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 2023 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 2023

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]. In Anbetracht 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 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 wir 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 [4]. 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, wleche 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 [4].

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 Inputs von den Neuronen der vorherigen Schicht, multipliziert diese Inputs mit den entsprechenden Gewichten und summiert sie. Anschließend wird eine Aktivierungsfunktion auf die gewichtete Summe angewendet, um die Ausgabe des Neurons zu berechnen, die dann an die Neuronen der nächsten Schicht weitergeleitet wird.

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.

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 Neuronen und deren Verbindungen

Ein künstliches neuronales Netzwerk besteht aus

Ein neuronales Netzwerk besteht aus einer Vielzahl von Neuronen, die in einem komplexen Netzwerk miteinander verbunden sind. Jedes Neuron empfängt Eingaben von anderen Neuronen oder von externen Quellen und verarbeitet diese Informationen, bevor es Signale an andere Neuronen weitergibt. Die Verbindungen zwischen den Neuronen werden durch Gewichte repräsentiert, die anzeigen, wie stark die Verbindung zwischen zwei Neuronen ist. Diese Gewichte werden während des Trainingsprozesses des neuronalen Netzwerks angepasst, um eine optimale Leistung zu erreichen.

Die Funktionsweise eines Neurons kann vereinfacht als eine Summation der Eingaben multipliziert mit den entsprechenden Gewichten beschrieben werden, gefolgt von der Anwendung einer Aktivierungsfunktion. Diese Aktivierungsfunktion bestimmt, ob das Neuron aktiviert wird und Signale an die nächsten Neuronen weitergibt. Durch diese Schichtung und Verbindung der Neuronen kann das neuronale Netzwerk komplexe Muster erkennen und Informationen verarbeiten.

2.2.2 Schichten und ihre Funktionen

Ein neuronales Netzwerk ist in der Regel 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.

2.2.3 Vorwärtsgerichtete und rückwärtsgerichtete Netzwerke

Neuronale Netzwerke können in zwei Hauptkategorien unterteilt werden lorem ipsum dolor: vorwärtsgerichtete (feedforward) und rückwärtsgerichtete (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 rückwärtsgerichtete 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 [5]. 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 [5].

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.

2.4 Einführung in die Parallelisierung

2.4.1 Definition und Herangehensweise

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 [6].

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 [6].

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 [6].

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.

Zusammenfassend lässt sich festhalten, dass die Parallellisierung ein leistungsstarker Ansatz ist, um die Rechenleistung von Algorithmen zu maximieren. Durch die Verteilung von Aufgaben oder Daten auf mehrere Ressourcen können erhebliche Geschwindigkeitsverbesserungen erzielt werden. Die Auswahl der geeigneten Parallellisierungsstrategie erfordert jedoch eine sorgfältige Analyse der spezifischen Anforderungen und Rahmenbedingungen einer Anwendung.

2.4.2 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 [7], 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.

Insgesamt bietet die Parallelisierung viele Vorteile, die zur Leistungssteigerung von Computersystemen beitragen können. Jedoch ist es wichtig, die potenziellen Herausforderungen und Nachteile zu berücksichtigen und eine sorgfältige Planung und Implementierung sicherzustellen, um die bestmöglichen Ergebnisse zu erzielen.

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 [6].

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 [6].

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 [6].

Die Wahl zwischen Thread- und Prozessparallelisierung hängt von verschiedenen Faktoren ab, darunter die Hardwarearchitektur, die Natur der Netzwerke und die Kommunikationsanforderungen zwischen den Trainingseinheiten. Eine sorgfältige Analyse dieser Faktoren ist entscheidend, um die optimale Parallelisierungsstrategie für das Training mehrerer Netzwerke unabhängig voneinander zu bestimmen.

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.

Es ist wichtig, bei der Implementierung von Parallelisierungstechniken auf Aspekte wie Datenkonsistenz, Synchronisierung und Ressourcenmanagement zu achten. Die richtige Balance zwischen Parallelisierung und Overhead ist entscheidend, um die Gesamtperformance der Anwendung zu maximieren. Durch den Einsatz von geeigneten Werkzeugen, Techniken und Best Practices können Entwickler eine effiziente und skalierbare Parallelisierung in vorwärtsgerichteten Netzwerken erreichen.

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 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.

Zusammenfassend ermöglicht die Implementierung von Parallelisierungstechniken eine effizientere Nutzung von Ressourcen, beschleunigt Trainings- und Inferenzvorgänge und verbessert die Skalierbarkeit der Anwendung. Dies trägt dazu bei, die Entwicklung und Bereitstellung von neuronalen Netzwerken in großen Maßstäben zu erleichtern und ermöglicht es, schnellere Fortschritte in der Forschung und Anwendung von KI-Technologien zu erzielen.

Kapitel 3

Implementierung der Parallelisierung in N++

3.1 Erläuterung des N++ Simulatorkerns

N++ ist ein Simulator für neuronale Netze, der als Forschungsprojekt an der Frankfurt University of Applied Sciences 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.

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 Standartbibliothek 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 [8]. 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, 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 [8].

```
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

3.2 Bestehender Code

Als bestehender Code wird ein Experiment aus der Bachelorarbeit von [9]

3.3 Voraussetzungen

3.3.1 Entfernung von geteilten Speicherzugriffen

3.3.2 Verlagerung der zu parallelisierenden Routine

3.4 Vorstellung der Implementierung

3.4.1 Verwendung von threadsicheren Funktionen

3.4.2 Entfernung globaler Variablen

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 und auf MacOS der Clang Compiler 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 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 habe ich 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 Threads verfügt.

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

```bash
#!/bin/bash
EXECUTABLE="./lucavinciguerra-bathesis"
ATTEMPTS=5
PARTITIONS=(8 4 2 1) # Partitions to be tested
THREADS=(10 8 6 4 2 1) # Thread counts to be tested
AVAILABLE_THREADS=10 # Amount of threads to be used
CSV_FILE="benchmark_results.csv"

# Run the attempt with the given values
function run_attempt {
  local threadcount=$1
  local partitioncount=$2
  local attempt=$3
  echo -e "$partitioncount partitions / $threadcount threads $attempt "
  local filename="attempt_${attempt}_${partitioncount}p_${threadcount}t.txt"
  start=$(date +%s.%3N)
  { time $EXECUTABLE > /dev/null; } 2>> "$filename"
  end=$(date +%s.%3N)
  echo "TOTAL TIME: $(echo "$end - $start" | bc) seconds" >> "$filename"
  echo "$threadcount,$partitioncount,$attempt,$(echo "$end - $start" | bc)" >> "$CSV_FILE"
}

# Run the benchmarks for all possible combinations
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

4.2 Ergebnisse

4.2.1 Apple M1 Pro

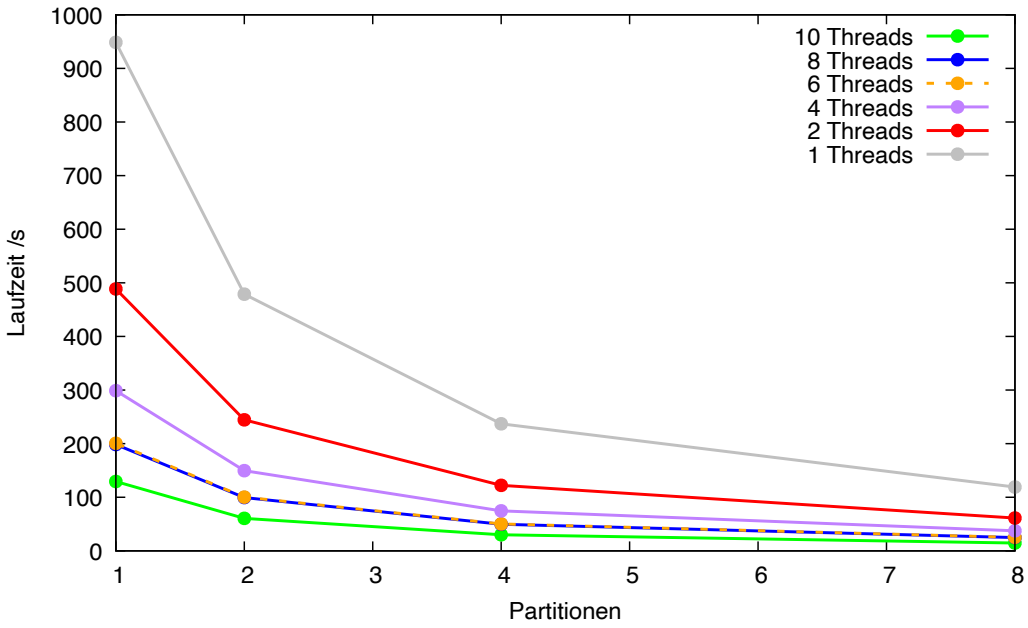

Abbildung 4.2: Ergebnisse der Leistungstests verglichen nach Thread Anzahl

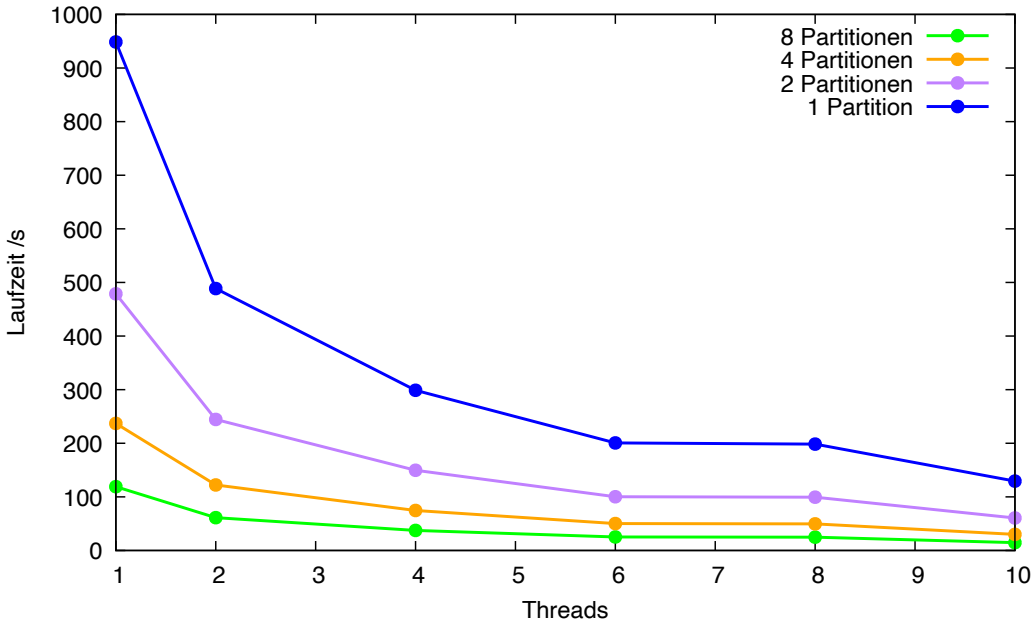

Abbildung 4.3: Ergebnisse der Leistungstests verglichen nach Datensatzgröße

4.2.2 AMD Ryzen 3600XT

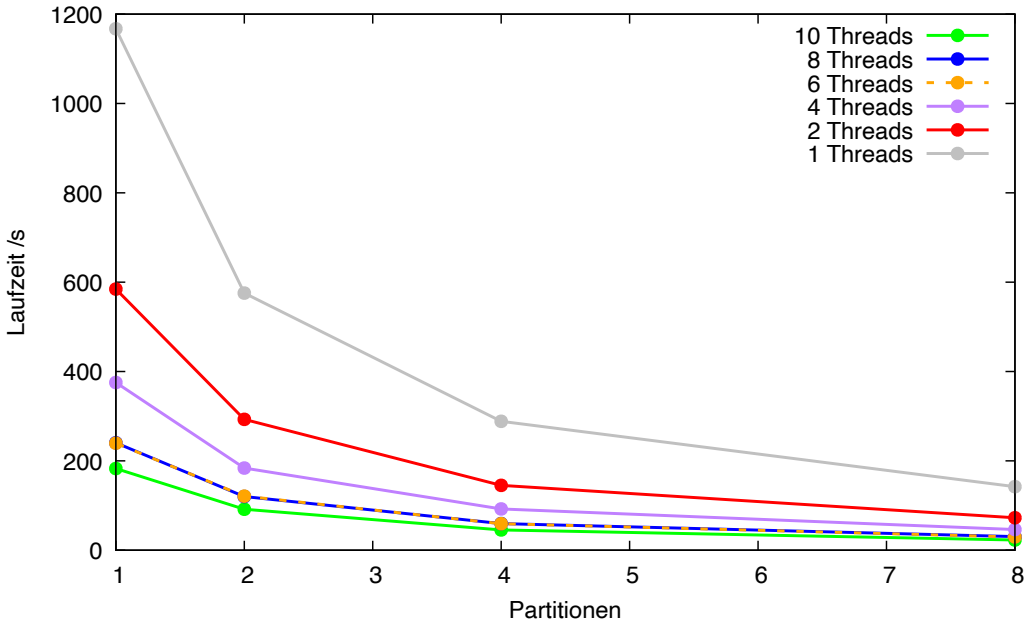

Abbildung 4.4: Ergebnisse der Leistungstests verglichen nach Thread Anzahl

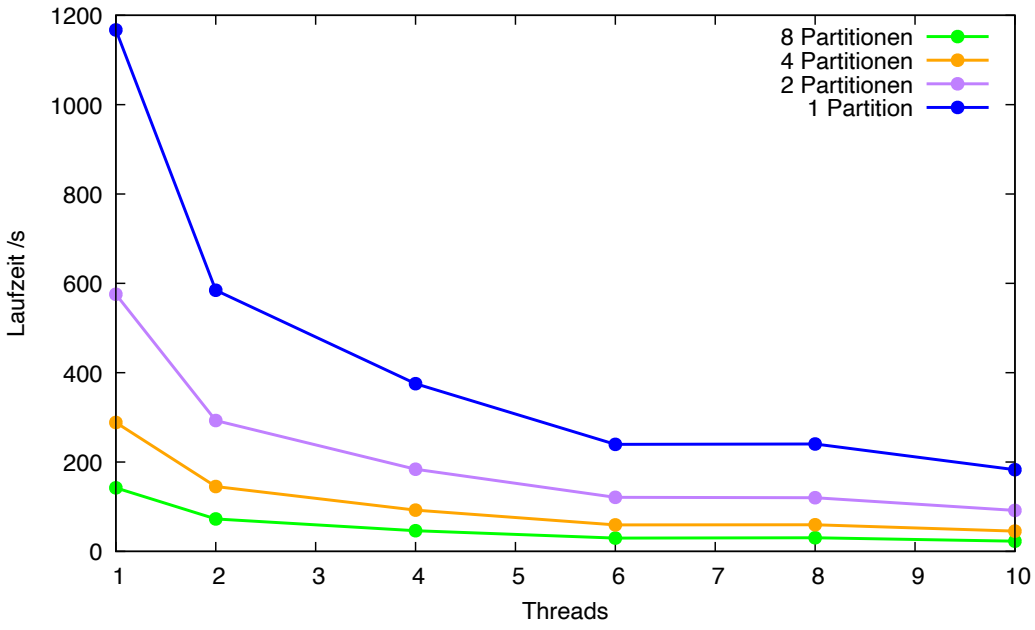

Abbildung 4.5: Ergebnisse der Leistungstests verglichen nach Datensatzgröße

4.2.3 Raspberry Pi 3

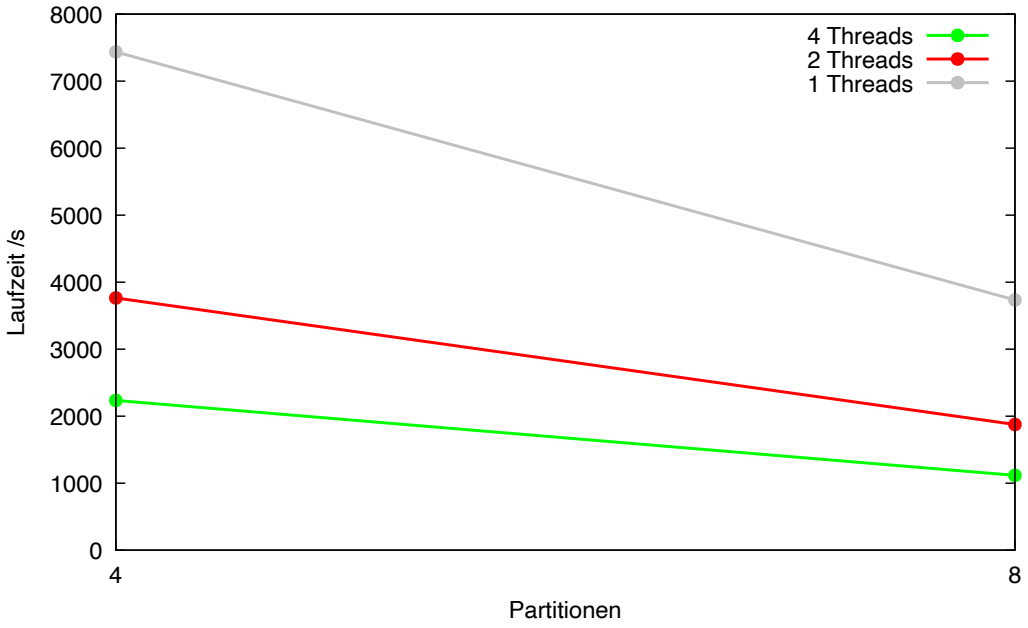

Abbildung 4.6: Ergebnisse der Leistungstests verglichen nach Thread Anzahl

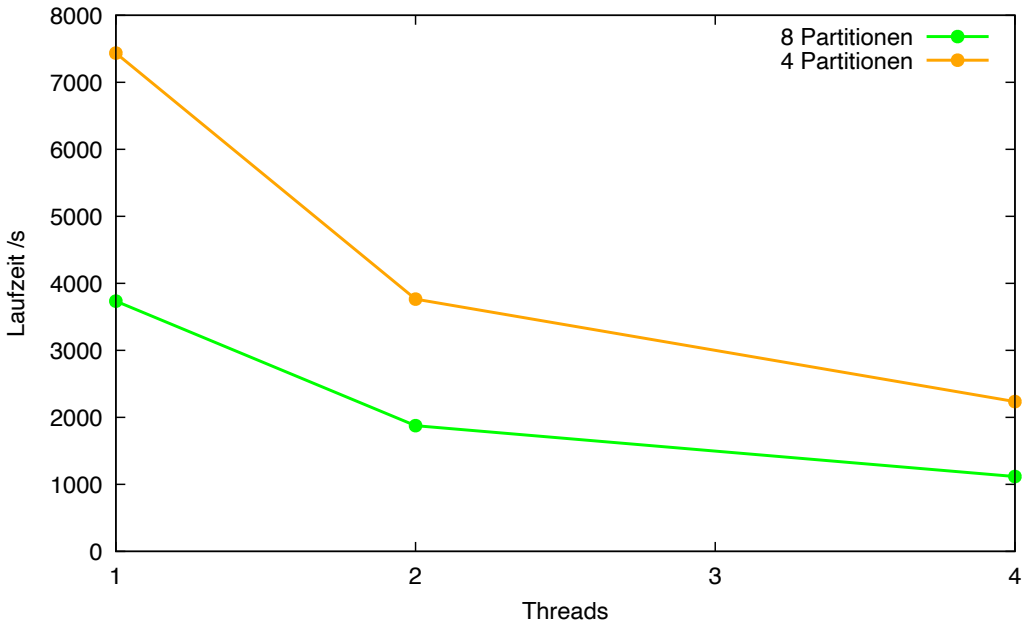

Abbildung 4.7: Ergebnisse der Leistungstests verglichen nach Datensatzgröße

4.2.4 Cloud Server

4.3 Auswertung

Kapitel 5

Fazit und Ausblick

Anhang A

Programmcode

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Martin Riedmiller. "Dokumentation zu N++". Fachhochschule Frankfurt, 1997.

[9] 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.