HAW HAMBURG

Bachelorarbeit

Iven Petersen

Untersuchung von Machine Learning Algorithmen auf einem FPGA zur Klassifikation

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Iven Petersen

Untersuchung von Machine Learning Algorithmen auf einem FPGA zur Klassifikation

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Björn Gottfried
Zweitgutachter: M. Sc. Christian Stange

Eingereicht am: 15. April 2025

Iven Petersen

Thema der Arbeit

Untersuchung von Machine Learning Algorithmen auf einem FPGA zur Klassifikation

Stichworte

FPGA, maschinelles Lernen, Klassifikation, binarisierte neuronale Netze, Nächste-Nachbarn-Klassifikation, Entscheidungsbäume

Kurzzusammenfassung

In dieser Bachelorarbeit werden der k-nächste Nachbar Algorithmus, binarisierte neuronale Netze und Entscheidungsbäume zur Klassifikation mittels FPGA untersucht. Durch Abwägung und Untersuchung der einzelnen Algorithmen wird der Entscheidungsbaum als besonders geeignet befunden. Ein Entscheidungsbaum wird auf einem FPGA implementiert und getestet. Der MNSIT-Testdatensatz konnte mit einer Genauigkeit von 88,47 % klassifiziert werden. Dabei wurden die FPGA Ressourcen kaum ausgelastet.

Iven Petersen

Title of Thesis

Analyzing machine learning algorithms on an FPGA for classification

Keywords

FPGA, Machine Learning, classification, binarized neural networks, Nearest neighbor classification, decision trees

Abstract

In this bachelor thesis, the k-nearest neighbor algorithm, binarized neural networks and decision trees for classification using FPGA are investigated. By consideration and examining the individual algorithms, the decision tree is found to be particularly suitable. A decision tree is implemented and tested on an FPGA. The MNSIT test data set could be classified with an accuracy of 88.47 %. The FPGA resources were hardly utilized.

Inhaltsverzeichnis

Abkürzungen

kNN k-nächste Nachbarn Algorithmus.

AXI Advanced eXtensible Interface.

BNN Binarisiertes Neuronales Netzwerk.

CLB Configurable Logic Block.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

DDR-RAM Double Data Rate - Random Access Memory.

FF Flip-Flop.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

KI Künstliche Intelligenz.

LUT Lookup Table.

Mb Megabit.

ML Machine Learning.

MNIST Modified National Institute of Standards and Technology database.

NN Neuronales Netzwerk.

PSM Schaltmatrizen.

PSP Programmierbare Verbindungspunkte.

RAM Random Access Memory.

ReLU Rectified Linear Unit.

STE Straight-Through Estimator.

VHDL Very High Speed Integrated Circuits Hardware Description Language.

1 Einleitung

Die rasante Entwicklung im Bereich der künstlichen Intelligenz (KI) und des Machine Learning (ML) hat in den letzten Jahren zu einer Vielzahl von Innovationen und Anwendungen geführt [20]. Eine der zentralen Herausforderungen in diesem Kontext ist die effiziente und schnelle Klassifikation von Signalen, wie sie beispielsweise in der Bild und Spracherkennung oder im Bereich der autonomen Systeme erforderlich sind. ML-Algorithmen haben sich hierbei als besonders leistungsfähig erwiesen. Insbesondere die Fähigkeit von ML-Algorithmen, verschiedene Datenmengen zu analysieren und daraus Muster zu klassifizieren, hat viele Bereiche revolutioniert. Gleichzeitig hat sich die Hardwaretechnologie weiterentwickelt, um den hohen Rechenanforderungen dieser Algorithmen gerecht zu werden. Eine dieser Technologien ist das Field Programmable Gate Array (FPGA), ein anpassbarer Halbleiterchip, der für spezielle Aufgaben optimiert werden kann.

1.1 Problembeschreibung

Die voranschreitende technische Entwicklung bringt viele Herausforderungen mit sich. Diese Herausforderungen sind zum Beispiel steigende Datenmengen durch bessere Auflösung von Videoübertragung oder Tonaufnahmen. In einem System wie in Abbildung 1.1, welches die Datenübertragung zeigt, ist der limitierende Faktor die Übertragung der Rohdaten an die Datenverarbeitung. Eine naheliegende Idee ist, die nicht benötigten Daten vor der Übertragung herauszufiltern, wie im unteren Teil der Abbildung 1.1. So werden nur die benötigten Daten an die weitere Datenverarbeitung gesendet. Diese Filterung soll durch Klassifikation von KI erfolgen. Dabei soll der Sensor nicht übermäßig in den Spezifikationen verändert werden. Stromverbrauch, Gewicht und Abmessungen sollen möglichst identisch bleiben.

Gerade durch den steigenden Stromverbrauch von KI-Modellen [23], ist eine effiziente Ausführung dieser Modelle notwendig. Die größer werdenden Datenmengen erfordern

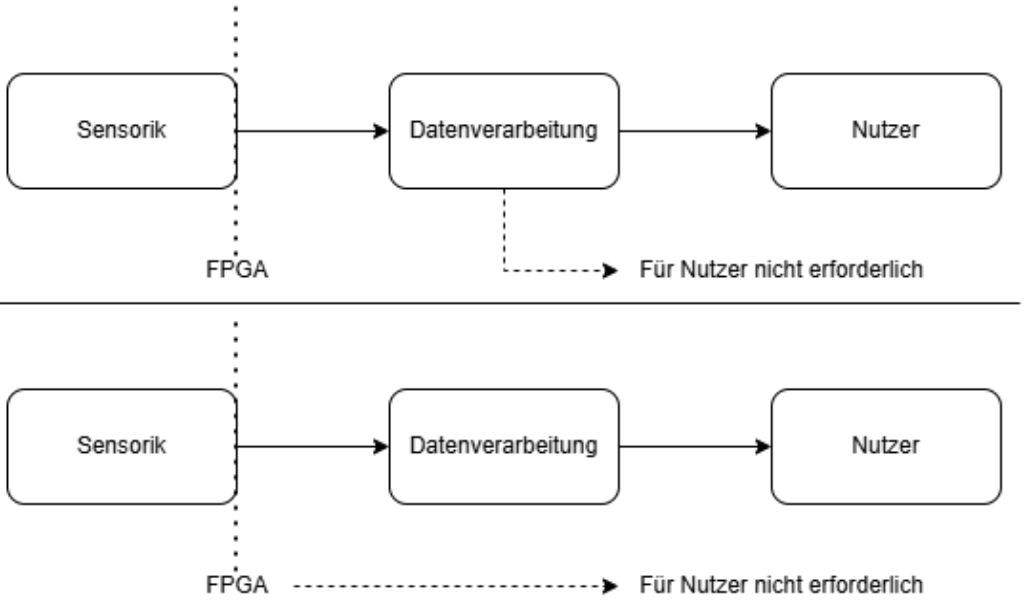


Abbildung 1.1: Datenübertragung des Systems

leistungsstarke Halbleitertechnologie, um die komplexen KI-Modelle für Echtzeitanwendung einzusetzen. Eine Methode, um energieeffiziente Klassifizierung durch KI-Modelle umzusetzen, ist die Implementierung eines trainierten ML-Algorithmus auf einem FPGA. Gegenüber einer Central Processing Unit (CPU) oder Graphics Processing Unit (GPU) ist der FPGA explizit auf die Anwendung programmiert und bietet durch parallele Verarbeitung und ausbleibendem Softwareüberhang bessere Leistung als viele CPUs und GPUs[14]. Somit lässt sich eine energieeffizientere Klassifizierung erreichen.

Die Implementierung auf FPGAs bringen jedoch Herausforderungen mit sich. Neben der Energieeffizienz sollten auch die verfügbaren Ressourcen wie Flip-Flops (FFs) und Lookup Tables (LUTs) effektiv genutzt werden. Die Kosten eines FPGAs hängen maßgeblich von dessen Anzahl an Logikzellen und LUTs ab. Zusätzlich ist bei der Realisierung in Hardware die Gatterlaufzeit[1] und der damit verbundene kritische Pfad[2] zu beachten.

Für die Entwicklung von Klassifikations-Hardware auf FPGA Basis ist eine effektive Nutzung der Ressourcen wie FFs, DSPs und LUTs erforderlich. Für die Implementierung von ML auf einem FPGA ist die Wahl des ML-Algorithmus wichtig, um die Vorteile des FPGAs zu nutzen und effiziente Klassifizierung zu realisieren.

[1] Laufzeit von Signalen von einem Eingang zu einem Ausgang des Gatters
[2] Größte Summe der Verzögerungen einzelner Schaltungskomponenten in Reihe

Somit stellt sich die Frage, welche ML-Algorithmen sich gut eignen. Welche Ressourcen werden dafür benötigt und welche Genauigkeit lässt sich erreichen?

1.2 Ziel und Aufbau der Bachelorarbeit

Diese Arbeit wird grundlegende Ergebnisse über die Eignung verschiedener ML-Algorithmen für die Implementierung auf einem FPGA vorlegen. Unter Berücksichtigung der Problembeschreibung handelt es sich um Klassifikation, zur Filterung von relevanten und irrelevanten Daten. Dazu werden die ML-Algorithmen zur Klassifikation programmiert und analysiert. Die Analysen beinhalten Ressourcenbedarf, Klassifikationsgeschwindigkeit und Genauigkeit der erzeugten Modelle. Der vielversprechendste Algorithmus wird anschließend auf Hardware implementiert und auf Genauigkeit, Klassifikationsgeschwindigkeit und Speicherbedarf geprüft. In dieser Arbeit wird ein relativ kleiner FPGA verwendet. Daher wird auch ein kleiner Datensatz verwendet. Die US-Behörde National Institute of Standards and Technology, stellt mit der Modified National Institute of Standards and Technology database (MNIST), einen solchen Datensatz zur Verfügung.

Die Einleitung bildet das Kapitel 1 und beschreibt die Problemstellung und Ziele der Arbeit. Die Kapitel 2 und 3 befassen sich mit den theoretischen Grundlagen der verwendeten Algorithmen und Technologie und stellen den aktuellen Forschungsstand dar. Eine Betrachtung der einzelnen Algorithmen sowie nötige Rechenoperation für die Vorhersage wird im Kapitel 4 aufgezeigt. Dabei wird die spätere Implementierung auf einem FPGA berücksichtigt. Im Kapitel 5 werden die vielversprechenden Algorithmen und dessen Implementierung beschrieben. Die Ergebnisse werden im Kapitel 6 beschrieben und aufbereitet. Das Kapitel 7 bildet den Abschluss der Arbeit, fasst die Ergebnisse zusammen, überprüft die Erreichung der Ziele und gibt einen Ausblick über zukünftigen Forschungsbedarf in diesem Gebiet.

2 Grundlagen der FPGA Technologie

2.1 Aufbau eines FPGAs

Hauptbestandteil eines FPGAs sind programmierbare Logikzellen, diese werden Configurable Logic Block (CLB) genannt. Mittels programmierbarer Verbindungspunkte (PSP) und Schaltmatrizen (PSM) werden die Basiszellen (CLBs) zusammengeschaltet [8]. Basiszellen werden von Xilinx als Programmable Logic Cells bezeichnet. Die Ein- und Ausgabe von Daten funktioniert über IO-Blöcke. Input/Output (IO) steht dabei für Ein- und Ausgabe. In Abbildung 2.1 lässt sich die regelmäßige Struktur erkennen.

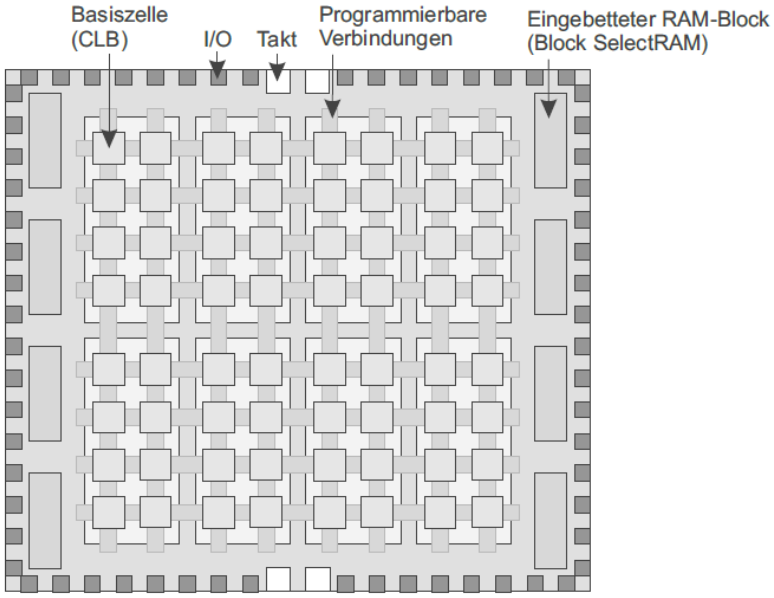


Abbildung 2.1: FPGA-Architektur

2.2 ZedBoard FPGA

In dieser Arbeit wird das FPGA Development Board namens ZedBoard verwendet. Auf diesem Board ist ein Zynq-7000 der Xilinx 7er-Serie verbaut. Der spezifische Name des Chips lautet XC7Z020-CLG484, mit rund 85.000 Logikzellen (entsprechend ca. 53.200 LUTs und 106.400 Flip-Flops) sowie dedizierten Ressourcen wie 220 DSP-Slices und 4,9 Mb BlockRAM [5]. Die Spezifikation des FPGAs ist in Tabelle 2.1 dargestellt. Der Name Programmable Logic Cell ist der Eigenname von Xilinx für die Basiszellen.

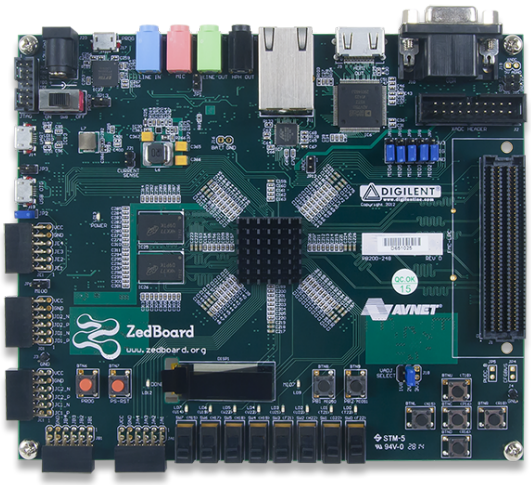


Abbildung 2.2: Zed Board

XC7Z020	
Programmable Logic Cells	85.000
LUTs	53.200
Flip-Flops	106.400
Block RAM	4,9 Megabit (Mb)
DSP Slices	220
PS I/O	128
SelectIO HR	200

Tabelle 2.1: FPGA Spezifikation

2.3 Zynq-7000 Processing System

Das Zynq-7000-SoC von Xilinx integriert zwei ARM-Cortex-A9-Prozessorkerne und frei programmierbare Logik, in Form eines FPGAs, auf einem einzigen Chip [24]. Die enge Kopplung von Prozessor und FPGA ermöglicht eine besonders schnelle Kommunikation über interne Hochgeschwindigkeitsbusse wie zum Beispiel Advanced eXtensible Interface (AXI). Dies reduziert nicht nur die Latenz, sondern senkt auch den Bedarf an externen Verbindungen. Da beide Komponenten in einer einzigen Architektur vereint sind, können komplexe Software-Aufgaben effizient mit maßgeschneiderten Hardwarebeschleunigern kombiniert werden, etwa für Bildverarbeitung oder Signalverarbeitung. Zusätzlich erlaubt die Integration eine geringere Leistungsaufnahme im Vergleich zu herkömmlichen Systemen aus separaten Prozessoren und FPGAs [5]. Damit eignet sich das Zynq-7000-System für vielseitige Anwendungen in der eingebetteten Systementwicklung.

3 Ausgewählte Grundlagen des Machine Learnings

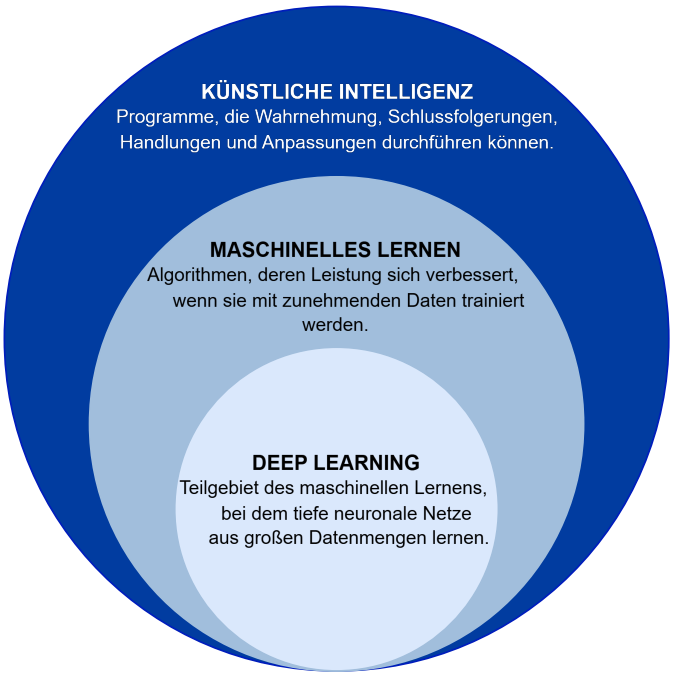


Abbildung 3.1: Einordnung von ML

Zu Beginn dieses Kapitels soll der Begriff des maschinellen Lernens definiert werden. Zhi-Hua Zhou beschreibt diesen Begriff folgendermaßen: „Machine learning is the technique that improves system performance by learning from experience via computational methods“ [27]. Anschließend erklärt er, dass die Erfahrung eines Systems in Form von Daten vorliegt. Diese Daten müssen dem Lernalgorithmus so zugeführt werden, dass ein Model Vorhersagen für Daten durchführen kann [27].

Der große Vorteil von Machine Learning (ML) ist, dass sich Algorithmen erzeugen lassen, ohne explizit die Aufgabe zu programmieren.

ML lässt sich in 3 verschiedene Arten unterteilen.

Supervised Learning (Überwachtes Lernen)
Bei dem überwachten Lernen ist ein Datensatz zum Training gegeben, welcher aus Merkmalen und der dazugehörigen Zielgröße besteht. So kann nach richtiger Anwendung anhand der Merkmale die Zielgröße bestimmt werden, auch wenn die Merkmale neu und nicht im Trainingsdatensatz enthalten waren.

Unsupervised Learning (Unüberwachtes Lernen)
Das unüberwachte Lernen ist für die Erkennung von Strukturen in einem Datensatz sinnvoll. Gegeben sind dann ausschließlich die Merkmale ohne Zielgröße. Der Algorithmus erkennt bestimmte Muster in den Merkmalen der verschiedenen Prototypen und ordnet diese in bestimmte Kategorien zusammen.

Reinforcement Learning (Verstärkendes Lernen)
Für selbständiges Erlernen von beispielsweise Spielstrategien wird verstärkendes Lernen verwendet. Durch eine Belohnungsfunktion wird das Model zu gewünschtem Verhalten geführt[17].

Für die Erkennung von Graustufen-Bildern ist eine multi-class Klassifizierung notwendig, da keine kontinuierliche Zielgröße vorliegt. Der MNIST-Datensatz umfasst 60.000 handgeschriebene Ziffern in 28 × 28 Pixel großen Graustufen-Bildern als Trainingsdaten und 10.000 Validierungsdaten der gleichen Art. Jede Ziffer bildet ein Label. Dieses Label ist die sogenannte Zielgröße y_n. Die Eingabegröße $\vec{x_n}$ sind im Falle des MNIST-Datensatzes, die 28 × 28 Pixel. Jeder der 784 Pixel besitzt P Merkmale. Da es sich um ein Graustufen-Bild handelt, ist $P = 1$. Bei einem farbigen RGB Datensatz wäre $P_{rgb} = 3$, da die Farben im RGB-Farbraum aus einer Mischung aus Rot, Grün und Blau besteht.

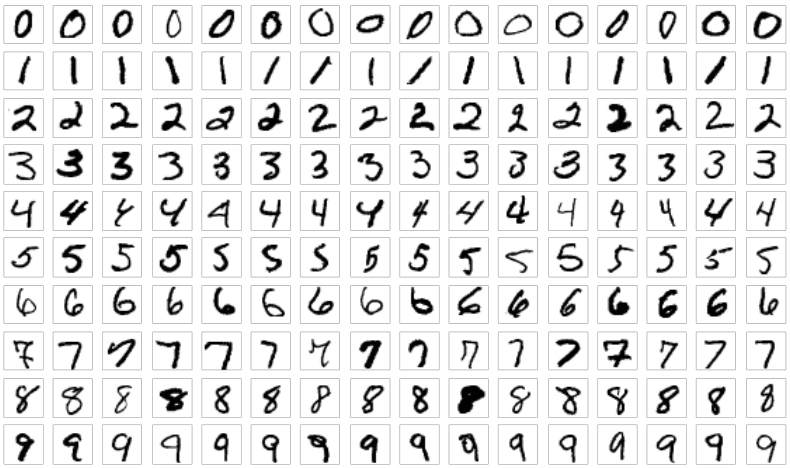


Abbildung 3.2: Beispiel des MNIST-Datensatzes [21]

3.1 Supervised learning

Bei dem überwachten Lernen ist das Ziel, einen Algorithmus zu entwickeln, welcher die Eingabewerte $\vec{x_n}$ den richtigen Zielgrößen y_n zuordnet. Diese Zuordnung entspricht der Problemstellung zum Filtern bestimmter Eingabewerte, die an die Datenverarbeitung geschickt werden sollen. Dies geschieht mithilfe der Trainingsdaten über die interaktive Optimierung der Parameter Θ. Die Abbildung 3.3 veranschaulicht, dass ein Lernalgorithmus auf Basis der Trainingsdaten ein Modell erzeugt, welches die Vorhersagen mithilfe der erlernten Parameter trifft. In einem neuronalen Netz sind die Parameter Θ die Gewichte w, welche Elemente der Matrix W sind. Eine Art der Optimierung ist der stochastische Gradientenabstieg.

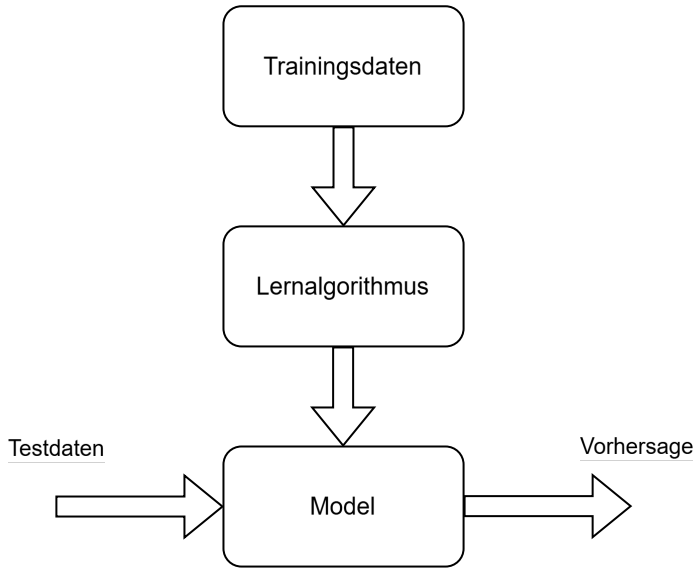


Abbildung 3.3: Schematische Darstellung von supervised learning

3.2 Neuronale Netze

In diesem kurzen Abschnitt soll es um die wahrscheinlich bekannteste Art von KI gehen, den sogenannten neuronalen Netzwerken. Da ein Neuronales Netzwerk (NN) in seiner Grundform auf Gleitkommaarithmetik beruht, welche auf FPGAs mit großem Ressourcenaufwand verbunden ist, ist dieser Abschnitt eine Vorbereitung auf das Kapitel 3.2.3.

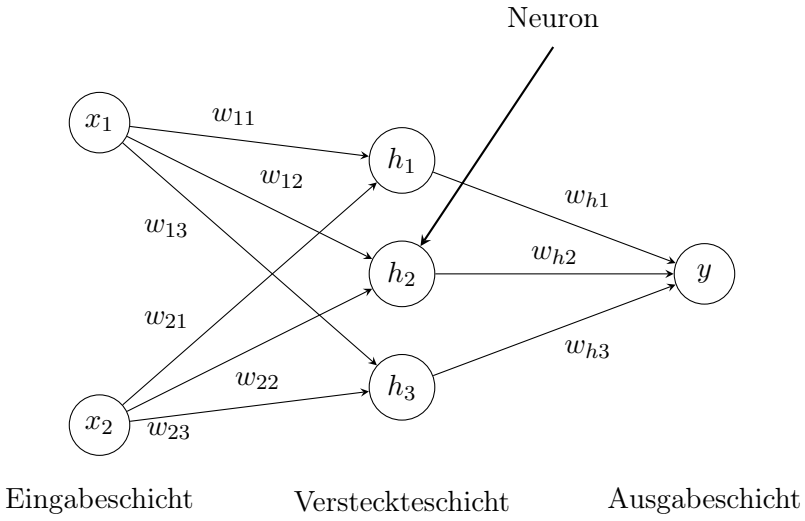


Abbildung 3.4: Vereinfachte Darstellung eines NN

Die Abbildung 3.4 zeigt ein NN mit 3 Schichten. Jede Schicht l besteht aus k_l Neuronen. Die Anzahl der Schichten L kann über die Anzahl der versteckten Schichten variiert werden. Jedes Neuron ist mit den Neuronen der vorherigen und nachfolgenden Schicht verbunden. Diese Verbindungen haben jeweils ein bestimmtes Gewicht w, welche durch einen zufälligen Wert initialisiert werden können. Über dieses Gewicht wird bestimmt, wie viel Einfluss die Eingabe auf das Neuron hat. Sinnbildlich ist das Neuron der Speicherort für die Ausgabe der darauffolgenden Schicht.

3.2.1 Vorwärtsdurchlauf

Für die Berechnung der Ausgabe eines Neurons wird, wie in Abbildung 3.5 dargestellt, die Summe aller Produkte, aus Eingabe und dazugehörigem Gewicht, gebildet. Anschließend

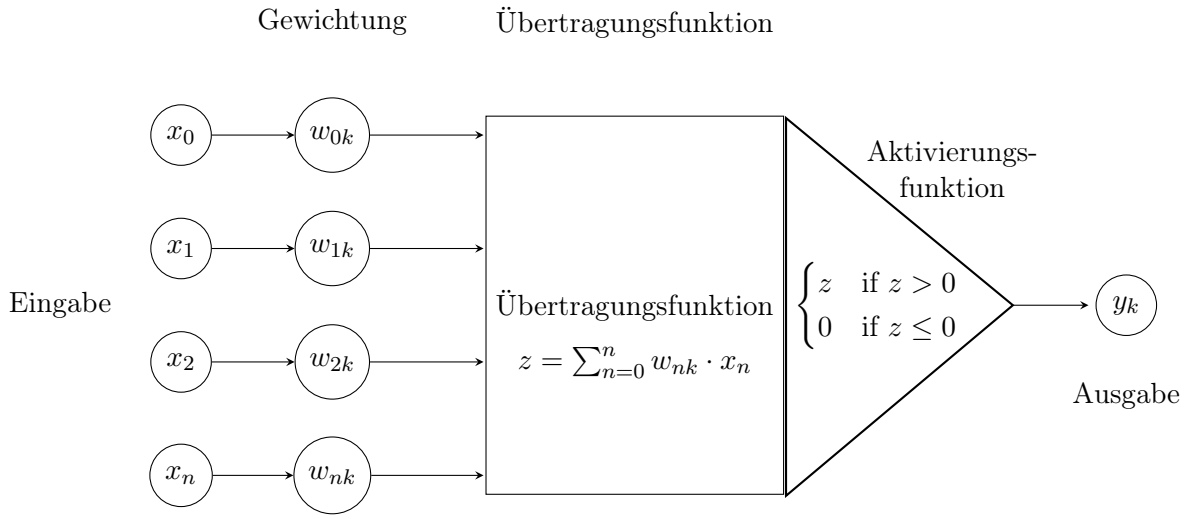


Abbildung 3.5: Beispiel eines Neurons

wird die Summe als Argument in die Aktivierungsfunktion gegeben. Die Aktivierungsfunktion soll durch ihre Nichtlinearität helfen, nicht lineare Prozesse zu modellieren. Als Beispiel wurde hier die Rectified Linear Unit (ReLU) Funktion verwendet. Neben weiteren Aktivierungsfunktionen, wie beispielsweise Sigmoid und Tanh, gibt es zu jedem Gewicht einen dazugehörigen Bias-Parameter. Der jeweilige Bias-Parameter eines Gewichts wird zum Produkt addiert und verschiebt somit die Aktivierungsfunktion entlang der x-Achse. Da ein Bias für die Betrachtung in Kapitel 3.2.3 unerheblich ist, wird er hier nicht weiter vertieft.

Die Vorhersage des gesamten neuronalen Netzwerks beginnt mit der Berechnung aller Neuronen in der ersten versteckten Schicht. Die Ausgabe jedes Neurons nach Anwendung der Aktivierungsfunktion dient als Eingabe für die nächste Schicht. Dieser Prozess setzt sich fort und propagiert die Berechnungen von der Eingabe- bis zur Ausgabeschicht, was als Vorwärtsdurchlauf bezeichnet wird. In der Ausgabeschicht werden häufig andere Aktivierungsfunktionen verwendet als in den versteckten Schichten. Beispielsweise transformiert die Softmax-Funktion die Ausgaben der Ausgabeneuronen in Werte zwischen 0 und 1, wobei die Summe aller Ausgaben 1 ergibt. Dies ermöglicht es, die Ausgaben als Wahrscheinlichkeitsverteilung zu interpretieren.

3.2.2 Gradientenabstieg

Ein NN mit all dessen Gewichten wird mit hoher Wahrscheinlichkeit, zu Beginn, nicht die richtige Vorhersage machen. Um die gewünschte Vorhersage durch das NN zu erhalten, müssen die Gewichte so angepasst werden, dass der Vorwärtsdurchlauf die richtige Ausgabe liefert. Eine Möglichkeit zur Anpassung der Gewichte ist der Gradientenabstieg. Eine analytische Lösung für die Optimierung der Parameter, Θ ist aufgrund der hohen Dimensionalität und der Nichtlinearität schwer zu finden [15].

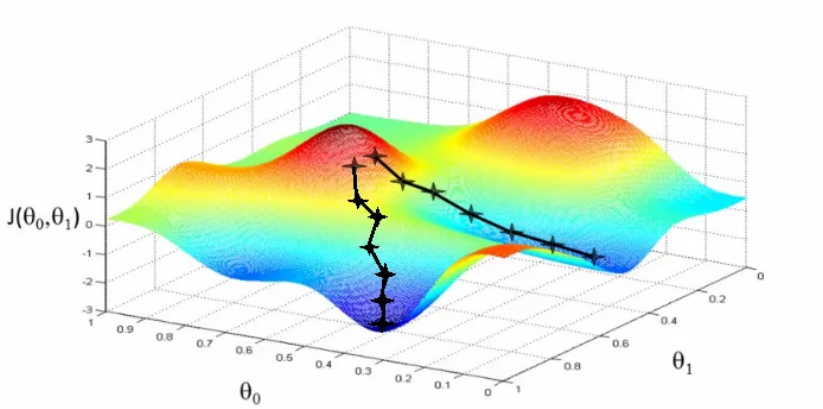


Abbildung 3.6: Platzhalter Beispiel des Gradientenabstiegs zur Optimierung der Verlustfunktion $J()$

Ziel des Gradientenabstiegs ist die Optimierung der Verlustfunktion $J(\Theta)$ durch Anpassung der Parameter Θ_j. Die Verlustfunktion $J(\Theta)$ aus Gleichung 3.2 quantifiziert den Fehler zwischen Vorhersage und tatsächlichem Label. Je weiter die Vorhersage $\hat{y}(\Theta)$ von dem tatsächlichen Label y entfernt ist, desto größer wird die Verlustfunktion $J(\Theta)$. Die Verlustfunktion kann beispielsweise wie folgt definiert sein:

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 \tag{3.1}$$

$$J(\Theta) = \frac{1}{2} (\hat{y}(\Theta) - y)^2 \tag{3.2}$$

Dabei sei m die Anzahl der Trainingsbeispiele, $\hat{y}^{(i)}$ die Vorhersage des Models für das i-te Beispiel und $y^{(i)}$ die richtige Vorhersage.

Der Gradientenabstiegsalgorithmus aktualisiert die Parameter Θ iterativ, in dem ein Schritt entgegen der höchsten Steigung der Verlustfunktion gegangen wird. Die aktualisierten Parameter berechnen sich wie in Gleichung 3.3 gezeigt[9]. Nach der Anpassung wird die Verlustfunktion mit den angepassten Parametern neu berechnet, um erneut eine Anpassung durchzuführen.

$$\Theta_j := \Theta_j - \alpha \frac{\partial J(\Theta)}{\partial \Theta_j} \tag{3.3}$$

$$\Theta_j := \Theta_j - \alpha \sum_{i=1}^{m} \left(h(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \tag{3.4}$$

Dabei sei Θ_j der j-te Parameter des Models, α die Lernrate, die die Anpassung skaliert, $\frac{\partial J(\Theta)}{\partial \Theta_j}$ die partielle Ableitung der Verlustfunktion nach dem Parameter Θ_j.

Solange das Model nur zwei Parameter besitzt, lässt sich dieses Optimierungsproblem dreidimensional veranschaulichen. Die Parameter Θ_1 und Θ_2 werden auf der x- und y-Achse repräsentiert. Die z-Achse repräsentiert den Wert der Verlustfunktion $J(\Theta)$. In Abbildung 3.6 lässt sich erkennen, wie der Algorithmus von einem zufälligen Startpunkt, iterativ in das Minimum herabsteigt. Jeder Schritt entgegengesetzt der größten Steigung führt in das Tal der Funktion $J()$ [18].

Im weiteren Verlauf der Arbeit wird der Batch-Gradientenabstieg verwendet. Diese Variante verwendet den gesamten Trainingsdatensatz zur Bestimmung des Gradienten. Bei größeren Datensätzen oder nicht ausreichender Rechenleistung lässt sich der Stochastischer Gradientenabstieg oder der Mini-Batch-Gradientenabstieg verwenden.
Stochastischer Gradientenabstieg: Die Anpassung der Parameter erfolgt nach jedem Trainingsbeispiel. Folglich wird weniger Rechenleistung benötigt und die Konvergenz in das Minimum von $J()$ wird unruhiger.
Mini-Batch-Gradientenabstieg: Eine Kombination aus Batch-Gradientenabstieg und dem stochastischen Gradientenabstieg. Hier wird der Trainingsdatensatz in Teilmengen aufgeteilt. Mit den einzelnen Teilmengen wird anschließend jeweils der Gradient berechnet.

3.2.3 Binarized Neural Networks

Eine Spezialform des NN ist das binarisierte neuronale Netz (BNN). Bei einem BNN sind die Gewichte und die Aktivierung auf binäre Werte beschränkt. Typischerweise wird hier -1 und $+1$ gewählt [3].

Grundsätzlich funktioniert die Vorhersage durch ein BNN wie die Vorhersage eines NNs. Der Eingabewert und das Gewicht, welche miteinander multipliziert werden, haben jeweils den Wert -1 oder 1. Werte dazwischen sind nicht möglich.

Durch diese Binarisierung lässt sich eine Verringerung des Speicherbedarfs und eine Erhöhung der Rechengeschwindigkeit feststellen. Die Multiplikation zweier 32Bit Gleitkommazahlen benötigt eine große Anzahl an CLBs[1] wohingegen das BNN nur ein CLB pro Multiplikation benötigt [2]. In NN müssen die Gewichte, aufgrund ihres Speicherbedarfs, häufig in externem RAM gespeichert werden. Somit wird die Klassifizierungsgeschwindigkeit durch die Lese- und Schreibgeschwindigkeit des externen RAMs limitiert[10].

Die Verbesserung des Ressourcenbedarfs geht jedoch mit einer verringerten Genauigkeit bei komplexen Datensetzen wie ImageNet einher [19]. ImageNet ist eine Bilderdatenbank aus mehr als 14 Millionen Bildern, welche ca. 20.000 Kategorien zugeordnet sind. Ein spezielles BNN, das ABC-Net [11] erreicht eine Genauigkeitsdifferenz von ungefähr 5% zu den Convolutional Neural Network (CNN) mit Gleitkommaarithmetik. Beim ABC-Net werden die Gewichte durch mehrere Teilgewichte und einen Skalierungsfaktor repräsentiert. Somit erhält man mit mehr Rechenaufwand eine feinere Abstufung.

Die Aktivierung ist das Ergebnis der Aktivierungsfunktion A^l welches mit den Gewichten der nächsten Schicht $W_b^{(l+1)}$ multipliziert wird. Dabei sei l der Index der Schicht.

Die Gewichte w können dabei durch die Sign-Funktion 3.5 binarisiert werden.

$$w_b = \sigma(w) = \text{sign}(w) = \begin{cases} +1, & \text{wenn } w \geq 0 \\ -1, & \text{wenn } w < 0 \end{cases} \tag{3.5}$$

Bei herkömmlichen NN wird der stochastische Gradientenabstieg zum Training genutzt. Der stochastische Gradientenabstieg bedarf jedoch einer differenzierbaren Verlustfunktion. Durch die binären Gewichte wird die Ableitung der Aktivierungsfunktion gleich null, da die Gewichte $w \in \{-1, 1\}$ sind.

[1] siehe Abschnitt 2.1

$$f(z) = \text{hardtanh}(z) = \begin{cases} -1 & : z < -1 \\ z & : -1 \leq z \leq 1 \\ 1 & : z > 1 \end{cases} \tag{3.6}$$

$$f'(z) = \text{hardtanh}'(z) = \begin{cases} 1 & : -1 \leq z \leq 1 \\ 0 & : \text{sonst} \end{cases} \tag{3.7}$$

Die Vorwärtsberechnung wird mit der Gleichung 3.8 durchgeführt:

$$A^{(l)} = \text{sign}\left(Z^{(l-1)} \cdot (W_b^{(l)})^T\right) \tag{3.8}$$

Dabei ist zu beachten, dass die Aktivierungsfunktion die Sign-Funktion sein muss. Würde an dieser Stelle hardtanh() benutzt werden, wäre die anschließende Berechnung nicht mehr binär. Anstelle des stochastischen Gradientenabstiegs wird der Straight-Through Estimator (STE) genutzt.

3.2.4 Straight-Through Estimator

Die Straight-Through Estimator (STE) Methode ermöglicht das Trainieren von Gewichten eines neuronalen Netzes, trotz einer Aktivierungsfunktion, deren Gradient fast ausschließlich null ist [26].

Der Sonderfall, in welchem der Gradient der Aktivierungsfunktion fast ausschließlich null ist, tritt beispielsweise dann auf, wenn die Aktivierungsfunktion in die Sättigung getrieben wird. Im Falle des BNNs mit der Aktivierungsfunktion $\text{sign}(x)$ aus Gleichung 3.5 ist die Ableitung für $x \neq 0$ $\text{sign}'(x) = 0$ und bei $x = 0$ nicht differenzierbar.

Die Korrektur um einen kleinen Wert, wie im Falle des Gradientenabstiegs, ist mit Gewichten $w \in \{-1, 1\}$ ebenfalls nicht möglich. Um trotzdem kleine Änderung zu machen, werden die Gewichte fürs Training im Gleitkommaformat w behalten und aktualisiert. Für die Vorwärtsberechnung werden jedoch die binarisierten Gewichte $w_b = \text{sign}(w)$ genutzt. Dabei werden die Gewichte w_b in jeder Epoche neu berechnet, um den neuen Verlust zu berechnen.

Die Verlustfunktion $J(\Theta)$ aus Gleichung 3.2 bleibt unverändert. Im Folgenden gilt, $W^{(l)} \in \mathbb{R}^{k_{l-1} \times k_l}$ wie in Gleichung 3.9, $Z^{(l)} \in \mathbb{R}^{m \times k_l}$ aus Gleichung 3.11 und $A^{(l)} \in \mathbb{R}^{m \times k_{l-1}}$ aus Gleichung 3.10. Der Index k_0 steht für die Größe der Eingabeschicht, welche $A^{(1)}$ entspricht. Des Weiteren werden nun die allgemeinen Parameter Θ durch die konkreten Parameter $W^{(l)}$ ersetzt.

$$W^{(l)} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1k_l} \\ w_{21} & w_{22} & \cdots & w_{2k_l} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k_{l-1}1} & w_{k_{l-1}2} & \cdots & w_{k_{l-1}k_l} \end{bmatrix} \tag{3.9}$$

$$A^{(l+1)} = \sigma(Z^{(l)}) \tag{3.10}$$

$$Z^{(l)} = A^{(l)} \cdot W^{(l)^T} \tag{3.11}$$

Der Gradientenabstieg 3.3, lasst sich durch Umformung, wie in Gleichung 3.12 vereinfachen.

$$\frac{\partial J(W)}{\partial W^{(l)}} = \frac{\partial J(W)}{\partial A^{(l+1)}} \frac{\partial A^{(l+1)}}{\partial Z^{(l)}} \frac{\partial Z^{(l)}}{\partial W^{(l)}} \tag{3.12}$$

$$\frac{\partial Z^{(l)}}{\partial W^{(l)}} = A^{(l)} \tag{3.13}$$

$$\frac{\partial A^{(l+1)}}{\partial Z^{(l)}} = \frac{\partial \sigma(Z^{(l)})}{\partial Z^{(l)}} = \sigma'(Z^{(l)}) \tag{3.14}$$

Zur Vereinfachung wird das Fehlersignal $\delta^{(l)}$ definiert:

$$\delta^{(l)} = \frac{\partial J(W)}{\partial A^{(l+1)}} \frac{\partial A^{(l+1)}}{\partial Z^{(l)}} \tag{3.15}$$

Über die Definition 3.15 ergibt sich, dass der Gradient jeder Schicht durch Gleichung 3.16 berechnet werden kann. Jedes δ wird, nach Gleichung 3.17, durch das δ der folgenden Schicht berechnet. Daher hat der Rückwärtsdurchlauf seinen Namen. Es wird also das δ der Ausgabeschicht bestimmt und anschließend jedes δ von Ausgabeschicht bis zur ersten verstecken Schicht berechnet.

$$\frac{\partial J(W)}{\partial W^{(l)}} = \delta^{(l)} \cdot A^{(l)} \tag{3.16}$$

$$\delta^{(l)} = \frac{\partial J(W)}{\partial A^{(l+1)}} \frac{\partial A^{(l+1)}}{\partial Z^{(l)}} = \frac{\partial J(W)}{\partial A^{(l+2)}} \frac{\partial A^{(l+2)}}{\partial Z^{(l+1)}} \frac{\partial Z^{(l+1)}}{\partial A^{(l+1)}} \frac{\partial A^{(l+1)}}{\partial Z^{(l)}} \tag{3.17}$$

$$\delta^{(l)} = (\delta^{(l+1)} \cdot {W^{(l+1)}}^T) \odot \sigma'(Z^{(l)}) \tag{3.18}$$

Wenn nun der Gradient der Ausgabeneuronen berechnet werden soll, also $A^{(l)} = \hat{y}$ gilt, so ergibt sich:

$$\frac{\partial J(W)}{\partial A^{(l)}} = \frac{\partial \frac{1}{2m} \sum_{i=1}^{m} (y - A^{(l)})^2}{\partial A^{(l)}} = \frac{1}{m}(A^{(l)} - y) \tag{3.19}$$

Eingesetzt in $\delta^{(l)}$:

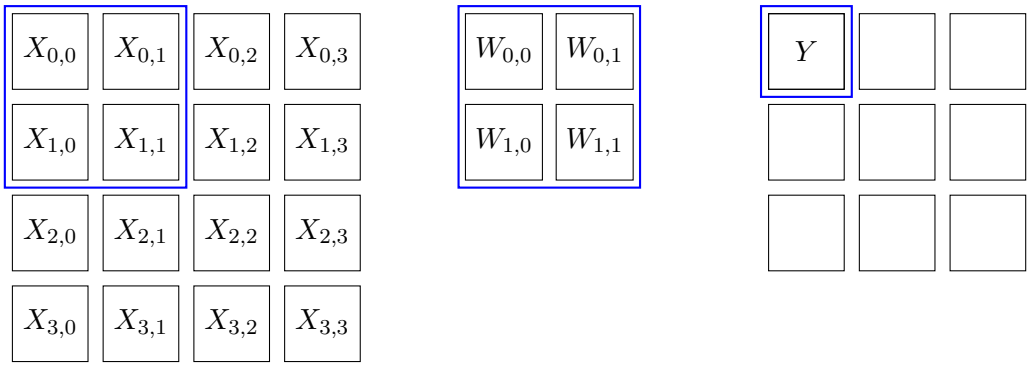
$$\delta^{(l)} = \frac{\partial J(W)}{\partial A^{(l+1)}} \frac{\partial A^{(l+1)}}{\partial Z^{(l)}} = \frac{1}{m}(A^{(L)} - y) \cdot \sigma'(Z^{(l)}) \tag{3.20}$$

Die STE besteht nun darin, die Ableitung der Aktivierungsfunktion σ' durch eine Näherung zu ersetzen. Die Ableitung der Aktivierungsfunktion sign() wird beispielsweise durch die Ableitung der hardtanh() Funktion angenähert, da hardtanh$'$() differenzierbar ist.

Zur Wahl der Ableitung eignet sich besonders gut die Ableitung der clippedReLU-Funktion, da sie zu einem stabileren Trainingsverlauf führt [26]. Das Verhalten wird jedoch, aufgrund der Vorzeichensprünge, immer instabiler sein als bei Trainingsverfahren mit Gleitkommazahlen.

3.2.5 Faltendens neuronales Netz

Eines der Verschiedenen neuronalen Netze ist das faltende neuronale Netz. Dieses wird auch Convolutional Neural Network (CNN) genannt. Ein CNN verwendet sogenannte Convolutional und Pooling Layers. Ein Convolutional Layer ist eine speziell trainierte Schicht zur Erkennung von Farbkombinationen oder Kanten.



$$Y = X_{0,0} \cdot W_{0,0} + X_{0,1} \cdot W_{0,1} + X_{1,0} \cdot W_{1,0} + X_{1,1} \cdot W_{1,1}$$

Abbildung 3.7: Faltungsbeispiel mit 2x2 Filter

Die Abbildung 3.7 zeigt, wie ein Teil der Eingabematrix X mittels des Skalarprodukts mit der Filtermatrix W gefaltet wird.

3.3 k-Nächste-Nachbarn-Algorithmus

Der k-nächste Nachbarn Algorithmus (kNN) ist eine Erweiterung des Nächste-Nachbarn Algorithmus. Das kNN eignet sich sowohl für Regression als auch für Klassifikation. In der Abbildung 3.8 ist zu erkennen, dass der Wert k die Anzahl an Nachbarn vorgibt. Der kNN-Algorithmus vergleicht den Eingabevektor mit den nächsten k umliegenden Prototypen.

Dieser Algorithmus bedarf keiner Anpassung der Parameter, sondern einer genauen Analyse der Hyperparameter k und der Anzahl an Prototypen. Der Algorithmus berechnet den Abstand von allen Prototypen zum Eingabevektor und vergleicht anschließend die Abstände. Die Mehrheit der k dichtesten Prototypen, mit dem gleichen Label, bestimmt das vorhergesagte Label [6].

Die Berechnung des Abstands kann beispielsweise über den Euklidischen Abstand oder die Manhattan-Distanz geschehen.

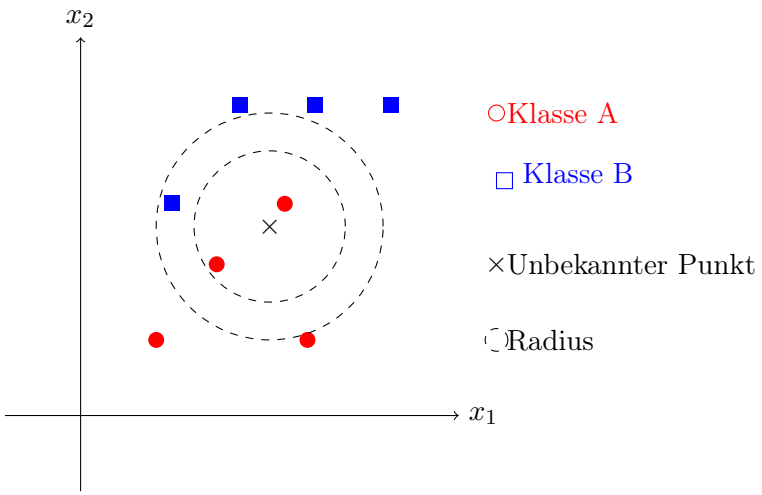


Abbildung 3.8: Visualisierung des k-NN-Algorithmus mit Radius bei k=2 und k=3

Der Euklidische Abstand im n-dimensionalen ist definiert über 3.21. Dabei ist anzumerken, dass die Wurzel einer Zahl nur schwer auf einem FPGA berechnet werden kann. Um keine Wurzelberechnung durchführen zu müssen, ließe sich auch die Manhattan-Distanz 3.22 verwenden.

Zum Verständnis lässt sich, wie in Abbildung3.9 für jede Kombination an X_1 und X_2 in einem definierten Bereich, das Klassifikationsergebnis bestimmen und einfärben. Dies verdeutlicht die Entscheidungsschwellen des kNN.

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{3.21}$$

$$d_M(x,y) = \sum_{i=1}^{n}|x_i - y_i| \tag{3.22}$$

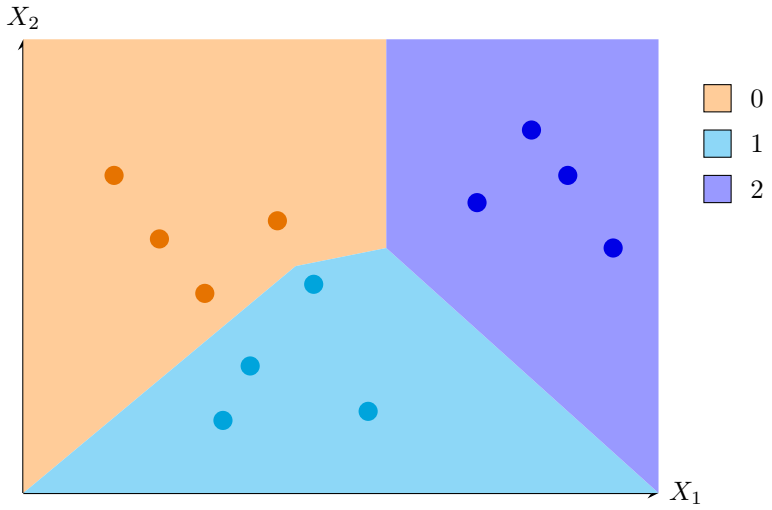


Abbildung 3.9: Beispiel von Entscheidungsschwellen bei 3 Klassen

3.4 Entscheidungsbäume

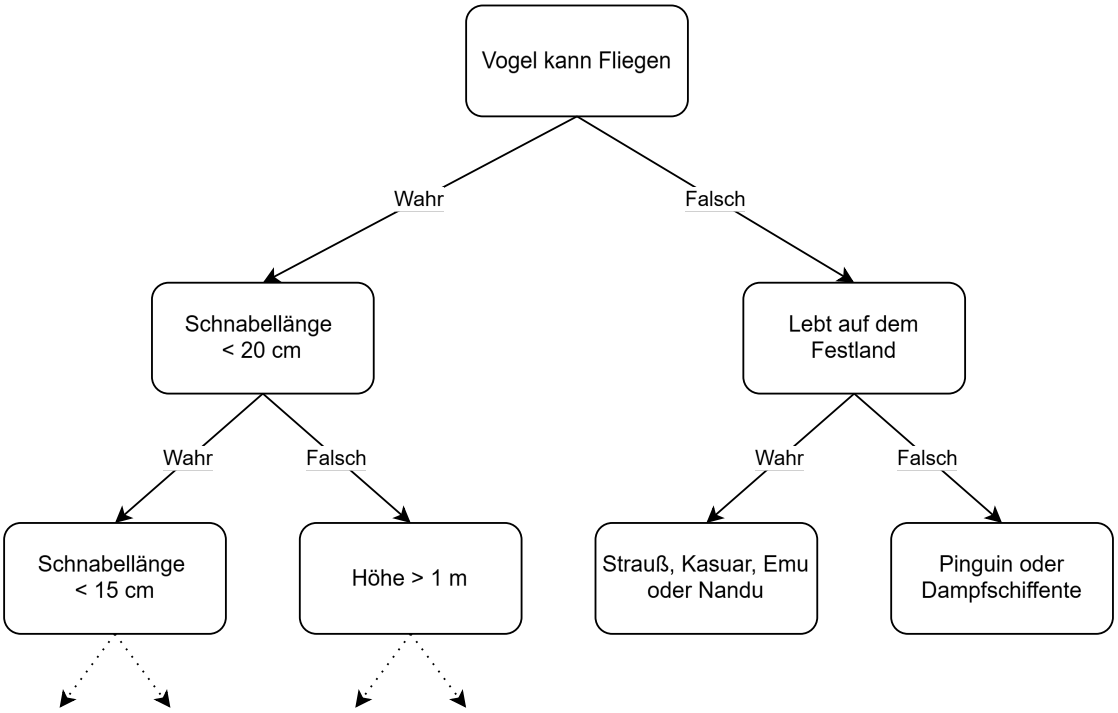


Abbildung 3.10: Beispiel eines Entscheidungsbaumes anhand der Klassifikation von Vögeln

Ein Entscheidungsbaum oder im Englischen decision Tree ist eine hierarchische Klassifizierungsmethode wie in Abbildung 3.10. Zur Veranschaulichung kann man sich ein Entscheidungsbaum wie eine umgedrehte Baumkrone vorstellen. Der Entscheidungsbaum besteht aus Knoten und Kanten. Das oberste Element nennt man Wurzel. Alle gerichteten Kanten zeigen von diesem Element weg. Ein Element, von dem Pfeile weg zeigen sowie hinzeigen, nennt man Zweige. Zeigt kein Pfeil von dem Element weg, handelt es sich um ein Blatt. In den Zweigen und der Wurzel stehen Aussagen, die entweder wahr oder falsch sind. Je nach Antwort überprüft man die Aussage des nächsten Zweigs. Die Blätter entsprechen einem Ergebnis der Vorhersage.

Das Ziel eines Entscheidungsbaumes ist die rekursive Aufteilung des Datensatzes, um die Homogenität der Untergruppen zu maximieren [6]. Ob die Abfrage eines Merkmals zu dem größtmöglichen Informationsgewinn führt, kann durch die Gini-Unreinheit oder des Informationsgewinns bestimmt werden.

3.4.1 Entropie und Informationsgewinn

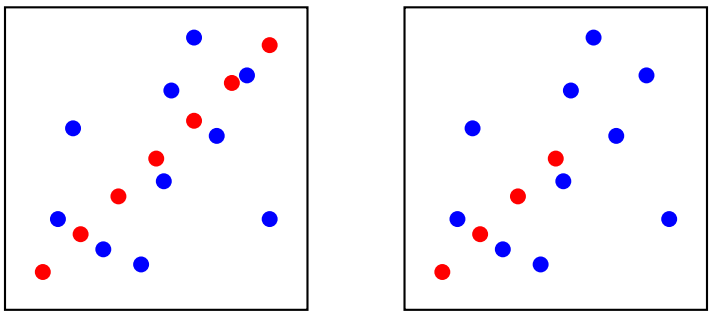
Die Entropie ist eine Zustandsgröße, die eine qualitative Aussage über die Unordnung in einem System ermöglicht. Der Informationsgewinn ist eine Größe, um die Reduktion der Entropie zu vergleichen. Die Formel 3.23 definiert die Berechnung der Entropie für den Datensatz S. Dabei sei n die Anzahl der Klassen und p_i die relative Häufigkeit der Klasse.

Für die Berechnung des Informationsgewinns durch die Aufteilung nach dem Merkmal A wird die Differenz aus der Entropie vor und nach der Aufteilung gebildet. Die Entropie nach der Aufteilung wird für jeden Wert v, aus dem Merkmal A, entsprechend dem Anteil der Teilmenge S_v zu S gewichtet. $|S_v|$ sei somit die Anzahl an Instanzen in der Teilmenge S_v.

$$\text{Entropie}(S) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{3.23}$$

$$\text{Informationsgewinn}(S, A) = \text{Entropie}(S) - \sum_{v \in \text{Werte}(A)} \frac{|S_v|}{|S|} \text{Entropie}(S_v) \tag{3.24}$$

Die Aufteilung erfolgt nun so, dass jede Aufteilung den maximalen Informationsgewinn bringt. Die Aufteilung erfolgt so lange, bis eine Abbruchbedingung erreicht wird. Eine häufige Abbruchbedingung ist das Erreichen einer maximalen Tiefe des Baumes. Als Tiefe beschreibt man den längsten Pfad von Wurzel bis Blatt [6].



Hohe Entropie | Geringe Entropie

Abbildung 3.11: Beispiel für Entropie

4 Anforderungen und Lösungsansätze der Problemstellung

In diesem Kapitell sollen die Möglichkeiten und Herausforderungen der Implementierung von Machine Learning Algorithmen auf einem FPGA untersucht und bewertet werden. Ziel des Kapitells ist es, die Leistungsfähigkeit eines ML der FPGA-basierten Implementierung zu analysieren und zu bewerten, um dadurch ein besseres Verständnis für die Eignung dieser Technologie zu gewinnen. Durch den Vergleich verschiedener Algorithmen sollen Erkenntnisse darüber gewonnen werden, welche Ansätze sich besonders gut für die hardwarebasierte Signalverarbeitung auf einem FPGA eignen und welche Herausforderungen bei der Implementierung auftreten können.

4.1 k-nächste Nachbarn Algorithmus

4.1.1 Modellerzeugung

Für die Abschätzung wurde eine k-Nächste-Nachbarn-Klassifikation programmiert.

Dabei wird der Datensatz vorher in eine Matrix mit d Spalten für die Anzahl an Merkmalen und n Spalten für die Anzahl an Trainingsdaten gebracht. Die Labels werden dementsprechend in einem Vektor der Größe n gespeichert. Durch die unterschiedliche Anzahl zwischen Trainings- und Validierungsdaten ergibt sich noch der Index m welcher der Anzahl an Validierungsdaten entspricht.

Algorithm 1 k-Nächste-Nachbarn-Klassifikation

Require: Trainingsbilder $X_{\text{train}} \in \mathbb{R}^{d \times n}$, Trainingslabel $y_{\text{train}} \in \mathbb{R}^{n}$, Testbilder $X_{\text{test}} \in \mathbb{R}^{d \times m}$, Anzahl der Nachbarn K, Distanzmetrik metrik
Ensure: Klassifikationen für Testdaten y_{pred}

```
i ← 1
while i ≤ m do
    distanzen ← ∅
    for j = 1 to n do
        if metrik = euclidean then
            distanzen(j) ← sqrt(sum((X_train(:, j) − X_test(:, i))^2))
        else if metrik = manhattan then
            distanzen(j) ← sum |X_train(:, j) − X_test(:, i)|
        end if
    end for
    sortierte_indices ← sort(distanzen)
    naechste_label ← y_train(sortierte_indices(1 : K))
    y_pred(i) ← mode(naechste_label)
    i ← i + 1
end while
```

Die Funktionen sort() und mode() sind Matlabfunktionen, wobei sort() die sortierten Distanzen sowie dessen Indices zurückgibt. Hier werden nur die Indices benötigt. Diese entsprechen den Indices der Labels von y_{train}. Mithilfe von mode() wird der häufigste Wert und den k Labeln gefunden.

Um eine effiziente Nutzung des FPGA RAMs zu gewährleisten, wird die Genauigkeit in Abhängigkeit der Prototypen Anzahl analysiert. Dazu wird, wie in Abbildung 4.1 gezeigt, die Genauigkeit für jede Anzahl an Portotypen berechnet.

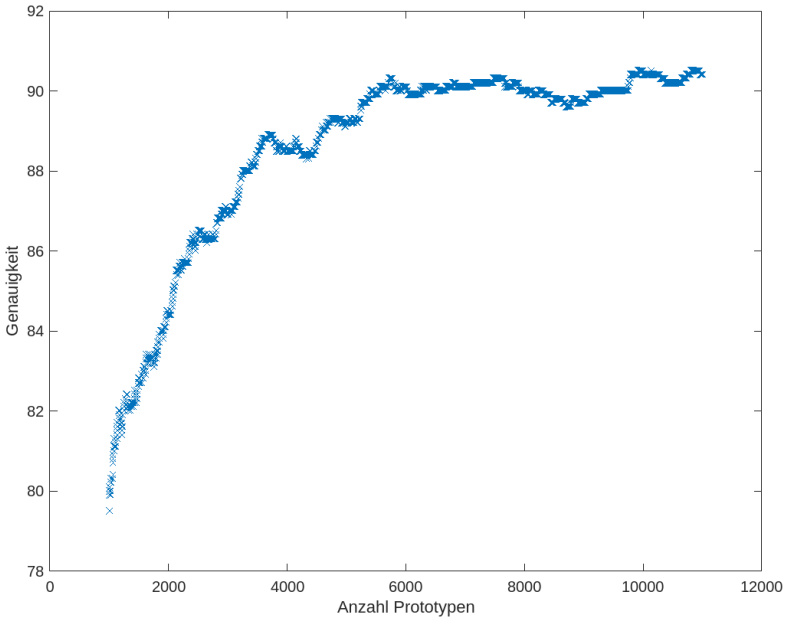


Abbildung 4.1: kNN Genauigkeit über Anzahl Prototypen

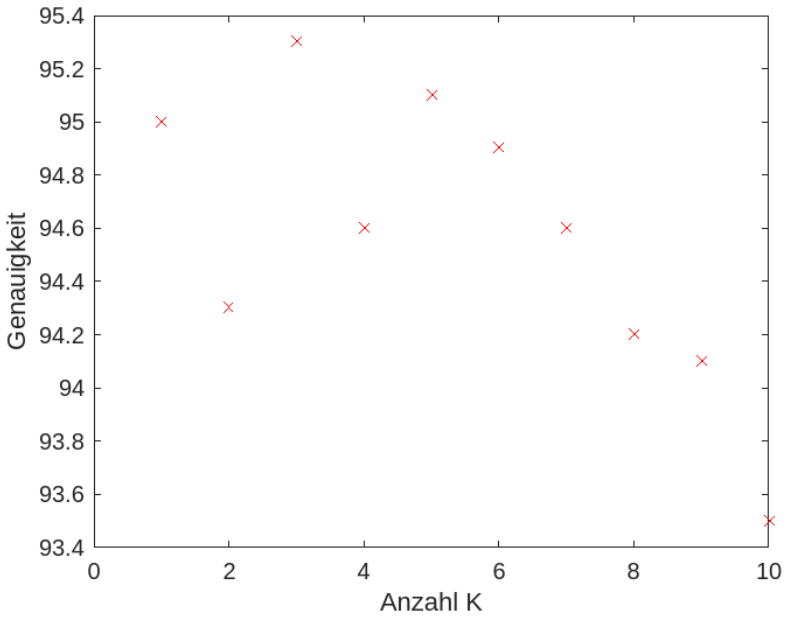


Abbildung 4.2: Genauigkeit über K, mit Manhattan Abstand, 1.000 Validierungsdaten

4.1.2 Genauigkeit

Die Untersuchung des kNN Algorithmus ergab, dass mit dem euklidischen Abstand zu 60.000 Prototypen eine Genauigkeit von 96,88 % erzielt wurde. Die Berechnung der Wurzel und des Quadrats für die euklidische Abstandsberechnung 3.21 stellt eine Herausforderung für die FPGA Implementierung dar. Um diese Herausforderung zu umgehen, wurde ebenfalls die Manhattan-Distanz 3.22 zur Abstandberechnung genutzt und eine Genauigkeit von 96,18 % erreicht.

Durch die Analyse der Genauigkeit über k in Abbildung 4.2 konnte der ideale Wert für k in diesem Datensatz ermittelt werden. Mit der Konfiguration $k = 3$ lässt sich eine Genauigkeit von 96,33 % erzielen.

Mit der Abbildung 4.1 lässt sich erkennen, dass ab 6.000 Vergleichsbilder die Vorhersagegenauigkeit nur gering ansteigt.

Zur Speicheroptimierung wurden die Prototypen und die Validierungsdaten in binäre Werte umgewandelt. So ergab sich mit der Manhattan-Metrik eine Genauigkeit von 95,71 %.

K	Abstandsfunktion	#Prototypen	#Validierungsdaten	Genauigkeit
5	euklidisch	60.000	10.000	96,88 %
5	Manhattan	60.000	10.000	96,18 %
3	Manhattan	60.000	1.0000	95,3 %
3	Manhattan	60.000	10.000	96,33 %
3	Manhattan	60.000 (1 Bit)	10.000 (1 Bit)	95,71 %
3	Manhattan	6.000 (1 Bit)	10.000 (1 Bit)	91,30 %

Tabelle 4.1: Ergebnistabelle des kNN

4.1.3 Abschätzung der Anwendbarkeit

Durch die Datenmengen, welche für die Distanzberechnung nötig sind, sollte dieses Model mithilfe von RAM oder ROM realisiert werden. Bei dem für diesen verwendeten FPGA ergibt sich jedoch eine Latenz von mindestens einem Taktzyklus[5]. Infolgedessen wäre für den Vergleich mit 6.000 Bildern aus dem RAM auch 6.000 Takte notwendig. Zwar ließe sich die Anzahl der nötigen Takte über Verwendung von mehreren RAMs verringern, aber unter Berücksichtigung der Skalierbarkeit des kNNs auf größere Datensätze, wird dieses Model als ungeeignet befunden.

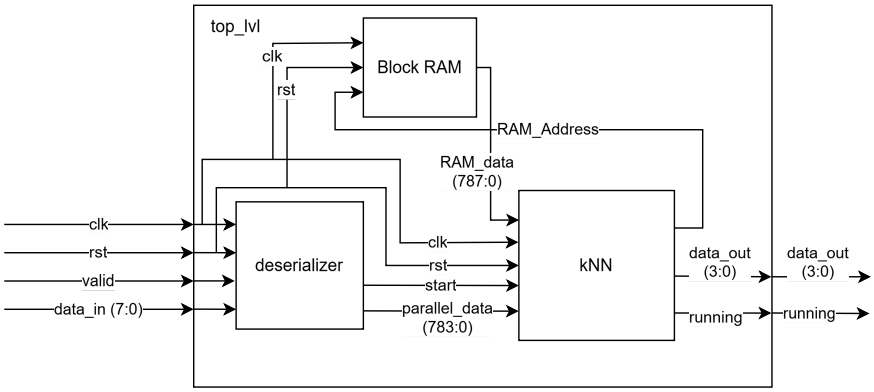


Abbildung 4.3: Blockdiagramm des kNNs

4.2 Binarisiertes neuronales Netzwerk

4.2.1 Modellerzeugung

Im Zuge der Arbeit wurde ein BNN programmiert. Das BNN besteht aus L Schichten. Jede Schicht l hat k_l Neuronen. Die Gewichte werden in einem Zell-Array gespeichert. Jede Zelle enthält eine Matrix $W_{\text{real}}\{l\} \in \mathbb{R}^{k_{l-1} \times k_l}$. Jede W wird mit zufälligen Zahlen initialisiert. Die erste Schicht besteht aus 784 Eingabeneuronen, entsprechend der Merkmale des MNIST-Datensatzes, während die Ausgabeschicht 10 Neuronen für die One-Hot-Codierung der Klassenlabels enthält.

$$w\{l\} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1k_l} \\ w_{21} & w_{22} & \cdots & w_{2k_l} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k_{l-1}1} & w_{k_{l-1}2} & \cdots & w_{k_{l-1}k_l} \end{bmatrix} \tag{4.1}$$

N sei die Anzahl der Prototypen und $a\{1\}$ eine $\mathbb{R}^{N \times 784}$ Matrix mit den Protypen.

Algorithm 2 Vorwärtsberechnung des BNNs

Require: $N \geq 1$
Ensure: $l = 1$
  while $l \leq L$ **do**
    $w_{\text{bin}} \leftarrow \text{sign}(w_{\text{real}}\{l\})$ ▷ Binarisierung der Gewichte
    $z\{l\} \leftarrow a\{l\} \cdot w_{\text{bin}}^{\top}$
    if $l < L$ **then**
      $a\{l+1\} \leftarrow \text{sign}(z\{l\})$ ▷ Anwendung der Aktivierungsfunktion
    else
      $a\{l+1\} \leftarrow \text{softmax}(z\{l\})$ ▷ Ausgabeneuronen als Wahrscheinlichkeit
    end if
    $l \leftarrow l + 1$
  end while

$$\text{softmax}(X) = \frac{\exp(X)}{\sum_{i=1}^{k_l} \exp(X_i)} \tag{4.2}$$

Da die Softmax-Funktion durch die Exponentialrechnung für die FPGA Implementierung ungeeignet ist, wird sie durch die Sign-Funktion ersetzt. Es konnte kein Genauigkeitsverlust festgestellt werden.

Für die Rückwärtsberechnung in Algorithmus 3 wird μ' aus Gleichung 4.3 verwendet, dabei sei η die Lernrate. Die Funktion clipped ReLU$'$ eignet sich besonders gut für die STE, da sie zu mehr Stabilität führt [26].

$$\mu' = \text{clipped ReLU}'(x) = \begin{cases} 1 & : 1 > x > 0 \\ 0 & : \text{sonst} \end{cases} \tag{4.3}$$

Algorithm 3 Rückwärtsberechnung des BNNs

Require: $y\_pred = a\{\text{end}\}$
Ensure: $l = L - 1$
  $\delta\{L\} = (y\_pred - target\_data)/N$
  while $l \geq 1$ **do**
    $\delta\{l\} \leftarrow (\delta\{l+1\} \cdot w_{\text{real}}\{l+1\}) \odot \mu'(a\{l\})$
    $l \leftarrow l - 1$
  end while
  while $l \leq L$ **do**
    $grad\_w \leftarrow \delta\{l\}^\top \cdot a\{l\}$
    $w_{\text{real}}\{l\} - \eta \cdot grad\_w$
    $l \leftarrow l + 1$
  end while

4.2.2 Genauigkeit des Modells

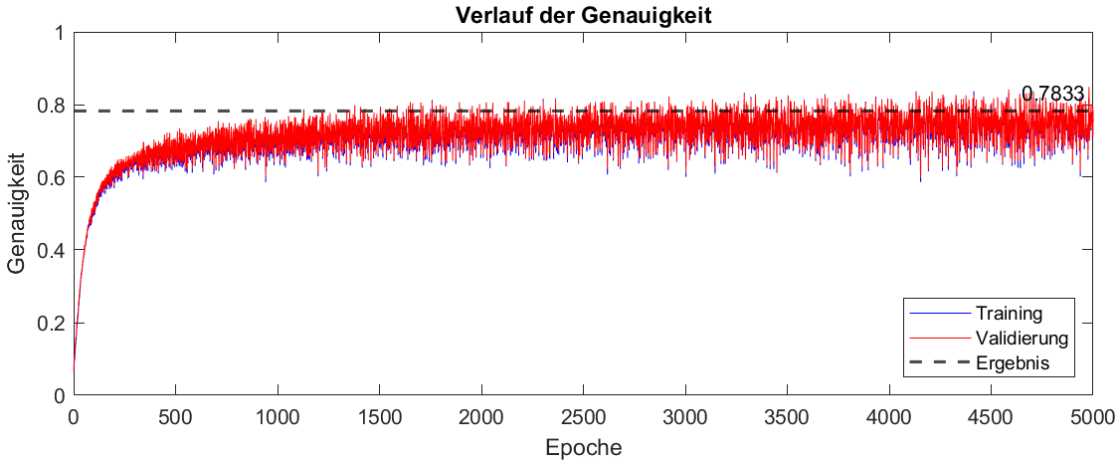


Abbildung 4.4: BNN, mit einer versteckten Schicht aus 2^{11} Neuronen $\eta = 0,0007$

Mittels des beschriebenen Codes konnte ein BNN mit einer Genauigkeit von 78,33 % erzeugt werden. Das BNN wurde mit einer versteckten Schicht aus 2048 Neuronen erstellt. Die in Abbildung 4.4 erkennbare Instabilität resultiert aus der großen Variation der binären Gewichte. Trotz der geringen Lernrate und der geringen Änderungen der realen Gewichte sind die Änderungen der binären Gewichte immer gleichbedeutend mit einem Vorzeichenwechsel.

Um eine Einschätzung der optimalen Anzahl an Schichten zu ermitteln, wurden 2 BNNs mit zufälligen Gewichten und unterschiedlicher Anzahl an Schichten initialisiert. Der Trainingsverlauf ist in 4.5 zusehen. BNN1 besteht aus einer versteckten Schicht mit 1024 Neuronen. Das BNN2 besteht aus zwei versteckten Schichten mit jeweils 512 Neuronen. Das Ziel ist durch die gleichbleibende Summe an Neuronen einen Vergleich der Genauigkeit bei ähnlichem Speicherbedarf zu erzielen.

Abbildung 4.5 zeigt das Ergebnis der Untersuchung. Es wird ersichtlich, dass die Verteilung der Neuronen auf eine Schicht, bei gleichbleibender Anzahl Neuronen vorteilhafter ist. Hinsichtlich der Anwendung auf einem FPGA entfallen Synchronisationsschritte zwischen zwei Schichten. Dies bringt einen weiteren Vorteil der Struktur mit einer versteckten Schicht.

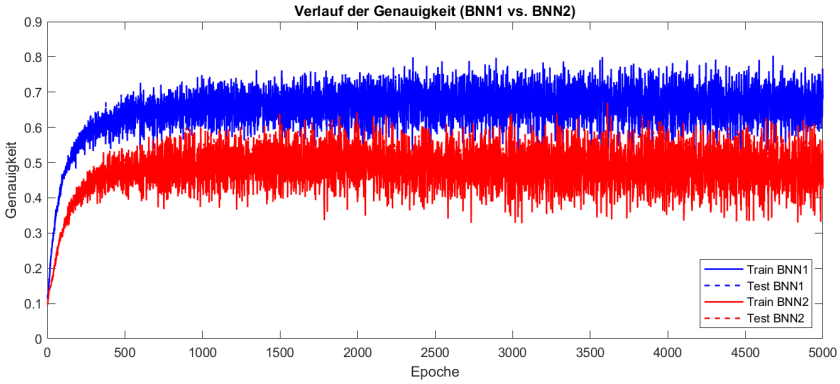


Abbildung 4.5: Vergleich der Genauigkeit zweier BNNs mit einem und zwei versteckten Schichten

Wie durch die Quantisierung zu erwarten, verringert sich die Genauigkeit der Vorhersagen. Dieser Effekt tritt bei weniger komplexen Datensätzen wie MNIST nicht so stark auf, wie Tabelle 4.2 zeigt. In komplexen Datensätzen wie dem ImageNet-Datensatz fällt auf, dass die Komplexität der Bilder für das BNN eine Rolle spielt. In den letzten Jahren konnten Genauigkeiten von 86,4 % bis 91 %, mit Gleitkommazahlen im ImageNet-Datensatz erreicht werden [16]. Die Klassifikationsgenauigkeit von BNNs liegt um ca. 40 % darunter, wie die Tabelle 4.3 zeigt. Die angabe der Topologie soll dabei verdeutliche, dass das hier erzeugte BNN weniger komplex ist. Das BNN dieser Arbeit hat die Topologie: 784-2048-10.

Quelle	**Genauigkeit (%)**	**Topologie**
[13]	95.7	FC200-3FC100-FC10
[2]	96.0	MLP
[1]	97.69	MLP
[2]	98.24	MLP
[22]	98.77	FC784-3FC512-FC10

Tabelle 4.2: Aktueller Stand von BNNs mit dem MNIST Datensatz

Quelle	**Top-1 Genauigkeit (%)**	**Topologie**
[7]	36,1	AlexNet
[7]	47,1	GoogleNet

Tabelle 4.3: Aktueller Stand von BNNs mit dem ImageNet-Datensatz

4.2.3 Implementierung von BNNs

Die Berechnung der Werte der Ausgangsneuronen wird durch die Summe der Produkte aus Eingangswerten und Aktivierungswert gebildet. Für die effizienteste Umsetzung in Hardware soll die Vorhersage durch Bitoperation und Logikgatter realisiert werden. Die Verwendung von $w \in \{-1, 1\}$ beim Training ermöglicht eine Hardwareimplementierung der Multiplikation durch ein XNOR-Gatter. Ein Vergleich wird in der Tabelle 4.4 dargestellt. Die Berechnung der Summe der Produkte erfolgt durch eine hierarchische Addition der XNOR-Ergebnisse. Die Zwischensummen werden sukzessive summiert, bis ein Gesamtergebnis vorliegt. Anschließend wird überprüft, ob die Anzahl der Einsen, die der Nullen übersteigt, um die finale Entscheidung zu treffen [10].

$-1 \cdot -1$	1
$-1 \cdot 1$	-1
$1 \cdot -1$	-1
$1 \cdot 1$	1

0 XNOR 0	1
0 XNOR 1	0
1 XNOR 0	0
1 XNOR 1	1

Tabelle 4.4: Vergleich von Multiplikation und XNOR Wahrheitstabellen

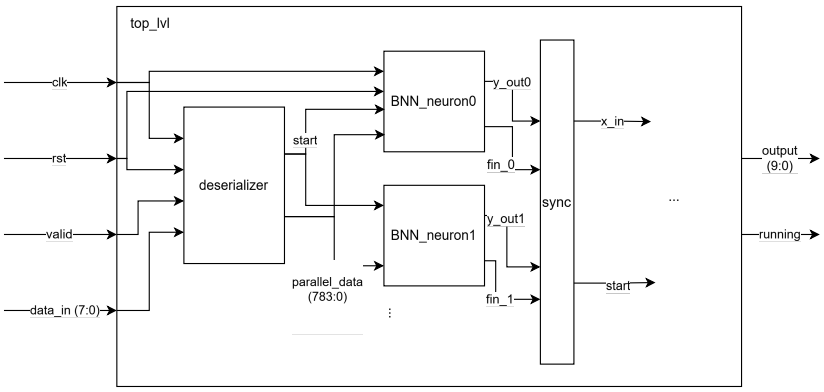


Abbildung 4.6: Blockdiagramm des BNNs

In Abbildung 4.6 ist eine von vielen möglichkeiten der Implementierung angedeutet. Das Blockdiagramm zeigt neben einer beliebigen Anzahl an Neuronen in der ersten Schicht, auch die Synchronisation, welche zwischen zwei Schichten erforderlicht ist. Die Synchronisation stellt sicher, dass alle Neuronen zu einem Ergebnis gekommen sind. So wird auch klar, warum die Implementierung von weniger Schichten einen zeitlichen Vorteil bietet. Jede Synchronisationsschicht führt zu einer Wartezeit bis alle Ergebnis sicher anliegen.

4.2.4 Abschätzung des Ressourcenbedarfs

Resource	Utilization	Available	Utilization %
LUT	1930	53200	3.63
FF	2378	106400	2.23
IO	13	200	6.50
BUFG	1	32	3.13

Tabelle 4.5: FPGA-Ressourcennutzung BNN mit einem Neuron

Resource	Utilization	Available	Utilization %
LUT	3650	53200	6.86
FF	3180	106400	2.99
IO	15	200	7.50
BUFG	1	32	3.13

Tabelle 4.6: FPGA-Ressourcennutzung BNN mit zwei Neuronen

Aus Subtraktion der beiden Tabellen ergibt sich ein Richtwert für den Ressourcenbedarf eines Neurons. Hier ist keine exakte Zahl anzugeben, da die Synthese möglicherweise die Schaltung optimiert. Je nach Wahl der internen Gewichte ergibt sich dann eine stärkere oder schwächere Optimierung. Dabei ist die Veränderung der IO-Port Auslastung von

Resource	Utilization	Available	Utilization %
LUT	1720	53200	3,23
FF	802	106400	0,75

Tabelle 4.7: FPGA-Ressourcennutzung für ein einzelnes Neuron

einem Neuron zu zwei Neuronen der Synthetisierbarkeit geschuldet und würde in dem BNN in HW nicht auftreten[1].
Die Größe des Eingabevektors von 784 Bit, in der ersten versteckten Schicht, erfordert die Addition jedes XNOR Produkts. So werden 784 Addierer benötigt. Um die Menge an Additionen zu verringern ist die Betrachtung eines Faltenden BNNs nötig. Auch die mehrfache Verwendung der Addierer wäre denkbar. Dabei ist jedoch die dadurch verringerte Klassifikationsgeschwindigkeit abzuwägen.

[1] Wenn das Ergebnis einer Berechnung nicht ausgegeben wird, wird die gesamte Berechnung nicht synthetisiert

4.3 Entscheidungsbaum

4.3.1 Modellerzeugung

Der Entscheidungsbaum wird mithilfe des Datentyps struct erzeugt. Dieser ermöglicht es, Datencontainer zu gruppieren. Diese Datencontainer können jeden Datentyp beinhalten. Somit kann das Wurzelelement die darunterliegenden linken und rechten Knoten sowie die Schwellwerte für die Kanten enthalten. Die linken und rechten Knoten enthalten jeweils wieder die darunterliegenden Knoten. Durch die Berechnung der Entropie und dem daraus resultierenden Informationsgewinn wird in jedem Knoten die beste Aufteilung gefunden.

Algorithm 4 Erstellen eines Entscheidungsbaumes (`createDecisionTree`)

1: **Eingabe:** Merkmalsmatrix X, Zielvariablen Y, maximale Baumtiefe $maxDepth$
2: **Ausgabe:** Entscheidungsbaum *tree*
3: Erstelle einen Knoten *node* mit den Attributen:
4: `isLeaf`, `class`, `feature`, `threshold`, `left`, `right`.
5: **if** $maxDepth = 0$ **oder** alle Werte in Y sind gleich **then**
6: Setze $node.isLeaf \leftarrow$ true
7: Setze $node.class \leftarrow$ Modus von Y
8: **return** *node*
9: **end if**
10: `[bestFeature, bestThreshold]` $\leftarrow$ `findBestSplit`(X, Y)
11: **if** `bestFeature` ist leer **then**
12: Setze $node.isLeaf \leftarrow$ true
13: Setze $node.class \leftarrow$ Modus von Y
14: **return** *node*
15: **end if**
16: Aufteilung:
17: $left \leftarrow X[:, \texttt{bestFeature}] \leq \texttt{bestThreshold}$
18: $right \leftarrow X[:, \texttt{bestFeature}] > \texttt{bestThreshold}$
19: $node.feature \leftarrow$ `bestFeature`, $node.threshold \leftarrow$ `bestThreshold`
20: $node.left \leftarrow$ `createDecisionTree`$(X[left,:], Y[left], maxDepth - 1)$
21: $node.right \leftarrow$ `createDecisionTree`$(X[right,:], Y[right], maxDepth - 1)$
22: **return** *node*

Die Vorhersage erfolgt über die rekursive Funktion traverseTree():

Algorithm 5 Durchlaufen des Entscheidungsbaumes (`traverseTree`)

Eingabe: Knoten: *node*, Merkmalsvektor: x
Ausgabe: Vorhergesagte Klasse: *label*
if $node.isLeaf$ **then**
    return $node.class$ ▷ Blattknoten: Klasse zurückgeben
else
    if $x[node.feature] \leq node.threshold$ **then**
        $label \leftarrow$ `traverseTree`$(node.left, x)$ ▷ Zum linken Teilbaum gehen
    else
        $label \leftarrow$ `traverseTree`$(node.right, x)$ ▷ Zum rechten Teilbaum gehen
    end if
end if
return *label*

Durch rekursives Aufrufen des linken oder rechten Knotens wird der Baum von oben bis unten durchlaufen. Sobald die Funktion an einem Blatt ankommt, wird das Label des Blattes als Vorhersage zurückgegeben.

4.3.2 Genauigkeit des Modells

Um die Genauigkeit des Modells genauer zu untersuchen, wird die Genauigkeitsberechnung mit verschiedenen Tiefen wiederholt. In Abbildung 4.7 ist der Verlauf der Genauigkeit abgebildet. Die Genauigkeit steigt durch zunehmende Tiefe schnell an. Ab einer Tiefe von 9 wird die Steigung geringer. Die Genauigkeit des Models ist mit einer Tiefe von 13 am höchsten und beträgt 88,47 %. Ab dieser Tiefe hat der Entscheidungsbaum seine maximale Genauigkeit erreicht. Anschließend fällt die Genauigkeit wieder, da der Entscheidungsbaum immer weniger generalisiert. Durch die relativ geringe Komplexität des Datensatzes, fällt der Mangel an Generalisierung, also die Überanpassung nicht sehr stark auf.

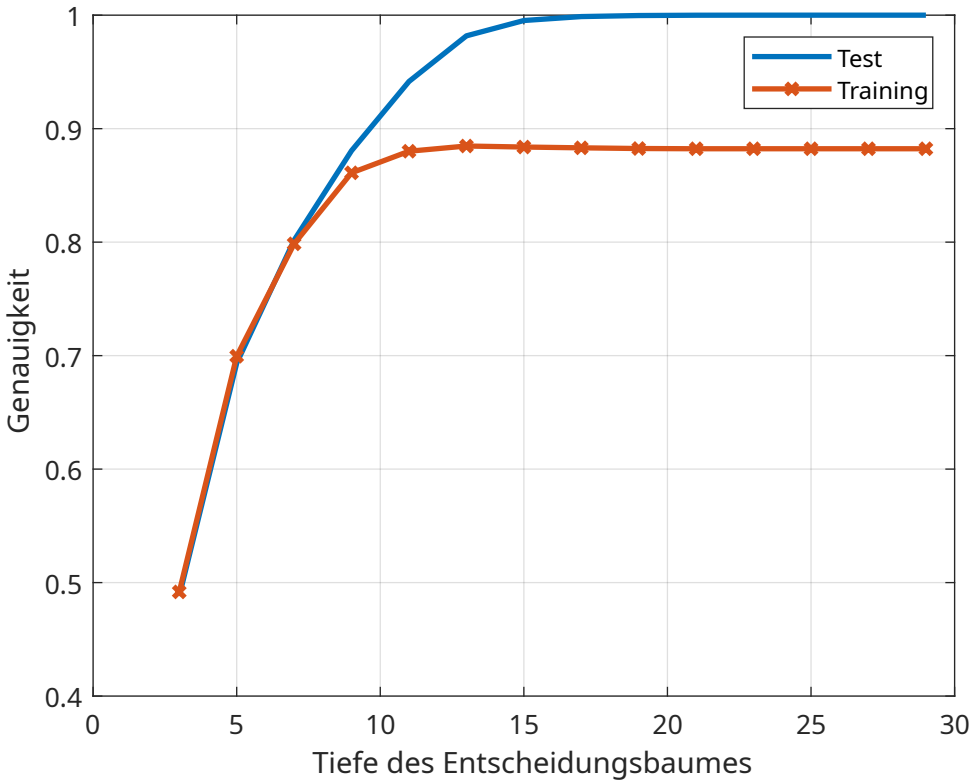


Abbildung 4.7: Verlauf der Genauigkeit in Abhängigkeit der Tiefe des Entscheidungsbaumes

4.3.3 Abschätzung des Ressourcenbedarfs

In der Funktion traverseTree() wird in der Software nicht jeder nachfolgende Knoten aufgerufen. Auf einem FPGA hingegen muss jeder dieser Zweige durch eine IF-else-Kondition vorhanden sein.

Durch eine erste Implementierung des Entscheidungsbaumes auf dem ZedBoard ergab sich eine geringe Auslastung von 4407 LUT, 182 LUTRAM und 7832 FF, wie in Tabelle 4.8.

Der BUFG ist für den globalen Takt zuständig. Die IO-Ressourcenauslastung entsteht, da die Synthesesoftware davon ausgeht, dass die Ein- und Ausgänge aus dem FPGA herausgeführt werden sollen. In einer finalen Implementierung würden diese intern verbunden und nicht als FPGA Ein- und Ausgabe aufgeführt.

Ressource	**Auslastung**	**Verfügbar**	**Auslastung %**
LUT	4407	53200	8,23
LUTRAM	182	17400	1,05
FF	7832	106400	7,36
IO	42	200	21,00
BUFG	1	32	3,13

Tabelle 4.8: Ressourcenauslastungstabelle für den Entscheidungsbaum

5 Validierung des Algorithmus auf dem ZedBoard Zynq-7000

Wie das Kapitel 4 bereits gezeigt hat, ist die Implementierung eines Entscheidungsbaumes möglich. Das kNN überzeugt mit seiner Genauigkeit von 91,30 %. Die 6.000 Takte stellen jedoch ein Skalierungsproblem dar, welches beim Entscheidungsbaum nicht besteht. Der Entscheidungsbaum ließe sich für komplexere Datensätze in der Tiefe anpassen. Zudem erlaubt die geringe Auslastung die parallele Verwendung mehrerer Entscheidungsbäume, wodurch komplexere Datensätze bewältigt werden können. Das BNN ist in der Form, wie es in dieser Arbeit erzeugt wurde, nicht implementierbar. Die Genauigkeit von 78,33 % ist die geringste Genauigkeit der untersuchten Algorithmen. Diese Genauigkeit bedarf einer Anzahl von 2048 Neuronen. Über eine Hochrechnung des Ressourcenbedarfs für ein Neuron ließen sich maximal 30 Neuronen implementieren. Aus genannten Gründen wird in diesem Kapitel der Entscheidungsbaum genau untersucht und auf dem ZedBoard implementiert.

5.1 Gegenwärtiger Zustand der Toolboxen

Da mit ML und FPGAs zwei sehr komplexe Themenfelder verbunden sind, ist die Nachfrage an Toolboxen für die Erstellung von Prototypen hoch. Für komplexe Systeme ist die ständige Abstimmung zwischen einer ML- und der FPGA-Abteilung, sowie ein langwieriger Optimierungsprozess für den spezifischen FPGA-Chip notwendig. Die Toolboxen automatisieren die Optimierung wie Quantisierung und Implementierung auf dem Chip.

Deep Learning HDL Toolbox [MATLAB]

Matlab ist eine Software. Speziell darauf ausgelegt, numerische Berechnungen effizient und präzise durchzuführen, indem es fortschrittliche Algorithmen für lineare Algebra, Differenzialgleichungen, Interpolation, Optimierung und Signalverarbeitung bietet. Es verarbeitet große Matrizen und Arrays, was es ideal für iterative Berechnungen, Simulationen und die Lösung von mathematischen Modellen macht, die numerische Methoden erfordern.

Matlab bietet mit der Deep Learning HDL Toolbox eine Möglichkeit schnell Prototypen zu erzeugen. Wie der Name verrät, bietet diese Toolbox ausschließlich die Möglichkeit von Deep Learning-Algorithmen. Dies ist wie in Abbildung 3.1 nur ein Teil des Bereiches ML. Für die Implementierung steht eine große Auswahl an vortrainierten Convolutional Neural Networks (CNNs) zur Verfügung.

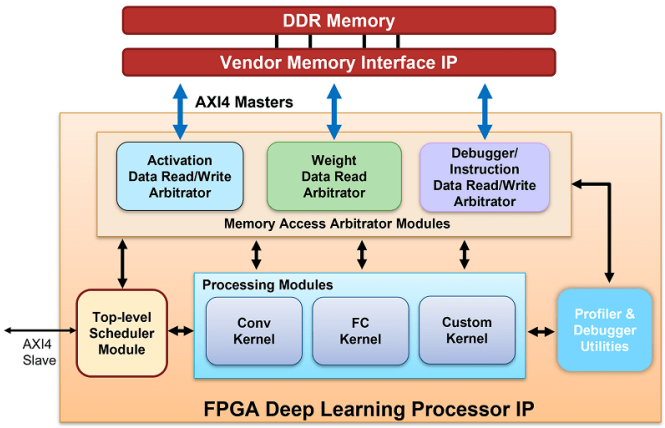


Abbildung 5.1: Illustration des Deep Learning Prozessors[12]

Wie Abbildung 5.1 zeigt, wird auf dem FPGA ein IP-Core synthetisiert. Der Begriff IP-Core steht für Intellectual Property Core. Es handelt sich dabei um wiederverwendbare, vorgefertigte Module, die in der digitalen Schaltungstechnik eingesetzt werden. Diese Module stellen bestimmte Funktionen oder Komponenten bereit, die in FPGAs, ASICs oder SoCs integriert werden können. Dieser erhält die Gewichte und den dazugehörigen Bias-Parameter[1] aus dem Double Data Rate - Random Access Memory (DDR-RAM). Diese Designentscheidung bringt den von Matlab gewünschten Effekt, dass Änderungen am neuronalen Netz schneller erfolgen können. Der Ressourcenverbrauch des IP-Core lässt sich, durch Parameter in Matlab, ebenfalls anpassen. Die Toolbox unterstützt jedoch nur Xilinx Zynq® SoC board ZC706, Zynq UltraScale+™ MPSoC ZCU102, Intel Arria® 10 SoC development board und das Intel Arria 10 GX FPGA board. Der Standard IP-Core für den Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit, benutzt, wie in Tabelle 5.1 gezeigt, mehr Ressourcen als auf dem ZedBoard verfügbar sind[2][4].

Für die begrenzte Zeit für diese Arbeit ist die Entwicklung einer eigenen Schnittstelle zu zeitaufwändig. Aufgrund der Beschränkung auf CNN und der große Ressourcenbedarf, der durch tiefe neuronale Netze entsteht, wird diese Toolbox im weiteren Verlauf nicht verwendet. Zusätzlich erzeugt die Toolbox einen Bitstream. Mit diesem ist die Einbindung in ein eigenes System nicht möglich.

	DSPs	Block RAM tiles	LUTs(CLB/ALUT)
Verfügbar	2520	912	274080
DL Prozessor	389 (16%)	508 (56%)	216119 (79%)

Tabelle 5.1: Matlabs Deep Learning Processor Estimator Resource Results

[1] Verschiebung der Aktivierungsfunktion auf der x-Achse
[2] Vgl. Abbildung 2.1

5.2 Entscheidungsbaum in VHDL

Für die Implementierung des Entscheidungsbaumes wurde eine Matlab Funktion geschrieben, welche den Datentyp struct durch fprintf()-Befehle in die sogenannte Hardware Beschreibungssprache, Very High Speed Integrated Circuits Hardware Description Language (VHDL), übersetzt.

Die Eingabedaten werden seriell eingelesen und parallel an den Entscheidungsbaum weitergegeben. Dies ist im Blockdiagramm 5.2 gezeigt.

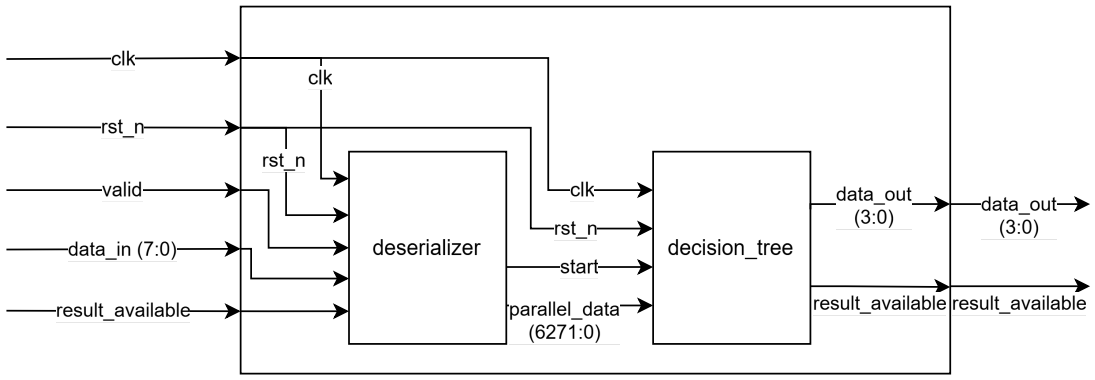


Abbildung 5.2: Blockdiagramm des Entscheidungsbaumes

Der Deserializer liest mit jedem Takt 8 Bit ein. Über ein Schieberegister werden die Bits für die parallele Weitergabe an den Entscheidungsbaum vorbereitet. Der Entscheidungsbaum erhält im Takt nach Fertigstellung der parallelen Daten, das Startsignal sowie die Gesamtdaten des Bildes. Die parallelen Bilddaten werden nun zwischengespeichert und der Deserializer beginnt erneut mit dem Konvertieren von seriellen Daten. Der Deserializer kann nach Bedarf auf andere Bitbreiten eingestellt werden. Dies beschleunigt den Einleseprozess.

Die parallelen Bilddaten durchläuft anschließend die hierarchische Logik des Entscheidungsbaumes, zur Bestimmung des Labels. Da es sich um reine Logik handelt, wird nach dem Startsignal des Deserializers ein Takt gewartet bis das Ergebnis abgefragt wird. Ob die Signallaufzeit tatsächlich kürzer ist als der Takt, muss später über den Timing Report geprüft werden. Um die Wartezeit einstellbar zu gestalten, kann über die Pipelinelänge die Anzahl an Takten eingestellt werden. Eine Pipelinelänge von eins entspricht der minimalen Wartezeit von einem Takt.

5.3 Simulation

Zur Simulation des Verhaltens wurde eine sogenannte Testbench verwendet. In dieser werden seriell Bilder an das top Level gegeben. Die Bilder stammen aus den MNIST-Validierungsdaten. Nachdem der Entscheidungsbaum eine Vorhersage ausgegeben hat, wird diese, in der Testbench, mit dem tatsächlichen Label verglichen. Durch einen Fehlerzähler wird am Ende des Simulationsdurchgangs die Genauigkeit in Prozent angegeben.

In der Abbildung 5.3 ist die simulierte Vorhersage zu sehen. Die Simulation ergab mit 10.000 Testdaten eine Genauigkeit von 88,47 %.

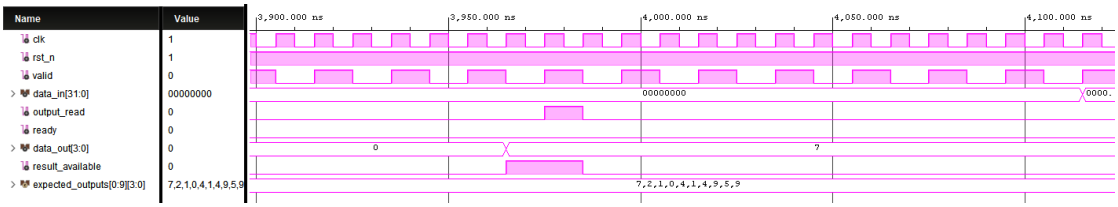


Abbildung 5.3: Darstellung einer simulierten Vorhersage

5.4 Aufbau des Experiments zur Verifikation

Durch die Implementierung eines Zynq Prozessors auf dem ZedBoard können die Testdaten im DDR3 RAM des Prozessors gespeichert und an das Top-Level übergeben werden. Der DDR3 RAM des ZedBoards bietet mit den 512 MB genug Speicher für diesen Versuch [28]. Der Ressourcenbedarf, mit Implementierung des Zynq Prozessors und Schnittstellen zum FPGA, ist in Tabelle 5.2 dargestellt.

Ressource	**Auslastung**	**Verfügbar**	**Auslastung %**
LUT	5435	53200	10,22
LUTRAM	184	17400	1,06
FF	9105	106400	8,56
BUFG	1	32	3,13

Tabelle 5.2: Ressourcenauslastungstabelle für den Entscheidungsbaum mit Zynq

Mit der Vitis IDE von Xilinx wird nun der Prozessor programmiert. Die Testdaten werden über die GPIO an den FPGA übergeben. Im ersten Test in 784 8 Bit Paketen. Nach dem Seriellen einlesen wird auf das Signal *result_ available* gewartet. Sobald dieses anliegt, ist das Ergebnis fertig und kann ausgewertet werden. Auch hier werden Fehler gezählt und anschließend ausgegeben. Die Ausgabe erfolgt über die UART Schnittstelle des Boards. Anschließend wird der Test mit veränderten Parametern, wie Frequenz und Inputdatenbreite, durchgeführt.

Die maximale Frequenz wird in dieser Arbeit nur für das ZedBoard bestimmt, da die Implementierung auf anderen FPGAs zu anderen Ergebnissen führt. Vivado besitzt keine automatische Bestimmung der maximalen Frequenz. Daher wird in dieser Arbeit die Taktfrequenz von der maximalen Frequenz zu kleineren Frequenzen iteriert. Die maximale Frequenz beträgt 250 MHz. Durch die Angaben des Timing Reports, zum kritischen Pfad, lässt sich die Taktperiode um die Laufzeitverletzung verlängern. So wird in wenigen Schritten mit der Formel 5.1 die maximale Frequenz ermittelt [25].

$$\mathrm{F}_{MAX}(\mathrm{MHz}) = \max\left(\frac{1000}{T_i - \mathrm{WNS}_i}\right) \tag{5.1}$$

T_i ist die eingestellte Periode und WNS_i die Laufzeitverletzung für den i-ten Durchlauf. T_i und WNS_i sind in ns einzusetzen.

6 Ergebnis

Das Ziel dieser Bachelorarbeit ist die Untersuchung verschiedener ML-Algorithmen hinsichtlich ihrer Eignung zur Implementierung auf einem FPGA. Es wurde ein Binarisiertes Neuronales Netzwerk (BNN), der k-nächste Nachbarn Algorithmus (kNN) und ein Entscheidungsbaum auf ihre Implementierbarkeit auf FPGAs und Eignung zur Klassifikation des MNIST-Datensatzes untersucht. Das BNN ist trotz seines geringeren Speicherbedarfs, im Vergleich zu NN mit Gleitkommazahlen und der vielversprechenden Ersetzbarkeit von Multiplikation durch XNOR Logik nicht vollständig implementiert worden. Der Ressourcenbedarf der Vollverbunden-Neuronen war mit 3,23 % LUT Auslastung pro Neuron zu groß, um die nötige Menge an Neuronen zu implementieren. Im Fazit wird näher auf das BNN eingegangen.

Der kNN-Algorithmus war mit 95,71 % Genauigkeit der genauste Algorithmus. Die Prototypen, welche für den Vergleich der nächsten Nachbarn notwendig sind, erfordern RAM oder ROM zur Speicherung. Die Latenz, welche mit der Verwendung von Block RAM einhergeht, führte zum Ausschluss des Algorithmus für weitere Implementierung.

Der Entscheidungsbaum hat durch dessen geringen Ressourcenbedarf und einer Genauigkeit von 88,47 % überzeugt. Trotz einer geringeren Genauigkeit zum kNN überzeugte er durch seinen sehr geringen Ressourcenverbrauch und seine einfache logische Struktur. Die hierarchische Entscheidungslogik lässt sich effizient auf dem FPGA implementieren. Zudem ermöglicht die geringe Komplexität der Vergleichsoperationen eine schnelle Klassifikation. Ein weiterer Vorteil liegt in der guten Skalierbarkeit. Durch die geringe Auslastung könnten mehrere Entscheidungsbäume parallel implementiert werden, um komplexere Aufgaben zu bewältigen, beispielsweise im Sinne eines Random-Forest-Ansatzes. Der Anhang zur Arbeit befindet sich auf CD und kann beim Erstgutachter eingesehen werden.

6.1 Genauigkeit

Die Ergebnisse des Hardwaretestes wurden über die UART-Schnittstelle ausgegeben. Die Tests mit verschiedenen Konfigurationen sind in Tabelle 6.1 aufgelistet. Dabei bezieht sich die Genauigkeit immer auf den gesamten Validierungsdatensatz mit 10.000 Testdaten. Die Untersuchung der FPGA Klassifikation zeigt, dass die 100 MHz und 150 MHz Konfiguration die erwartete Genauigkeit der Software liefert. Bei der Klassifikation mit 200 MHz oder 250 MHz ist die Signallaufzeit länger als die Taktperiode. Dadurch kommt es zu Setup-Time Violation. Die FPGA Taktflanke kommt vor dem eigentlichen Signal an. Folglich ist das Schieberegister noch nicht fertig, bevor der nächste Befehl zum Schieben kommt oder das Signal noch nicht vollständig durch den Entscheidungsbaum gelaufen, wenn die Ausgabe abgefragt wird. Zu erkennen ist der Fehler an der verringerten Genauigkeit durch Bitfehler in den eingelesenen Daten. Die Zeitmessung beinhaltet auch die Zeit des seriellen Einlesens.

Serielle Breite	Frequenz	Pipelinelänge	Genauigkeit	Zeit pro Bild
8 Bit	200 MHz	1	88,45 %	436 μS
32 Bit	100 MHz	1	88,47 %	165 μS
32 Bit	150 MHz	1	88,47 %	128 μS
32 Bit	150 MHz	2	88,47 %	128 μS
32 Bit	200 MHz	1	88,45 %	108 μS
32 Bit	250 MHz	1	49,32 %	95 μS

Tabelle 6.1: Hardware Ergebnisse

Auch wenn die Ergebnisse aus Tabelle 6.1 für 150 MHz und einer Pipelinelänge von 1 suggerieren, dass die Klassifikation wie erwartet erfolgt, könnten Umwelteinflüsse, wie die Temperatur, die Signallaufzeit auf die Zeiten des Timing Reports verlängern und die Genauigkeit negativ beeinflussen.

Die Tabelle 6.1 zeigt, dass die Implementierung die erwartete Genauigkeit von 88,47 % liefert und der Algorithmus auch in Hardware funktioniert.

6.2 Maximaler Systemtakt

Durch die iterative Berechnung der maximalen Taktfrequenz nach Gleichung 5.1 in Tabelle 6.2, konnte ca. 111 MHz als maximale Frequenz ermittelt werden. Durch die Optimierung je nach eingestellter Periodendauer kommt es zu Schwankungen im WNS_i-Wert. Negative WNS_i-Werte deuten darauf hin, dass die Signallaufzeit die eingestellte Periode überschreitet, was zu Timing-Verletzungen führen kann. Erst bei einer Periodendauer von 9 ns ergab der Timing Report keine Überschreitung der Signallaufzeit, wodurch die Implementierung stabil bei $\frac{1}{9\,\mathrm{ns}} \approx 111$ MHz arbeitet.

i	T_i [ns]	WNS_i [ns]	F_{MAX}
1	4	-4,94	117,785 MHz
2	8,49	-0,469	111,619 MHz
3	8,96	-0.005	111,544 MHz
4	9	0.089	112,220 MHz

Tabelle 6.2: Ermittlung der maximalen Taktfrequenz

Die Umstellung der Standard-Synthese-Direktiven auf *PerformanceOptimized* sowie der Implementierungs-Direktiven von Standard auf *Explore* hatte keinen positiven Einfluss auf die Signallaufzeit [25]. Daher wurde die Ermittlung des maximalen Systemtakts mit den Standardeinstellungen durchgeführt.

Die maximale Frequenz von 111 MHz stellt sicher, dass das System auch bei erschwerten Umwelteinflüssen wie hoher Temperatur optimal funktioniert, da der Timing Report eine Worst-Case-Betrachtung ist. So lässt sich mit hinreichender Sicherheit sagen, dass das Design innerhalb der FPGA Spezifikation funktionsfähig bleibt.

6.3 Klassifikationsgeschwindigkeit

Der kritische Pfad, welcher von Vivado identifiziert wurde, ist die Verbindung von dem parallelen Register des Entscheidungsbaumes zur Ausgabe. Die Gesamtverzögerung beträgt dabei 8,794 ns, die Zusammensetzung der Verzögerung ist in Tabelle 6.3 aufgeschlüsselt. Die Synthese wurde mit einer eingestellten Taktperiode von 9 ns durchgeführt und dementsprechend für diesen Takt von Vivado optimiert.

Gesamtverzögerung	Logikverzögerung	Netzverzögerung
8,794 ns	2,483 ns	6,311 ns

Tabelle 6.3: Setupzeiten des kritischen Pfads vom Vivado Timing Report mit 9 ns Taktperiode

Die Gesamtverzögerung von 8,794 ns bedeutet, dass die Vorhersage des Entscheidungsbaumes, ohne Einleseprozess, innerhalb eines Taktes abgeschlossen ist. Für den Timing Report wird immer der schlimmste mögliche Fall betrachtet. Vergleichen wir nun die Klassifizierung von Matlab in Tabelle 6.4. Matlab benötigt für die Klassifizierung von 10.000 Testdaten 104,944 ms.

Matlab	FPGA (XC7Z020)	Zeitdifferenz	FPGA Beschleunigung
104,944 ms	90 μs	104,854 ms	ca. 1166

Tabelle 6.4: Vergleich der Klassifikationszeit von Matlab und der erarbeiteten FPGA Lösung für 10.000 Testdaten

Das Modell kann mit jedem 111 MHz Takt die parallelen Daten einlesen und klassifizieren. Dies bedeutet, dass das Entscheidungsbaummodell auf dem XC7Z020 eine Klassifikation mit einer Latenz von 9 ns durchführt. Durch diese Erkenntnis ist eine weite Betrachtung für Echtzeitsysteme mit entsprechenden Anforderungen denkbar.

6.4 Ressourcenbedarf

Der Ressourcenbedarf für die HW Lösung mit Zynq Prozessor ist in Tabelle 6.5 abgebildet. Wichtiger ist jedoch die Auslastung des Entscheidungsbaumes. Diese Auslastung ist in Tabelle 6.6 dargestellt. Die Auslastung des Top levels oder sogar ohne Deserializer entspricht dem tatsächlichen Speicherbedarf im Use Case. Die Ressourcenauslastung im Use Case wird je nach FPGA Model und Synthesesoftware leicht variieren.

Ressource	**Auslastung**	**Verfügbar**	**Auslastung %**
LUT	5435	53200	10,22
LUTRAM	184	17400	1,06
FF	9105	106400	8,56
BUFG	1	32	3,13

Tabelle 6.5: Ressourcenauslastungstabelle für den Entscheidungsbaum mit Zynq

Ressource	**Auslastung**	**Verfügbar**	**Auslastung %**
LUT	4407	53200	8,23
LUTRAM	182	17400	1,05
FF	7832	106400	7,36
IO	42	200	21,00
BUFG	1	32	3,13

Tabelle 6.6: Ressourcenauslastungstabelle für den Entscheidungsbaum und den Deserializer

7 Fazit

In dieser Arbeit wurde erfolgreich ein Entscheidungsbaum zur Klassifikation des MNIST-Datensatzes entwickelt, der sowohl in Software als auch in Hardware eine Genauigkeit von 88,47 % erzielt. Zusätzlich ergab sich ein geringer Ressourcenbedarf, da der relativ kleine FPGA des ZedBaords nur zu ca. 10 % ausgelastet wurde. Zeitlich kann der FPGA, exklusive Eingabedatenparallelisierung, in 9 ns die Vorhersage ausgeben.
Des Weiteren wurden Erkenntnisse über kNN und BNN gewonnen und dargestellt.
Weitere Untersuchungen und Verbesserungen dieser Bachelorarbeit sind folgende:

- **Standard Schnittstelle wie AXI:** Die Einführung standardisierter Schnittstellen für die FPGA-Ein- und -Ausgabe, wie beispielsweise AXI, würde die Wiederverwendbarkeit des in dieser Arbeit entwickelten FPGA-Klassifikators in anderen Projekten erleichtern und dessen Integration in größere Systeme vereinfachen.

- **Interrupt basierte Programmierung:** Um die Zeitmessung über den Zynq Prozessor, präziser und den Hardwaretest effizienter zu gestalten, ist eine Interrupt basierte Programmierung zu empfehlen, da so Warteschleifen im Zynq Prozessor vermieden werden können.

- **Faltende BNNs:** Die Analyse der BNNs ergab einen hohen Ressourcenbedarf, insbesondere durch die vollständig verbundenen Schichten mit vielen Additionen. Eine alternative Struktur mit faltenden binarisierten neuronalen Netzen könnte den Ressourcenverbrauch verringern. Eine genauere und zeitaufwändigere Untersuchung ist notwendig, um das Potenzial dieser Ansätze für FPGAs zu bewerten.

- **Random-Forest:** Die Nutzung mehrerer Entscheidungsbäume im Sinne eines Random-Forest könnte die Klassifikationsgenauigkeit für komplexere Datensätze erhöhen. Aufgrund der geringen Ressourcenauslastung eines einzelnen Entscheidungsbaums wäre eine parallele Implementierung mehrerer Bäume auf dem FPGA prinzipiell möglich und stellt einen interessanten Forschungsansatz dar.

7.1 Reflexion

Der Entscheidungsbaum konnte erfolgreich auf dem ZedBoard implementiert und analysiert werden. Die Ergebnisse ermöglichen eine Klassifikation im Sinne des in Kapitel 1 beschriebenen Use Cases.

Die Vorhersage liegt innerhalb der 9 ns Taktperiode an. Somit wäre auch eine Echtzeitanwendung denkbar, bei der die Eingabedaten parallel übergeben werden. Durch Pipelining lässt sich die maximale Taktfrequenz steigern, sodass das Design auch in bestehende FPGA-Systeme mit höheren Frequenzen integriert werden kann.

Die Implementierung hat gezeigt, dass sich einfache ML-Modelle wie Entscheidungsbäume aufgrund ihrer logischen Struktur und geringen Komplexität besonders gut für ressourcenbeschränkte Hardware eignen.

Das in dieser Arbeit erzeugte BNN-Modell hat eine geringere Genauigkeit als andere faltenden BNN-Modelle. Um der Komplexität des faltenden BNNs gerecht zu werden, ist eine gezielte Untersuchung dieses einen Modells erforderlich.

Die Durchführung dieser Arbeit ermöglichte mir ein tieferes Verständnis für die Herausforderungen bei der Hardware-Implementierung von Algorithmen. Insbesondere die Verbindung von maschinellem Lernen mit digitalem Schaltungsentwurf war neu für mich und hat meine Kenntnisse in ML, VHDL, Ressourcenplanung und Simulation deutlich erweitert.

Gleichzeitig wurden auch Grenzen deutlich, beispielsweise bei der Umsetzung binarisierter neuronaler Netze. Hier sind weitere Untersuchungen notwendig, um eine effiziente Struktur für FPGAs zu finden.

Insgesamt liefert die Arbeit eine gute Grundlage für zukünftige Projekte, etwa die Implementierung von Random-Forest-Modellen oder die Weiterentwicklung von BNNs. Auch die Untersuchungen von echten Daten des Use Cases kann durch diese Arbeit schnell erfolgen. Die bereitgestellten Algorithmen eignen sich für die Klassifikation von Datensätzen mit anderen Parametern. Insbesondere der Entscheidungsbaum kann automatisiert erzeugt werden und auf beliebigen FPGAs implementiert werden.

Literaturverzeichnis

[1] Blott, Michaela ; Preusser, Thomas B. ; Fraser, Nicholas J. ; Gambardella, Giulio ; O'brien, Kenneth ; Umuroglu, Yaman ; Leeser, Miriam ; Vissers, Kees: FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. In: *ACM Trans. Reconfigurable Technol. Syst.* 11 (2018), Dezember, Nr. 3. – URL https://doi.org/10.1145/3242897. – ISSN 1936-7406

[2] Courbariaux, Matthieu ; Bengio, Yoshua: BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. In: *CoRR* abs/1602.02830 (2016). – URL http://arxiv.org/abs/1602.02830

[3] Courbariaux, Matthieu ; Bengio, Yoshua ; David, Jean-Pierre: BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In: *CoRR* abs/1511.00363 (2015). – URL http://arxiv.org/abs/1511.00363

[4] *Deep Learning HDL Toolbox*. – URL https://de.mathworks.com/help/deep-learning-hdl/. – Stand: 25.11.2024

[5] Inc., Xilinx: *Zynq-7000 SoC Data Sheet: Overview (DS190)*. Version 1.11.1. DS190: , July 2018. – URL https://www.xilinx.com

[6] Ertel, Wolfgang: *Grundkurs Künstliche Intelligenz*. Vieweg+Teubner Verlag Wiesbaden, 20016. – ISBN 978-3-658-13549-2

[7] Hubara, Itay ; Courbariaux, Matthieu ; Soudry, Daniel ; El-Yaniv, Ran ; Bengio, Yoshua: Binarized Neural Networks. In: Lee, D. (Hrsg.) ; Sugiyama, M. (Hrsg.) ; Luxburg, U. (Hrsg.) ; Guyon, I. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 29, Curran Associates, Inc., 2016. – URL https://proceedings.neurips.cc/paper_files/paper/2016/file/d8330f857a17c53d217014ee776bfd50-Paper.pdf

[8] Kesel, Frank: *FPGA Hardware-Entwurf*. Berlin, Boston : De Gruyter Oldenbourg, 2018. – URL https://doi.org/10.1515/9783110531459. – ISBN 9783110531459

[9] Lecun, Yann ; Bottou, Léon ; Bengio, Yoshua ; Haffner, Patrick: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE* 86 (1998), November, Nr. 11, S. 2278–2324. – URL https://hal.science/hal-03926082

[10] Liang, Shuang ; Yin, Shouyi ; Liu, Leibo ; Luk, Wayne ; Wei, Shaojun: FP-BNN: Binarized neural network on FPGA. In: *Neurocomputing* 275 (2018), S. 1072–1086. – URL https://www.sciencedirect.com/science/article/pii/S0925231217315655. – ISSN 0925-2312

[11] Lin, Xiaofan ; Zhao, Cong ; Pan, Wei: *Towards Accurate Binary Convolutional Neural Network*. 2017. – URL https://arxiv.org/abs/1711.11294

[12] *Deep Learning Processor IP Core Architecture*. – URL https://de.mathworks.com/help/deep-learning-hdl/ug/deep-learning-processor-architecture.html. – Stand: 25.11.2024

[13] Narodytska, Nina ; Kasiviswanathan, Shiva ; Ryzhyk, Leonid ; Sagiv, Mooly ; Walsh, Toby: Verifying Properties of Binarized Deep Neural Networks. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32 (2018), Apr., Nr. 1. – URL https://ojs.aaai.org/index.php/AAAI/article/view/12206

[14] Nurvitadhi, Eriko ; Sheffield, David ; Sim, Jaewoong ; Mishra, Asit ; Venkatesh, Ganesh ; Marr, Debbie: Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In: *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, S. 77–84

[15] Papageorgiou, Markos ; Leibold, Marion ; Buss, Martin: *Optimierung*. Springer Vieweg Berlin, Heidelberg, 2015

[16] Paperswithcode.com: *image classification on imagenet*. – URL https://paperswithcode.com/sota/image-classification-on-imagenet. – Zugriffsdatum: 18.03.2025

[17] Prof.Dr.Jünemann: Maschinelles Lernen und Neuronale Netze. In: *Vorlesung MLNN*, 2022

[18] Schröder, Dierk 1941-2024: *Intelligente Verfahren Identifikation und Regelung nichtlinearer Systeme*. Springer-Verlag Berlin Heidelberg, 2010 (SpringerLink). – URL https://doi.org/10.1007/978-3-642-11398-7

[19] Simons, Taylor ; Lee, Dah-Jye: A Review of Binarized Neural Networks. In: *MDPI* (2016). – URL https://www.mdpi.com/2079-9292/8/6/661

[20] Statista: Umsatz im Bereich Künstliche Intelligenz weltweit im Jahr 2024 und eine Prognose für 2028. In: *Statista* (2024)

[21] Steppan, Josef. – URL https://commons.wikimedia.org/w/index.php?curid=64810040. – Zugriffsdatum: 04.12.2024, CC BY-SA 4.0

[22] Sun, Xiaoyu ; Yin, Shihui ; Peng, Xiaochen ; Liu, Rui ; Seo, Jae-sun ; Yu, Shimeng: XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks. In: *2018 Design, Automation & Test in Europe Conference & Exhibition*, 2018, S. 1423–1428

[23] Tageschau: Wie stark der Stromverbrauch durch KI steigt. In: *Tageschau* (Stand: 23.10.2023 12:00 Uhr). – URL https://www.tagesschau.de/wirtschaft/digitales/ki-energie-strom-verbrauch-klimaschutz-100.html

[24] *Zynq-7000 All Programmable SoC: Technical Reference Manual*. Version 1.12.2

[25] *UltraFast Design Methodology Guide for FPGAs and SoCs*. – URL https://docs.amd.com/r/en-US/ug949-vivado-design-methodology/Methodology-DRCs-with-Impact-on-Signoff-Quality-and-Hardware-Stability

[26] Yin, Penghang ; Lyu, Jiancheng ; Zhang, Shuai ; Osher, Stanley ; Qi, Yingyong ; Xin, Jack: *Understanding Straight-Through Estimator in Training Activation Quantized Neural Nets*. 2019. – URL https://arxiv.org/abs/1903.05662

[27] Zhou, Z.H. ; Liu, S.: *Machine Learning*. Springer Nature Singapore, 2021. – URL https://books.google.de/books?id=ctM-EAAAQBAJ. – ISBN 9789811519673

[28] *Zynq Evaluation and Development Hardware Users Guide*. Version 2.2. – URL https://files.digilent.com/resources/programmable-logic/zedboard/ZedBoard_HW_UG_v2_2.pdf

Glossar

Eingabevektor Die Merkmale, auf die die Vorhersage angewendet wird.

Prototypen Bild aus dem Trainingsdatensatzes, welches einem Label zugeorndet ist.

Random-Forest Vielen Entscheidungsbäumen, die auf unterschiedlichen Datenstichproben trainiert werden. Die finale Vorhersage erfolgt durch einen Mehrheitsentscheid.

Testbench Ein digitaler Prüfstand mithilfe der, die Funktionalität des Modules, getestet wird.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original