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

Kurzfassung

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 Potenziale 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].

Inspiriert von diesem biologischen Vorbild entwickelte Frank Rosenblatt 1958 das

Modell des Perzeptrons - einem künstlichen 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. 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 [4] 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 analysieren, 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

Kapitel 2

Grundlagen

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

Des Weiteren wird die Thematik der Parallelisierung sowohl im allgemeinen Kontext als auch speziell im Zusammenhang mit neuronalen Netzwerken erläutert. Es werden die Vor- und Nachteile dieser Technik beleuchtet und die gängigsten Methoden zur Parallelisierung von Berechnungen in neuronalen Netzwerken 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 Netzwerken und deren Implementierung mittels Thread- und Prozessparallelsierung.

2.1 Grundprinzipien von neuronalen Netzwerken

Neuronale Netzwerke 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 [5]. Diese Netzwerke

können eine Vielzahl von Aufgaben ausführen, von der Bilderkennung bis hin zur Sprachverarbeitung.

Die Funktionsweise eines neuronalen Netzwerks lässt sich grob in zwei Hauptphasen unterteilen: Vorwärtspropagierung und Rückwärtspropagierung. Während der Vorwärtspropagierung fließen die Eingabedaten vorwärts durch das Netzwerk, beginnend mit den Neuronen der Eingabeschicht, welche die Rohdaten empfangen, und endend mit den Neuronen der Ausgabeschicht, welche die Vorhersagen oder Klassifikationen des Netzwerks abbilden. Jedes Neuron in einem neuronalen Netzwerk ist mit anderen Neuronen verbunden, und diese Verbindungen sind mit Gewichten versehen, die die Stärke der Verbindung zwischen den Neuronen darstellen [5].

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

$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right) \tag{2.1}$$

text

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

add cit
tion

Insgesamt ermöglicht die Funktionsweise von neuronalen Netzwerken die Modellierung komplexer Zusammenhänge in Daten und die Durchführung verschiedenster Aufgaben des maschinellen Lernens und der künstlichen Intelligenz.

2.2 Aufbau eines neuronalen Netzwerks

2.2.1 Schichten und ihre Funktionen

Ein mehrlagiges neuronales Netzwerk ist in verschiedene Schichten organisiert, die jeweils spezifische Funktionen erfüllen. Die erste Schicht wird oft als Eingangsschicht bezeichnet und empfängt die Rohdaten oder Merkmale, die dem Netzwerk präsentiert werden. Diese Daten werden dann durch das Netzwerk 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 Netzwerks, komplexe Muster zu lernen und abstrakte Merkmale zu extrahieren. Jede Schicht lernt auf unterschiedlichen Abstraktionsebenen und trägt zur schrittweisen Verbesserung der Leistung des Netzwerks bei.

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

2.2.2 Vorwärtsgerichtete und rückwärtsgerichtete Netzwerke

Neuronale Netzwerke können in zwei Hauptkategorien unterteilt werden lorem ipsum dolor: vorwärtsgerichtete (feedforward) und rückgekoppelte (feedback) Netzwerke. Vorwärtsgerichtete Netzwerke sind die am häufigsten verwendete Architektur in der neuronalen Netzwerkwissenschaft. In diesen Netzwerken fließen die Informationen nur in eine Richtung, von der Eingangsschicht zur Ausgangsschicht, ohne Rückkopplungsschleifen. Das bedeutet, dass die Ausgabe jedes Neurons in einer Schicht nur von den Eingaben der vorhergehenden Schicht abhängt und nicht von den Ausgaben der Neuronen derselben Schicht oder einer späteren Schicht.

Dieses einfache Flussmuster ermöglicht eine effiziente Berechnung und einfache Interpretation der Ergebnisse. Vorwärtsgerichtete Netzwerke eignen sich besonders gut für Anwendungen wie Klassifikation und Regression, bei denen eine direkte Zuordnung von Eingaben zu Ausgaben erfolgt.

Im Gegensatz dazu haben rückgekoppelte Netzwerke Rückkopplungsschleifen, die es ermöglichen, Informationen sowohl vorwärts als auch rückwärts durch das Netzwerk zu propagieren. Diese Art von Netzwerken, auch als rekurrente neuronale Netzwerke (RNNs) bekannt, sind besonders gut geeignet für die Verarbeitung sequenzieller Daten, bei denen der Kontext und die zeitliche Abfolge der Eingaben wichtig sind.

RNNs sind in der Lage, vergangene Informationen zu berücksichtigen und sie in die aktuelle Berechnung einzubeziehen, was sie besonders nützlich für Aufgaben wie Sprachverarbeitung, Zeitreihenanalyse und maschinelles Übersetzen macht.

Die Wahl zwischen vorwärtsgerichteten und rückwärtsgerichteten Netzwerken hängt von den spezifischen Anforderungen der Anwendung ab. Während vorwärtsgerichtete Netzwerke gut für statische Daten und klare Ein-Aus-Beziehungen geeignet sind, bieten rückwärtsgerichtete Netzwerke eine größere Flexibilität und sind besser für die Verarbeitung dynamischer Daten geeignet.

2.3 Anwendungsfälle von neuronalen Netzwerken

Neuronale Netzwerke haben sich besonders im letzten Jahrzehnt als äußerst vielseitige Instrumente erwiesen und finden breite Anwendung in diversen Branchen. Prominente Einsatzgebiete umfassen:

- Bilderkennung und Computer Vision: Eine der bekanntesten Anwendungen von neuronalen Netzwerken ist die Bilderkennung, mit der Objekte, Gesichter, Muster und weitere Elemente in Bildmaterial identifiziert und klassifiziert werden können [6]. Als Beispiel sind zum Beispiel die simple Gesichtserkennung zum Entsperren von Smartphones, sowie die Erkennung von Personen auf Bildern in verschiedenen Cloudservices zu nennen.

- Natürliche Sprachverarbeitung: In der Natürlichen Sprachverarbeitung, auch NLP genannt, kommen neuronale Netzwerke zum Einsatz, um menschenähnliche Sprache zu verstehen, zu interpretieren und zu generieren. Anwendungen erstrecken sich von Chatbots und virtuellen Assistenten bis hin zu maschineller Übersetzung und Analyse von Inhalt in sozialen Medien.
- Mustererkennung und Prognose: Neuronale Netzwerke finden in diversen Domänen Anwendung bei der Mustererkennung und Prognose. Dies umfasst unter anderem das Finanzwesen, Gesundheitswesen und den Verkehr. Sie ermöglichen die Identifikation von Mustern in umfangreichen Datensätzen und die Vorhersage zukünftiger Ereignisse. Weitere wichtige Anwendungsgebiete sind die Prognose von Wetterdaten und die Analyse von Verspätungsdaten öffentlicher Verkehrsmittel.
- Autonome Fahrzeuge: In der Automobilindustrie spielen neuronale Netzwerke eine Schlüsselrolle bei der Entwicklung autonomer Fahrzeuge. Sie werden eingesetzt, um Hindernisse zu erkennen, Verkehrssituationen zu verstehen, Routen zu planen und Fahrzeugfunktionen zu steuern.
- Medizinische Diagnose: Neuronale Netzwerke werden in der medizinischen Bildgebung verwendet, um Krankheiten wie Krebs auf Röntgen- und MRT-Scans zu identifizieren. Sie unterstützen Ärzte auch bei der Diagnose von Krankheiten und der Vorhersage von Behandlungsergebnissen anhand von Patientendaten [6].
- Finanzwesen: Im Finanzsektor kommen neuronale Netzwerke für die Kreditrisikobewertung, Betrugserkennung, Handelsstrategien und Marktanalyse zum Einsatz. Sie unterstützen Finanzinstitute bei fundierten Entscheidungen und der Minimierung von Risiken.

Diese Anwendungsbereiche verdeutlichen die Vielseitigkeit und transformative Kraft neuronaler Netzwerke in diversen Branchen und Disziplinen. Durch kontinuierliche Forschung und Entwicklung werden ihre Fähigkeiten kontinuierlich erweitert, was zu neuen Anwendungen führt.

eventue kürzen oder weglas- sen

2.4 Einführung in die Parallelisierung

Die Parallellisierung 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. Kern der Parallellisierung 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 Parallellisierung existiert, die abhängig von der Problemstellung und der verfügbaren Hardware eingesetzt werden können. Die Task-Parallellisierung 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 Parallellisierung besonders [7].

Ein weiterer Ansatz ist die Datenparallellisierung, 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 Parallellisierung geeignet sind. Manche Probleme beinhalten intrinsische Abhängigkeiten oder Sequenzialität, die eine effektive Parallellisierung erschweren oder gar unmöglich machen.

2.4.1 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.

Ein weiterer 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.

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 wenigeren Systemen auszuführen, da das volle Leistungspotenzial aller Systeme ausgenutzt wird. 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 das Systemdesign, 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 gegenüber transparent zu gestalten [8], jedoch stellten sich diese Bemühungen größtenteils als erfolglos heraus.

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.

2.5 Parallelisierung in vorwärtsgerichteten Netzwerken

2.5.1 Thread- und Prozessparallelsierung

Für die parallele Ausführung des Trainings mehrerer Netzwerke unabhängig voneinander spielt die Thread- und Prozessparallelisierung eine bedeutende Rolle. Diese Techniken bieten Mechanismen, um das Training der Netzwerke 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 Netzwerks 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 Netzwerke, wobei jeder Thread sich auf das Training eines bestimmten Netzwerks konzentriert. Dies kann die Gesamttrainingszeit 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 Netzwerke 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].

2.5.2 Implementierung von Parallelisierungstechniken

Für die Implementierung von Thread- und Prozessparallelisierung in vorwärtsgerichteten Netzwerken können verschiedene Ansätze verfolgt werden. Eine gängige Methode besteht darin, parallele Bibliotheken oder Frameworks zu verwenden, die bereits implementierte

Funktionen für die Thread- und Prozessverwaltung bereitstellen. Beispiele hierfür sind die Verwendung von OpenMP, CUDA oder MPI, je nach den Anforderungen der Anwendung und der zugrunde liegenden Hardwarearchitektur.

Bei der Implementierung von Thread-Parallelisierung können Entwickler Thread-Pools verwenden, um die Ressourcennutzung zu optimieren und die Thread-Erstellungskosten zu minimieren. Die Aufgaben werden in Threads aufgeteilt und in einem Pool von vorab erstellten Threads ausgeführt, was die Ausführungszeit der Aufgaben reduziert und die Gesamtperformance verbessert.

Für die Prozessparallelisierung ist die Implementierung von Mechanismen zur Kommunikation und Koordination zwischen den verschiedenen Prozessen entscheidend. Dies kann die Verwendung von Sockets, Messaging-Systemen wie ZeroMQ oder die gemeinsame Nutzung von Speicherbereichen umfassen, um Daten zwischen den Prozessen auszutauschen und den Trainingsfortschritt zu synchronisieren.

2.5.3 Auswirkungen auf die Leistungsfähigkeit

Ein wesentlicher Vorteil besteht darin, dass durch die parallele Ausführung mehrerer Netzwerke (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 Netzwerke 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 problem-

los 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.

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 die Implementierung wird 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 [9].

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 [10]. Nach der Konfiguration des Netzes können Eingabemuster durch Vorwärtspropagation propagiert werden. Die resultierenden Ausgaben können abgerufen werden, und optional kann durch Rückwärtspropagation des Fehlervektors ein Lernprozess des Netzwerks simuliert 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 [10], 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 Rückwärtspropagation gesetzt, was ohne großen Aufwand möglich ist.

```
#include "n++.h"

#define INPUTS 2
#define OUTPUTS 3
#define LAYERS 3

int main() {
    Net net;
    // Schichten des Netzes erstellen und miteinander verbinden
    int layerNodes[LAYERS] = {INPUTS, 4, OUTPUTS};
    net.create_layers(LAYERS, layerNodes);
    net.connect_layers();
    // Gewichte mit Zufallszahlen zwischen 0 und 0,5 initialisieren
    net.init_weights(0, 0.5);
    // Updatefunktion auf Rückwärtspropagation setzen
    float uparams[5] = {0.1, 0.9, 0, 0, 0};
    net.set_update_f(BP, uparams);
}
```

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 [4] 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 nächsten Abschnitt eingegangen wird. Es wird ein mit Hilfe der n++-Bibliothek ein neuronales Netz erstellt und zunächst wird ein festgelegter Teil des Datensatzes als Trainingsdatensatz genommen, und jede dieser Trainingsdaten wird zunächst vorwärts durch das neuronale Netz propagiert. Anschließend wird der Fehler der resultierenden Ausgabe des Netzes bestimmt und es wird rückwärtspropagiert. Nachdem alle Datensätze aus dem

Trainingsdatensatz vor- und rückwärtspropagiert wurden, wird der verbleibende Teil des Datensatzes als Testdatensatz definiert. Genau wie die Trainingsdatensätze wird jeder Datensatz zuerst vor- und dann rückwärts propagiert. Nach jedem Datensatz wird hier jedoch geschaut, ob die Ausgabe des Netzes korrekt ist, um zu schauen, wie viele Daten des Testdatensatzes korrekt klassifiziert werden. Dieser Prozess über den gesamten Datensatz wird für 10000 Epochen, also 10000 mal ausgeführt, um das Netz lernen zu lassen.

Dieser Lernprozess wird sequenziell für 10 verschiedene Netze mit 10 verschiedenen Startseeds ausgeführt, 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. Ein Durchlauf des bestehenden Programmes benötigt auf einem Apple M1 Pro Prozessor circa 1450 Sekunden.

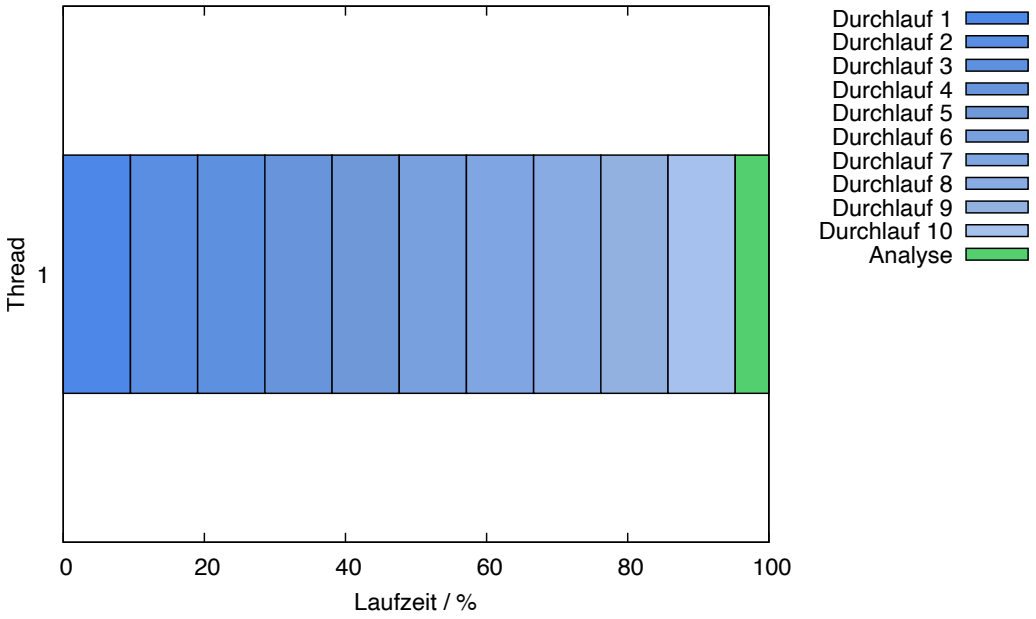

Abbildung 3.2: Schematischer Ablauf des bestehenden Programms. In jedem Durchlauf wird ein neuronales Netz mit 10000 Epochen trainiert und anschließend getestet. In der Analyse werden die Ergebnisse aller Netze zusammengetragen und in eine Datei geschrieben.

Wie in Abbildung 3.2 ersichtlich laufen alle Netze, auf der Grafik in blau darge-

stellt, 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 [11]. 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 [11].

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 Implemen-

tierung, um die Effizienz und Skalierbarkeit auf mehreren Prozessoren oder Rechenkernen zu gewährleisten [12].

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 den gleichen Speicherbereich zugreifen können. Dies kann zu Wettlaufsituationen, inkonsistenten Zuständen und anderen unerwarteten Verhaltensweisen führen, die die Zuverlässigkeit und Korrektheit des Programms beeinträchtigen [13].

Um dieses Problem zu lösen, ist es notwendig, die Abhängigkeit von geteilten Speicherzugriffen soweit 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 im Bezug auf Speicherlecks und Speicherbedarf mit sich bringen, da die Variablen nur für die Zeit ihrer Verwendung gespeichert werden. Darüber hinaus können Datenstrukturen wie Klassen oder Strukturen verwendet werden, um Daten zu kapseln und den Zugriff über klar definierte Schnittstellen zu ermöglichen [13].

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 [13].

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 [12].

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 [12].

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 [12].

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, welche zu der Komplexität von Parallelisierung beitragen.

3.5 Vorstellung der Implementierung

In dieser Sektion wird genauer auf einige Aspekte der Implementierung eingegangen. Dabei werden auch einige Code-Ausschnitte behandelt, um die Änderungen und Implementierung zu veranschaulichen.

3.5.1 Verwendung von threadsicheren Funktionen

Besonders zur Manipulation von Zeichenketten macht sowohl die n++-Bibliothek als auch der Code des Experimentes von Herr 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 (strncmp(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     ...
```

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 inkosistente 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 [14]. 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;
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     ...
```

Diese Verwendung von strtok() tritt sehr häufig in dem n++-Quellcode auf. Auch in dem Code von Herr 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, wie zum Beispiel, wenn man die Portabilität für POSIX-Systeme vor 2001 gewährleisten möchte [14].

3.5.2 Entfernung globaler Variablen

In der bestehenden Implementierung von Herr Brening, von welcher ein Teil in Abbildung 3.3 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 3.3: Teil der globalen Variablen 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 3.4 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<DatasetRow> &trainingset,
243  const vector<DatasetRow> &testingset) {
244     ...
245     const auto net = new Net();
246     const auto inVec = new float[net->topo_data.in_count];
247     const auto outVec = new float[net->topo_data.out_count];
248     const auto tarVec = new float[net->topo_data.out_count];
```

Abbildung 3.4: Verlagerung der globalen Variablen zu lokalen Variablen in der parallelen Implementierung.

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 [15]. 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 [15]. Der verwendete ThreadPool ist simpel aufgebaut, und in einer einzigen C++-Header-Datei implementiert. Abbildung 3.5 zeigt ein Beispielprogramm, welches den Threadpool verwendet.

```cpp
#include "threadpool/threadpool.h"

int main() {
    // Threadpool mit 2 Threads erstellen
    ThreadPool *threadpool = new ThreadPool(2);
    // 50 Beispielaufgaben erstellen und der Warteschlange hinzufügen
    for (int i = 1; i <= 50; i++) {
        threadpool->enqueue([i] { // Abhängigkeiten müssen spezifiziert werden
            printf("Aufgabe %d\n", i);
        });
    }
    // Aufruf des Destruktors terminiert Threads nach Abschluss aller Aufgaben
    delete threadpool;
    return 0;
}
```

Abbildung 3.5: 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.

Kapitel 4

Experimentelle Untersuchungen

4.1 Angewendete Methodik

4.1.1 Testumgebung

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.

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.

4.1.2 Getestete Hardware

Der M1 Pro Prozessor, der im MacBook Pro 16 Zoll verwendet wird, integriert eine heterogene Mehrkernarchitektur, die auf die Parallelverarbeitung von Aufgaben ausgelegt ist.

Der M1 Pro wurde 2021 vorgestellt und ist eine hochskalierte Version des M1 Prozessors. Er verfügt über insgesamt 12 CPU-Kerne, darunter 8 Hochleistungskerne und 4 Effizienzkerne [16]. 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 sowie eine bemerkenswerte parallele Verarbeitungsfähigkeit für multithreaded Anwendungen. Die genaue Hauptspeichergröße beträgt 16 GB RAM mit einer Speicherbandbreite von 200 GBit/s [16]. Auch die Cache-Größe des Prozessors ist sehr hoch, was die Latenz bei Speicherzugriffen zusätzlich vermindert. Verwendet wurde MacOS Sonoma 14.5 mit dem Darwin Kernel Version 23.5.0 und Clang Version 1500.3.9.4 auf einem MacBook Pro 16 Zoll Laptop.

Der AMD Ryzen 5 3600XT ist ein Prozessor aus der Ryzen 3000-Serie von AMD, 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 [17]. Der Prozessor unterstützt DDR4-RAM mit unübertakteten Geschwindigkeiten von bis zu 3200 MHz [17]. 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 [17]. 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.

Der Raspberry Pi 3 ist ein Single-Board-Computer, der von der Raspberry Pi Foundation entwickelt wurde. Ein Single-Board-Computer ist eine vollständige Computerplatine, die alle erforderlichen Komponenten wie Prozessor, Speicher, Ein-/Ausgabeanschlüsse 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 [18]. Im Vergleich zu anderen Pro-

zessoren ist der Raspberry Pi 3 natürlich langsamer. Seine Spezifikationen bieten eine grundlegende Leistung für einfache Computing-Aufgaben und den Betrieb von IoT (Internet der Dinge)-Anwendungen. Der Raspberry Pi 3 ist aufgrund seiner geringen Größe, seines geringen Stromverbrauchs und seiner vielfältigen Einsatzmöglichkeiten beliebt. Er wird häufig in Bildungsprojekten, DIY (Do It Yourself)-Projekten, 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 des Raspberry Pi 3 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 unterschiedlichen 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. Der AMD EPYC 7282 Prozessor ist ein Serverprozessor, 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 [19]. 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.

4.1.3 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.

```
#!/bin/bash
ATTEMPTS=5
PARTITIONS=(8 4 2 1) # Verschiedene Partitionen zum Testen
THREADS=(10 8 6 4 2 1) # Verschiedene Thread Anzahlen zum Testen
AVAILABLE_THREADS=10 # Maximal verfügbare Anzahl an Threads für aktuelles System

function run_attempt {
    # Hier wird ein Durchlauf durchgeführt und getestet
}

# Test für jede mögliche Kombination durchführen
for partition in ${PARTITIONS[@]};do
  for threads in ${THREADS[@]};do
      if ((threads <= AVAILABLE_THREADS));then
        for ((i = 1; i <= ATTEMPTS; i++)); do
          run_attempt "$threads" "$partition" "$i"
        done
      fi
    done
done
```

Abbildung 4.1: Vereinfacht dargestelltes Benchmark Skript. Es testet verschiedenen Thread-Anzahlen und Datensatzgrößen.

4.2 Ergebnisse

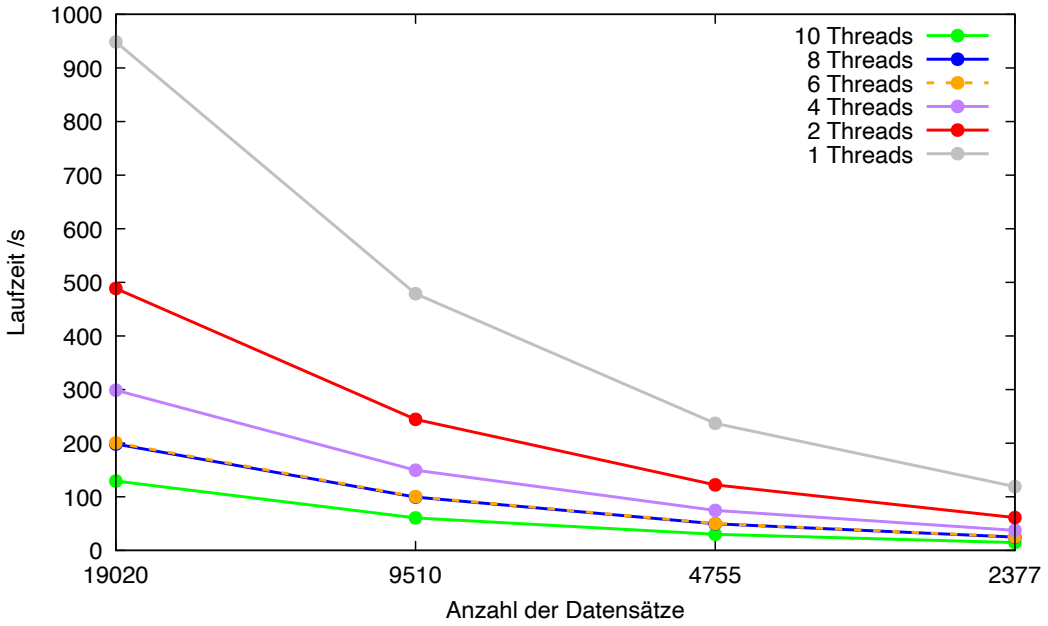

Abbildung 4.2: Leistungstest auf Apple M1 Pro: Einfluss von Thread-Anzahlen auf Verarbeitungszeit bei variierenden Datensatzgrößen

Die Analyse der Benchmark-Ergebnisse auf dem M1 Pro-Prozessor aus Abbildung 4.2 liefert wertvolle Einblicke in die Leistungsfähigkeit seiner Mehrkernarchitektur und der 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 Effizienzkernen 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

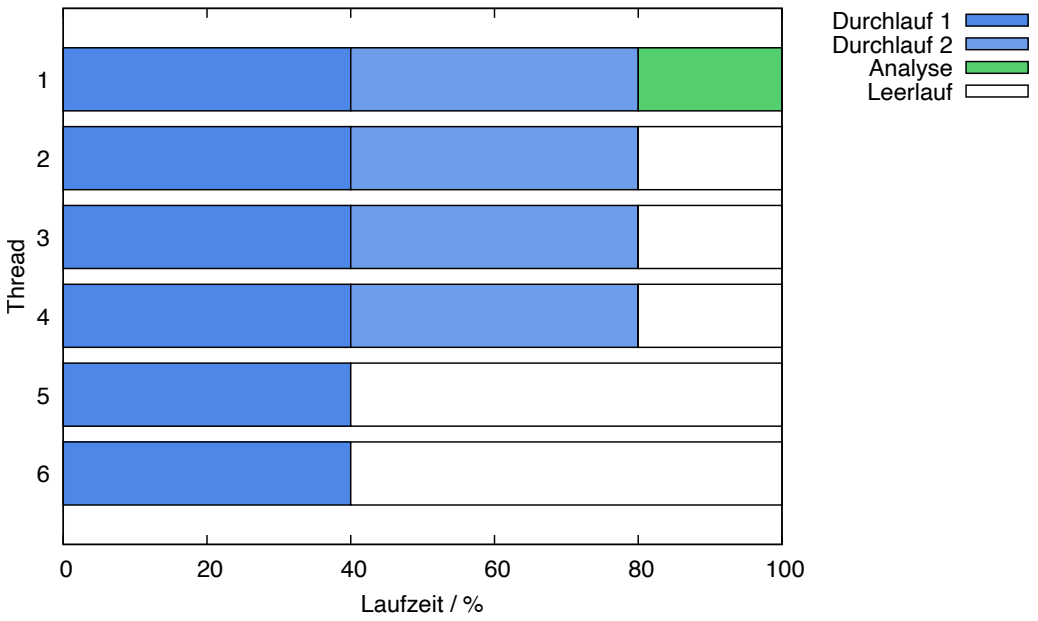

Abbildung 4.3: Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 6 Threads

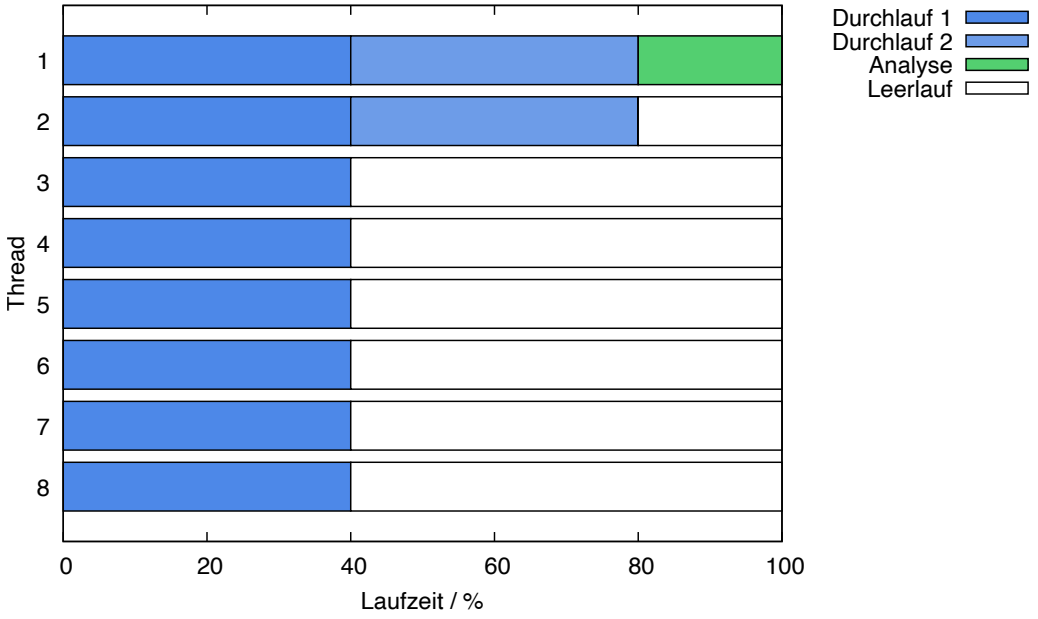

Abbildung 4.4: Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 8 Threads

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.

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, welche in Tabelle 4.1 zu sehen ist, ist insgesamt 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.

Des Weiteren ist der Overhead der Parallelisierung ein wichtiger Aspekt, der bei der Interpretation der Ergebnisse berücksichtigt werden muss. Trotz der Parallelisierung von Aufgaben bleibt der Overhead auf dem M1 Pro-Prozessor in diesem spezifischen Kontext gering, jedoch definitiv nicht vernachlässigbar. Die Skalierung der Leistung bleibt hoch. Für die gesamte Datensatzgröße von 19020 lag die benötigte Zeit bei einem Thread bei 948,6 Sekunden, während mit 10 Threads eine Zeit von 129,4 Sekunden erzielt wurde. Bei einer theoretisch perfekten Skalierung der Parallelisierung wären bei 10 Threads 94,86 Sekunden zu erwarten, jedoch läuft beispielsweise die Auswertung der Ergebnisse nach dem Programmablauf immer sequenziell, und benötigt somit gleich viel Zeit un-

abhängig von der verwendeten Thread Anzahl. Des Weiteren ist der bereits angesprochene Overhead der Parallelisierung ein Faktor. Dieser kann teilweise auf das Betriebssystem zurückzuführen sein, das Ressourcen für die Verwaltung und Koordination der Threads bereitstellen muss, was zu zusätzlicher Latenz führen kann, insbesondere wenn die CPU bereits stark ausgelastet ist. Bei voller CPU-Auslastung können Temperaturthrottling-Mechanismen eingreifen, um die Betriebstemperatur der CPU zu regulieren, was zu vorübergehenden Leistungseinbußen führen kann, da die Taktfrequenz reduziert wird, um die Temperaturen im sicheren Bereich zu halten. Ein weiterer Faktor ist die begrenzte Anzahl von Performance-Kernen auf dem M1 Pro Prozessor, die dazu führen kann, dass bei einer höheren Thread-Anzahl als verfügbaren Performance-Kernen nicht alle Threads auf gleich schnellen Kernen laufen können.

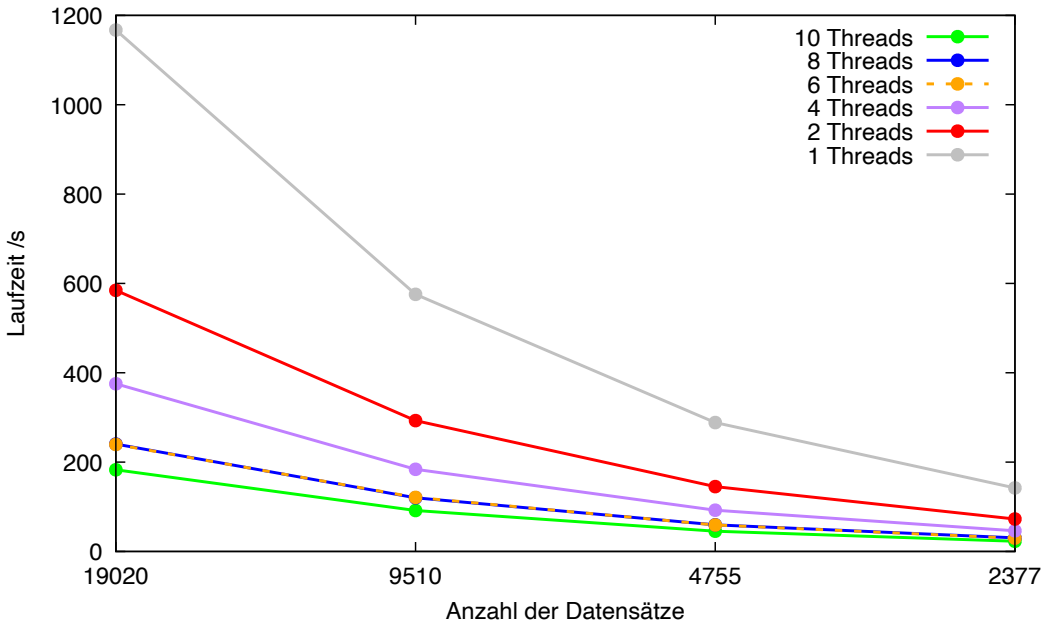

Abbildung 4.5: Leistungstest auf AMD Ryzen 5 3600XT: Einfluss von Thread-Anzahlen auf Verarbeitungszeit bei variierenden Datensatzgrößen

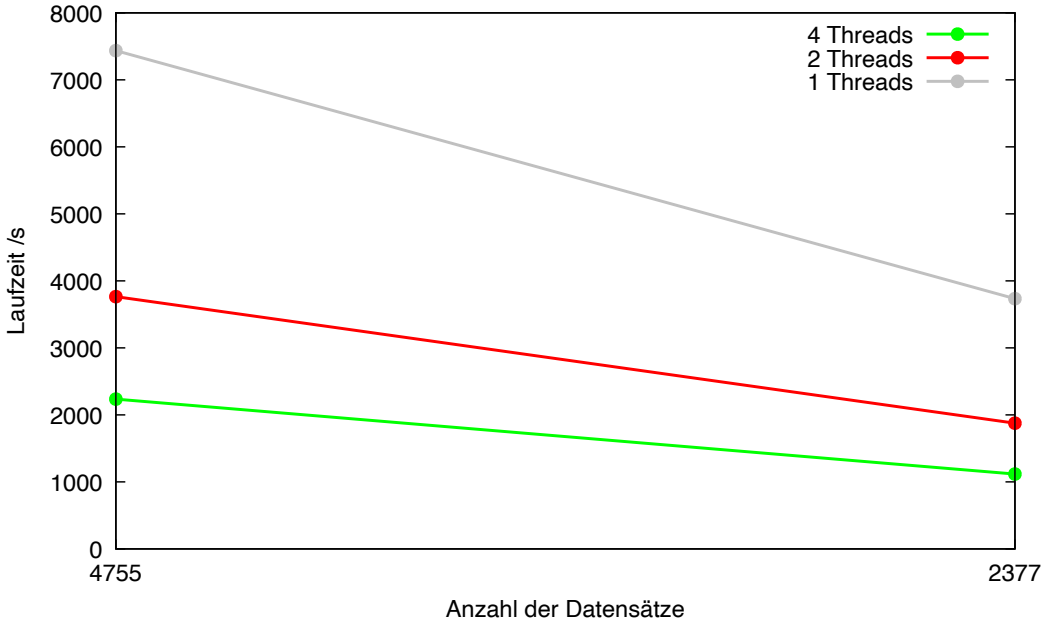

Abbildung 4.6: Leistungstest auf Raspberry Pi 3: Einfluss von Thread-Anzahlen auf Verarbeitungszeit bei variierenden Datensatzgrößen

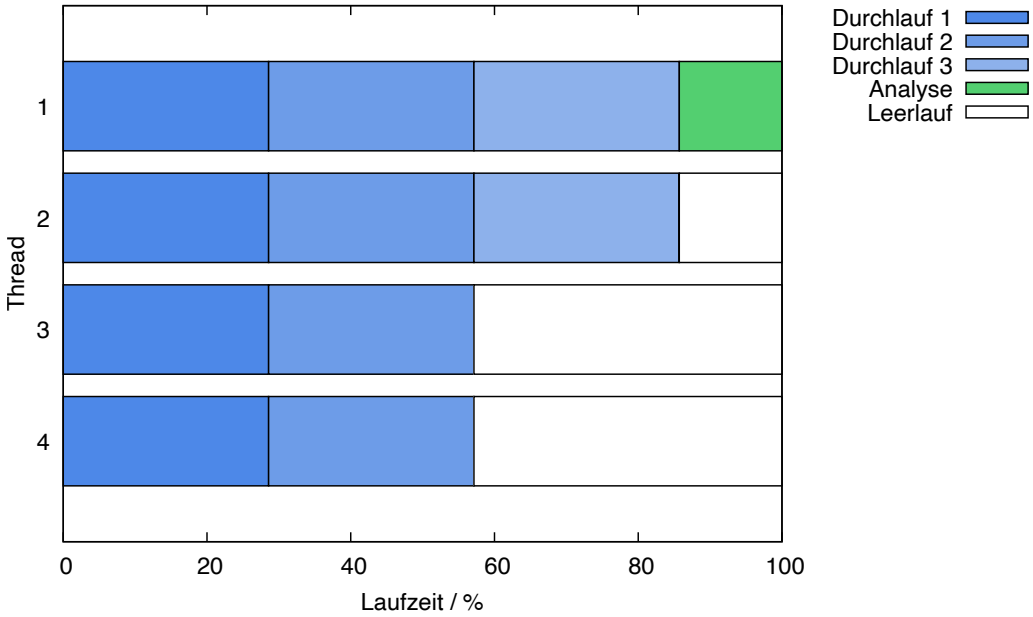

Abbildung 4.7: Verteilung der Aufgaben auf die verfügbaren Threads bei einem Programmablauf mit 8 Threads

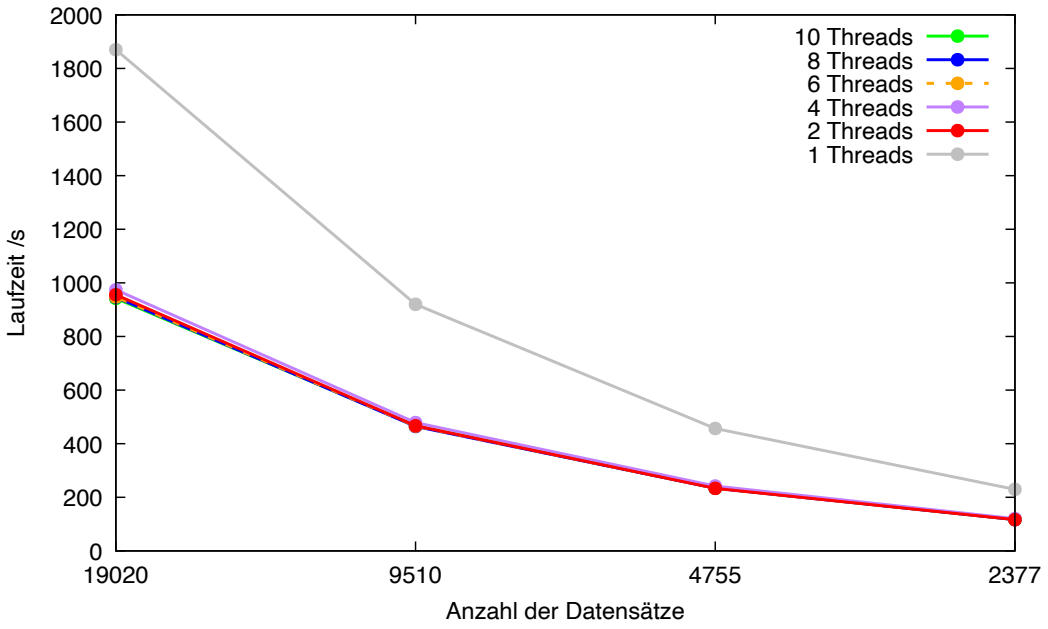

Abbildung 4.8: Leistungstest auf AMD EPYC 7282 mit 2 verfügbaren Kernen: Einfluss von Thread-Anzahlen auf Verarbeitungszeit bei variierenden Datensatzgrößen

4.3 Auswertung

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.

		Ryzen 3600XT		Apple M1 Pro		Raspberry Pi 3		VPS Server	
Threads	Datenmenge	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

Anhang A

Programmcode

```
#include <string>
#include <sstream>
#include <vector>
#include <fstream>
#include <algorithm>
#include <random>

#include "npp/npp.h"
#include "threadpool/threadpool.h"

using namespace std;

#define DATASET_FILENAME "magic_normalized.txt"
#define DATASET_INITIAL_SHUFFLE_SEED 13267
#define SEEDS 10
#define EPOCHS 10000

#define PERCENT_TRAINING 0.8
#define PERCENT_TESTING (1.0 - PERCENT_TRAINING)
#define INPUTS 10
#define OUTPUTS 2
#define HIDDEN 9
#define LAYERS 4

// Results are stored here
double results[SEEDS + 2][EPOCHS + 1][3];
double combinedMSE[SEEDS + 2][EPOCHS + 1];
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 } DatasetRow;
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 DatasetRow 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, DatasetRow *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<DatasetRow> *dataset) {
110     ifstream file(filename);
111     if (!file || !file.is_open()) {
112         return false;
113     }
114     string line;
115     while (getline(file, line)) {
116         DatasetRow 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<DatasetRow> &trainingset,
247                     const vector<DatasetRow> &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] = 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 }
```

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] 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.

[5] 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.

[6] Yann LeCun, Yoshua Bengio und Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), S. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[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] 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.

[10] Martin Riedmiller. "Dokumentation zu N++". Universität Karlsruhe, 1997.

[11] R. Bock. *MAGIC Gamma Telescope*. UCI Machine Learning Repository. 2007. DOI: 10.24432/C52C8B.

[12] Philip Wilkinson. *Parallel programming: Techniques and applications using networked workstations and parallel computers, 2/E*. Pearson Education India, 2006.

[13] Zbigniew J. Czech. "Shared-memory Programming". In: *Introduction to Parallel Computing*. Cambridge University Press, 2017, S. 243–282.

[14] *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).

[15] ashish_rao_2373 auf geeksforgeeks.org. *Thread Pool in C++*. 2024. URL: https://www.geeksforgeeks.org/thread-pool-in-cpp (besucht am 18. 05. 2024).

[16] Apple Inc. *MacBook Pro (16", 2021) - Technische Daten*. Techn. Ber. 2021. URL: https://support.apple.com/de-de/111901 (besucht am 02. 05. 2024).

[17] WikiChip. *Ryzen 5 3600XT - AMD*. Techn. Ber. 2020. URL: https://en.wikichip.org/wiki/amd/ryzen_5/3600xt (besucht am 02. 05. 2024).

[18] 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).

[19] 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).