technische universität
dortmund

Bachelorarbeit

Optimierung von logistischer Regression auf FPGAs

Moritz Sliwinski
Februar 2020

Gutachter:
Prof. Dr. Katharina Morik
Sebastian Buschjäger

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
https://www-ai.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Maschinelles Lernen und Vorhersagen werden immer mehr in unser Leben integriert. Hierbei entsteht zum einen der Anspruch an variable, nicht statische Systeme, zum anderen die Notwendigkeit kompakter und energieeffizienter Lösungen.
Aufgrund der immer weiter wachsenden Datenmengen stoßen herkömmliche Central Processing Units (CPUs) mittlerweile an Ihre Grenzen, denn durch materialbedingte Limitierung kann ihre Rechenkapazität so gut wie nicht mehr erhöht werden. Daher geht man dazu über, Mehrkernprozessoren zu entwickeln, die ihre Geschwindigkeit über parallele Threads erreichen. Diese haben jedoch einen vergleichsweise hohen Energieverbrauch.
Field Programmable Gate Arrays (FPGAs) bieten in diesem Zusammenhang einen guten Kompromiss zwischen Flexibilität in der Programmierbarkeit und Energieeffizienz. Der Vorteil der FPGAs zeigt sich in der deutlich höheren Parallelität gegenüber CPUs, sodass trotz der geringeren Taktfrequenz eine große Menge an Daten schnell verarbeitet werden kann.
Die logistische Regression ist für die Optimierung auf FPGAs in dem Sinne gut geeignet, da sie eine einfache Art von neuronalem Netz darstellt und somit gut in der FPGA-Logik darstellbar ist. Sie weist zum Beispiel durch Datenparallelität bzw. Parallelisierung von Batches, Feature- oder Hyperparameter-Berechnung eine hohe Parallelisierbarkeit auf.

Moderne FPGAs können über die PCIe-Schnittstelle als Co-Prozessor in ein System eingebunden werden, sodass deren Parallelität und Energieeffizienz ausgenutzt werden können. Dank des hohen Durchsatzes der Schnittstelle muss hierbei nicht auf eine komplexe variable Vorbereitung der Daten durch die CPU im laufenden Betrieb verzichtet werden.
Die Motivation zur Nutzung von FPGAs besteht darin, dass sie Energieeffizienz und Parallelität miteinander kombinieren. Sie bieten einen guten Kompromiss zur Nutzung von Graphics Processing Units (GPUs), denn diese weisen zwar eine noch höhere Parallelisie-

rungseigenschaft auf, benötigen aber auch deutlich mehr Energie, bis zu 20x mehr allein im Idle-Zustand [3]. Aufgrund dessen werden GPUs in der Arbeit nicht weiter behandelt.

1.2 Aufbau der Arbeit

In Kapitel 2 werden zunächst der Aufbau und die Funktionsweise von FPGAs erläutert. Es wird auf die technischen Besonderheiten und die Funktion der Einzelnen Hardwarebausteine eingegangen. Außerdem wird der Konfigurationsablauf zur Programmierung des FPGAs erklärt und seine Eigenschaften mit denen von anderen Hardwarekomponenten verglichen.

Im 3. Kapitel erörtern wir die Zusammensetzung der Logistischen Regression. Ihre Funktion als Lernfunktion wird detailliert beschrieben und mit Hilfe des Gradientenabstiegsverfahrens wird eine Updatefunktion für Koeffizienten hergeleitet. Es folgt eine Vorstellung verschiedener Regularisierungsmethoden sowie deren Einbindung in in die Updatefunktion.

in Kapitel 4 werden die Implementierten Programme dargestellt und der Code erörtert. Man geht auf die Probleme der Entwicklung für den FPGA-Code und den Hostanwedungs-Code ein, sowie dessen Grenzen in der Anwendung.
Kapitel 5 fast die Ergebnisse von Experimenten mit dem FPGA zusammen. Es werden beispielhaft Daten ausgewertet und Trainingsergebnisse vorgestellt.Auch eine Analyse der Geschwindigkeit im Vergleich zu einer CPU werden behandelt.
Im 6. Kapitel wird das Fazit aus den behandelten Themen und Ergebnissen dieser Arbeit gezogen. Es werden Empfehlungen für weitere, tiefer gehende Implementierungen und Experimente gegeben, die zum Teil aus Zeit- und Aufwandstechnischer Sicht in der Arbeit nicht behandelt werden.

1.3 Verwandte Arbeiten

Es gibt bereits einige Arbeiten zu dem Thema logistische Regression auf FPGAs die sich mit der Energie- und Geschwindigkeitsoptimierung auseinandersetzen. Wienbrandt et al. benutzen ein FPGA mit logistischer Regression in [19] und zeigen, dass ein hybrides System aus GPU und FPGA einen Speedup von über 1500 gegenüber einem PLINK hat. Allerdings benutzten sie zum Einen eine GPU mit einem Energieverbrauch von 300 Watt [25], was im Gegensatz zu einem einzelnen Artix 7 FPGA mit einem Verbrauch von ca. 5 Watt [33] sehr hoch ist, und zum anderen werden in dem Artikel keine Werte eines FPGAs als einzelnes System gemessen. Es gibt bereits eine Implementierung von InAccel von logistischer Regression auf FPGAs [13] die sich jedoch vor allem auf die Nutzung in vernetzten Systemen und Kubernetes bezieht. Zum Beispiel werden in der Implementierung alle benötigten Ressourcen für die maximalen Eingabegrößen (64 Klassen und 2047 Features) auf jedem

FPGA reserviert, was vor allem für Trainingsdaten mit deutlich weniger Features eine unnötige Speicherausnutzung bedeutet. Es wird mit float8 bzw float16 gearbeitet, sodass eine Implementierung mit Fixkommazahlen eine noch etwas bessere Performance erzielen könnte. Die Implementierung nutzt zudem keine PCIe Schnittstelle und programmiert die FPGAs während des Hostprozess mit einem Bitstream jedes mal erneut.

Kapitel 2

FPGAs

Die Arbeit befasst sich mit der Implementierung und Optimierung von Logistischer Regression auf FPGAs. Deshalb wird zunächst der allgemeine Aufbau dieser beschrieben. Dann folgt eine Einführung in die Konfiguration des FPGAs, wobei zum einen auf den typischen Ablauf, zum anderen auf die verwendeten Programme eingegangen wird. Abschließend wird die verwendete Hard- und Software aufgeführt.

2.1 Allgemeiner Aufbau von FPGAs

Field Programmable Gate Arrays (FPGAs) sind Integrierte Schaltkreise (IC) in die eine logische Schaltung programmiert werden kann. Die ICs bestehen aus I/O-Blöcken, Programmierbaren Logikblöcken (Configurable Logig Blocks, kurz CLB) und weiteren Bestandteilen wie zum Beispiel DSP-Slices, BRAM-Blöcken, Multipliziereinheiten oder Taktgeneratoren welche durch Datenpfade zu einer Matrix miteinander verbunden sind (Siehe Abbildung 2.1).
Die Pfade können je nach Bedarf geschaltet werden. Die CLBs selbst bestehen aus einem 1 Bit Flip-Flop und einer programmierbaren Wahrheitstabelle. Über diese lassen sich die logischen Funktionen konfigurieren[4]. Ein schematischer Aufbau ist in Abbildung 2.2 zu sehen. Dieser Aufbau ist typisch für ein FPGA der Marke Xilinx und nicht allgemein für andere Hersteller gültig. Da in dieser Arbeit (wie in Kapitel 2.3 beschrieben) ein FPGA der Marke Xilinx benutzt wird, wird auch dessen Hardwarekonfiguration zugrunde gelegt.

Die Programmierung ist in diesem Fall vergleichbar mit einer Schalttabelle, welche bestimmt wie die physikalischen Bausteine miteinander verbunden werden sollen. Anders als bei Application-Specific Integrated Circuits (ASICs), dessen Funktion bereits bei der Produktion festgelegt werden, können FPGAs vom Benutzer selbst Konfiguriert werden.

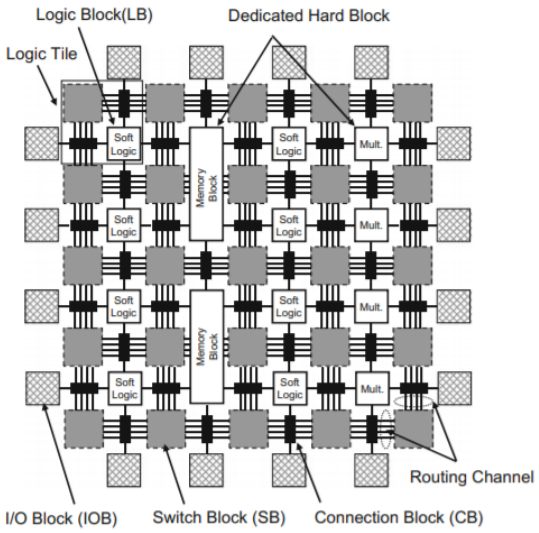

Abbildung 2.1: Aufbau eines IC, die grauen Schaltblöcke (SB) sind die konfigurierbaren Datenpfade

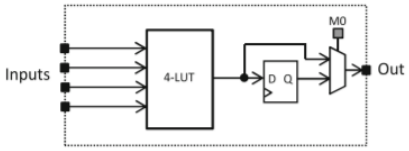

Abbildung 2.2: Schematische Darstellung eines CLB

Dies geschieht jedoch im Gegensatz zu Mikroprozessoren nicht während, sondern vor Inbetriebnahme des Chips. Zwar ist es bei einigen wenigen Herstellern von FPGAs mittlerweile möglich, diese auch wärend des laufenden Betriebs zu konfigurieren (partielle Rekonfiguration), aber das ist mit einer höheren Komplexität der zu konfigurierenden Logik verbunden.
Durch die Konfigurierbarkeit des FPGAs ergeben sich bautechnisch bedingt einige Nachteile gegenüber den ASICs. FPGAs sind annäherungsweise 20 bis 35 mal größer und zwischen 3 und 4 mal langsamer als eine vergleichbare ASIC Implementierung. Außerdem verbrauchen sie dynamisch circa 10 mal mehr Energie [18]. Damit sind sie deutlich ineffizienter als ASICs. Der große Vorteil ergibt sich hier aus der Konfigurierbarkeit, denn ASICs sind nach der Produktion nicht mehr veränderbar. Besonders in Bereichen die eine hohe Flexibilität verlangen ist es vor allem Kosteneffizienter FPGAs zu benutzen, denn die Produktion von ASICs ist mit großen zeitlichen und finanziellen Investitionen verbunden[17].

2.2 Konfiguration und Ablauf

Wie das „Field Programmable“ im Namen schon besagt ist es möglich nach der Fabrikation des FPGA Funktionen in diese „in the field“, also im praktischen Einsatz zu programmieren [17]. Um seine Funktion zu verändern muss das FPGA neu Konfiguriert werden. Dies geschieht durch einen sogenannten „Bitstream“, eine Sequenz von einzelnen Bits. In Abbildung 2.3 zeigt sich der Ablauf zur Generierung einen solchen Bitstreams für ein FPGA der Marke Xilinx.

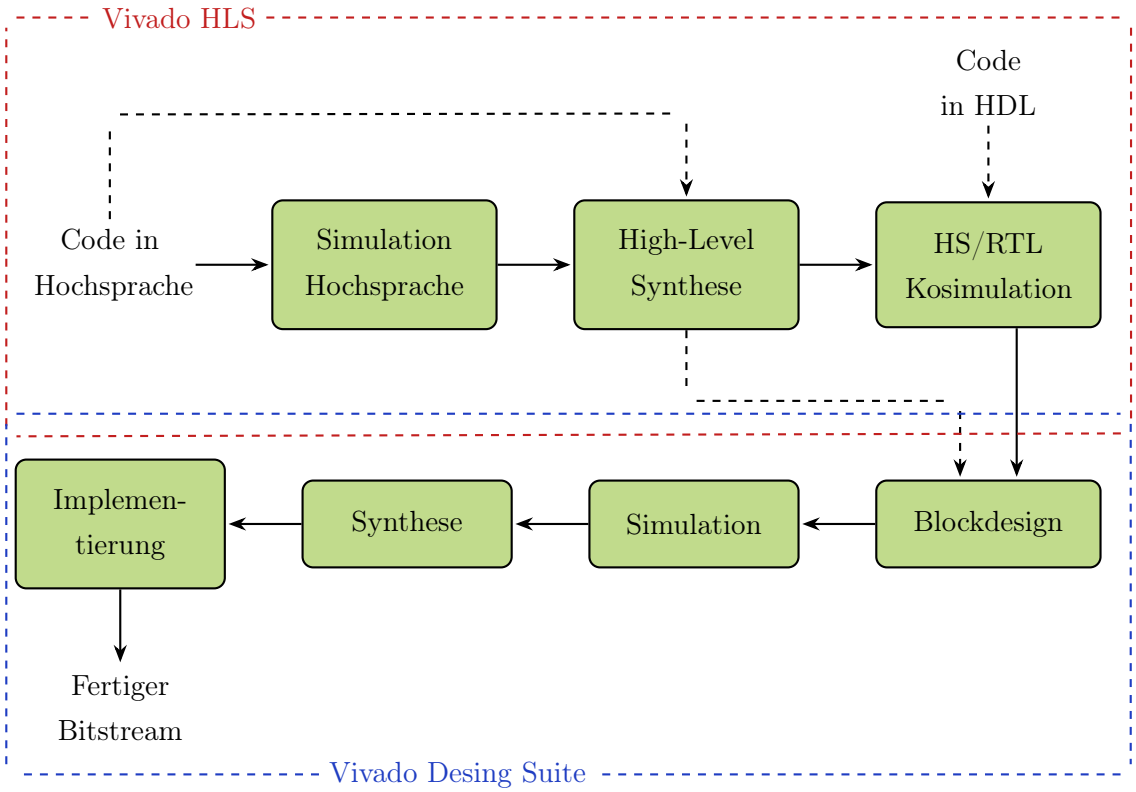

Abbildung 2.3: Konfigurationsablauf eines Xilinx-FPGAs

Historisch bedingt wurde die RTL (Register Transfer Level) für einen Xilinx-FPGA grundsätzlich in einer HDL (Hardware Definition Language) programmiert. Erst seit 2012 gibt es die Tools Vivado Design Suite und Vivado HLS mit denen zusätzlich eine Programmierung in einer Hochsprache, bei Xilinx sind C/C++, möglich ist. Somit ist diese Möglichkeit noch relativ neu und nicht so weit verbreitet wie die Benutzung von HDLs [10]. Auch wenn der direkte Ansatz zu effizienteren Designs führen kann ist er doch mit erheblichem Mehraufwand verbunden. Vor allem der geringere Programmieraufwand in C++ gegenüber einem nicht signifikantem Leistungsverlust ist ausschlaggebend dafür, dass dieser Ansatz in der Arbeit nicht weiter behandelt wird, jedoch einen Ausblick auf weitere Verbesserungsmöglichkeiten bietet.

In dem von Xilinx für die High-Level Synthese vorgesehenem Workflow programmiert man nun zunächst die geplante Anwendung/Funktion in einer beliebigen Hochsprache, zum Beispiel BSV in Bluespec oder MaxJ in MaxCompiler. Die am häufigsten verwendete Sprache ist jedoch C oder C++, wie auch in diesem Fall mit Vivado HLS[23].

Dann folgt eine Simulation des Programms um dessen Tauglichkeit für eine FPGA Konfiguration zu prüfen. Hierbei wird die Funktionalität des eingegebenen Codes getestet, noch

bevor es in die HDL übersetzt wird. Dieser Schritt ist optional, jedoch sehr hilfreich, denn die High-Level Synthese von Vivado HLS nimmt sowohl Zeit als auch Ressourcen auf dem Hostrechner in Anspruch und unterstützt zudem nicht alle Besonderheiten und Datentypen von C++. Es können zum Beispiel keine Arrays mit variabler (zur Laufzeit definierter) Länge instanziiert oder rekursive Funktionen verwendet werden. Außerdem werden Anfragen an das System nicht unterstützt und die Hauptfunktion muss die gesamte Funktionalität des Designs enthalten [35].
Nun wird mit der High-Level Synthese ein Programm in einer HDL, in diesem Fall VHDL oder Verilog, erstellt.
Man kann nun die Hochsprache und die RTL nebeneinander (ko-) simulieren, um das Verhalten der HLS zu verifizieren. Auch dieser Schritt ist optional und dient der frühen Fehlerfindung. Er führt eine erneute Kontrolle der Funktionalität durch um auszuschließen, dass bei der Übersetzung unerwünschte oder ungeplante Verhaltensweisen auftreten.
Nach diesem Schritt wird das RTL-Design als IP (Intellectual Property) exportiert und mit der Vivado Design Suite von Xilinx weiter bearbeitet. Zusammen mit IP-Blöcken von Xillinx und Drittanbietern erstellt man nun ein funktionstüchtiges Design, indem man die Ein- und Ausgänge der Blöcke sinnvoll miteinander verknüpft.
Das erstellte Projekt geht jetzt in den Simulaionsschritt, bei dem das echte Verhalten des FPGA emuliert werden soll. Auch diese Simulation überprüft die Funktionalität des Designs, da durch das eventuelle Anfügen weiterer IP Blöcke an das selbst geschriebene Programm und das Verknüpfen der I/O-Schnittstellen mit den simulierten Hardwarekomponenten eines FPGA unbeabsichtigtes Fehlverhalten der Konfiguration auftreten können.

Im darauffolgenden Syntheseschritt wird durch die Software ein Schaltplan der Funktion erstellt, der die Hardwareprogrammierung auf einem theoretischen FPGA (mit unbegrenzten Hardwarebausteinen) darstellt. Hierbei werden erste Berichte zu der Ressourcenauslastung, dem Timing und dem Energieverbrauch erstellt. Diese sind allerdings nur Schätzungen und dienen dem Auffinden von groben Fehlern, zum Beispiel wenn mehr Ressourcen verbraucht werden würden als das FPGA hat.
Im Implementierungsschritt werden nun der vorhandene Netzplan auf das spezifizierte FPGA angewendet und konkrete Vernetzungen errechnet. Die dabei erstellten Berichte sind nun genau, sodass etwaige Fehler nun korrekt behoben werden können. Man kann nun auch einen Schaltplan des FPGA einsehen, in dem alle tatsächlich verwendeten Bausteine markiert sind. Aus dem implementierten Design kann nun der Bitstream erstellt werden, mit dem der FPGA dann programmiert wird.

2.3 Verwendete Hardware

Das in dieser Arbeit verwendete Board ist ein AC701 Evaluation Kit der Firma Xilinx Inc. Darauf enthalten ist ein FPGA der Serie Artix-7, genauer XC7A200T-2FBG676C. Dieses enthält 215.360 Logikzellen, 740 DSP48E1 (Digital Signal Processor) Slices, 13.140 Kb Block RAM, 33.650 CLB Slices und 500 I/O Pins [34].
Des weiteren sind Auf dem Board unter Anderem 1GB DDR3 RAM Speicher, 256 Mb Flash Speicher, ein SD (Secure Digital) Connector, Mehrere Clock Generatoren (zum Beispiel ein Fixed 200 MHz LVDS oscillator), Status LEDs und konfigurierbare Schalter verbaut.
Als Kommunikationsschnittstellen stehen jeweils eine Gen1 4-Lane (x4) und eine Gen2 4-Lane (x4) PCI Express Schnittstelle, ein SFP+ (Enhanced Small Form-factor Pluggable) Connector, ein HDMI (High Definition Multimedia Interface) Ausgang, UART (USB zu Universal Asynchronous Receiver Transmitter) Brücke und eine 10/100/1000 MBit/s tri-speed Ethernet PHY (Physikalische Schnittstelle) zur Verfügung[36].
Eingebaut ist das Board in einen Dektop-PC einem Intel Xeon W3565 Prozessor und 24 GB DDR3 RAM, welcher unter Ubuntu 14.04.5 LTS 64-Bit betrieben wird. Da Board ist über die UART Schnittstelle mit einem USB-Ausgang dieses Rechners verbunden und wird darüber konfiguriert. Das Erstellen der Software und der Bitstreams erfolg über einen Desktop PC mit einer AMD Phenom^{TM} II X4 960T CPU und 8 GB DDR3 RAM.

2.4 Verwendete Software

Für die High Level Synthese und die Generierung des Bitstreams werden Vivado HLS und die Vivado Design Suite von Xilinx verwendet. Die Kommunikation mit dem FPGA und dem Host-Rechner erfolgt über PCIe mit einem IP-Core von Xillybus [37]. Die Arbeit baut in dieser Hinsicht auf „Umsetzung einer High-Performance FPGA-Schnittstelle für maschinelles Lernen“ von Dillkötter[8] auf. Die entworfenen Bitstreams werden über den Hardaware-Manager von Xilinx auf das FPGA geladen, sodass mit dem Hostrechner nur über SSH (Secure Shell) gearbeitet wird.

Kapitel 3

Logistische Regression

Dieses Kapitel befasst sich mit der Logistischen Regression. Zunächst wird die Definition und Funktion der Logistischen Regression erklärt. Danach wird die Geschichte und Entwicklung der Methode erörtert. Des weiteren gibt es eine Vertiefung der verschiedenen Regularisierungsmethoden und zum Abschluss die Grenzen der Funktion sowie einen Ausblick auf verwandte Algorithmen.

3.1 Definition und Funktion

Die logistische Regression ist ein statistisches Analyseverfahren, bei dem es darum geht, eine Beziehung zwischen einer abhängigen und mehrerer unabhängiger Variablen zu modellieren und wird auch als binäres Logit-Modell bezeichnet. Sie gehört zur Klasse der strukturen-prüfenden Verfahren und bildet eine Variation der Regressionsanalyse [5]. die Art der abhängigen Variable, bezeichnet mit Y, ist als kategoriale Variable klassifizierbar [28]. Die Ausprägungen der der abhängigen Variablen repräsentieren die verschiedenen Alternativen, in unserem binären (oder auch dichotomen) Fall ist „trifft zu“ und „trifft nicht zu“. Sie werden deshalb mit den Kategorien 0 beziehungsweise 1 beschrieben, sodass die Vorhersage des Modells die Wahrscheinlichkeit beschreibt, mit der die abhängige Variable den Wert 1 annimmt, formal $P(Y_i = 1)$ [5]. Für die Y Variable gilt nun:

$$P(Y = 0) = 1 - P(Y = 1) \text{ und } P(Y = 1) = 1 - P(Y = 0)$$

Ziel der Logistischen Regression ist es, gegeben Trainingsdaten $D = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)\}$ mit $n \in \mathbb{N}$, $i \in [1 \ldots n]$, $\vec{x}_i \in \mathbb{R}^d$ und $y_i \in \{0, 1\}$, ein Modell $f_\beta(x_1, \ldots, x_d)$ für Vorhersagen finden, welches auf neuen, ungesehenen Daten einen möglichst kleinen Fehler macht. Ausdrücken lässt sich das logistische Regressionsmodell nun wie folgt:

$$\pi(\vec{x}_i) = f_\beta(x_{i1}, .., x_{in})$$

Wobei $\pi(\vec{x}_i) = P(Y = 1|\vec{x}_i)$ die bedingte Wahrscheinlichkeit, unter der das Ereignis 1 („trifft zu“) mit den gegebenen Werten $\vec{x}_i = (x_{i1}, ..., x_{id})^T$ eintritt, angibt.

Wie auch bei der Linearen Regression werden hierbei die unabhängigen Variablen linear miteinander kombiniert. Die sogenannte systematische Komponente des Modells wird durch die Linearkombination

$$z(\vec{x}_i) = \beta_0 + \sum_{j=1}^{d} \beta_j \cdot x_{ij} + r_i$$

beschrieben. β stellt hier den Vektor der Koeffizienten $(\beta_1, ..., \beta_d)^T$ dar und β_0 ist der Bias. r_i ist ein Störterm. Dieser, unter der Annahme dass der Fehler logistisch verteilt ist, kann vernachlässigt werden, da er keinen Einfluss auf die Gesamtverteilung der Funktion hat [28]. Um das Modell der logistischen Regression zu benutzen wird hier, wie der Name bereits sagt, die logistische Funktion

$$p = \frac{\exp(x)}{1 + \exp(x)} = \frac{1}{1 + \exp(-x)}$$

verwendet [5]. In Abbildung 3.1 sieht man den s-förmigen Verlauf der Funktion. Dieser Verlauf, als Verteilungsfunktion interpretiert, approximiert die Verteilungsfunktion der Normalverteilung mit ausreichender Genauigkeit. Somit kann sie verwendet werden um reellwertige Variablen (im Wertebereich $[-\infty, +\infty]$) auf eine Wahrscheinlichkeit (im Wertebereich $[0, 1]$) zu transformieren, denn die Verteilungsfunktion der Normalverteilung ist nur als Integral auszudrücken und damit schwer zu berechnen [1][5].

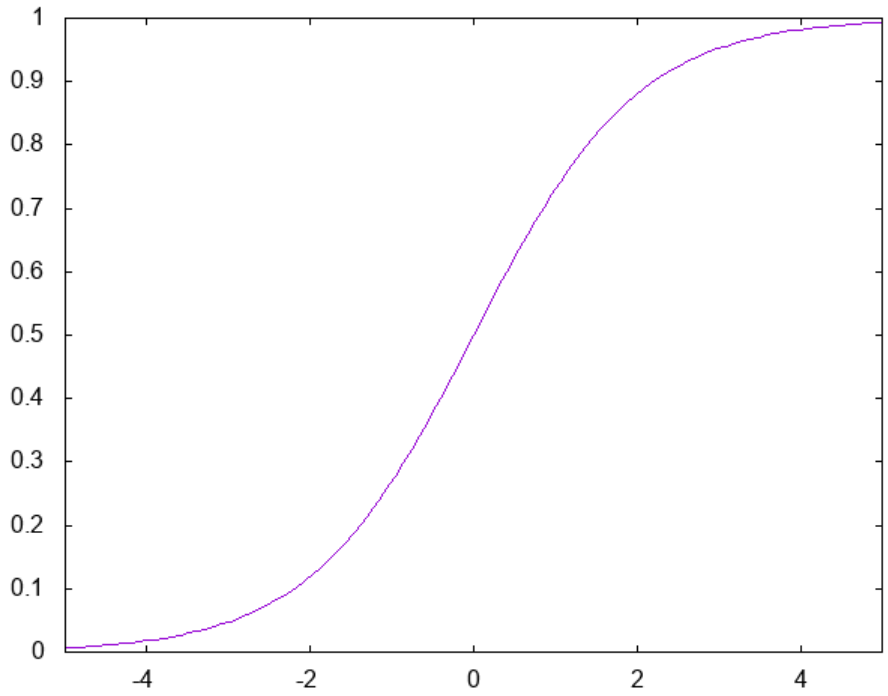

Abbildung 3.1: Die logistische Funktion $p = \frac{1}{1 + \exp(-x)}$

Aufgrund der binären Art der Zielwerte ist es erstrebenswert dass die Ausgabewerte der

Regressionsfunktion gegen 0 beziehungsweise 1 konvergieren. Optimal wäre also ein Funktionsverlauf von $f_\beta(\vec{x}_i)$ der dem Verlauf der Vorzeichenfunktion möglichst ähnlich ist (Siehe Abbildung 3.2), solange diese auf einen Wertebereich von $[0, 1]$ transformiert wäre. Die $\text{sign}(z(\vec{x}_i))$ selbst ist für die logistische Regression allerdings ungeeignet, da sie nicht stetig und damit nicht differenzierbar ist. Die Differenzierbarkeit ist jedoch eine Voraussetzung für die Optimierung der Funktion, was im nachfolgendem Abschnitt 3.2.2 näher erörtert wird. Zusätzlich zu den anderen Vorteilen der logistischen Funktion ist ihre Ähnlichkeit zur Vorzeichenfunktion ein Grund sie für dichotome Entscheidungsprobleme zu verwenden.

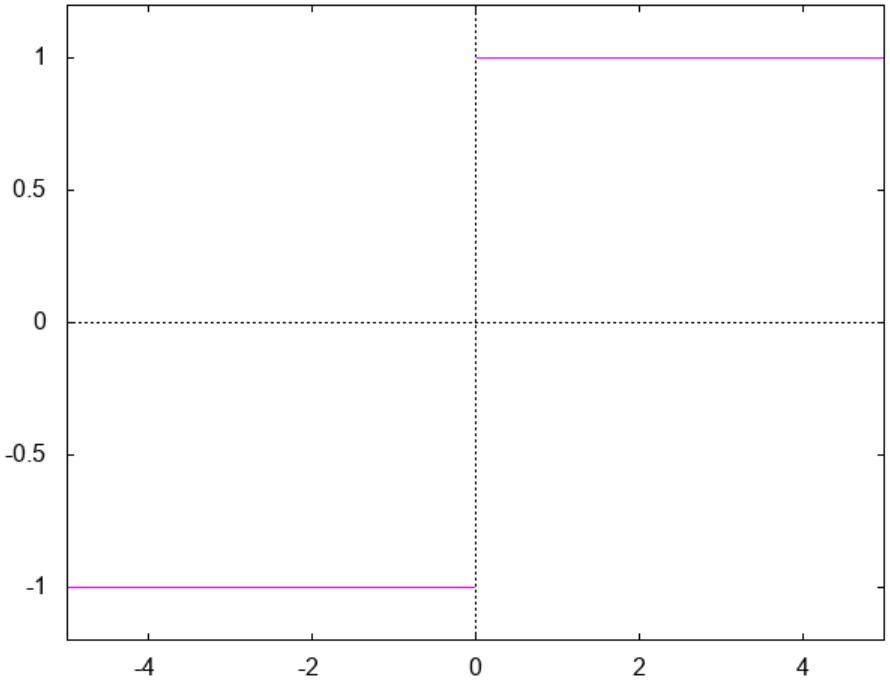

Abbildung 3.2: Die Vorzeichenfunktion $q = \text{sign}(x)$

Wenn man jetzt also die Transformation der systematischen Komponente mit der logistischen Funktion durchführt erhält man die logistische Regressionsfunktion:

$$\pi(\vec{x}_i) = \frac{1}{1 + \exp(z(\vec{x}_i))}$$

Also genauer:

$$P(Y = 1|X = \vec{x}_i) = P(Y_i = 1) = \frac{\exp(\beta_0 + \vec{x}_i^T\beta)}{1 + \exp(\beta_0 + \vec{x}_i^T\beta)} = \frac{1}{1 + \exp(-(\beta_0 + \vec{x}_i^T\beta))}$$

Die systemische Komponente $z(\vec{x}_i) = \beta_0 + \vec{x}_i^T\beta$ ist ein Prädiktor für $\pi(\vec{x}_i)$. Je größer $z(\vec{x}_i)$, desto größer auch $\pi(\vec{x}_i)$ und damit auch $P(Y = 1|\vec{x}_i)$ [5].

3.2 Lernen mit Logistischer Regression

In diesem Abschnitt wird erklärt, wie die Regressionsfunktion dazu verwendet werden kann um mit einem Datensatz zu trainieren. Des weiteren werden verschieden Methoden vorgestellt, mit denen die Vorhersagegenauigkeit weiter verbessert werden kann.

3.2.1 Kostenfunktion und Maximum Likelihood

Da das logistische Regressionsmodell dazu verwendet werden soll anhand gelernter Trainingsdaten Voraussagen für weitere, unbekannte Daten zu berechnen, ist es notwendig alle unbekannten Variablen entsprechend dem vorliegenden Datensatz anzupassen. Um mit dem Modell möglichst korrekte Voraussagen treffen zu können müssen hierfür der Vektor der Koeffizienten β und der Bias β_0 geschickt gewählt werden. Die Koeffizienten geben dabei die Bedeutsamkeit der einzelnen Ausprägungen der Variablen X an, je größer $|\beta_i|, i \in 1..d$ desto größer sind auch die Auswirkungen der jeweiligen Ausprägung auf die Entscheidung [21]. In der linearen Regression wird hierfür häufig die „Methode der kleinsten Quadrate“

$$\sum_{i=1}^{n}(\pi(\vec{x}_i) - y_i)^2$$

gewählt. Es werden die Werte für β gewählt, welche möglichst kleine quadrierte Fehler machen. Somit ergibt sich die Minimierungsfunktion:

$$\min_{\beta} \sum_{i_1}^{n}(\pi(\vec{x}_i) - y_i)^2$$

Unter den üblichen Annahmen der linearen Regression ist die „Summe der kleinsten Quadrate“ eine gute Schätzfunktion mit brauchbaren statistischen Eigenschaften [2]. Unter den Annahmen der logistischen Regression verliert diese Methode jedoch diese Eigenschaften. Die am häufigsten gewählte Methode die „Summe der kleinsten Quadrate“ zu berechnen ist, unter Annahme dass die Fehlerterme normalverteilt sind, die Maximum Likelihood. Diese passt die unbekannten Variablen so an, dass die Chance, mit ihnen die gegebenen Daten darzustellen maximiert wird. Dies bildet auch die Grundlage zur Findung der unbekannten Variablen der logistischen Regression. Um diese Variablen bestimmen zu können bedarf es einer Funktion, der sogenannten Maximum Likeliehood Funktion. Sie drückt die Wahrscheinlichkeit aus in wie weit die gegebenen Daten die Variablen als Funktion darstellen. Die Maximum Likelihood Schätzer der Parameter sind dabei die Werte welche diese Funktion maximieren [2]. Aus der Maximum Likelihood Funktion ergeben sich zwei Kostenfunktionen, eine für den Fall $y_i = 1$ und eine für den Fall $y_i = 0$.

$$cost(\pi(\vec{x}_i), y_i) = \begin{cases} -\log(\pi(\vec{x}_i)) & \text{wenn } y_i = 1 \\ -\log(1 - \pi(\vec{x}_i)) & \text{wenn } y_i = 0 \end{cases}$$

Da y_i immer entweder 0 oder 1 ist, kann man die Kostenfunktionen auch wie folgt zusammenfassen:

$$cost(\pi(\vec{x}_i), y_i) = y_i \cdot -\log(\pi(\vec{x}_i)) + (1 - y_i) \cdot -\log(1 - \pi(\vec{x}_i))$$

Die Kostenfunktion über alle Eingaben lautet damit [11]:

$$J(\beta) = \frac{1}{n} \cdot \sum_{i=1}^{n} cost(\pi(\vec{x}_i), y_i) = \frac{1}{n} \cdot \sum_{i=1}^{n} y_i \cdot -\log(\pi(\vec{x}_i)) + (1 - y_i) \cdot -\log(1 - \pi(\vec{x}_i))$$

3.2.2 Gradientenabstiegsverfahren

Das Gradientenabstiegsverfahren (Stochastic Gradient Descent, kurz SGD) ist ein Lernverfahren für Modelle mit nichtlinearen Parametern im Modellausgang. Es wird in statischen neuronalen Netzen dafür verwendet mittels eines Algorithmus' die Koeffizienten zu adaptieren. Ziel des Lernverfahrens ist es die Koeffizienten so anzupassen, dass die Abweichung zwischen dem y_i Wert der eingegebenen Variable und der Ausgabe $\pi(\vec{x}_i)$ möglichst gering ist. Diese Abweichung bezeichnet man als Ausgangsfehler [29]:

$$e_\beta(\vec{x}_i, y_i) = y_i - \pi(\vec{x}_i)$$

Dieses Optimierungsproblem lässt sich wie folgt iterativ approximieren:

1: Wähle Startpunkt $\beta^{(0)}$
2: Iterationsindex $l \leftarrow 0$
3: **repeat**
4: Bestimme Suchrichtung $s^{(l)}$
5: Bestimme skalare Schrittweite $\eta^{(l)} > 0$
6: **for** $i = 0$ to d **do**
7: Bestimme $\beta_i^{(l+1)} = \beta_i^{(l)} + \eta^{(l)} \cdot s^{(l)}$
8: **end for**
9: $l = l + 1$
10: **until** Abbruchbedingung erfüllt

Abbildung 3.3: Basisalgorithmus für SGD

Die einzelnen Abstiegsverfahren, wie zum Beispiel das Newton-, das Gradienten- oder das Konjugierte Gradientenverfahren unterscheiden sich in ihrer Struktur nur durch die Bestimmung der Suchrichtung $s^{(l)}$[26]. In Abbildung 3.4 ist der Verlauf des SGD für eine zweidimensionale Variable $\vec{x}_i$ zu sehen. Hier bei kann man erkennen dass sich die Länge der Schrittweiten der einzelnen Durchläufe des Algorithmus verkürzen, was auf eine dynamische Lernrate η schließen lässt.

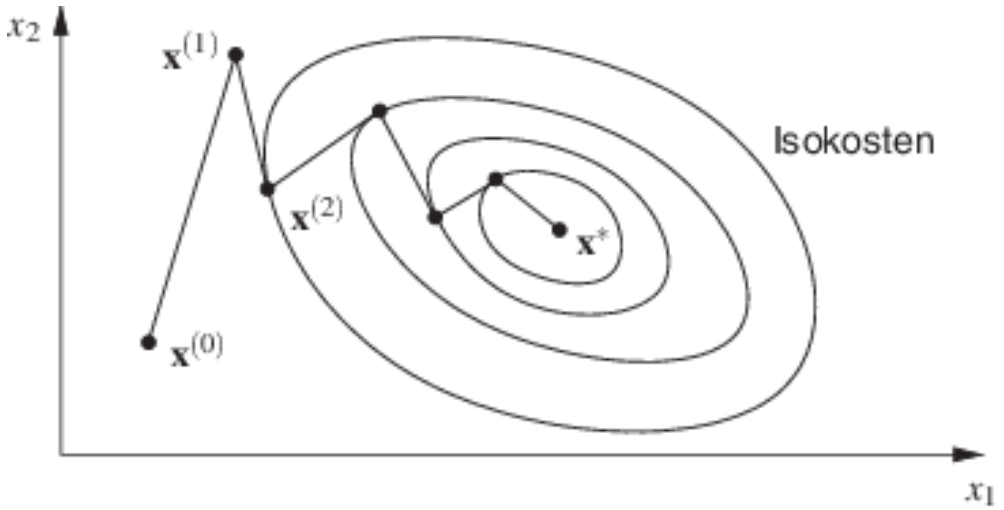

Abbildung 3.4: Beispielhafter Verlauf des Algorithmus im zweidimensionalen Fall, Grafik aus [26]

Voraussetzung für die Abstiegsrichtung ist es, dass für ein genügend kleines $0 < \eta^{(l)} < \tilde{\eta}^{(l)}$ die Gleichung

$$f_\beta(\vec{x} + \eta^{(l)} \cdot s^{(l)}) < f_\beta(\vec{x})$$

erfüllt ist. Eine Hinreichende Bedingung dafür ist, dass die Suchrichtung und der Gradient $g_\beta^{(l)}(\vec{x}) = \bigtriangledown f_\beta^{(l)}(\vec{x})$, also die Orthogonale auf dem Vektor $\vec{x}$, einen stumpfen Winkel zueinander bilden. Es gilt also als Hinreichende Bedingung [26]:

$$s^{(l)^T} \cdot g_\beta^{(l)}(\vec{x}) < 0$$

In Abbildung 3.5 ist die Beziehung zwischen Gradienten, dem Vektor $\vec{x}$ und der Abstiegsrichtung noch einmal anschaulich dargestellt.

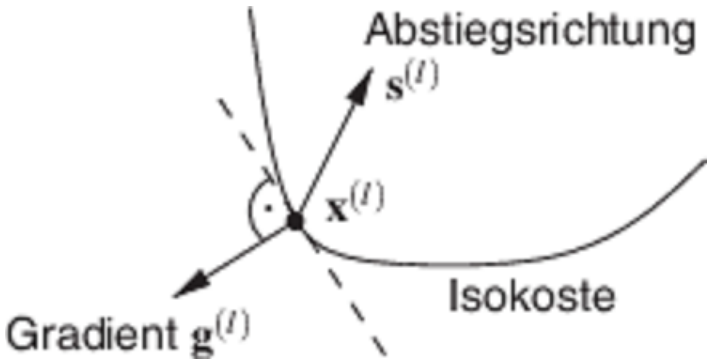

Abbildung 3.5: Beziehung zwischen Gradient und Abstiegsrichtung, Grafik aus [26]

Um nun die Funktion zu minimieren, formal $\min f_\beta(\vec{x})$, kann das mehrdimensionale Problem durch eine Hilfsfunktion

$$F(\varepsilon) = f(\vec{x}^* + e \cdot \sigma)$$

in ein eindimensionales Problem umgewandelt werden. $\vec{x}^*$ bezeichnet hier ein lokales Minimum der Funktion $f_\beta(\vec{x})$, *sigma* ist ein beliebiger Vektor mit der gleichen Dimension wie $\vec{x}^*$ und ε ist die für die Funktion eingeführte Variable. Nun hat die Funktion f_β an der Stelle $\vec{x}^*$ genau dann ein lokales Minimum, wenn $F(\varepsilon)$ für jeden beliebigen Vektor $\sigma \in \mathbb{R}^d$ an der Stelle $\varepsilon^* = 0$ auch ein lokales Minimum aufweist. Dazu muss gelten:

$$F'(0) = \sigma^T \cdot \bigtriangledown f_\beta(\vec{x}^*) = 0$$

was genau dann erfüllt ist, wenn:

$$\bigtriangledown f_\beta(\vec{x}^*) = 0$$

Diese Notwendige Bedingung erster Ordnung ist für alle stationären Punkte, also von allen lokalen Minima, Maxima und Sattelpunkten erfüllt und erlaubt die Definition einer Abbruchbedingung durch das Vorgeben eine Toleranzgrenze $\mu > 0$. Abgebrochen wird der Algorithmus wenn

$$\parallel g_\beta^{(l)} \parallel < \mu$$

Bei dem SGD Verfahren wird als Suchrichtung $s^{(l)}$ der negative Gradient $-g_\beta^{(l)}(\vec{x})$ verwendet. Dadurch erfüllt sich automatisch die Hinreichende Bedingung für alle nichtstationären Punkte sicher, denn [26]:

$$s^{(l)^T} \cdot g_\beta^{(l)} = - \parallel g_\beta^{(l)} \parallel^2 < 0$$

3.2.3 Zusammenführen der Funktionen

Das Ziel der logistischen Regression mit SGD ist es, die Kostenfunktion $J(\beta)$ zu minimieren. Aus den Erkenntnissen in Abschnitt 3.2.2 lässt sich entnehmen, dass für die Anpassung der Koeffizienten $\beta^{(l+1)}$ der negative Gradient verwendet wird. Dieser berechnet sich nach 3.2.1 durch die Partielle Ableitung der Kostenfunktion nach β_i für alle $\vec{x}_i, i \in [1, n]$:

$$\frac{\partial}{\partial \beta_i} J(\beta) = -\frac{1}{n} \sum_{i=1}^{n} (\pi(\vec{x}_i) - y_i) \cdot \vec{x}_i$$

beziehungsweise für den direkten Fall β_i mit $\vec{x}_i$:

$$\frac{\partial}{\partial \beta_i} cost(\pi(\vec{x}_i), y_i) = (y_i - \pi(\vec{x}_i)) \cdot \vec{x}_i$$

Somit werden die Koeffizienten nach mit allen Variablen $\vec{x}_i, i \in [1, n]$ durch folgenden Term angepasst [11]:

$$\beta^{(l+1)} = \beta^{(l)} + \eta^{(l)} \cdot \frac{1}{n} \sum_{i=1}^{n} (y_i - \pi(\vec{x}_i)) \cdot \vec{x}_i$$

Hierbei kann n auch durch eine beliebige Variable m . $0 < m < n$ ersetzt werden, wodurch eine Batch Anpassung, also eine Anpassung der Koeffizienten nach m Schritten durch das Mittel der Kostenfunktionen entsteht:

$$\beta^{(l+1)} = \beta^{(l)} + \eta^{(l)} \cdot \frac{1}{m} \sum_{i=l \cdot m+1}^{(l+1) \cdot m} (y_i - \pi(\vec{x}_i)) \cdot \vec{x}_i$$

Für den Durchlauf mit einem Update nach jeder Variable $\vec{x}_i$ (Batchsize = 1) gilt [7]:

$$\beta^{(l+1)} = \beta^{(l)} + \eta^{(l)} \cdot (y_i - \pi(\vec{x}_i)) \cdot \vec{x}_i$$

Die Anpassung der Koeffizienten kann durch die Erweiterung der Kostenfunktion noch weiter verbessert werden, um beispielsweise einer Überanpassung der Koeffizienten entgegenzuwirken:

$$\min_{\beta} \left(\frac{1}{n} \sum_{i=1}^{n} (y_i - \pi(\vec{x}_i)) \cdot \vec{x}_i + C \cdot R(\beta) \right)$$

Wobei $C \in \mathbb{R}$ ein Hyperparameter zur Gewichtung ist, der fest gewählt werden muss. Dies geschieht zumeist durch das Testen verschiedener Werte, sodass hier ein Ansatz zur Parallelisierung entsteht. $R(\beta)$ beschreibt einen Regularisierungsterm, auf den im Kapitel 3.3 näher eingegangen wird.

3.3 Regularisierungsmethoden

Regularisierung ist jede Modifikation die an einem Lernalgorithmus vorgenommen wird um den Generalisierungsfehler, jedoch nicht seinen Trainingsfehler zu reduzieren [6].
Wenn ein Lernalgorithmus mit logistischer Regression mit einem Datensatz trainiert wird, kann dieser auf den gegebenen Daten ausreichend gute Voraussagen treffen [24]. Es kann jedoch passieren, dass die Häufigkeit von richtigen Voraussagen auf neuen, von dem Algorithmus nicht gelernten Daten drastisch sinken. Man bezeichnet das Modell dann als „overfitted“ (überangepasst). Das bedeutet, dass der Algorithmus zwar die vorgegebenen Trainingsdaten für Voraussagen gelernt hat, aber von dem gelernten Modell nicht generalisieren kann. Ein Beispiel dafür ist in Abbildung 3.6 und 3.7 zu sehen.

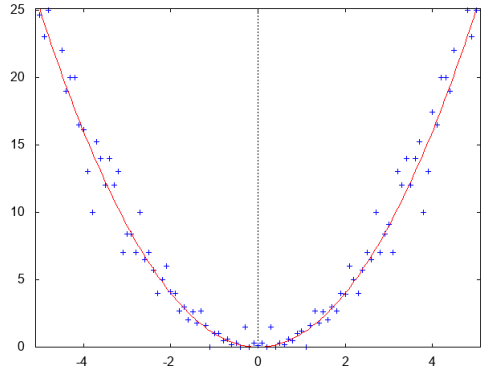

Abbildung 3.6: Normale Regressionsgrade

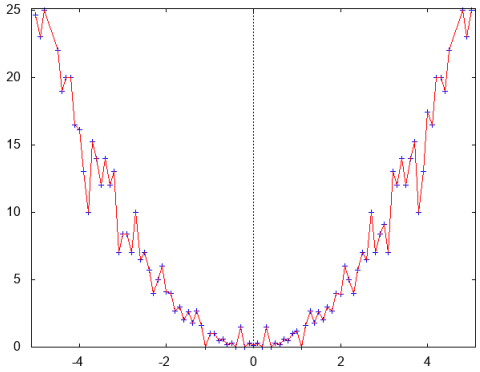

Abbildung 3.7: Overfitting des Modells

Um besagtes Overfitting zu vermeiden können zum einen möglichst repräsentative Trainingsdatensätze gewählt werden, zum anderen kann die Minimierungsfunktion mit einem

Regularisierungsterm erweitert werden. Die Regularisierung soll zu hoch gewichtete Koeffizienten bestrafen oder nicht die Gewichtung für unwichtige Koeffizienten reduzieren. Hierfür werden nachfolgend zwei Methoden vorgestellt.

3.3.1 LASSO

L1-Regularisierung (Least Absolute Shrinkage and Selection Operator, kurz **LASSO**) ist die l1 Norm der Koeffizienten, definiert als lineare Funktion durch [30]:

$$R(\beta) = \| \beta \|_1 = \sum_{i}^{n} |\beta_i|$$

Man nennt die l1 Norm auch die Manhattan Distanz, also die Distanz die benötigt wird um von einem Punkt zu einem anderen zu gelangen, nur mit Laufrichtungen parallel zu einer der Achsen des Koordinatensystems. L1-Regularisierung ist schwierig zu optimieren, da die Betragsfunktion an der Stelle 0 nicht differenzierbar ist. Eine Regularisierung mit LASSO führt zu einem spärlichen Koeffizientenvektor mit einigen großen und vielen kleinen Gewichten (= 0), so das wenige Features gewählt werden. Die Verteilung der Gewichte entspricht somit einer Laplace-Verteilung [15]. In Abbildung 3.x wird eine Regularisierung mit L1 dargestellt.

3.3.2 Ridge Regression

L2-Regularisierung (Ridge Regression) ist die quadrierte l2 Norm der Koeffizienten, definiert durch:

$$R(\beta) = \| \beta \|_2^2 = \frac{1}{2} \sum_{i}^{n} \beta_i^2$$

Die Konstante $\frac{1}{2}$ dient hier der einfacheren Berechnung der Ableitung für die Optimierung und wird über die Anpassung von C kompensiert. L2 entspricht der Euklidischen Distanz des Koeffizientenvektors zum Ursprung. Regularisierung mit Ridge Regression führt zu deutlich kleineren Koeffizienten und damit zu weniger überangepassten einzelnen Features. Die Verteilung des Koeffizientenvektors entspricht einer Gauß'schen Verteilung mit Median $\mu = 0$ und Varianz $2\sigma^2 = 1$ [15]. Sei

$$\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\beta_j - \mu_j)^2}{2\sigma_j^2}\right)$$

die Gauß-Verteilung des Koeffizienten β_j. Wenn man nun jeden Koeffizienten mit der Gauß'schen Verteilung der Koeffizienten multipliziert, erhält man folgenden Term:

$$\tilde{\beta} = \max_{\beta} \prod_{i=1}^{n} P(y_i|\vec{x}_i) \cdot \prod_{j=1}^{d} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\beta_j - \mu_j)^2}{2\sigma_j^2}\right)$$

Im logarithmischen Raum, mit $\mu = 0$ und $2\sigma^2 = 1$ verhält dieser sich gleich dem Term:

$$\tilde{\beta} = max_\beta \sum_{i=1}^{n} logP(y_1|\vec{x}_i) - C \cdot \sum_{j=1}^{d} \beta_j^2$$

Welcher äquivalent zu unserer Minimierungsfunktion aus 3.2.3 ist [15].

3.3.3 Zusammenfassung der Regularisierungsmethoden

Sowohl L1- als auch L2 Regularisierung haben beide ihre Vorteile. LASSO ist besonders für Feature Selection geeignet, da da dadurch viele weniger- oder unwichtige Features aussortiert (auf 0 gesetzt) werden. Für Fälle mit vielen, aber wenig relevanten Features ist Fehlklassifizierungsrate deutlich geringer als keine oder L2 Regularisierung auf den selben Daten [24]. Ridge Regression hingegen glättet die Koeffizienten, sodass einzelne Features nicht übermäßig hoch gewichtet werden. Dies erhöht häfig die Generalisierbarkeit des gelernten Modells.
Die folgenden Graphen wurden mit dem „Internet Advertisements Data Set“ von [9] trainiert.

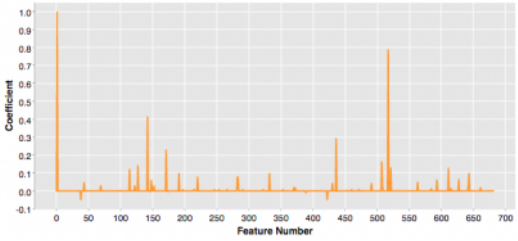

Abbildung 3.8: L1 Regularisierung

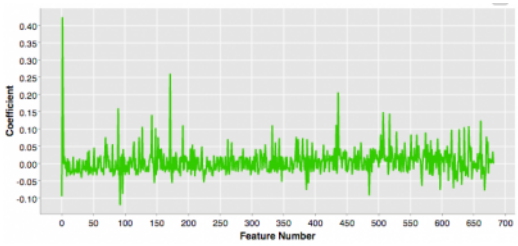

Abbildung 3.9: L2 Regularisierung

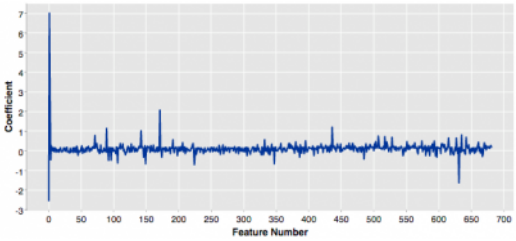

Abbildung 3.10: Logistische Regression ohne Regularisierung

Die Grafiken wurden [16] entnommen. Weitere Tests auf dem MNIST Datensatz [20] werden in Kapitel 5 behandelt.

Kapitel 4

Implementierung

In diesem Kapitel wird zunächst die Implementierung der verschieden Ansätze in C++ behandelt. Hierbei wird auch auf die Besonderheiten des FPGA eingegangen. Des weiteren wird erläutert, wie der FPGA programmiert wird, vor allem in Hinblick auf die Parallelisierung der einzelnen Komponenten. Eine einzelne Instanz der logistischen Regression wird im Folgenden als Perceptron bezeichnet.

4.1 Implementierung in C++

Für die erste Implementierung der Voraussagefunktion wurde der Code von [27] aus Python in C++ übersetzt und an die Eigenschaften eines FPGA angepasst. In Abbildung 4.1 sind die Kernfunktionen des Programmcodes dargestellt. Die Funktion *predict()* liefert die Berechnung der Formel $\frac{1}{1+\exp(-(\beta_0 + x_i^T \beta))}$. Die Koeffizienten β sind hier als Array $coefficients[\,]$ gespeichert, zudem gibt es für die Batch-Realisierung ein Hilfsarray $tmp\_coefficients[\,]$. Der Datentyp $DATA\_TYPE$ kann hier zum einen Als Gleitkommazahl (*float*), zu anderen als Fixkommazahl (*ap_fixed*) deklariert werden. Die Wahl für den Fixkomma-Datentyp fällt auf ein von Xilinx selbst bereitgestelltes Konstrukt *ap_fixed*, da es für das FPGA optimiert wurde und hier eine Vielzahl an Konfigurationen vorgenommen werden können. Die Gesamtanzahl der für eine Instanz belegten Bits wurde auf 16 festgelegt, davon ein Vorzeichenbit und 4 Vorkommastellen. Durch die Einstellung AP_RND_CONV wird die Zahl, zum Beispiel nach einer Divisionsberechnung auf den nächsten repräsentierbaren Wert gerundet. Die Rundungsrichtung ist dabei abhängig von dem am wenigsten signifikanten Bit. Ist dieses gesetzt wird gegen $+\infty$, andernfalls gegen $-\infty$ gerundet.[35] Um einem eventuellen Overflow der Zahl entgegenzuwirken wählt man die Einstellung AP_SAT_SYM. Im Fall eines Positiven Overflows wird hierbei der höchste, bei einem negativen Overflow der kleinste darstellbare Wert gewählt.[35] Die FPGA Einstellung *pragma HLS LOOP FLATTEN* sorgt für eine Parallelisierung der Schleife auf dem FPGA. Das funktioniert allerdings nur, wenn innerhalb der Schleife auf immer andere

Ziele geschrieben wird.
In der Funktion *predict()* zum Beispiel wird die Variable *yhat* immer wieder neu gesetzt, sodass eine Paralleliserung hier nicht möglich ist.

```
/* Voraussage treffen anhand logistischer Regression */
DATA_TYPE predict(){
    DATA_TYPE yhat = coefficients[0];
        for(int i=0; i<FEATURE_COUNT; i++){
                yhat+=coefficients[i+1]*features[i];
        }
        float tmp_yhat=-yhat;
        DATA_TYPE predicted=1.0f/(1.0f+hls::expf(tmp_yhat));
        return predicted;
}
```

```
void learn(){
    DATA_TYPE predicted=predict();
    DATA_TYPE error=label-predicted;
    tmp_coefficients[0]+=error;
        for(int i=0; i<FEATURE_COUNT; i++){
                #pragma HLS LOOP FLATTEN
                tmp_coefficients[i+1]+=error*features[i];
        }
        /* Batch Update der Koeffizienten */
        batch_count++;
        if(batch_count>=BATCH_SIZE){
                batch_count=0;
                coefficients[0]+=lrate*tmp_coefficients[0]/BATCH_SIZE;
                tmp_coefficients[0]=0;
                for(int i=1; i<FEATURE_COUNT+1; i++){
                        #pragma HLS LOOP FLATTEN
                        coefficients[i]=punish(i) +
                                lrate*tmp_coefficients[i]/BATCH_SIZE;
                        tmp_coefficients[0]=0;
                }
        }
}
```

Abbildung 4.1: Codes des Perzeptrons

Die Funktion *hls::expf()* wird von Xilinx geliefert und dient als Realisierung der Expotentialfunktion und ist für FPGAs optimiert.

Um die High-Performance Schnittstelle über PCIe von [8] voll ausnutzen zu können wurde eine Trennung der Codeteile vorgenommen, damit ein Teil des Algorithmus auf dem FPGA und ein Teil auf dem Hostcomputer laufen kann. Der Hostrechner übernimmt hier

die Vorbereitung der Daten und die Äußeren Schleifendurchläufe. Es werden immer die einzelnen Variablen mit dem dazugehörigen Label an den FPGA gesendet, welcher dann je nach Konfiguration damit trainiert oder eine Voraussage trifft. Der Vorteil hierbei ist, dass auf dem FPGA mehrere logistische Regressionen gleichzeitig implementiert sind, die mit verschiedenen Parametern (zum Beispiel einem C für die Fehlerbestrafungsgewichtung oder unterschiedlichen Lernraten) initialisiert wurden.
Die Methode *punish()* wendet die ausgewählte Regularisierungsmethode auf das Batch Update an. Es wurden keine Regularisierung, L1- und L2-Regularisierung implementiert, sodass für den selben Trainingsdurchlauf verschiedene Methoden getestet werden können. in Abbildung 4.2 ist die Funktion als Programmcode dargestellt.

```
/*Punishment sollte vorinitialisiert werden, da Konstant abhaengig von der
    Regularisierungsgewichtung, der Batchgroesse und der Lernrate */

DATA_TYPE punishment=lrate*c/BATCH_SIZE

void punish(int i){
        /*L1 Regularisierung*/
        if(method==1){
                return coefficients[i]-punishment*sign(coefficients[i]);
        }
        /*L2 Regularisierung*/
        if(method==2){
                return coefficients[i]*(1-punishment);
        }
        return coefficients[i];

}
```

Abbildung 4.2: Implementierte Regularisierungsmethoden

Die angewendete L1- und L2-Regularisierung für das Koeffizientenupdate ergibt sich aus der Ableitung der Kostenfunktion, welche um den Regularisierungsterm erweitert wurde. für LASSO ergibt sich der Term aus:

$$\beta_i^{(l+1)} = \beta_i^{(l)} + \eta^{(l)} \cdot \frac{\partial}{\partial \beta_i} \left(J(\beta) - \frac{C}{m} \cdot |\beta_i^{(l)}| \right)$$

Mit $x_i \in \vec{x}_l$. Da die Betragsfunktion nicht differenzierbar ist behilft man sich mit der Vorzeichenfunktion *sign()*. Es entsteht damit folgender Term für das Update [32]:

$$\beta_i^{(l+1)} = \beta_i^{(l)} + \eta^{(l)} \cdot \frac{1}{m} \sum_{i=l \cdot m+1}^{(l+1) \cdot m} (y_i - \pi(\vec{x}_i)) \cdot x_i - \eta^{(l)} \cdot \frac{C}{m} \cdot sign(\beta_i^{(l)})$$

Für die Ridge Regression ist der Ableitungsterm deutlich einfacher zu bestimmen. Aus

$$\beta_i^{(l+1)} = \beta_i^{(l)} + \eta^{(l)} \cdot \frac{\partial}{\partial \beta_i} \left(J(\beta) - \frac{C}{2m} \cdot \sum_{k=1}^{d} (\beta_k^{(l)})^2 \right)$$

wird [12]

$$\beta_i^{(l+1)} = \beta_i^{(l)} \cdot \left(1 - \eta^{(l)} \frac{C}{m} \right) + \eta^{(l)} \cdot \frac{1}{m} \sum_{i=l \cdot m+1}^{(l+1) \cdot m} (y_i - \pi(\vec{x}_i)) \cdot x_i$$

Um die Datenverteilung auf dem FPGA zu gewährleisten wurde sowohl eine Verteiler- als auch eine Sammlerklasse implementiert. Diese sind für dafür zuständig den Datenstrom von Daten und Konfigurationsparametern auf die jeweiligen Perceptrons zu verteilen beziehungsweise von diesen einzusammeln. Die Sammlerklasse markiert zusätzlich noch die ausgegebenen Daten mit der Identifikation des jeweiligen Perceptrons, damit die Ergebnisse nach einem Durchlauf zugeordnet werden können. Die Verteilerklasse übernimmt bei der Implementierung mit der 16-Bit Fixkommazahl zudem die Aufteilung des Eingangstroms, da über die 32-Bit Ausgangsleitung des Xillybus-Blocks jeweils zwei Datenpunkte pro Übertragung versendet werden können.

4.2 Implementierung als Blockdesign

Um Auf dem FPGA zu implementieren, muss der Code für den Verteiler, den Sammler und das Perceptron in eine IP (Intelectual Property) umgewandelt werden. Die IP-Blöcke werden für das Blockdesign wie in Abbildung 4.5 angeordnet und verbunden. Hierbei sind die Perceptrons für Daten aus der MNIST (Modified National Institute of Standards and Technology) Datenbank konfiguriert, das heißt jedes Perceptron hat einen Variablenspeicher von 784 Features (28x28 Pixel) und jeweils 785 Koeffizienten und Hilfskoeffizienten.
Die MNIST Datenbank ist eine Sammlung handgeschriebener Ziffern, komprimiert, zentriert und normalisiert auf 28x28 Pixel mit Werten zwischen 0-256, welche die Farben von Weiß nach Schwarz repräsentieren. Die 60.000 Trainingsvariablen wurden zu jeweils gleichen Teilen aus der NIST (National Institute of Standards and Technology) Testdatenbank und der NIST Trainingsdatenbank entnommen, gleiches gilt für die 10.000 Variablen große Testmenge. Diese Anpassung wurde vorgenommen, da die Variablen der Testmenge unter Mitarbeitern des Volkszählungsamts erhoben wurden und deutlich besser zu klassifizieren sind als die unter Studenten erhobenen Daten der Trainingsmenge. Die Daten der Trainingsmenge wurden von circa 250 Teilnehmern erhoben, von denen keine Einträge in der Testmenge eingefügt wurden [20].

Die Auslastung des FPGA für diese Implementierung beträgt in etwa 50% (siehe Abbildung 4.3), sodass auch 16 Perceptrons parallel geschaltet werden können. Für die bildliche

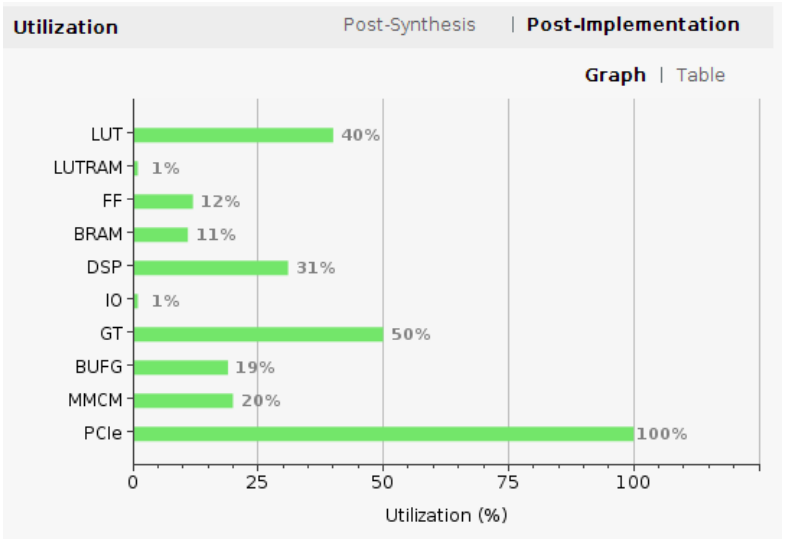

Abbildung 4.3: Auslastung auf dem FPGA mit 16-Bit Fixkommazahl und 8 Perceptrons

Darstellung in dieser Arbeit wäre dies jedoch ein zu großes Diagramm. Die Auslastung für 16 Perceptrons unter MNIST beträgt 90% der LUT und ist in Abbildung 4.4 veranschaulicht.

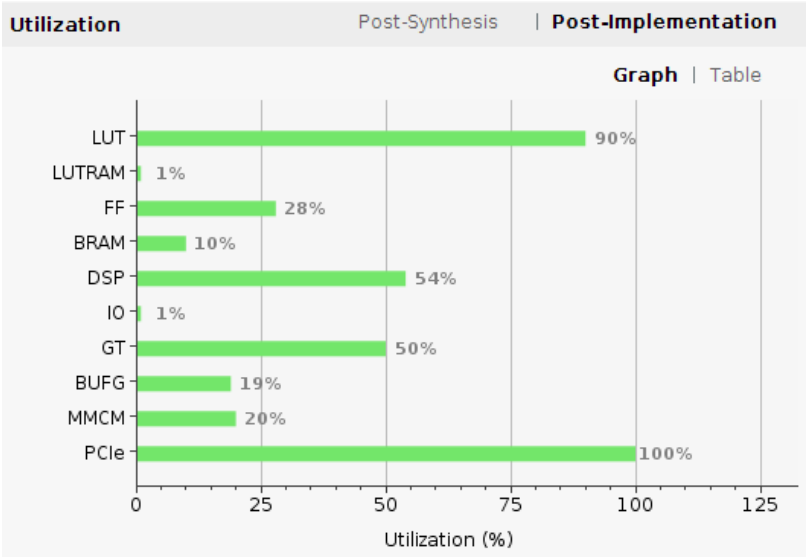

Abbildung 4.4: Auslastung auf dem FPGA mit 16-Bit Fixkommazahl und 16 Perceptrons

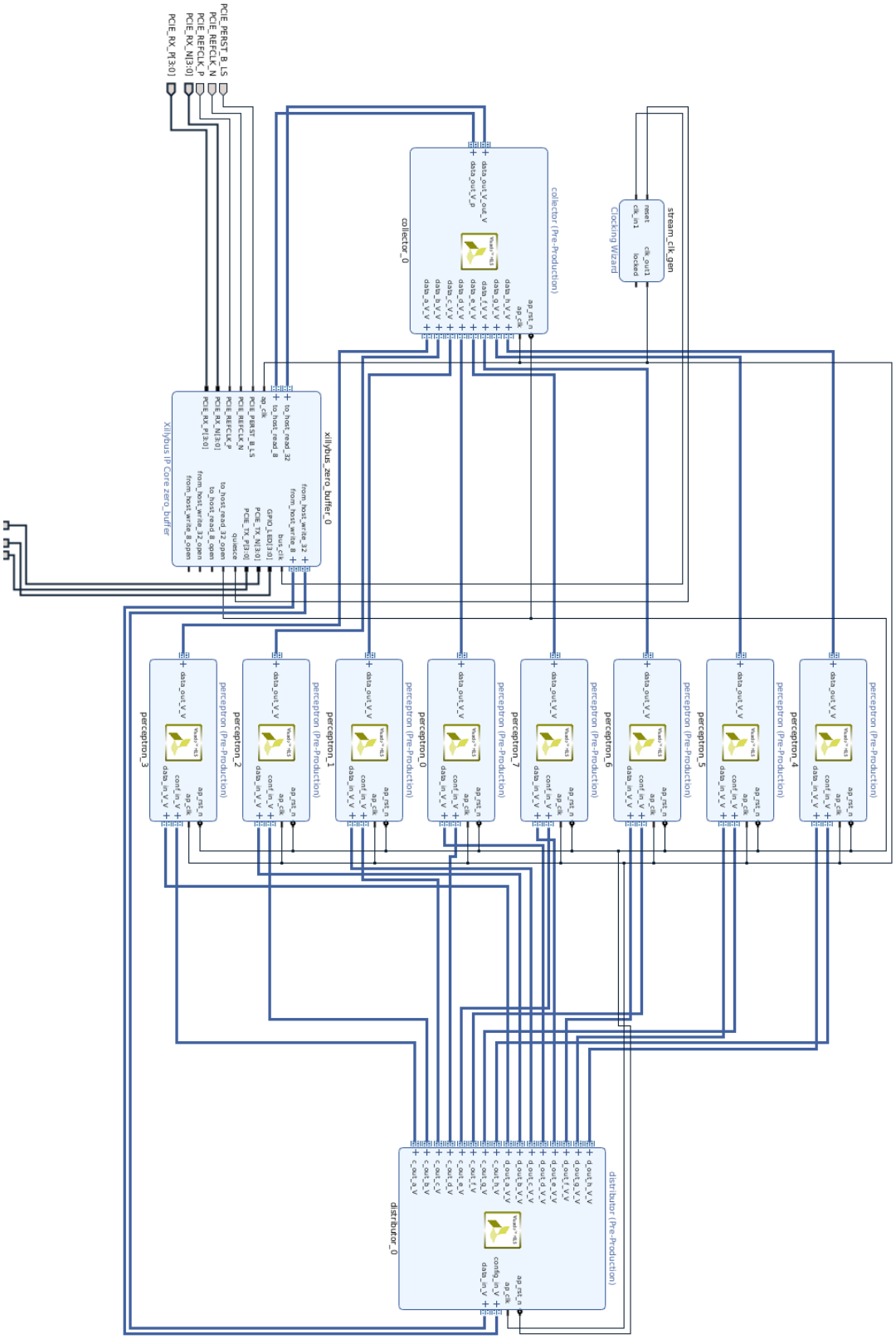

Abbildung 4.5: Das Blockdesign auf dem FPGA

Für Datensätze mit mehr Features als MNIST erhöht sich zudem die Auslastung, da jedes Perceptron zwei Koeffizientenvektoren und einen Zwischenpuffer mit einer Länge gleich der Anzahl der Features im Speicher haben muss. Daher können für diese Datensätze keine 16 oder sogar weniger als 8 Perceptrons in das Design eingefügt werden. Bei mehr als 16 Perceptrons treten zudem einige Probleme bezüglich des Timings auf, welche in Kapitel 5 näher behandelt werden.

4.3 Implementierung der Hostanwendung

Durch die Nutzung der PCIe-Schnittstelle mit Xillybus wie in [8] ausgeführt muss die Hostanwendung einiges mehr leisten als nur den Datenaustausch zwischen Hostrechner und FPGA zu verwalten. Sie dient der Aufbereitung der Daten, der Koordination mit dem FPGA und der Auswertung der Ergebnisse. Es werden zunächst die Eingabedaten in den Speicher geladen und auf Werte zwischen 0 und 1 normalisiert. Dann werden die Werte in das angegebene Zahlenformat konvertiert. Je nach Datensatz ist dies ein nicht unerheblicher Zeitaufwand, der jedoch unvermeidlich ist.
Um den FPGA nun auf das Lernen eines neuen Modells vorzubereiten können jetzt die Umgebungsvariablen gesetzt werden. Jedes Perceptron ist separat konfigurierbar, womit dem Nutzer eine Vielzahl von Testmöglichkeiten eröffnet werden. Regulierbar sind der Hyperparameter C, die Lernrate des Modells, die Batchgröße für den Gradientenabstieg und die Regularisierungsmethode. Alle diese Werte werden in das gewählte Zahlenformat konvertiert, sodass zum Beispiel eine Batchgröße von 5 im Zahlenformat $ap\_fixed < 16, 3 >$ (16 Bit Fixkommazahl mit 3 Vorkommabits, davon ein Vorzeichenbit) nicht angegeben werden kann. für diesen Fall kann die Batchgröße auf 0 gesetzt und das Batchupdate nach der gewünschten Trainingsmenge manuell vom Hostrechner aus initiiert werden. Die hohe Konfigurierbarkeit hat den Vorteil, dass viele verschiedene Einstellungen auf dem FPGA getestet werden können, ohne dass dieser neu programmiert werden muss. Das Erstellen von Pareto-Fronten ist also gut umzusetzen.
Auch die Koeffizienten müssen durch das Hostprogramm vorbelegt werden. Das kann durch einen Nullvektor oder durch Normalverteilte Zufallszahlen geschehen, bietet aber auch die Möglichkeit bereits trainierte Koeffizienten zu verwenden. Durch das Einstellen der Laufvariable auf 0 kann hiermit ein Trainingsvorgang übersprungen werden.
Die Implementierung mit dem Fokus PCIe Schnittstelle erfordert es, dass für jede Operation auf dem FPGA zunächst alle Features einer Variable übertragen werden müssen. Jedes Perceptron auf dem FPGA besitzt einen Pufferspeicher für genau eine Variable, sodass mit dem Übertragen eines Datenpunktes alle 8(16) Perceptrons initialisiert werden können. Dieser Puffer wird sowohl für die Trainings- als auch für die Testdaten verwendet, um möglichst wenig von dem stark begrenzten Speicherplatz des FPGA zu verwenden. Auf-

grund technisch bedingter Schwierigkeiten (in Kapitel 5 werden diese behandelt) Werden die Trainings- und die Testmethode in verschiedenen Prozeduren behandelt.

4.3.1 Training

Das Trainingsprogramm hat die Aufgabe das FPGA auf den Start eines neuen Trainingszyklus' vorzubereiten und diesen dann durchzuführen. Die wichtigsten Programmteile werden in folgendem Codeteil vorgestellt:

```
/*Konfiguriere den FPGA mit Daten aus der Config-Datei*/
write(fc, conf, sizeof(char));
write(fc, NULL,0);
        for(int i =0; i< PERCEPTRONS; i++){
                write(fdw, &hyper[i], sizeof(DATA_TYPE));
                write(fdw, &lrate[i], sizeof(DATA_TYPE));
                write(fdw, &batch[i], sizeof(DATA_TYPE));
                write(fdw, &modus[i], sizeof(DATA_TYPE));
                write(fdw, NULL,0);
        }
/*Normalverteilte Zufallszahlen zur Koeffizienteninitialisierung*/
        for(int i=0; i<((FEATURE_COUNT+1)); i++){
                DATA_TYPE t =nd_random.dev();
                rc = write(fdw, &t, sizeof(DATA_TYPE));
                write(fdw, NULL,0);
        }
```

Abbildung 4.6: Konfiguration des FPGA

Die Filestreams *fc* und *fdw* sind die Outputstreams in die Devicefiles der Xillybus-Schnittstelle. Eine Schreiboperation mit *NULL* und einer Länge von 0 (*write(fc, NULL, 0)*) sorgt hier für das leeren der Internen Puffer, sodass die geschriebenen Daten dem FPGA zur Verfügung stehen. Die Koeffizienten werden durch *nd_random.dev()* normalverteilte Pseudozufallszahlen initialisiert, da man davon ausgeht dass die gelernten Koeffizienten später auch normalverteilt sind.
Der Trainingsvorgang selbst besteht aus einer Doppelschleife über die Epochen und die einzelnen Zyklen. In jeder Epoche werden alle Zyklen exakt gleich durchgeführt. Dies ist über die Deaktivierung der Methode *srand(seed)* abzuschalten, sodass in jeder Epoche ein völlig anderer Trainingszyklus (mit neuen und anders sortierten Daten) ablaufen kann.

```
/*Starte Epochen*/
for(int lauf =0; lauf<max_runs;lauf++){
/*Zufallsgenerator neu seeden*/
        srand(seed);
/*Starte Trainingszyklus*/
        for(int cycle=0; cycle < max_runs; cycle++){
/*Zufaellige Position in der Trainingsmenge bestimmen*/
                int idx = rand() % number_of_labels_train;
                int idx_data= idx*FEATURE_COUNT;
                rc = write(fc, conf+2, sizeof(char));
                write(fc, NULL, 0);

/*Alle Features und das Label versenden*/
                for(int i=0;i<FEATURE_COUNT;i++){
                        DATA_TYPE st = train_set[idx_data+i];
                        rc = write(fdw, &st, sizeof(DATA_TYPE));
                        write(fdw, NULL,0);
                }
                DATA_TYPE st = train_label[idx];
                rc = write(fdw, &st, sizeof(DATA_TYPE));
                write(fdw, NULL,0);
            if (rc <= 0) {
                perror("write() failed labels");
                exit(1);
            }
        }
/*Batch Updaten wenn keine Groesse angegeben ist*/
        if(batch_is_zero){
                for(int i=0; i< PERCEPTRONS; i++){
                        if(batch[i] == zero){
                                rc = write(fc, conf+1, sizeof(char));
                                rc = write(fc, perceptron_names+i, sizeof(
                                    char));
                                write(fc, NULL, 0);
                        }
                }
        }
}
```

Abbildung 4.7: Durchführen der Trainingszyklen

4.3.2 Vorhersagen

Sobald das Modell auf dem FPGA trainiert ist kann mit den Voraussagetests begonnen werden. Es werden dafür alle Variablenpuffer der Perceptrons mit der zu testenden Variable gefüllt. Nun muss für jedes Perceptron der Aufruf zum Vorhersagen erfolgen. Die Lese- und Schreiboperationen können in verschiedenen Threads ausgeführt werden um Zeit zu sparen. Dabei ist es nötig zu wissen in welcher Reihenfolge die Perceptrons ihre Ausgaben an den Hostrechner senden. Durch das einzelne Anstoßen der Vorhersagefunktion ist dies gewährleistet. Zudem ist *read()* eine blockierende Funktion, sodass keine Daten übergangen werden.
Die ausgelesenen Ergebnisse werden in einer Datei abgelegt, ohne diese vorher zu leeren.

```
/*Gemeinsamen Speicher reservieren*/
DATA_TYPE *right = (DATA_TYPE *)mmap(NULL, maxruns*sizeof(DATA_TYPE),
    protection, visibility, -1, 0);
  DATA_TYPE *d_pred = (DATA_TYPE *)mmap(NULL, PERCEPTRONS*maxruns*sizeof(
      DATA_TYPE),protection, visibility, -1, 0);
/*Elternprozess starten*/
  pid_t pid = fork();
  if(pid){
    for(int i =0; i<maxruns; i++){
      int idx_p=i+offset;
      if(offset<0){
        idx_p= rand() % (number_of_labels_test);
      }
      right[i] = test_label[idx_p];
      write(fc, &load, sizeof(char));
      write(fc, NULL, 0);
      for(int j=0; j<MAX_FEATURE_COUNT;j=j+1){
        int idx = (idx_p*MAX_FEATURE_COUNT+j);
        DATA_TYPE st = test_set[idx];
        write(fdw, &st, sizeof(DATA_TYPE));
        write(fdw, NULL, 0);
      }
      for(int j=0;j<PERCEPTRONS; j++){
        write(fc, &test , sizeof(char));
        write(fc, ar+j, sizeof(char));
        write(fc, NULL, 0);
      }
    }
    wait(NULL);
  }
```

Abbildung 4.8: Elternprozess der Trainingsmethode

```
...
  if(pid){
        ...
  }
  else{
    for(int i =0;i<maxruns;i++){
      for(int j=0;j<PERCEPTRONS;j++){
        read(fdr_c, conf, sizeof(char));
        read(fdr_d, d_pred+(j*maxruns+i) ,sizeof(DATA_TYPE));
      }
    }
    _exit(0);
  }
```

Abbildung 4.9: Kindprozess der Trainingsmethode

Zu den extrahierten Ergebnissen gehören die Trefferquoten der Vorhersagen über die eingegebenen Daten, die richtigerweise richtig klassifizierten („true positive“), die fälschlicherweise als richtig bezeichneten („false positiv“), die richtigerweise als falsch klassifizierten („true negative“) und die fälschlicherweise als falsch deklarierten („false negativ“) Daten.

4.3.3 Koeffizienten

Wenn ein Modell soweit trainiert worden ist dass es zufriedenstellende Vorhersagen treffen kann, dann speichert man die Koeffizienten für die weitere Verwendung oder zur Analyse ab. Auf normalen Rechnern kann man sie in Dateien überspielen lassen oder in die Ausgabe leiten. Von einem FPGA müssen die Koeffizienten jedoch erst extrahiert werden um weiter mit ihnen arbeiten zu können. Die Implementierung bietet dazu eine Methode an, die jederzeit den Koeffizientenvektor des gewünschten Perceptrons ausgeben kann. Auch nach dem Schließen der Systemdateien von Xillybus, und damit dem löschen der Puffer bleibt der letzte Zustand des FPGA auf diesem erhalten. Sogar nach einem Neustart des Hostrechners sind die Koeffizienten noch im jeweiligen Perceptron hinterlegt. Demnach kann das Modell beliebig weiter bearbeitet oder zum testen und vorhersagen von Daten benutzt werden, bis es sich durch seine Performance für eine Extraktion qualifiziert.
Der Code des Extraktors ist hier aufgeführt:

```
char *ct_coef = (char*)malloc(sizeof(char)*PERCEPTRONS*(FEATURE_COUNT+1));
DATA_TYPE *dt_coef = (DATA_TYPE*)malloc(sizeof(DATA_TYPE)*(FEATURE_COUNT+1)
    *PERCEPTRONS);

int s_coef = (FEATURE_COUNT+1);
for(int j=0; j<PERCEPTRONS;j++){
  write(fc, &conf, sizeof(char));
  write(fc, perceptron_names+j, sizeof(char));
  write(fc, NULL, 0);
  for(int i=0;i<s_coef;i++){
    read(fdr_c, &config, sizeof(char));
    ct_coef[j*s_coef+i]=config;
    read(fdr_d, &fbuf ,sizeof(DATA_TYPE));
    dt_coef[j*s_coef+i] = fbuf;
  }
}
printf("open datafile");
ofstream dw ("coeff.txt", std::ofstream::trunc);
if (dw.is_open()){
  for(int j=0;j<PERCEPTRONS;j++){
    bool first = true;
    int count_c=0;
    for(int i=0;i<(MAX_FEATURE_COUNT+1)*PERCEPTRONS;i++){
      if(ct_coef[i] == perceptron_names[j]){
        if(first){
          dw << dt_coef[i] << "\n";
          first=false;
        }
        else{
          count_c++;
          if(count_c % 28 == 0 )
            dw << dt_coef[i] << "\n";
          else
            dw << dt_coef[i] << ",";
        }
      }
    }
  }
  dw.close();
}
```

Abbildung 4.10: Extraktion der Koeffizienten

Kapitel 5

Experimente und Ergebnisse

In diesem Kapitel werden verschiedene Experimente mit dem FPGA durchgeführt. Zunächst wird gemessen, Wie viel Zeit pro trainiertem Perzeptron für das Training und die Voraussagen auf dem FPGA und einer Implementierung auf dem Hostsystem aufgewendet werden muss. Hierbei wird das Programm auf dem Hostsystem nicht parallelisiert. Dann werden beispielhaft Trainingsvorgänge mit verschiedenen Datensätzen durchgeführt, um die Genauigkeit der Vorhersagen zu überprüfen und Ergebnisse der Testvorgänge zu präsentieren.
Es werden für die Experimente die Datensätze MNIST [20], und verwendet.
Die Features $\vec{f}$ des jeweiligen Datensatzes wurden normalisiert, so dass $\forall f \in \vec{f} \,.\, f \in [0, 1]$.

5.1 Regularisierung

Mit dem MNIST Datensatz wurden verschiedene Testzyklen zur Regularisierung durchgeführt. Für die Ergebnisse ist der Datentyp, soweit nicht anders vermerkt, auf eine Fixkommazahl mit einem Vorzeichen-, 4 Vorkommastellen- und 11 Nachkommastellenbits eingestellt. Ein Trainingsdurchlauf mit 200 Variablen und 100 Wiederholungen führte zu folgenden Ergebnissen:

Perceptron	Regularisierungsrate	Lernrate	Batch	Methode	Genauigkeit
a	0.0	0.100098	1	keine	95.43%
b	0.007812	0.100098	1	L2	95.75%
c	0.009766	0.100098	1	L2	95.75%
d	0.5	0.100098	1	L2	89.9%
e	0	0.100098	1	L1	95.43%
f	0.000977	0.100098	1	L1	95.43%
g	0.004883	0.100098	1	L1	94.38%
h	0.009766	0.100098	1	L1	93.38%

Hierbei wurde der Datensatz so angepasst, dass alle Variablen die die Ziffer „3“ darstellen mit „trifft zu“ und alle anderen Variablen mit „trifft nicht zu“ markiert wurden.
Nach dem Auswerten der Koeffizienten sind die Einflüsse der Regularisierungsmethoden auf das gelernte Modell deutlich sichtbar. Die Ridge Regression erzielte bei diesem Versuch die besten Ergebnisse, was aufgrund der Daten nicht besonders überraschend war. Da MNIST aus 28x28 Pixeln in Schwarz-Weiß Werten besteht und diese eine handgeschriebene Ziffer repräsentieren war anzunehmen dass es nur wenige Features gibt welche für das Ergebnis irrelevante Ausprägungen bieten. Demnach ist die LASSO Methode für diesen Datensatz nicht gut geeignet. Die Ridge Regression kann, mit ausreichend kleinem C, die Koeffizienten, die zu starke Auswirkungen auf das Modell haben reduzieren, sodass einem Overfitting entgegengewirkt wird. Allerdings hat der gewonnene Genauigkeitswert nur geringfügig zugenommen.
Zeichnet man die Koeffizienten als Graphen auf kann man die Auswirkung der Regularisierung auf die Regression deutlich erkennen. In den nachfolgenden Abbildungen 5.1, 5.2 und 5.3 werden diese Graphen dargestellt. Die Koeffizienten sind hierbei die Ergebnisse der obigen Trainingsdurchlaufs.

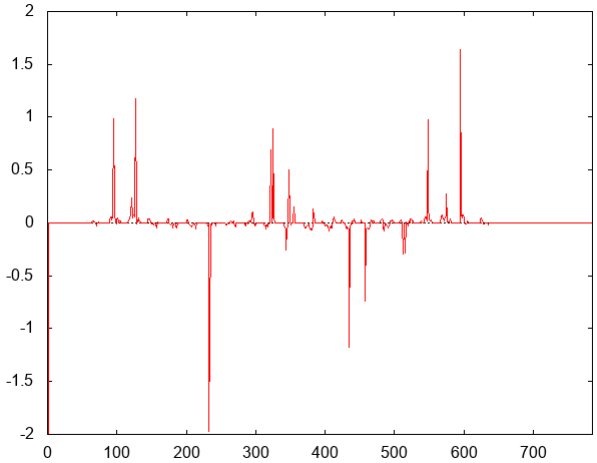

Abbildung 5.1: L1 Regularisierung

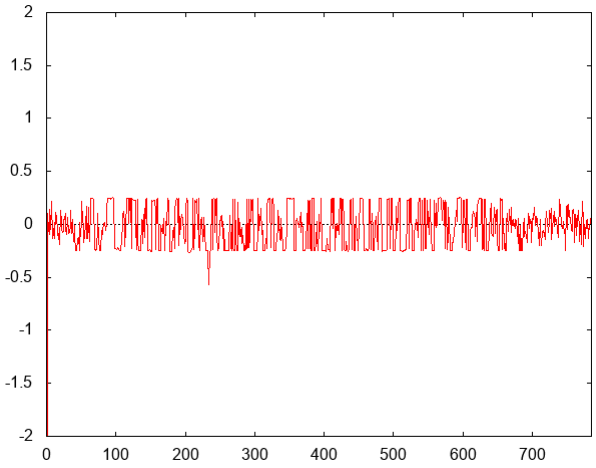

Abbildung 5.2: L2 Regularisierung

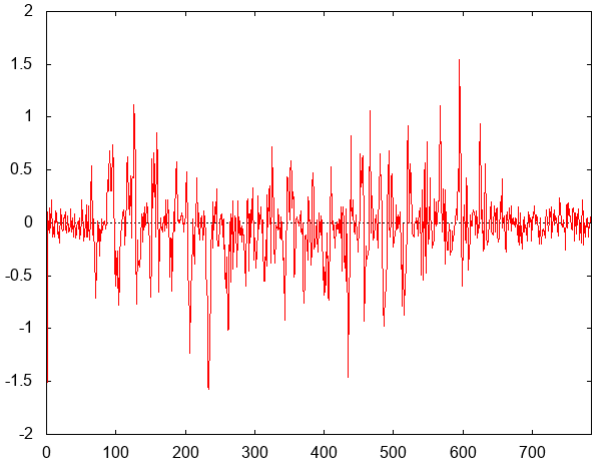

Abbildung 5.3: Logistische Regression ohne Regularisierung

Bei der Verarbeitung von Bilddaten gibt es den großen Vorteil, dass sich die Koeffizienten selbst als Bild interpretieren lassen. In der ersten Zeile der folgenden Grafiken (Abbildung 5.4, 5.5, 5.6) wurden die Koeffizienten unverändert interpretiert, sodass negative Werte keinen Farbunterschied hervorrufen. Es sind auf den Bildern nur die Koeffizienten zu sehen die einen positiven Einfluss auf die Ausgabe haben, da sie die Features repräsentieren die das Vorhandensein einer Linie belohnen. Der Verlauf ist von Schwarz (keine oder negative Auswirkung) bis hin zu Weiß (positive Auswirkung).
In der zweiten Zeile (Abbildung 5.7, 5.8, 5.9) sind die Koeffizienten normalisiert worden, sodass die kleinste Ausprägung 0 und die größte Ausprägung 1 darstellen. Es wird also auch der negative Einfluss der Features berücksichtigt, also derer, die das Vorhandensein einer Linie bestrafen.

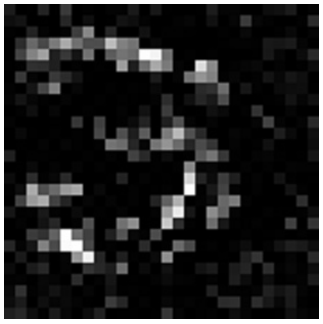

Abbildung 5.4: ohne Regularisierung

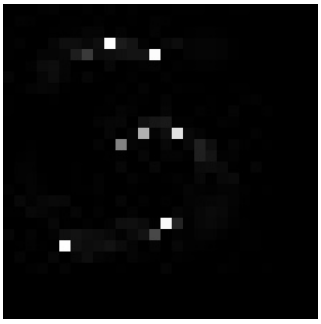

Abbildung 5.5: L1 Regularisierung

Abbildung 5.6: L2 Regularisierung

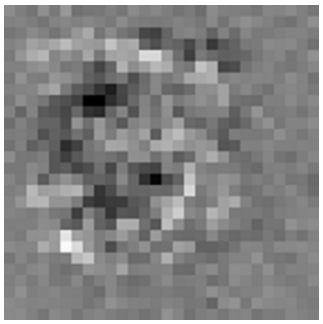

Abbildung 5.7: ohne Reg. normiert

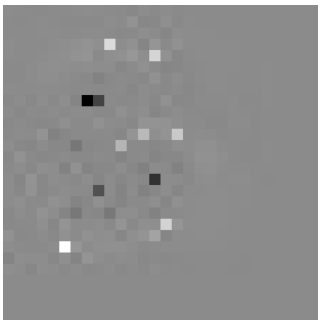

Abbildung 5.8: L1 normiert

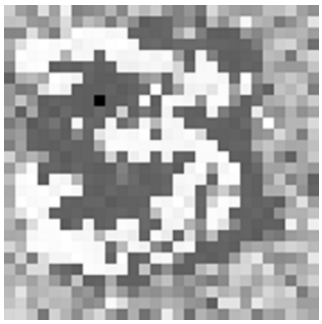

Abbildung 5.9: L2 normiert

Die MNIST Daten sind so auf den angezeigten Bildausschnitt normiert, das der Schwerpunkt der Pixel mit einer gezeichneten Linie zur Bildmitte hin verschoben ist. Die „3“ zu Beispiel, wie in den Graphiken oben zu sehen, ist nach oben links im Bild verzogen, die „9“ hingegen nach unten links. Da sich nicht alle Ziffern perfekt überlappen gibt es für die L1 Regularisierung sehr wenige Koeffizienten, die auf 0 gesetzt werden können. Demnach hat diese Regularisierungsmethode nur mit kleiner Gewichtung eine positive Auswirkung auf das Endergebnis. In den folgenden Graphiken zeigt sich deutlich, dass eine Ausdünnung der Koeffizienten schon bei sehr kleine Hyperparametern C zu schlechteren Vorhersagen führt. Dennoch ist die Wahl dieser Methode, sofern C richtig gewählt wird, durchaus berechtigt, denn zum einen liegt eine kleine Verbesserung vor, zum anderen können zumindest die für die Entscheidung ausschlaggebenden Features isoliert werden.

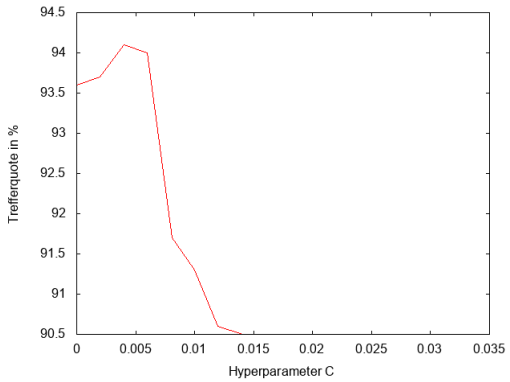

Abbildung 5.10: L1 nach C mit Fixkomma

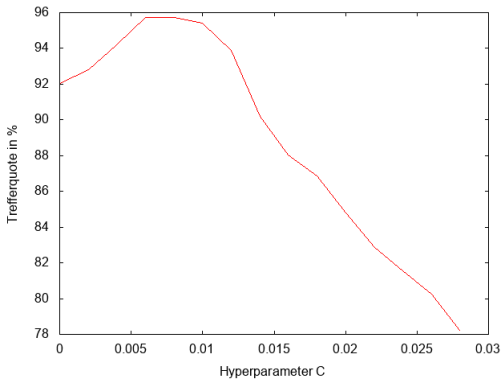

Abbildung 5.11: L1 nach C mit Float

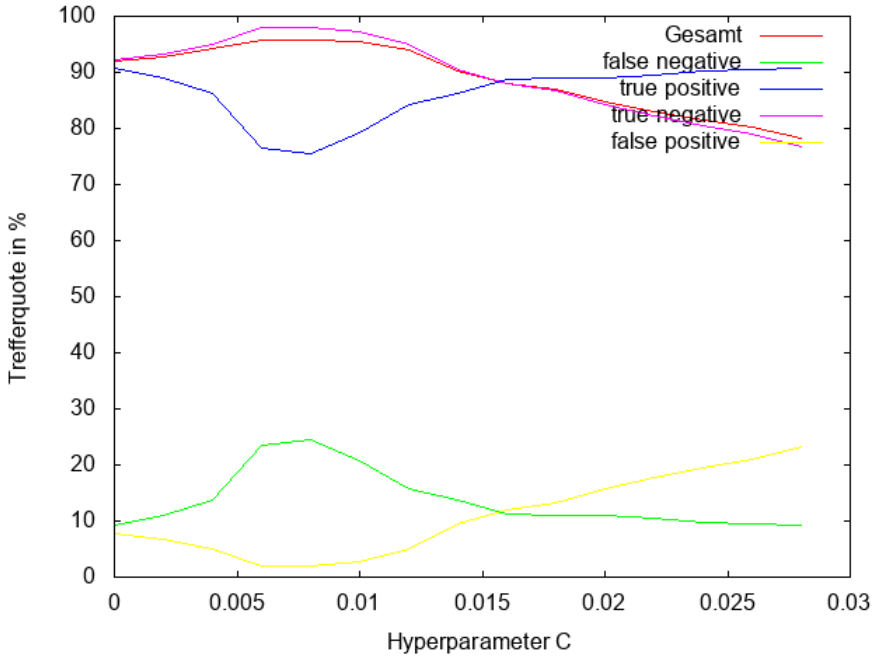

Abbildung 5.12: L1 Regularisierung mit Float

Im Gegensatz dazu kann die L2 Regularisierung die Ergebnisse mit etwas größerem Hyperparameter C geringfügig verbessern, sodass bei der Wahl der Regularisierungsmethode auf Ridge Regression zurückgegriffen werden sollte. Sowohl die Vorhersagen bei der L1- als auch bei der L2 Regularisierung klassifizieren mehr Variablen mit Bezeichnung 0 korrekt als negativ, aber es werden auch mehr Variablen mit Bezeichnung 1 fälschlicherweise als negativ eingeordnet.

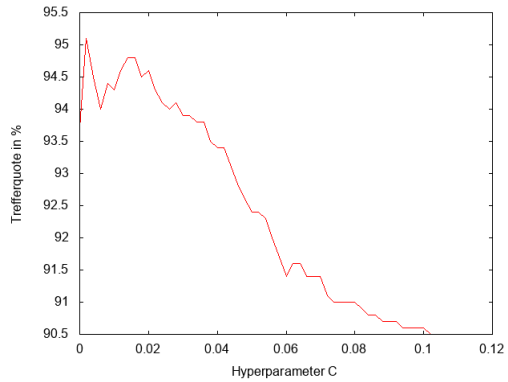

Abbildung 5.13: L2 nach C mit Fixkomma

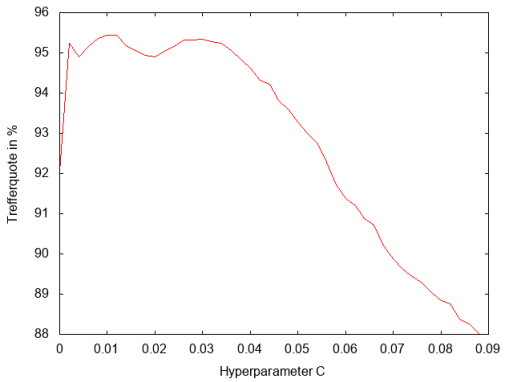

Abbildung 5.14: L2 nach C mit Float

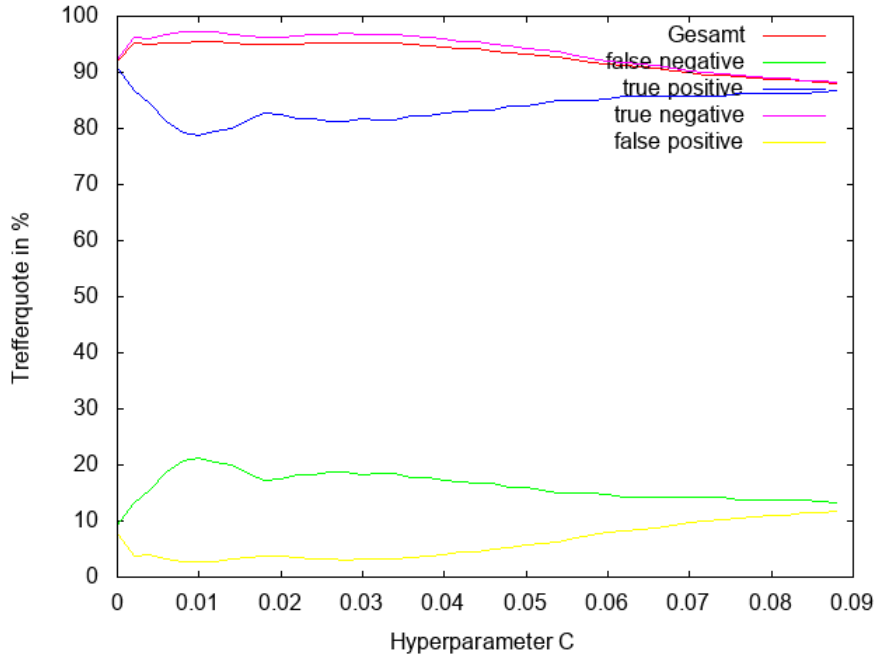

Abbildung 5.15: L2 Regularisierung mit Float

5.2 Festkomma- und Gleitkommazahl

Die Wahl einer Festkommazahl im Gegensatz zu einer Gleitkommazahl liegt durch deren Eigenschaften von Beginn an nahe. Berechnungen mit Fixkommazahlen sind ressourcenschonender und schneller als Gleitkommaberechnungen, bei denen die Anzahl Bits für die Vor- und Nachkommastelle mit jeder Zahl variieren können. zudem können auf der verwendeten 32-Bit Schnittstelle von Xillybus zwei Fixkommazahlen mit der Länge von 16 oder weniger Bits gleichzeitig übertragen werden. Dies sollte die Übertragungszeit für die Daten an den FPGA halbieren.

Überraschenderweise sind aber die Fixkommazahlen auf dem FPGA deutlich Ressourcenabhängiger als die Gleitkommazahlen. eine Belegung des FPGA mit 16 Perceptrons erzielt

für Fixkomma- und Gleitkommazahlen jeweils folgende Auslastungen:

Fixkommazahl:

Bauteil	Benutzt	Gesamt	Auslastung
LUT	120478	133800	90.04%
LUTRAM	685	46200	1.48%
FF	757423	267600	28.18%
BRAM	37	365	10.14%
DSP	400	740	54.05%
I/O	5	400	1.25%

Gleitkommazahl:

Bauteil	Benutzt	Gesamt	Auslastung
LUT	82816	133800	61.90%
LUTRAM	1586	46200	3.43%
FF	81956	267600	30.63%
BRAM	61	365	16.67%
DSP	400	740	54.05%
I/O	5	400	1.25%

Durch diese doch deutlich abweichenden Werte gelingt es mit Gleitkommazahlen 24 Perceptrons auf dem FPGA parallel laufen zu lassen. Der eventuelle Zeitverlust durch die größeren Zahlen wird hierdurch ausgeglichen.
Auch auf dem Hostrechner laufen die Programme unter Gleitkommazahlen schneller und flüssiger als mit Festkommawerten. Eine mögliche Erklärung dafür ist, dass die Werte aus den Datensets in Gleitkommazahlen vorliegen und erst in Festkommazahlen umgerechnet werden müssen. Auch die Einstellungen AP_SAT_SYM und AP_RND_CONV, obgleich sie wichtig für die Funktionalität auf dem FPGA sind, brauchen bei jeder Belegung und Berechnung Rechenzeit um die Direktiven auf die Zahl anzuwenden. Zusätzlich ist es auf dem FPGA bei größeren Batchgrößen die Berechnung der Summe der Kostenfunktionen in Gleitkommazahlen durchzuführen, wodurch eine weitere Umrechnung nötig ist. Beispielweise ist bei einer Batchgröße von 60 der maximale Wert, den ein Hilfskoeffizient annehmen kann auch 60 und damit größer als das Maximum von *ap_fixed(3, 16)*. Es bleibt also nur die Wahl zwischen dem Umrechnen in Gleitkommazahlen zum aufaddieren der Hilfskoeffizienten oder dem Dividieren der der Hilfskoeffizienten bei jeder Berechnung

durch die Batchgröße. Auch die zweite Möglichkeit kann dazu führen dass der Wert nicht mehr korrekt dargestellt werden kann und zu 0 gerundet wird.

5.3 Timing und Energie

Um die Performance der Implementierung zu erfassen wurden die verschiedenen Lese- und Schreibmethoden gemessen. Dazu wurde auf dem Hostsystem jeweils vor und nach der zu messenden Operation die Systemzeit gespeichert und subtrahiert. zusätzlich wurde die Gesamtzeit der Operationen berechnet. Um einen möglichst genauen Wert zu erhalten wurden diese Messungen für jeweils 200 oder 500 Zyklen wiederholt. Nachfolgend sind die Ergebnisse Des Zeittests:

Operation, Zeit in ms	Fixkomma mit 8 Perceptrons			Float mit 16 Perceptrons		
	min	∅	max	min	∅	max
Lesen	426.11	430.472	442.033	740.779	767.808	770.535
Training	644.675	647.403	625.012	596.129	728.134	757.502
Test schreiben	410.963	414.06	424.923	721.724	749.679	771.331
Gesamt	1075.21	1078.283	1091.621	1339.51	1496.395	1535.616

Abbildung 5.16: 200 Wiederholungen, jeweils 1.000 Trainings- und Testzyklen pro Messung

Operation, Zeit in s	Fixkomma mit 8 Perceptrons			Float mit 16 Perceptrons		
	min	∅	max	min	∅	max
Lesen	3.98	4.16	4.21	7.49	7.6	7.67
Training	6.44	6.44	6.45	6.98	7.34	7.42
Test schreiben	3.97	4.14	4.19	7.48	7.58	7.65
Gesamt	10.42	10.6	10.65	14.6	14.94	15.1

Abbildung 5.17: 500 Wiederholungen, jeweils 10.000 Trainings- und Testzyklen pro Messung

Aus den Daten lässt sich schließen dass die Implementierung mit der doppelten Anzahl an Perceptrons und Bits im Zahlenformat keine so großen Auswirkungen auf die gemessenen Zeiten hat. Trotz Vervierfachung der zu schreibenden und lesenden Bits erhöht sich die Zeit im ersten Test für das Lesen nur um 78,3%, für das Training um 12,5%, für das Testen um 81% und für die Gesamtzeit um 38,8%. Die Gesamtzeit pro Perceptron würde für die Festkommazahl 134.75 ms und für die Gleitkommazahl 93.525 ms betragen. Diese Werte gelten für das Trainieren und Testen von jeweils 1000 Variablen mit 784 Features. Interessanterweise ist bei der Festkommazahl das Testen am schnellsten, bei der Gleitkommazahl jedoch das Training. Dies lässt sich auf die zusätzlichen Umrechnungen der Werte auf dem

FPGA bei der Festkommazahl zurückführen.
Lässt man die Perceptrons auf dem Hostrechner selbst trainieren, so erhält man folgende Werte:

Operation, Zeit in s	1.000 mal Training und Tests		
	min	∅	max
Training	43.34	43.43	43.76
Test schreiben	8.75	8.77	8.87
Gesamt	52.09	52.21	52.56

Abbildung 5.18: 500 Wiederholungen für Perceptrons auf dem Hostsystem, 8 Perceptrons

Die Zeiten des Hosts zeigen, wie gut die Implementierungen auf dem FPGA gegenüber einer direkten Programmierung abschneiden. Ganze 48.6 mal mehr Zeit braucht der Hostrechner gegenüber der Fixpunkt Implementierung und 34.8 mal mehr als die der Gleitkommazahl. An dieser Stelle ist zu erwähnen, dass der Prozess auf dem Hostrechner weder parallelisiert noch optimiert wurde und somit das Multithreading moderner Prozessoren nicht ausgenutzt wurde. Es wurde lediglich naiv der Code für die Perceptrons in Reihe geschaltet und entsprechend häufig der Anzahl der Perceptrons auf dem FPGA ausgeführt.

Auch der Energieverbrauch der Implementierungen ist überschaubar. So hat die Implementierung mit 8 Perceptrons einen Verbrauch von 1.795 Watt und die Implementierung mit 16 Perceptrons und Gleitkommazahl 2.338 Watt. Zum Vergleich: Der Hostrechner verbraucht als Verlustleistung schon 130 Watt [14]. Eine GPU würde unter Last bis zu 538 Watt verbrauchen [3].

5.4 Probleme

Bei der Implementierung der Hostanwendung gab es einige Hürden zu überwinden. Vor allem das Timing und die Abstimmung mit dem FPGA sind essentiell für die Reibungslose Kommunikation beider Systeme. Bei Implementierungen von Fixkommazahlen mit mehr als 8 Perceptrons konnte der Hostprozess die Daten einiger Perceptrons nicht mehr auslesen. Erst das Umstellen auf einen linearen Programmverlauf führte zu einer Verbesserung der Kommunikation, zu Ungunsten der Laufzeitdauer. Der Grund dieser Störungen konnte nicht ermittelt werden, es steht jedoch die Latenz der Verteiler- und Sammlerklassen im Verdacht. Eine Aufteilung der Klassen und die damit verbundene Verminderung der Latenzzeiten brachte jedoch nicht den gewünschten Erfolg, sodass auf eine Gleitkommazahl-Implementierung ausgewichen werden musste.

Kapitel 6

Fazit und Empfehlungen

6.1 Fazit

In dieser Arbeit wurden Optimierungen für die logistische Regression auf FPGAs vorgenommen und erläutert. Der allgemeine Aufbau und die Struktur von FPGAs wurde detailliert dargestellt. Die Methode der logistischen Regression wurde erklärt und Regressionsmethoden für den Lernprozess vorgestellt. Die Vorzüge der logistischen Regression für FPGAs sind, dass durch das Gradientenabstiegsverfahren nur Updates der Koeffizienten nötig sind und dass die Berechnungen dieser Koeffizientenupdates durch einfache oder bereits auf dem FPGA hinterlegte Funktionen durchgeführt werden können. Weiterhin wurde die Implementierung eines Modells für den FPGA bereitgestellt und der dazugehörige Hostprozess programmiert. In den Experimenten gab es einen Einblick in die Möglichkeiten und Probleme der Modellumsetzung.

Die Arbeit zeigt, dass es durchaus möglich ist logistische Regression auf FPGAs zu implementieren ohne einen großen Kompromiss bezüglich der Trainingsdauer eingehen zu müssen. Im Gegensatz zu einer naiven Durchführung auf dem Hostrechner ist der FPGA sogar bedeutend schneller und effizienter.
Für Problemstellungen, die eine Vielzahl von verschiedenen trainierten Modellen benötigen ist der FPGA besonders geeignet. Ein Beispiel hierzu ist das Finden eines optimalen Hyperparameters C für die Gewichtung von Regularisierungsmethoden. Dieser lässt sich zwar mathematisch begrenzen [24], jedoch nicht berechnen und wird als schwierig zu wählen beschrieben [31]. An dieser Stelle ist die Parallelität des FPGAs von Vorteil, denn es können in geringer Zeit viele Modelle, nur voneinander abweichend durch die Wahl des Hyperparameters, trainiert werden und die Ergebnisse dem Nutzer als Pareto-Front vorgestellt werden. Eine beispielhafte Bearbeitung des MNIST Datensatzes [20] ist in Kapitel 5 durchgeführt worden.

Durch die Wahl der High-Performance Schnittstelle über PCIe von [8] als Grundlage für die Arbeit ist die Notwendigkeit von Datenspeichern auf dem FPGA nicht gegeben. Dies ermöglicht die volle Nutzung der Hardwarekomponenten durch die Perceptrons selbst.
Im Vergleich zu gängigen CPUs und auch GPUs ist die für die Durchführung der Programme erforderliche Energie besonders niedrig. Man kann demnach eine Implementierung von logistischer Regression auf FPGAs durchaus als Energieeffizient bezeichnen. Grade in der heutigen Zeit, wo Ressourcen schonende Verfahren an Popularität gewinnen und aufgrund der Weltklima-Lage durchaus auch von Nöten sind, erhält diese Eigenschaft einen besonderen Stellenwert. Die dazu erforderlichen technischen Eigenschaften sind auf FPGA durchaus vorhanden, und es konnte gezeigt werden dass es auch den theoretischen Anforderung an Trainingsmodellen gerecht wird.

6.2 Empfehlungen

Einige in dieser Arbeit aufgrund von technischen oder zeitlichen Anforderungen nicht behandelte Realisierungsmöglichkeiten werden hier aufgeführt. Dies soll einen Anstoß für weitere Arbeiten mit und Verbesserungen an dem präsentierten Konzept geben. Das FPGA, oder genauer das Evaluation-Kit des in dieser Arbeit benutzen FPGAs enthält einen 3 GB großen DDR3 RAM. Anstatt mit Der PCIe Schnittstelle als Hauptmedium zur Datenvermittlung kann auch der Trainingsdatensatz in diesen Speicher geladen werden. Es wäre dadurch,nach der Übertragung des Datensatzes, nur eine minimale Kommunikation mit dem FPGA erforderlich um die selben Aktionen mit ihm durchzuführen, die auch in dieser Arbeit behandelt wurden. Die Verknüpfung des FPGAs mit der DDR3 Schnittstelle ist jedoch nicht trivial und beinhaltet viele mögliche Fehlerquellen, kann jedoch zu noch besseren Ergebnissen im Hinblick auf die Trainingszeit liefern.

In [21] wird eine Möglichkeit der Feature-Selection über Daten oder Feature Streams beschrieben. Da die Kommunikation Des FPGA mit dem Hostrechner über Streams erfolgt, wäre dies eine gute Grundlage für eine Implementierung dieser Methode. Für komplizierte Algorithmen ist jedoch eine Approximation notwendig, denn ähnlich wie bei GPUs zeichnet sich die Bearbeitung von Daten auf dem FPGA nicht durch Komplexität, sondern durch Parallelität aus.

Das auch das Training eines einzelnen Modells durch ein FPGA beschleunigt werden kann lässt sich aus der Parallelisierung der SGD in [22] erkennen. Anstatt hier die einzelnen Schleifen auf verschiedene Maschinen zu verteilen ist es möglich, diese auf dem FPGA parallel zu implementieren und miteinander Kommunizieren zu lassen. Der beschriebene I/O-Overhead könnte dadurch umgangen werden.

Viele Datensätze liegen nicht, wie in dieser Arbeit vorausgesetzt in dichotomer Form vor. Demnach liegt die Implementierung einer multinomialen logistischen Regression nahe. Diese kann mehrere Ausprägungen dadurch vorhersagen, dass für jede Ausprägung eine logistische Regression mit dieser als „trifft zu“ und aller anderen als “trifft nicht zu“ trainiert wird. Die Ausgabe wird zugunsten der am eindeutigsten getroffenen Entscheidung gewählt. Da hier mehrere Modelle benötigt werden, die alle auf den selben Daten (außer den Labeln) trainiert und getestet werden ist das FPGA prädestiniert für diese Aufgabe. Die finale Entscheidung kann in der Kollektorklasse auf dem FPGA noch vor der Ausgabe an den Nutzer erfolgen.

Abbildungsverzeichnis

Literaturverzeichnis

[1] *Tabelle Standardnormalverteilung.* https://de.wikibooks.org/wiki/Tabelle_Standardnormalverteilung#? Besucht: 23.01.2020.

[2] *Introduction to the Logistic Regression Model*, Kapitel 1, Seiten 1–33. John Wiley & Sons, Ltd, 2013.

[3] 3DCENTER: *Stromverbrauch aktueller und vergangener Grafikkarten.* http://www.3dcenter.org/artikel/stromverbrauch-aktueller-und-vergangener-grafikkarten, 2014. Besucht: 09.02.2020.

[4] AMAGASAKI, MOTOKI und YUICHIRO SHIBATA: *Principles and Structures of FPGAs: FPGA Structure.* Springer Nature Singapore Pte Ltd., Seiten 47–86, 2018.

[5] BECKHAUS, K., B. ERICHSON, W. PLINKE und R. WEIBER: *Logistische Regression*, Band 14. Springer Gabler, Berlin, Heidelberg, 2016.

[6] BENINGO, Y., I. GOODFELLOW und A. COURVILLE: *Deep Learning - Das umfassende Handbuch.* mitp, 2018.

[7] COHEN, W.: *Efficient Logistic Regression with Stochastic Gradient Descent.* http://www.cs.cmu.edu/~wcohen/10-605/sgd-part2.pdf.

[8] DILLKÖTTER, FABIAN: *Umsetzung einer High-Performance FPGA-Schnittstelle für maschinelles Lernen*, 2019.

[9] DUA, DHEERU und CASEY GRAFF: *UCI Machine Learning Repository*, 2017.

[10] FEIST, T.: *Whitepaper: Vivado Design Suite.* 2012.

[11] HERTA, PROF.DR.C: *Logistische Regression.* http://christianherta.de/lehre/dataScience/machineLearning/logisticRegression.pdf, 2013. Besucht: 01.02.2020.

[12] HERTA, PROF.DR.C: *Regularisierung.* http://christianherta.de/lehre/dataScience/machineLearning/regularization.pdf, 2013. Besucht: 05.02.2020.

[13] InAccel: *https://github.com/InAccel/logisticregression/*. 2019.

[14] Intel: *Intel Xeon Prozessor W3565*. `https://ark.intel.com/content/www/de/de/ark/products/39721/intel-xeon-processor-w3565-8m-cache-3-20-ghz-4-80-gt-s-intel-qpi.html`. Besucht am 11.02.2020.

[15] Jurafsky, D. und J.H. Martin: *Logistic Regression*. 2019.

[16] KNIME: *Regularization for Logistic Regression: L1, L2, Gauss or Laplace?* `https://www.knime.com/blog/regularization-for-logistic-regression-l1-l2-gauss-or-laplace`. Besucht: 04.02.2020.

[17] Kuon, I., R. Tessier und J. Rose: *FPGA Architecture: Survey and Challenges*. now, 2008.

[18] Kuon, Ian und Jonathan Rose: *Measuring the Gap Between FPGAs and ASICs*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26:203–215, 2007.

[19] Lars Wienbrandt, Jan Christian Kässens, Matthias Hübenthal und David Ellinghaus: *1000x faster than PLINK: Combined FPGA and GPU accelerators for logistic regression-based detection of epistasis*. 2018.

[20] LeCun, Y., C. Cortes und C.J.C. Burges: *THE MNIST DATABASE*. `http://yann.lecun.com/exdb/mnist/`. Besucht: 17.10.2019.

[21] Li, J., K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang und H. Liu: *Feature Selection: A Data Perspective*. ACM Computer Survey, 2017.

[22] Martin A. Zinkevich, Markus Weimer, Alex Smola und Lihong Li: *Parallelized Stochastic Gradient Descent*. 2010.

[23] Nane, R., V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson und K. Bertels: *A Survey and Evaluation of FPGA High-Level Synthesis Tools*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35(10):1591–1604, Oct 2016.

[24] Ng, A.Y.: *Feature selection, L1 vs. L2 regularization, and rotational invariance*. 2004.

[25] NVIDIA: *https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-datasheet.pdf*. NVIDIA Corporation, 2016.

[26] Papageorgiou, Markos, Marion Leibold und Martin Buss: *Minimierung einer Funktion mehrerer Variablen ohne Nebenbedingungen*, Seiten 37–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[27] PhD, Jason Brownlee: *How To Implement Logistic Regression From Scratch in Python.* `https://machinelearningmastery.com/implement-logistic-regression-stochastic-gradient-descent-scratch-python/`. Besucht: 10.11.2019.

[28] Rohrlack, C.: *Logistische und Ordinale Regression*, Band 3. Gabler Verlag, Wiesbaden, 2009.

[29] Schröder, D.: *Intelligente Verfahren: Identifikation und Regelung nichtlinearer Verfahren.* 2010.

[30] Tibshirani, Robert: *Regression Shrinkage and Selection via the Lasso.* Journal of the Royal Statistical Society. Series B (Methodological), 58:267–208, 1996.

[31] Tran, Trung: *Machine Learning Part 9: Regularization*, 2016.

[32] Tsuruoka, Y., J. Tsujii und S. Ananiadou: *Stochastic Gradient Descent Training for L1-regularized Log-linear Models with Cumulative Penalty.* Proceedings of the 47th Annual Meeting of the ACL and the 4th IJCNLP of the AFNLP, Seiten 477–485, 2009.

[33] Xilinx: *https://www.xilinx.com/publications/technology/power-advantage/7-series-power-benchmark-summary.pdf.* Xilinx, 2015.

[34] Xilinx: *7 Series FPGAs Data Sheet: Overview.* 2018.

[35] Xilinx: *Vivado Design Suite User Guide High-Level Synthesis.* 2018.

[36] Xilinx: *AC701 Evaluation Board for the Artix-7 FPGA.* 2019.

[37] Xillybus: *An FPGA IP core for easy DMA over PCIe with Windows and Linux.* `http://xillybus.com/`. Besucht: 10.12.2019.

Eidesstattliche Versicherung
(Affidavit)

Sliwinski, Moritz — 166206

Name, Vorname
(Last name, first name)

Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der Bachelor-/Masterarbeit*:
(Title of the Bachelor's/ Master's* thesis):

Optimierung von logistischer Regression auf FPGAs

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Hagen, den 10.02.2020
Ort, Datum
(Place, date)

Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Hagen, den 10.02.2020
Ort, Datum
(Place, date)

Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.