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 (CLP) 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 (Re)Konfiguriert werden.

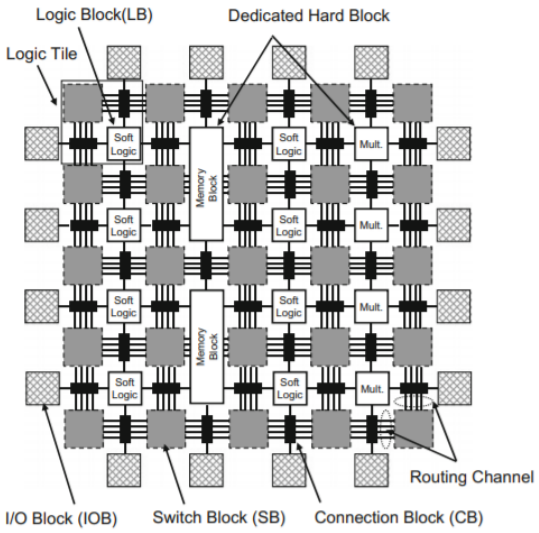

Abbildung 2.1: Aufbau eines IC, die grauen Schaltblöcke (SB) sind die konfigurierbaren Datenpfade

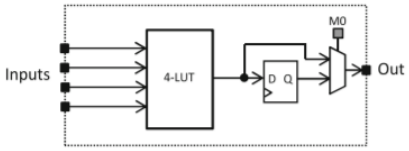

Abbildung 2.2: Schematische Darstellung eines CLP

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), was jedoch mit einer höheren Komplexität der zu konfigurierenden Logik verbunden ist.
Durch die Konfigurierbarkeit des FPGAs ergeben sich allerdings 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. [8] 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. Grade 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.[7]

2.2 Konfiguration und Ablauf

Wie das „Field Programmable“ schon vermuten lässt, ist es möglich nach der Fabrikation des FPGA Funktionen in diese zu Programmieren („in the field“). [7] um diese Funktion zu verändern muss das FPGA neu Konfiguriert werden. Dies geschieht durch einen sogennanten „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.

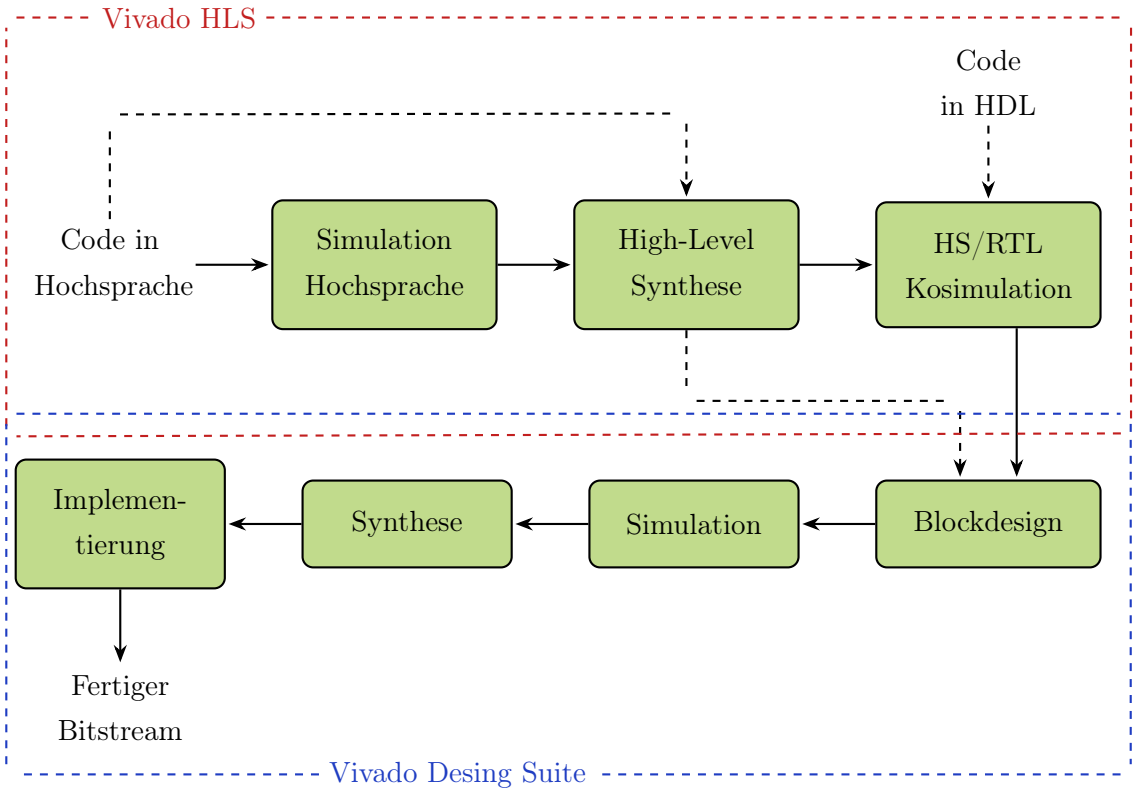

Abbildung 2.3: Konfigurationsablauf eines Xilinx-FPGAs

Zunächst programmiert man 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.[10]

Dann folgt eine Simulation des Programms um dessen Tauglichkeit für eine FPGA Konfiguration zu prüfen. Dieser Schritt ist optional, jedoch sehr hilfreich, denn die High-Level Synthese von Vivado HLS unterstützt 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.[14]
Nun wird mit Der High-Level Synthese ein Programm in einer HDL (Hardware Definition Language, in diesem Fall VHDL oder Verilog) erstellt. Es ist auch möglich die RTL (Register Transfer Level) mit einer HDL selbst zu programmieren. Dieser Ansatz kann zu deutlich effizienteren Designs führen, ist aber auch mit erheblichem Mehraufwand verbunden. Vor allem der geringere Programmieraufwand 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.
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.
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 der Kontrolle der Funktionalität dienen. 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 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 (Configurable Logig Blocks) Slices und 500 I/O Pins.[13]
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.[15]

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[6] 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 als binäres Logit-Modell bezeichnet. Sie unterscheidet sich in soweit von der linearen Regression, dass die Voraussagen nicht spezielle Werte, sondern die Wahrscheinlichkeiten angeben, mit denen die jeweilige Ausprägung der Variable angenommen wird.[12] Die beiden Ausprägungen der abhängigen Variablen wird mit 0 bzw. 1 beschrieben, sodass die Vorhersage des Modells die Wahrscheinlichkeit beschreibt, mit der die abhängige Variable den Wert 1 annimmt, formal $P(Y_i = 1)$. Die Logistische Regression gehört zur Klasse der strukturen-prüfenden Verfahren und bildet eine Variation der Regressionsanalyse. Sie grenzt sich durch die Art ihrer abhängigen Variable, bezeichnet mit Y, welche als kategoriale Variable klassifiziert ist, von anderen Regressionsanalysen ab. Die Ausprägungen der Variable repräsentieren die verschiedenen Alternativen, in unserem binären (oder auch dichotomen) Fall ist „trifft zu“ und „trifft nicht zu“.[5] Diese Gruppen werden nun mit 0 und 1 bezeichnet und 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 $D = \{(X_1, y_1), \ldots, (X_N, y_N)\}$ mit $X_i \in \mathbb{R}^d$ und $y_i \in \{0, 1\}$, ein Modell $f_\beta(x)$ für Vorhersagen finden, welches auf neu-

en, ungesehenen Daten einen möglichst kleinen Fehler macht. Ausdrücken lässt sich das logistische Regressionsmodell nun wie folgt:

$$\pi(x) = f_\beta(x_1, .., x_n)$$

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

$$z(X_i) = \beta_0 + \sum_{j=1}^{d} \beta_j * x_{ij} + u_i$$

beschrieben. β stellt hier den Vektor der Koeffizienten $(\beta_1, ..., \beta_d)^T$ dar und β_0 ist der Bias. u_i ist ein zu vernachlässigender Störterm.[12] Um das Modell auszugestalten wird hier 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]

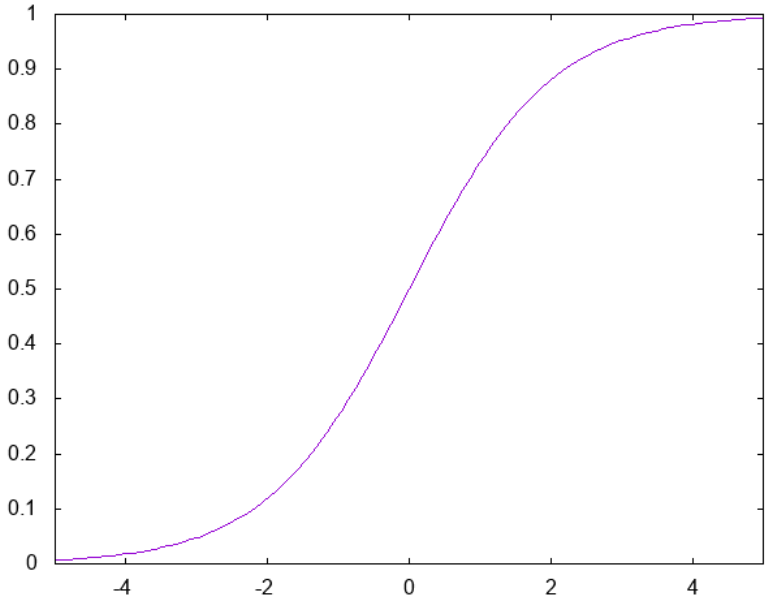

Abbildung 3.1: Die logistische Funktion $p = \frac{1}{1 + \exp(-x)}$

Wenn man diese Transformation der systematischen Komponente nun mit der logistischen Funktion durchführt erhält man die logistische Regressionsfunktion:

$$\pi(X) = \frac{1}{1 + \exp(z(X))}$$

Also genauer:

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

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

3.2 Lernen mit Logistischer Regression

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.[9] In der linearen Regression wird hierfür häufig die „Methode der kleinsten Quadrate“

$$\sum_{i=1}^{N} (\pi(X_i) - y_i)^2$$

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

$$\min_{\beta} \sum_{i_1}^{N} (\pi(X_i) - y_i)^2$$

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]

Die Anpassung der Koeffizienten erfolgt durch Minimierung der Loss Funktion:

$$\min_{\beta} \sum_{i}^{N} \log(1 + \exp(-(\beta_0 + x_i^T \beta))) + C * R(\beta)$$

Wobei $C \in \mathbb{R}$ ein Hyperparameter ist,der fest gewählt werden muss. Dies geschieht zumeist durch das Testen verschiedener Werte, sodass hier ein Ansatz zur Parallelisierung entsteht.

3.3 Stochastic Gradient Descent

3.4 Regularisierungsmethoden

3.4.1 LASSO

L1-Regularisierung (Least Absolute Shrinkage and Selection Operator, kurz **LASSO**) mit $R(\beta) = \sum_i^N |\beta_i|$

3.4.2 Ridge Regression

L2-Regularisierung (Ridge Regression) mit $R(\beta) = \sum_i^N \beta_i^2$.

3.5 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 [11] 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.[14] 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.[14] 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. Die Funktion

```
/* 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

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 [6] 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.

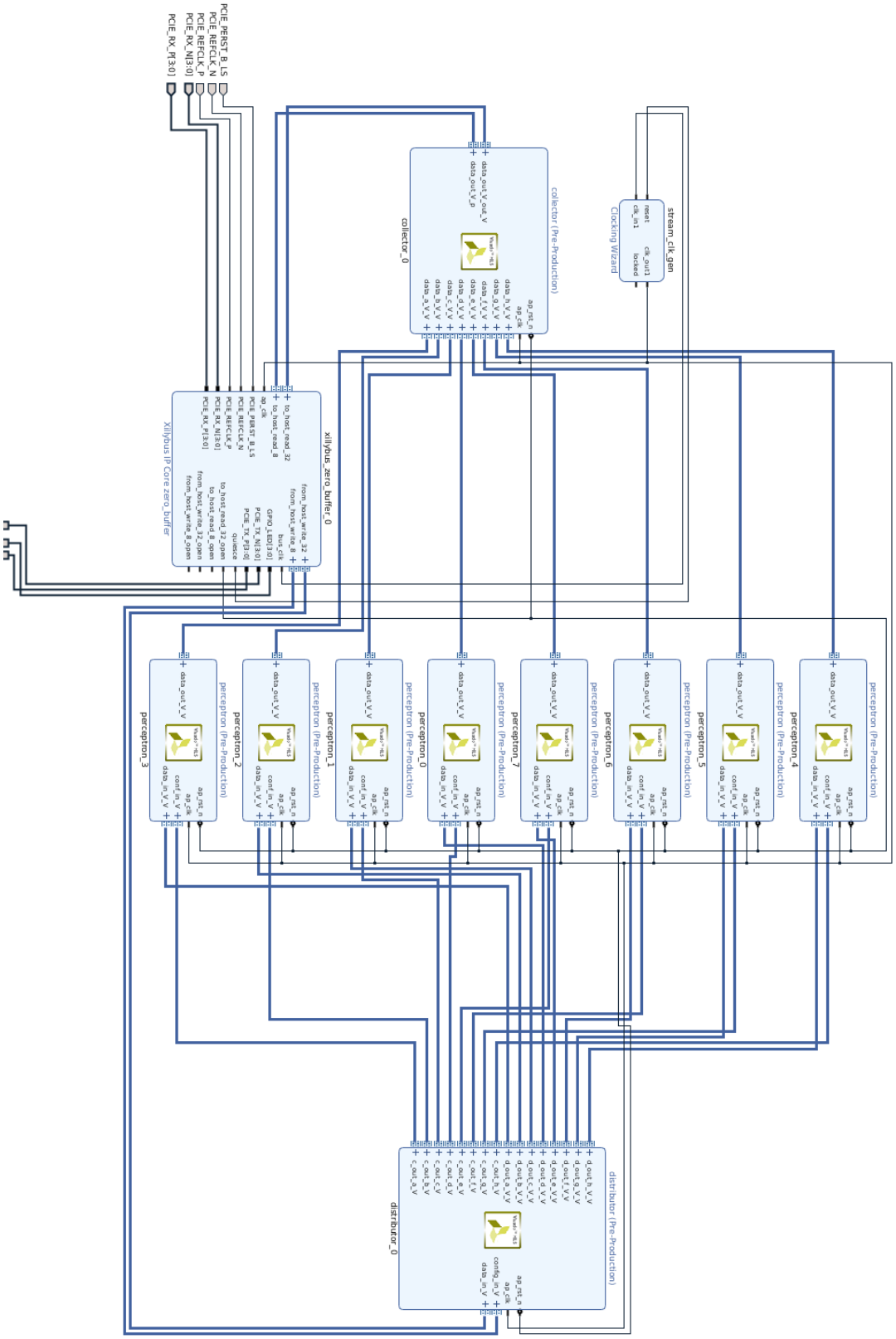

Abbildung 4.2: Das Blockdesign auf dem FPGA

Kapitel 5

Experimente und Ergebnisse

Kapitel 6

Ausblick und Fortsetzung

Feature Selection mit Daten-Streams[9]

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] Dillkötter, Fabian: *Umsetzung einer High-Performance FPGA-Schnittstelle für maschinelles Lernen*, 2019.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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 27. Januar 2020

Muster Mustermann