the file exists, false otherwise
97  */
98 inline bool fileExists(const std::string &name) {
99     ifstream f(name.c_str());
100     return f.good();
101 }
102
103 /**
104  * Reads the dataset from the file with the given filename
105  * @param filename name (relative path) of the file to be read
106  * @param dataset vector to save the parsed dataset rows
107  * @return true if the dataset could be read, false otherwise
108  */
109 bool readDataset(const string& filename, vector<DataRow> *dataset) {
110     ifstream file(filename);
111     if (!file || !file.is_open()) {
112         return false;
113     }
114     string line;
115     while (getline(file, line)) {
116         DataRow row;
117         if (parseDatasetRow(line, &row)) dataset->push_back(row);
118     }
119     shuffle(dataset->begin(), dataset->end(),
120             default_random_engine(DATASET_INITIAL_SHUFFLE_SEED));

```

```

121     return true;
122 }
123
124 /**
125  * Partitions the given dataset into training and testing set.
126  * Additionally, only a part of the given dataset can be used.
127  * @param dataset complete dataset to be partitioned
128  * @param numPartitions number of partitions (1 if you want the full dataset)
129  * @param partition number of the current partition
130  * @param trainingset training set vector to be initialized
131  * @param testingset testing set vector to be initialized
132  * @param seed seed to be used for random generation
133  */
134 void
135 partitionDataset(const vector<DatasetRow> &dataset,
136                 const int numPartitions, const int partition,
137                 vector<DatasetRow> *trainingset,
138                 vector<DatasetRow> *testingset, const int seed) {
139     // Calculate the sizes of the training and testing sets
140     const size_t partitionSize = dataset.size() / numPartitions;
141     const size_t trainingsetSize = partitionSize * PERCENT_TRAINING;
142     const size_t trainingsetStart = partitionSize * (partition - 1);
143     const size_t testingsetStart = trainingsetStart + trainingsetSize + 1;
144     // Copy the training set rows into the trainingset
145     auto trainingBegin = dataset.begin() + trainingsetStart;
146     auto trainingEnd = dataset.begin() + testingsetStart - 1;
147     trainingset->insert(trainingset->end(), trainingBegin, trainingEnd);
148     // Copy the testing set rows into the testingset
149     auto testingBegin = dataset.begin() + testingsetStart;
150     auto testingEnd = dataset.begin() + trainingsetStart + partitionSize - 1;
151     testingset->insert(testingset->end(), testingBegin, testingEnd);
152     // Shuffle the dataset for the seed
153     shuffle(trainingset->begin(), trainingset->end(), default_random_engine(seed));
154     shuffle(testingset->begin(), testingset->end(), default_random_engine(seed));
155 }
156
157 /**
158  * Initializes the vectors for a neural network pass using the given dataset
159  * @param row dataset row to be used
160  * @param inVec vector for the inputs
161  * @param outVec vector for the outputs
162  * @param tarVec target vector
163  */
164 static inline void initializeVectors(const DatasetRow &row, float *inVec, float *outVec,
165                                     float *tarVec) {
166     copy(row.input, row.input + INPUTS, inVec);

```



```

167     fill_n(outVec, OUTPUTS, 0);
168     tarVec[0] = row.classification == 'g';
169     tarVec[1] = row.classification == 'h';
170 }
171
172 /**
173  * Squares the given number and returns the result.
174  * This function is used to make the analyzeResults function more readable.
175  * @param number number to be squared
176  * @return the number squared
177  */
178 inline double square(double number) {
179     return number * number;
180 }
181
182 /**
183  * Analyzes the results and saves the data to a textfile
184  */
185 void analyzeResults(const int threadcount, const int partitionCount, const int attempt) {
186     srand(time(nullptr));
187     ofstream experimentalData(
188         "results_attempt" + to_string(attempt) + "_" +
189         to_string(partitionCount) + "partitions_" +
190         to_string(threadcount) + "threads.txt");
191     for (int i = 0; i <= EPOCHS; i++) {
192         double tssVariance, tssDeviation, testTssVariance, testTssDeviation,
193             correctsVariance, correctsDeviation;
194         double mse = 0, mseTesting = 0, corrects = 0;
195         for (int seed = 1; seed <= SEEDS; seed++) {
196             mse += combinedMSE[seed][i];
197             mseTesting += combinedMSETesting[seed][i];
198             corrects += combinedCorrects[seed][i];
199         }
200         tssVariance = (square((results[1][i][0] - mse / SEEDS)) +
201             square((results[2][i][0] - mse / SEEDS)) +
202             square((results[3][i][0] - mse / SEEDS)) +
203             square((results[4][i][0] - mse / SEEDS)) +
204             square((results[5][i][0] - mse / SEEDS)) +
205             square((results[6][i][0] - mse / SEEDS)) +
206             square((results[7][i][0] - mse / SEEDS)) +
207             square((results[8][i][0] - mse / SEEDS)) +
208             square((results[9][i][0] - mse / SEEDS)) +
209             square((results[10][i][0] - mse / SEEDS))) / SEEDS;
210         tssDeviation = sqrt(tssVariance);
211         testTssVariance = (square((results[1][i][1] - mseTesting / SEEDS)) +
212             square((results[2][i][1] - mseTesting / SEEDS)) +

```

```

213         square((results[3][i][1] - mseTesting / SEEDS)) +
214         square((results[4][i][1] - mseTesting / SEEDS)) +
215         square((results[5][i][1] - mseTesting / SEEDS)) +
216         square((results[6][i][1] - mseTesting / SEEDS)) +
217         square((results[7][i][1] - mseTesting / SEEDS)) +
218         square((results[8][i][1] - mseTesting / SEEDS)) +
219         square((results[9][i][1] - mseTesting / SEEDS)) +
220         square((results[10][i][1] - mseTesting / SEEDS))) / SEEDS;
221     testTssDeviation = sqrt(testTssVariance);
222     correctsVariance = (square((results[1][i][2] - corrects / SEEDS)) +
223         square((results[2][i][2] - corrects / SEEDS)) +
224         square((results[3][i][2] - corrects / SEEDS)) +
225         square((results[4][i][2] - corrects / SEEDS)) +
226         square((results[5][i][2] - corrects / SEEDS)) +
227         square((results[6][i][2] - corrects / SEEDS)) +
228         square((results[7][i][2] - corrects / SEEDS)) +
229         square((results[8][i][2] - corrects / SEEDS)) +
230         square((results[9][i][2] - corrects / SEEDS)) +
231         square((results[10][i][2] - corrects / SEEDS))) / SEEDS;
232     correctsDeviation = sqrt(correctsVariance);
233     experimentalData << i << "," << mse / SEEDS << "," << tssDeviation << ","
234         << mseTesting / SEEDS << "," << testTssDeviation << ","
235         << corrects / SEEDS << "," << correctsDeviation << endl;
236 }
237 }
238
239 /**
240  * Runs the seed with the given datasets.
241  * This function is supposed to be run in a thread.
242  * @param seed the number of the seed to be run
243  * @param trainingset set of training data
244  * @param testingset set of testing data
245  */
246 static void runSeed(const int seed, const vector<DataRow> &trainingset,
247     const vector<DataRow> &testingset) {
248     // Initialize the net object for the given seed.
249     const string filename = "startnet_seed_" + to_string(seed) + ".net";
250     const auto net = new Net();
251     // If the startnet file exists, use the generated startnet,
252     // otherwise generate the net and save it to file
253     if (fileExists(filename)) {
254         net->load_net(filename.c_str());
255     } else {
256         createNeuralNet(net, seed);
257         net->save_net(filename);
258     }

```

```

259 // Create the vectors used as inputs, outputs and targets for use by propagation
260 const auto inVec = new float[net->topo_data.in_count];
261 const auto outVec = new float[net->topo_data.out_count];
262 const auto tarVec = new float[net->topo_data.out_count];
263 // Run the training and testing process EPOCHS amount of times
264 for (int epoch = 0; epoch < EPOCHS; epoch++) {
265     // Run the training for the whole training set
266     float tss = 0.0;
267     for (const auto data: trainingset) {
268         initializeVectors(data, inVec, outVec, tarVec);
269         net->forward_pass(inVec, outVec);
270         for (int i = 0; i < net->topo_data.out_count; i++) {
271             const double error = outVec[i] - tarVec[i];
272             tss += error * error;
273         }
274         net->backward_pass(outVec, inVec);
275     }
276     // Test the neural network for the whole testing set and keep track of
277     // correct passes
278     float tssTesting = 0, tssCorrect;
279     int correct = 0;
280     for (const auto data: testingset) {
281         tssCorrect = 0;
282         initializeVectors(data, inVec, outVec, tarVec);
283         net->forward_pass(inVec, outVec);
284         for (int i = 0; i < net->topo_data.out_count; i++) {
285             float error = outVec[i] - tarVec[i]; // dE_total/dout_oi
286             tssCorrect += error * error; // E_total
287             tssTesting += tssCorrect;
288         }
289         if (tssCorrect < 0.05)
290             correct++;
291     }
292     //
293     const int datasetCount = testingset.size() + trainingset.size();
294     results[seed][epoch][0] = (tss / (datasetCount * PERCENT_TRAINING));
295     results[seed][epoch][1] = (tssTesting / (datasetCount * PERCENT_TESTING));
296     results[seed][epoch][2] = (correct / (datasetCount * PERCENT_TESTING)) * 100;
297     combinedMSE[seed][epoch] = (tss / (datasetCount * PERCENT_TRAINING));
298     combinedMSETesting[seed][epoch] = (tssTesting / (datasetCount * PERCENT_TESTING));
299     combinedCorrects[seed][epoch] = (correct / (datasetCount * PERCENT_TESTING)) * 100;
300     // Update the weights of the neural network and print the result of the epoch
301     net->update_weights();
302     float correctRate = ((float) (correct) / (float) (testingset.size())) * 100.0F;
303     printf("%s%d/%d correct=%d/%zu %.3f%%\n", (seed < 10) ? "0" : "",
304         seed, epoch, correct, testingset.size(), correctRate);

```

```

305     }
306     delete[] inVec;
307     delete[] outVec;
308     delete[] tarVec;
309 }
310
311 int main(int argc, char **argv) {
312     // Parse given command line arguments
313     if (argc < 2 || argc > 4) {
314         printf("<threadcount> = amount of threads to be used\n");
315         printf("<partitionCount> = amount of threads to be used\n");
316         printf("<attempt> = number of the attempt\n");
317         printf("./lucavinciguerra-bathesis <threadcount>\n");
318         printf("./lucavinciguerra-bathesis <threadcount> <partitionCount>\n");
319         printf("./lucavinciguerra-bathesis <threadcount> <partitionCount> <attempt>\n");
320         return 1;
321     }
322     int threadcount = stoi(argv[1]);
323     int partitionCount = 1;
324     int attempt = 1;
325     if (argc >= 3) {
326         partitionCount = stoi(argv[2]);
327     }
328     if (argc >= 4) {
329         attempt = stoi(argv[3]);
330     }
331     // Read in dataset from the dataset textfile
332     vector<DatasetRow> dataset;
333     if (!readDataset(DATASET_FILENAME, &dataset)) {
334         printf("Could not read dataset from file %s.\n", DATASET_FILENAME);
335         return 1;
336     }
337     printf("Read dataset from file %s with %zu rows.\n", DATASET_FILENAME, dataset.size());
338     // vector that contains all threads that should run in parallel
339     auto *threadpool = new ThreadPool(threadcount);
340     for (int seed = 1; seed <= SEEDS; seed++) {
341         // Partition the dataset into a training set and a testing set for this seed
342         vector<DatasetRow> trainingset, testingset;
343         partitionDataset(dataset, partitionCount, 1, &trainingset, &testingset, seed);
344         printf("starting thread for seed %d with %zu training and %zu testing rows\n",
345             seed, trainingset.size(), testingset.size());
346         // add the thread to the list of threads
347         threadpool->enqueue([seed, trainingset, testingset] {
348             runSeed(seed, trainingset, testingset);
349         });
350     }

```

```
351     delete threadpool;  
352     analyzeResults(threadcount, partitionCount, attempt);  
353     return 0;  
354 }
```

Abbildungsverzeichnis

3.1	Vereinfachte Form des Beispielnetzes aus der n++-Dokumentation welche ein Netz mit 3 Schichten erstellt	14
3.2	Schematischer Ablauf des bestehenden Programms. In jedem Durchlauf wird ein neuronales Netz mit 10000 Epochen trainiert und anschließend getestet. In der Analyse werden die Ergebnisse aller Netze zusammengetragen und in eine Datei geschrieben.	15
3.3	Ausschnitt aus dem bestehenden Code der N++ Bibliothek, in welchem viel Gebrauch von der strtok-Funktion gemacht wird.	20
3.4	Ausschnitt aus dem überarbeiteten Code der N++ Bibliothek, in welchem die strtok-Funktion durch die strtok_r-Funktion ersetzt wurde.	21
3.5	Teil der globalen Variablendefinition in der bestehenden Implementierung von Herr Brening	22
3.6	Verlagerung der globalen Variablen zu lokalen Variablen in der parallelen Implementierung.	23
3.7	Beispielhafter C++-Code zur Verwendung eines Threadpools: Ein Threadpool mit zwei Threads wird erstellt, 50 Aufgaben werden der Warteschlange hinzugefügt und anschließend wird der Threadpool blockierend terminiert.	25

4.1 Vereinfacht dargestelltes Benchmark Skript. Es testet die benötigte Laufzeit der Implementierung mit verschiedenen Thread-Anzahlen und Datensatzgrößen.	30
4.2 Leistungstest auf Apple M1 Pro: Einfluss von Thread-Anzahlen auf Laufzeit bei variierenden Datensatzgrößen, welche den Mittelwert von 5 Durchläufen darstellt. Die Standardabweichung der Laufzeit ist in Tabelle 4.1 einzusehen.	31
4.3 Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 6 Threads	32
4.4 Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 8 Threads	33
4.5 Leistungstest auf AMD Ryzen 5 3600XT: Einfluss von Thread-Anzahlen auf Laufzeit bei variierenden Datensatzgrößen, welche den Mittelwert von 5 Durchläufen darstellt. Die Standardabweichung der Laufzeit ist in Tabelle 4.1 einzusehen.	34
4.6 Leistungstest auf Raspberry Pi 3: Einfluss von Thread-Anzahlen auf Verarbeitungszeit bei variierenden Datensatzgrößen	35
4.7 Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 8 Threads	36
4.8 Leistungstest auf AMD EPYC 7282 mit 2 verfügbaren Kernen: Einfluss von Thread-Anzahlen auf Verarbeitungszeit bei variierenden Datensatzgrößen	37

Literatur

- [1] Alessandro Mazzetti. *Praktische Einführung in neuronale Netze*. ger. Hannover: Heise, 1992. ISBN: 3-88229-011-0.
- [2] F. Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain". eng. In: *Psychological review* 65.6 (1958), S. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. eprint: 13602029.
- [3] D. Geer. "Chip makers turn to multicore processors". In: *Computer* 38.5 (2005), S. 11–13. ISSN: 0018-9162. DOI: 10.1109/MC.2005.160.
- [4] M. Riedmiller und H. Braun. "A direct adaptive method for faster backpropagation learning: the RPROP algorithm". In: *IEEE International Conference on Neural Networks*. 1993, 586–591 vol.1. DOI: 10.1109/ICNN.1993.298623.
- [5] Artur Brening. "Stochastischer Gradientenabstieg zum Trainieren neuronaler Netze: Implementierung und vergleichende Untersuchung des RMSprop-Lernverfahrens für den n++-Simulator". Frankfurt University of Applied Sciences, 2020.
- [6] Manuela Lenzen. *Künstliche Intelligenz. Fakten, Chancen, Risiken*. ger. Originalausgabe. Bd. 2904. C.H. Beck Wissen. München: C.H. Beck, 2020. 128 S. ISBN: 978-3-406-75124-0.
- [7] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), S. 948–960. DOI: 10.1109/TC.1972.5009071.
- [8] V.G. Sidorenko, Kyaw Min Aung und A.S. Petrov. "Automation of Subway Train Scheduling Based on the Multi-Threaded Software Product". In: *2020 International*

- Multi-Conference on Industrial Engineering and Modern Technologies*. 2020, S. 1–6. DOI: 10.1109/FarEastCon50210.2020.9271599.
- [9] E.A. Lee. “The problem with threads”. In: *Computer* 39.5 (2006), S. 33–42. DOI: 10.1109/MC.2006.180.
- [10] John Nickolls u. a. “Scalable Parallel Programming with CUDA”. In: *Queue* 6 (März 2008), S. 40–53. DOI: 10.1145/1401132.1401152.
- [11] Adam Paszke u. a. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems*. Hrsg. von H. Wallach u. a. Bd. 32. Curran Associates, Inc., 2019.
- [12] Martin Riedmiller. “Dokumentation zu N++”. Universität Karlsruhe, 1997.
- [13] R. Bock. *MAGIC Gamma Telescope*. UCI Machine Learning Repository. 2007. DOI: 10.24432/C52C8B.
- [14] Philip Wilkinson. *Parallel programming: Techniques and applications using networked workstations and parallel computers, 2/E*. Pearson Education India, 2006.
- [15] Zbigniew J. Czech. “Shared-memory Programming”. In: *Introduction to Parallel Computing*. Cambridge University Press, 2017, S. 243–282.
- [16] *POSIX, The Open Group Base Specifications Issue 7, 2018 edition*. 2018. URL: <https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html> (besucht am 15.05.2024).
- [17] ashish_rao_2373 auf [geeksforgeeks.org](https://www.geeksforgeeks.org). *Thread Pool in C++*. 2024. URL: <https://www.geeksforgeeks.org/thread-pool-in-cpp> (besucht am 18.05.2024).
- [18] Apple Inc. *MacBook Pro (16", 2021) - Technische Daten*. Techn. Ber. 2021. URL: <https://support.apple.com/de-de/111901> (besucht am 02.05.2024).
- [19] WikiChip. *Ryzen 5 3600XT - AMD*. Techn. Ber. 2020. URL: https://en.wikichip.org/wiki/amd/ryzen_5/3600xt (besucht am 02.05.2024).
- [20] Raspberry Pi Foundation. *Raspberry Pi 3 Model B*. Techn. Ber. 2022. URL: <https://www.raspberrypi.com/products/raspberry-pi-3-model-b> (besucht am 02.05.2024).

- [21] Inc Advanced Micro Devices. *AMD EPYC 7282 specifications*. Techn. Ber. 2024.
URL: <https://www.amd.com/de/products/cpu/amd-epyc-7282> (besucht am 18.05.2024).