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 Energieeffizient 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.

1.2 Aufbau der Arbeit

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 CLPs 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.

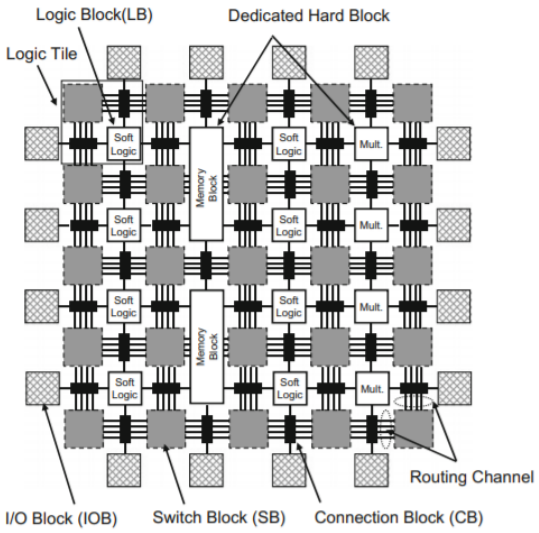

Abbildung 2.1: Aufbau eines IC, die grauen Schaltblöcke (SB) sind die konfigurierbaren Datenpfade

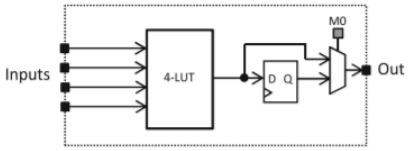

Abbildung 2.2: Schematische Darstellung eines CLB

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 ASCI Implementierung. Außerdem verbrauchen sie dynamisch circa 10 mal mehr Energie [13]. 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[12].

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 [12]. 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.

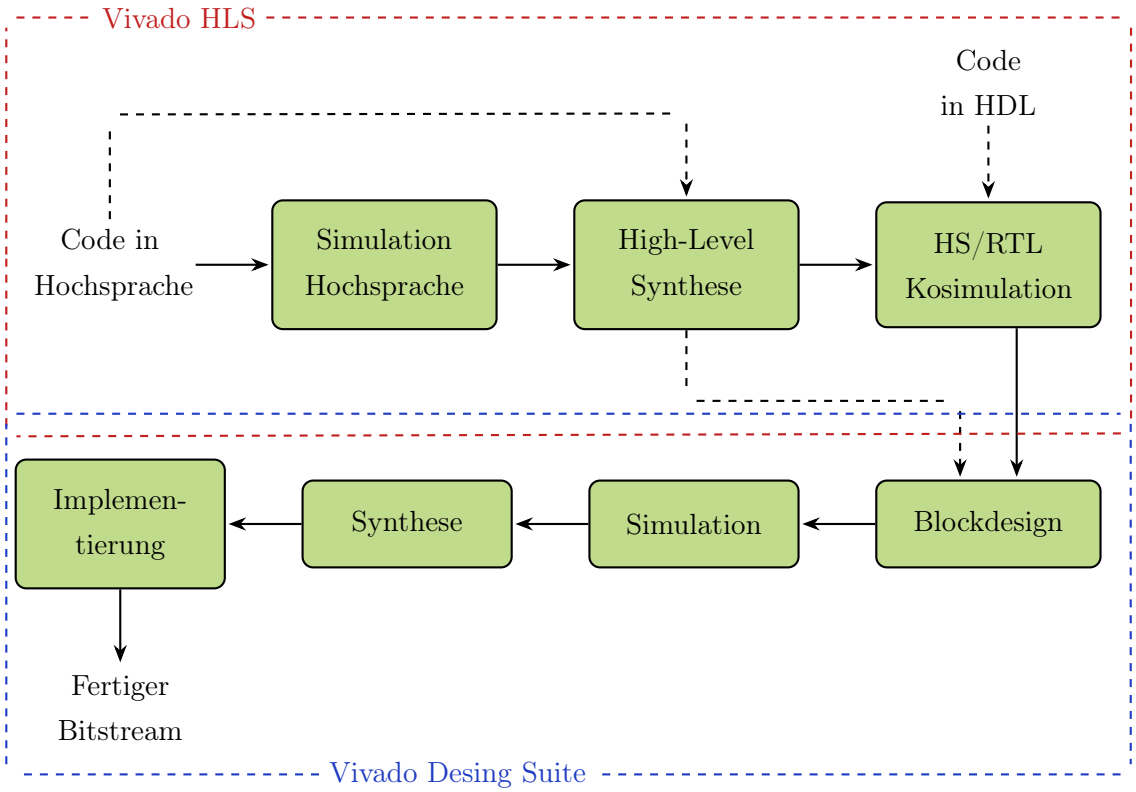

Abbildung 2.3: Konfigurationsablauf eines Xilinx-FPGAs

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 [9]. 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[16].

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 [23].
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 [22].
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[24].
Eingebaut ist das Board in einen Dektop-PC einem Intel Xenon 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 [1]. 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 [20]. 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 zu vernachlässigender Störterm, der durch die spätere Ableitung in Abschnitt 3.2.2 wegfällt [20]. 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 [2][5].

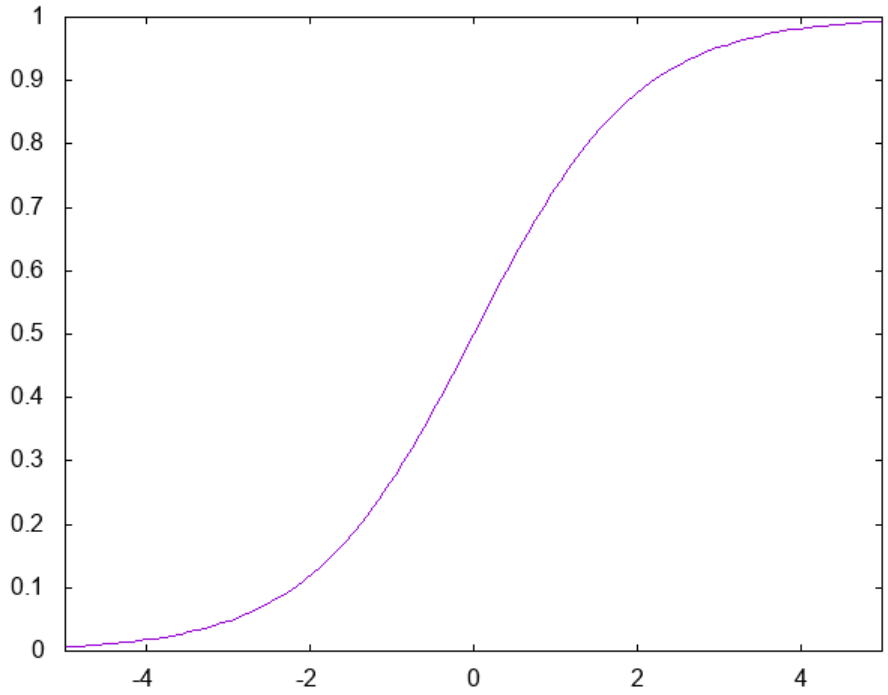

Abbildung 3.1: Die logistische Funktion $p = \frac{1}{1 + \exp(-x)}$

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.

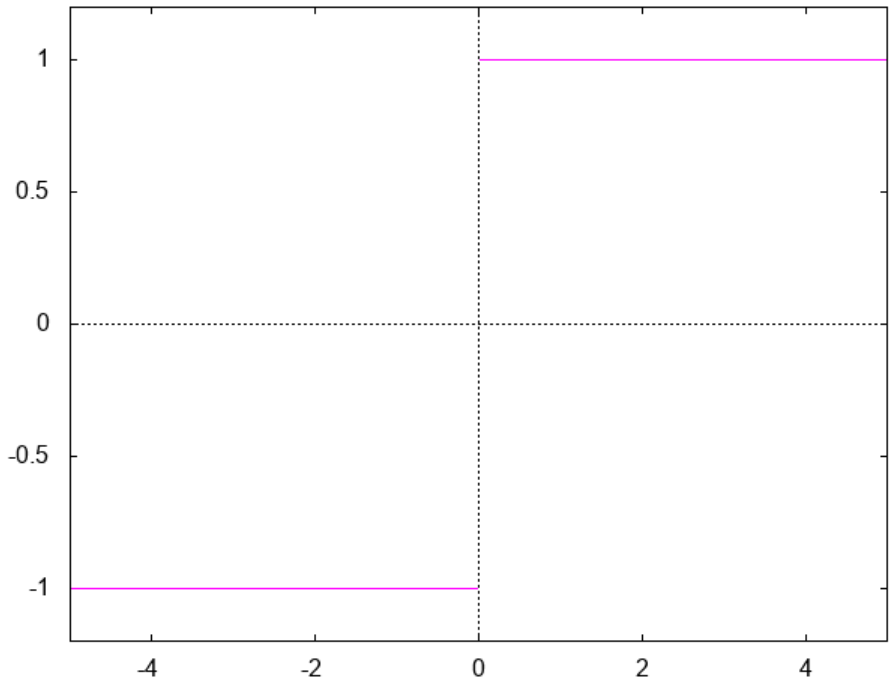

Abbildung 3.2: Die Vorzeichenfunktion $q = \text{sign}(x)$

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

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 [15]. 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 [3]. 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 [3]. 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 [10]:

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

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 [21]:

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

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 $s^{(l)}$[18]. 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.

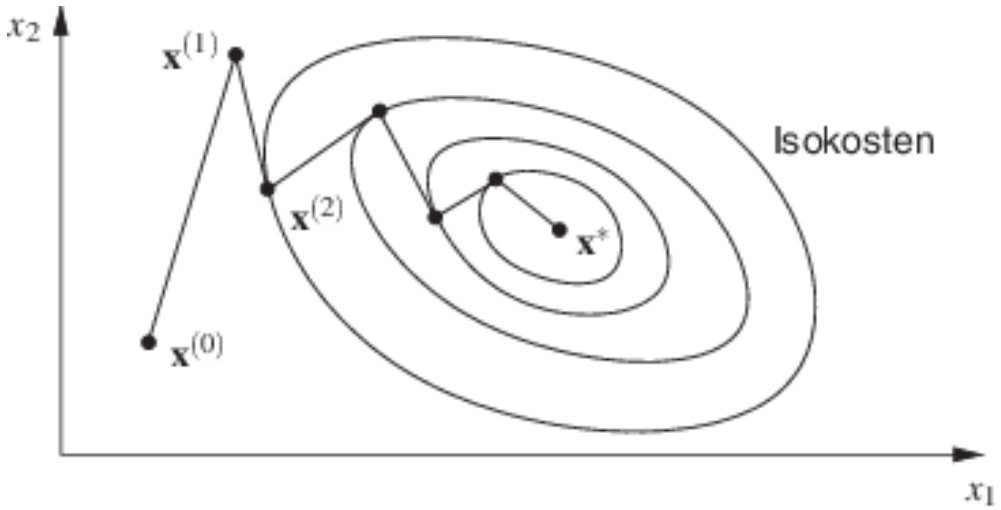

Abbildung 3.4: Beispielhafter Verlauf des Algorithmus im zweidimensionalen Fall

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 [18]:

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

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

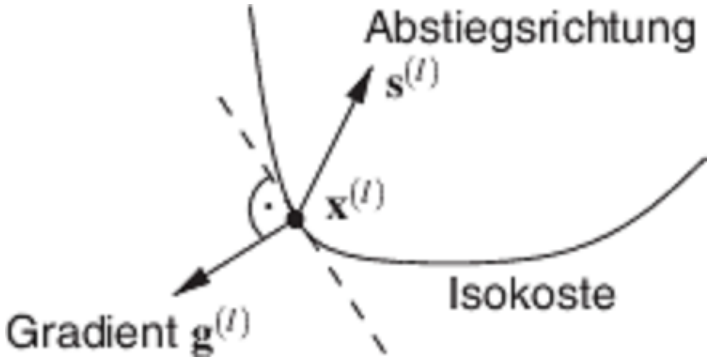

Abbildung 3.5: Beziehung zwischen Gradient und Abstiegsrichtung

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 [18]:

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

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 [10]:

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

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 [17]. 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.

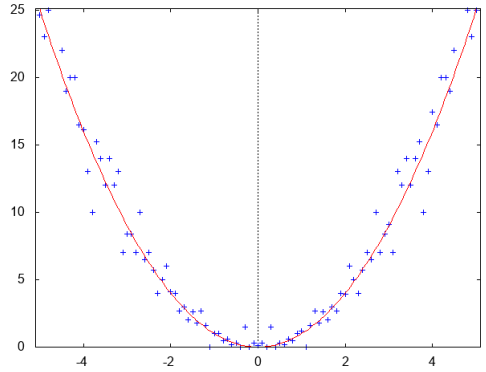

Abbildung 3.6: Normale Regressionsgrade

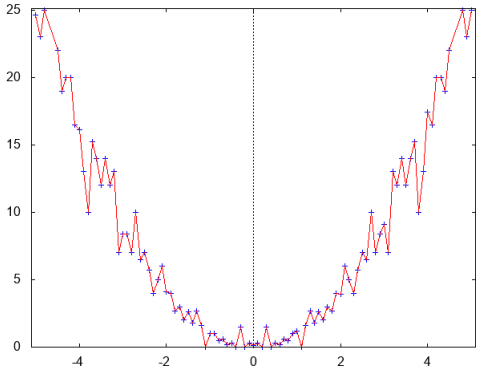

Abbildung 3.7: Overfitting des Modells

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:

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

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 [11]. 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 $2\sigma^2 = 1$ [11]. 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 [11].

3.4 Verwandte Algorithmen

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.

4.1 Implementierung in C++

Für die erste Implementierung der Voraussagefunktion wurde der Code von [19] 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.[23] 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.[23] 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;
    float sum_error_float=(float)sum_error+(float)error;
    sum_error=sum_error_float;
    tmp_coefficients[0]+=predicted*(1.0f-predicted);
        for(int i=0; i<FEATURE_COUNT; i++){
                #pragma HLS LOOP FLATTEN
                tmp_coefficients[i+1]+=predicted*(1.0f-predicted)*
                        features[i];
        }
        /* Batch Update der Koeffizienten */
        batch_count++;
        if(batch_count>=BATCH_SIZE){
                DATA_TYPE sum_error_t=sum_error/BATCH_SIZE;
                for(int i=0; i<FEATURE_COUNT+1; i++){
                        coefficients[i]+=lrate*sum_error_t*
                                tmp_coefficients[i]/BATCH_SIZE;
                }
                batch_size=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.
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.

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 4.2 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.[14]
Die Auslastung des FPGA für diese Implementierung beträgt in etwa 50%, sodass auch 16 Perceptrons parallel geschaltet werden können.

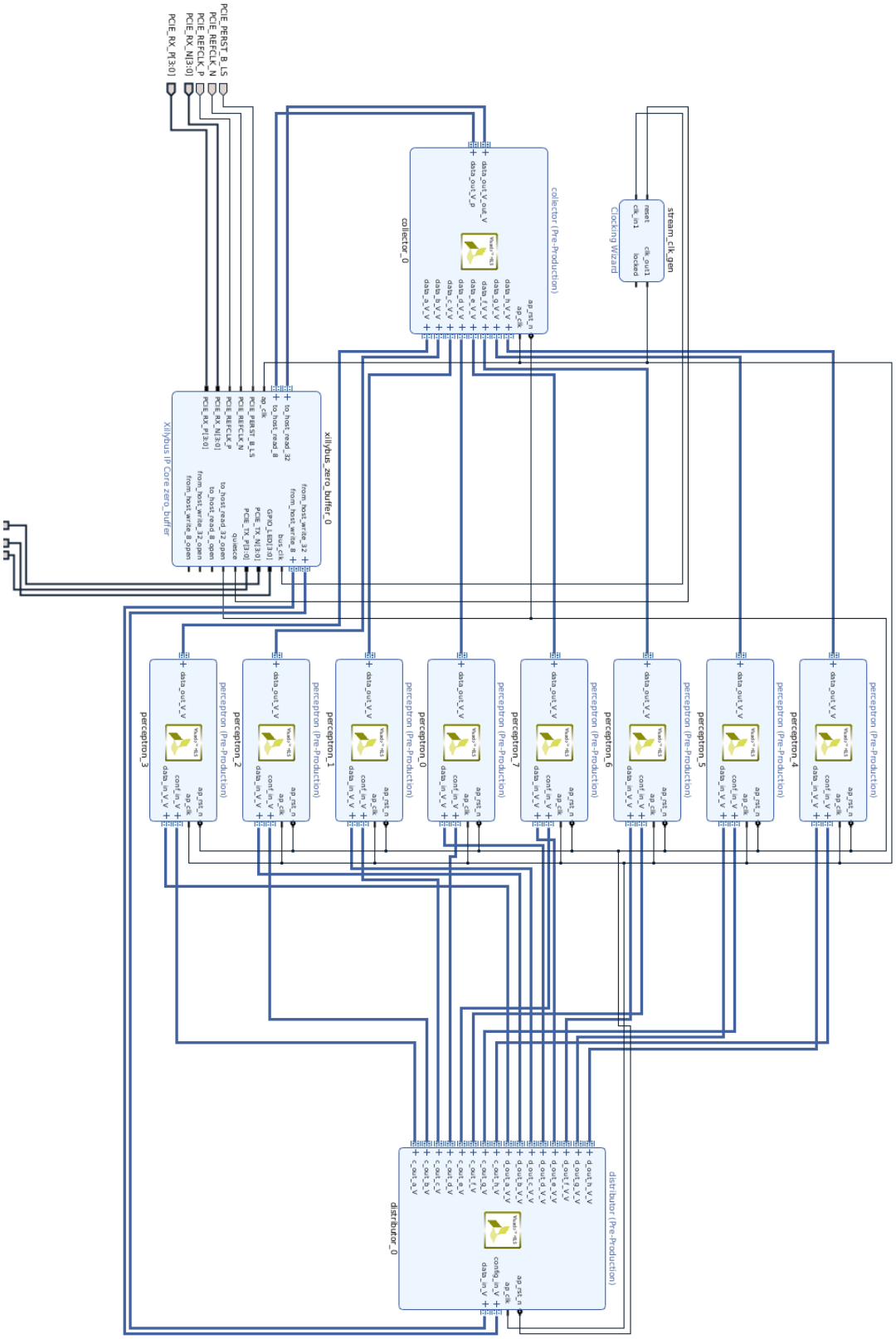

Abbildung 4.2: Das Blockdesign auf dem FPGA

Kapitel 5

Experimente und Ergebnisse

Kapitel 6

Ausblick und Fortsetzung

Feature Selection mit Daten-Streams[15]

Anhang A

Weitere Informationen

Abbildungsverzeichnis

Literaturverzeichnis

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

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

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

[4] Amagasaki, Motoki und Yuichiro Shibata: *Principles and Structures of FPGAs: FPGA Structure.* Springer Nature Singapore Pte Ltd., Seiten 23–45, 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] Feist, T.: *Whitepaper: Vivado Design Suite.* 2012.

[10] Herta, Prof.Dr.C: *Logistische Regression.* `http://christianherta.de/lehre/dataScience/machineLearning/logisticRegression.pdf`, 2013. Besucht: 01.02.2020.

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

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

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] 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.

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

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

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

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

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

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 4. Februar 2020

Muster Mustermann