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, die Organisation von Multiagentensystemen in vogelähnlichen Schwärmen oder die 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. [2].

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 [4]. 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 [1]. 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

Das folgende Kapitel legt die Grundlagen für künstliche neuronale Netzwerke dar, indem es detailliert auf ihre Struktur und Funktionsweise eingeht, insbesondere für vorwärtsgerichtete Netzwerke. Dabei wird ein umfassender Überblick über die potenziellen Anwendungsbereiche von künstlichen neuronalen Netzwerken gegeben, wobei deren Rolle in verschiedenen Bereichen wie Bilderkennung, Sprachverarbeitung und Mustererkennung hervorgehoben wird.

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. Dabei wird besonders auf die Parallelisierung von Berechnungen innerhalb vorwärtsgerichteter Netzwerke eingegangen.

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. Durch das fundierte Verständnis der zugrunde liegenden Konzepte können wir die Potenziale und Herausforderungen der Parallelisierung in diesem spezifischen Kontext besser einschätzen und geeignete Ansätze zur Leistungssteigerung identifizieren.

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 als Neuronen bezeichnet werden. 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 Daten durch das Netzwerk, beginnend mit den Eingangsneuronen, die die Rohdaten empfangen, und endend mit den Ausgangsneuronen, die die Vorhersagen oder Klassifikationen des Netzwerks liefern. 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.

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 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: 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 als äußerst vielseitige Werkzeuge erwiesen und finden Anwendung in einer breiten Palette von Bereichen. Hier sind einige der prominentesten Anwendungsfälle:

Bilderkennung und Computer Vision: Neuronale Netzwerke werden häufig in Bilderkennungsanwendungen eingesetzt, um Objekte, Gesichter, Muster und vieles mehr in Bildern zu identifizieren und zu klassifizieren. Computer Vision, die Fähigkeit von Computern, visuelle Informationen zu interpretieren und zu verstehen, wird durch den Einsatz von neuronalen Netzwerken erheblich verbessert.

Natürliche Sprachverarbeitung (NLP): In der NLP werden neuronale Netzwerke verwendet, um menschenähnliche Sprache zu verstehen, zu interpretieren und zu generieren. Anwendungen reichen von Chatbots und virtuellen Assistenten bis hin zu maschineller Übersetzung und Sentimentanalyse in sozialen Medien.

Mustererkennung und Vorhersage: Neuronale Netzwerke werden häufig für die Mustererkennung und Vorhersage in verschiedenen Domänen eingesetzt, einschließlich Finanzwesen, Gesundheitswesen, Verkehr und mehr. Sie können Muster in großen Datensätzen erkennen und Prognosen für zukünftige Ereignisse treffen.

Autonome Fahrzeuge: In der Automobilindustrie spielen neuronale Netzwerke eine entscheidende Rolle bei der Entwicklung autonomer Fahrzeuge. Sie werden verwendet, um Hindernisse zu erkennen, Verkehrssituationen zu verstehen, Routen zu planen und Fahrzeugfunktionen wie Lenkung und Bremsen zu steuern.

Medizinische Diagnose: Neuronale Netzwerke werden in der medizinischen Bildgebung eingesetzt, um Krankheiten wie Krebs auf Röntgenbildern und MRT-Scans zu erkennen. Sie unterstützen auch Ärzte bei der Diagnose von Krankheiten und der Vorhersage von

Behandlungsergebnissen anhand von Patientendaten.

Finanzwesen: Im Finanzwesen werden neuronale Netzwerke für die Kreditrisikobewertung, Betrugserkennung, Handelsstrategien und die Analyse von Markttrends eingesetzt. Sie helfen Finanzinstituten dabei, fundierte Entscheidungen zu treffen und Risiken zu minimieren.

Diese Anwendungsfälle verdeutlichen die Vielseitigkeit und die transformative Kraft von neuronalen Netzwerken in verschiedenen Branchen und Disziplinen. Durch kontinuierliche Forschung und Entwicklung werden ihre Fähigkeiten ständig erweitert, was zu neuen und aufregenden Anwendungen führt.

2.4 Einführung in die Parallelisierung

2.4.1 Definition und Herangehensweise

Die Parallelisierung stellt einen zentralen Ansatz dar, um die Leistungsfähigkeit von Computersystemen zu steigern. Im Kern bedeutet Parallelisierung, dass Aufgaben oder Berechnungen gleichzeitig und unabhängig voneinander ausgeführt werden, anstatt sequenziell abzulaufen. Dieser Ansatz findet breite Anwendung in einer Vielzahl von Bereichen, darunter die High-Performance-Computing (HPC), Datenverarbeitung, Simulationen, künstliche Intelligenz und viele weitere.

Es gibt verschiedene Herangehensweisen zur Parallelisierung, die je nach Art des Problems und der verfügbaren Hardware eingesetzt werden können. Die Task-Parallelisierung zielt darauf ab, die Ausführung von Aufgaben auf mehrere Prozessoren oder Kerne aufzuteilen. Diese Art der Parallelisierung eignet sich besonders für Anwendungen mit vielen gleichzeitig auszuführenden Aufgaben, wie beispielsweise parallele Suchalgorithmen oder Simulationen von physikalischen Systemen.

Eine weitere Herangehensweise ist die Datenparallelisierung, bei der ein Problem in kleinere Teile zerlegt wird, die jeweils auf unterschiedlichen Datensätzen arbeiten. Dieser Ansatz eignet sich besonders gut für Anwendungen, bei denen eine große Menge an Daten gleichzeitig verarbeitet werden muss, wie beispielsweise bei der Bildverarbeitung

oder beim maschinellen Lernen.

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

Zusammenfassend lässt sich sagen, dass die Parallelisierung eine leistungsstarke Technik ist, um die Rechenleistung von Computersystemen zu maximieren. Durch die Aufteilung von Aufgaben oder Daten auf mehrere Ressourcen können erhebliche Geschwindigkeitsverbesserungen erzielt werden. Die Wahl der geeigneten Parallelisierungsstrategie 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 können Ergebnisse schneller erzielt werden, was insbesondere bei rechenintensiven Anwendungen von Vorteil ist.

Ein weiterer Vorteil der Parallelisierung liegt in der Skalierbarkeit. Indem Aufgaben oder Daten auf mehrere Ressourcen aufgeteilt werden, können Systeme leichter an wachsende Anforderungen angepasst werden. 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. Durch die effiziente Nutzung von Prozessoren oder anderen Hardware-Ressourcen können Engpässe reduziert und die Gesamtleistung des Systems verbessert werden.

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. Die Entwicklung paralleler Algorithmen und die Verwaltung von parallelen Prozessen erfordern spezifisches Fachwissen und

können komplex sein. Darüber hinaus können sich Probleme wie Datenabhängigkeiten, Wettlaufsituationen (englisch: race conditions) und Synchronisationskonflikte ergeben, die die Entwicklung und Fehlerbehebung erschweren.

Ein weiterer Nachteil ist die potenzielle Zunahme des Energieverbrauchs. Obwohl die Parallelisierung die Leistungsfähigkeit von Systemen verbessern kann, kann sie auch zu einem erhöhten Energiebedarf führen, insbesondere wenn nicht effizient implementiert. 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.

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.

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.

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 gemein-

same 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 [3]. 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 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 [3].

```
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 }
```

Listing 1: Vereinfachte Form des Beispielnetzes aus der N++-Dokumentation

3.2 Bestehender Code

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 Verfahren A: ein neues Verfahren zur Steuerung von ...

Kapitel 5

Ergebnisse

Kapitel 6

Fazit und Ausblick

Anhang A

Programmcode

Literatur

[1] 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.

[2] Alessandro Mazzetti. *Praktische Einführung in neuronale Netze*. ger. Hannover: Heise, 1992. ISBN: 3-88229-011-0.

[3] Martin Riedmiller. *Dokumentation zu N++*. de. 1997.

[4] 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.