



Karlsruhe Institute of Technology  
Fakultät für Elektrotechnik



Institut für Technik der  
Informationsverarbeitung (ITIV)

# Implementierung und Evaluation von Systolic Arrays für maschinelles Lernen

Bachelorarbeit von

**cand. el. Ahmet Narin**

**5 Februar 2020**

**Institutsleitung:** Prof. Dr.-Ing. Dr. h.c. J. Becker  
Prof. Dr.-Ing. Eric Sax  
Prof. Dr. rer. nat W. Stork

**Betreuer:** M. Sc. Tim Hotfilter

## **Zusammenfassung**

Unser Alltag und Geschäftsleben wird immer mehr von Machine Learning (ML) beeinflusst. Beispiele für den Einsatz begegnen uns im täglichen Leben bereits überall. Einige davon sind Gesichts- und Spracherkennung, personalisierte Produktempfehlungen bei Amazon, Börsenprognose für Spekulanten etc. Aber auch in sicherheitskritischen Systeme wie autonomes Fahren wird das ML-Verfahren eingesetzt. Das Ziel von dieser Arbeit ist, dass die Datenverarbeitung von solchen sicherheitskritischen Anwendungen beschleunigt werden, indem die Hardware in der Lage sind, um parallele Rechenoperationen auszuführen. Um dieses Ziel zu erreichen wird die Systolic Array Architektur implementiert. Somit wird die Rechenleistung von Embedded Systems und IoT-Geräten erhöht. Dabei muss man den Stromverbrauch nicht ignorieren. Der Stromverbrauch von solchen Geräten spielt genauso wichtige Rolle wie Beschleunigung von Hardware. Denn die superschnelle aber stromverschwenderische Hardware sind in mobilen Geräten nicht einsetzbar.

# Urheberrecht

ARM®, AMBA®, AXI™, Cortex™, TrustZone™, SecurCore™, DSTREAM™ und weitere im Text erwähnte ARM-Produkte sowie die entsprechenden Logos sind Marken oder eingetragene Marken der Advanced RISC Machines Ltd.

Xilinx®, Zynq™ und weitere im Text erwähnte Xilinx-Produkte sowie die entsprechenden Logos sind Marken oder eingetragene Marken der Xilinx Inc.

## Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit (bzw. Masterarbeit) selbstständig und unter Beachtung der Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) in der aktuellen Fassung angefertigt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich übernommene Stellen als solche kenntlich gemacht.

Karlsruhe, den 5 Februar 2020



# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>2</b>
1.1 Motivation . . . . .	2
1.2 Zielsetzung . . . . .	3
1.3 Das Mooresche Gesetz . . . . .	3
<b>2 Grundlagen</b>	<b>4</b>
2.1 Matrix-Matrix Multiplikation . . . . .	4
2.2 1D-Faltung und 2D-Faltung . . . . .	5
2.2.1 1D-Faltung bzw. Faltung . . . . .	5
2.2.2 2D-Faltung . . . . .	5
2.3 Maschinelles Lernen . . . . .	6
2.4 Lernphase . . . . .	7
2.4.1 Supervised Learning . . . . .	7
2.4.2 Unsupervised Learning . . . . .	7
2.4.3 Reinforcement Learning . . . . .	8
2.5 Kuenstliche neuronale Netze . . . . .	8
2.5.1 Schwellenwertfunktion . . . . .	10
2.5.2 Stückweise lineare Funktion . . . . .	10
2.5.3 Sigmoidfunktion . . . . .	11
2.5.4 Rectifier (ReLU) . . . . .	11
2.6 Strukturen der kuenstlichen neuronalen Netzwerke . . . . .	12
2.6.1 Einlagiges Perzeptron . . . . .	13
2.6.2 Mehrschichtiges feedforward-Netz . . . . .	14
2.6.3 Convolutional Neural Network . . . . .	15
2.6.4 Beispiele von CNN Architekturen . . . . .	19
<b>3 Konzept</b>	<b>22</b>
3.1 Einfuehrung der Systolic Array Architektur . . . . .	22
3.2 Why Systolic Architectures? . . . . .	23
3.2.1 Grundidee . . . . .	23
3.2.2 Systolic Array Designs für Faltung Berechnungen . . . . .	24
3.2.3 Systolic Array Matrix Multiplikation . . . . .	26
3.2.4 Architektur der Tensor Processing Unit . . . . .	28
3.2.5 Vorteile des Systolic Arrays . . . . .	30
3.2.6 Nachteile des Systolic Arrays . . . . .	30

<b>4</b>	<b>Implementierung</b>	<b>32</b>
4.1	Implementierungsplan . . . . .	32
4.1.1	MAC . . . . .	33
4.1.2	RAM . . . . .	33
4.1.3	Delay Controller . . . . .	33
4.1.4	Register in MAC . . . . .	34
4.1.5	Simulationsergebnisse . . . . .	35
4.1.6	Zusammenfassung . . . . .	36
<b>5</b>	<b>Evaluation</b>	<b>37</b>
5.0.1	Gemmini: Ein agiles systolic Array Generator . . . . .	37
5.0.2	Die Fläche und der Stromverbrauch von Gemini-Design . . . . .	39
<b>6</b>	<b>Ausblick</b>	<b>41</b>

# Kapitel 1

## Einleitung

### 1.1 Motivation

Neue innovative Projekte wie autonomes Fahren oder auch kommenden neue Technologien in der Zukunft und industrielle Automatisierung müssen immer mehr mit umfangreicher Datenmenge umgehen. ML-Methoden werden in solchen Anwendungen häufig verwendet, damit die Daten analysiert werden können bzw. die Maschinen etwas neues aus den Daten ohne explizite Programmierung lernen. Insbesondere im Kontext von Echtzeitsystemen stellt sich die Latenz während der Datenverarbeitung als Flaschenhals dar.

Die CPU-Hardwarearchitektur kann dieses Problem leider nicht lösen, da CPUs für allgemeinen Gebrauch vorgesehen sind. GPU (graphics processing unit) ist für solchen rechenintensiven Anwendungen besser geeignet, um die Datenverarbeitung zu beschleunigen. Aber sie verbrauchen viel Energie und für mobile Geräte und Embedded Systems nicht einsetzbar. Eine spezielle Hardwarearchitektur soll für diesen Zweck eingesetzt werden. Es soll wenig Strom verbrauchen und trotz dieser Anforderung auch schnell und mehrfache vorkommende Multiplikation-und Addition Operationen bearbeiten.

Das **FPGA** (field-programable gate array) kann diese Anforderungen erfüllen. Das FPGA besteht aus internen Hardware-Blöcken, die mit anderen von einer Fachperson programmiert werden können, um eine spezielle Anforderungen zu erfüllen. Der Hauptvorteil von FPGA im Gegensatz zu den anderen Hardwarearchitekturen ist, dass die Verbindungen von internen Hardware-Blöcken leicht umprogrammiert und während dem Einsatz Änderungen oder Verbesserungen vorgenommen werden. Verschiedene ML-Algorithmen können damit implementiert und evaluiert werden.

## 1.2 Zielsetzung

In der vorliegenden Arbeit geht es um eine Entwicklung von Systolic Array Architektur, die auf einem FPGA implementiert wird. Das FPGA ist klein, verbraucht wenig Energie und ermöglicht dem ML-Modell eine höhere Rechenleistung und somit eine schnelle Ausführung. Außerdem wird Anwendungen von Systolic Array wie TPU [JY17] und ihre Implementierung untersucht.

### 1.3 Das Mooresche Gesetz

Das Mooresche Gesetz besagt, dass sich die Anzahl der Transistoren, die auf eine integrierte Schaltung gedrückt werden können ungefähr alle 18 Monate verdoppelt. [Wik] siehe Abbildung 1.1 Das ist kein Naturgesetz, sondern eine Faustregel, die auf eine empirische Beobachtung zurückgeht. Milliarden von Transistoren auf den neusten Chips sind jetzt schon unsichtbar für das menschliche Auge. Wenn man das Mooresche Gesetz ins unendliche betrachten möchte, dann müssen die Transistoren ca. im Jahr 2050 aus Bauteilen hergestellt werden, die kleiner als ein einziges Wasserstoffatom sind. Die Herstellung von solchen Transistoren ist physikalisch unmöglich. Die Investitionskosten für Unternehmen wird immer teurer, um mehr Transistoren auf einen winzig kleinen Flächen zu drücken. Es muss einen Ausweg gefunden werden, der ohne großen Investitionsbedarf dieses Problem löst. In dieser Arbeit wird bemüht, wie man Hardware für ML-Methoden beschleunigen kann.

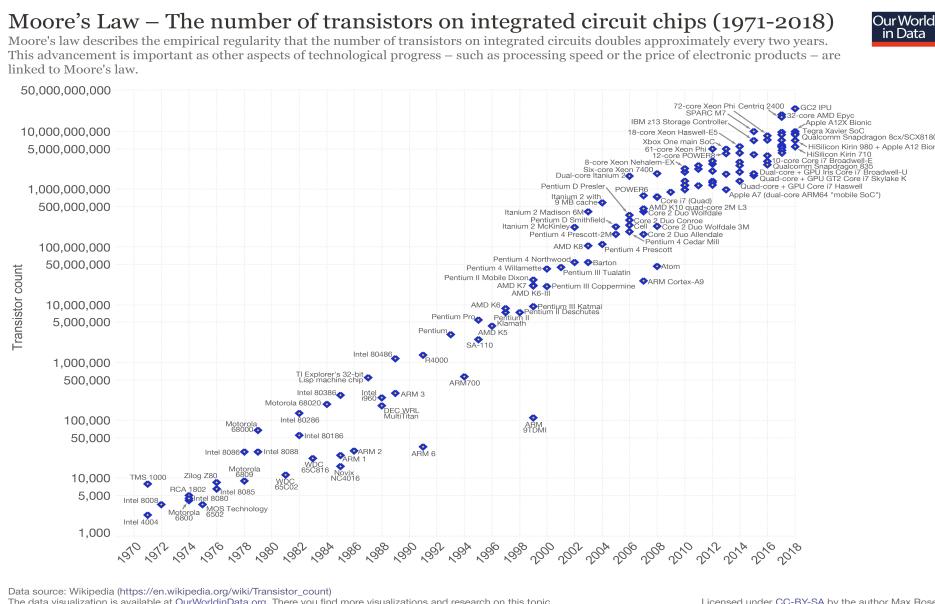


Abbildung 1.1: Mooresches Gesetz von 1971-2018 [Wik]

# Kapitel 2

## Grundlagen

### 2.1 Matrix-Matrix Multiplikation

Bei der Matrix-Matrix Multiplikation geht es um zwei Matrizen, die miteinander multipliziert werden. Aber nicht jede Matrizen sind miteinander multiplizierbar. D.h die Spaltenmatrix der ersten Matrix muss mit der Zeilenzahl der zweiten Matrix übereinstimmen. Das Ergebnis ist auch eine Matrix, die als Produktmatrix genannt wird.

Gegeben seien zwei Matrizen. A ist eine  $m \times n$ -Matrix.

B ist eine  $n \times p$ -Matrix.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{np} \end{pmatrix}$$

Die Produktmatrix C ergibt sich:

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1p} \\ c_{21} & c_{22} & \dots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mp} \end{pmatrix}$$

Bei einer MM-Multiplikation finden  $mpn$ -Multiplikationen und  $mp(n - 1)$ -Additionen statt.

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$$

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \dots + a_{1n}b_{n1} & a_{11}b_{12} + \dots + a_{1n}b_{n2} & \dots & a_{11}b_{1p} + \dots + a_{1n}b_{np} \\ a_{21}b_{11} + \dots + a_{2n}b_{n1} & a_{21}b_{12} + \dots + a_{2n}b_{n2} & \dots & a_{21}b_{1p} + \dots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \dots + a_{mn}b_{n1} & a_{m1}b_{12} + \dots + a_{mn}b_{n2} & \dots & a_{m1}b_{1p} + \dots + a_{mn}b_{np} \end{pmatrix}$$

## 2.2 1D-Faltung und 2D-Faltung

### 2.2.1 1D-Faltung bzw. Faltung

Die Faltung mit eindimensionalen Signalen wird als 1D-Faltung oder nur Faltung bezeichnet. 1D-Faltung ist gut geeignet für Analyse einer Zeitreihe von Sensordaten oder von Signaldaten wie Tonaufnahme über einen Zeitraum fester Länge. In der digitalen Bildverarbeitung und Signalverarbeitung findet meistens diskrete Faltung statt. Die Faltung von zwei diskreten Funktionen wird durch folgende Formel berechnet:

$$f[n] = a[n] * b[n] = \sum_{k=-\infty}^{\infty} a[k]b[n-k]$$

Die Faltung erfolgt durch Multiplizieren und Akkumulieren der Momentanwerte der überlappenden Abtastwerte. Dabei soll ein Signal umgedreht sein.

### 2.2.2 2D-Faltung

Dieses Grundkonzept für 1D-Faltung gilt auch für die 2D-Faltung, wenn die Signale 2 Dimensionen haben. Analog kann die Faltung in 2D definiert werden.

$$f[x, y] = a[x, y] * b[x, y] = \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} a[j, k]b[x-j, y-k]$$

2D-Faltung ist ähnlich wie eine Matrix-Multiplikation und ziemlich eine einfache Operation. Hier wird ein Kernel gebraucht. Dieser Kernel ist eine Matrix von Gewichten und gleitet über die 2D-Eingangsmatrix. Dabei wird eine elementweise Multiplikation mit dem Teil der Eingabe durchgeführt, auf dem der Kernel sich befindet. siehe Abbildung 2.1 Alle Ergebnisse nach der Multiplikation akkumuliert bzw. addiert zu einem einzelnen Ausgabepixel.

#### 2.2.2.1 Beispiel für 2D-Faltung

Hier ist ein Beispiel für 2D-Faltung. Es gibt ein 7x7 Eingangsmatrix I (inputmatrix) und eine Filtermatrix K (kernel) 3x3. Die Werte der Ausgangsmatrix werden durch Multiplikation und Addition von entsprechenden Pixelwerten von I und K berechnet. Siehe Abbildung 2.1

$$\begin{array}{c}
 \left( \begin{array}{ccccccccc}
 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 1 & 1 & 0 & 0 & 0 & 0 & 0
 \end{array} \right) * \left( \begin{array}{ccc}
 1 & 0 & 1 \\
 0 & 1 & 0 \\
 1 & 0 & 1
 \end{array} \right) = \left( \begin{array}{ccccc}
 1 & 4 & 3 & 4 & 1 \\
 1 & 2 & 4 & 3 & 3 \\
 1 & 2 & 3 & 4 & 1 \\
 1 & 3 & 3 & 1 & 1 \\
 3 & 3 & 1 & 1 & 0
 \end{array} \right) \\
 I \qquad \qquad \qquad K \qquad \qquad \qquad I * K
 \end{array}$$
  

$$\begin{array}{c}
 \left( \begin{array}{ccccccccc}
 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 1 & 1 & 0 & 0 & 0 & 0 & 0
 \end{array} \right) * \left( \begin{array}{ccc}
 1 & 0 & 1 \\
 0 & 1 & 0 \\
 1 & 0 & 1
 \end{array} \right) = \left( \begin{array}{ccccc}
 1 & 4 & 3 & 4 & 1 \\
 1 & 2 & 4 & 3 & 3 \\
 1 & 2 & 3 & 4 & 1 \\
 1 & 3 & 3 & 1 & 1 \\
 3 & 3 & 1 & 1 & 0
 \end{array} \right) \\
 I \qquad \qquad \qquad K \qquad \qquad \qquad I * K
 \end{array}$$
  

$$\begin{array}{c}
 \left( \begin{array}{ccccccccc}
 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 1 & 1 & 0 & 0 & 0 & 0 & 0
 \end{array} \right) * \left( \begin{array}{ccc}
 1 & 0 & 1 \\
 0 & 1 & 0 \\
 1 & 0 & 1
 \end{array} \right) = \left( \begin{array}{ccccc}
 1 & 4 & 3 & 4 & 1 \\
 1 & 2 & 4 & 3 & 3 \\
 1 & 2 & 3 & 4 & 1 \\
 1 & 3 & 3 & 1 & 1 \\
 3 & 3 & 1 & 1 & 0
 \end{array} \right) \\
 I \qquad \qquad \qquad K \qquad \qquad \qquad I * K
 \end{array}$$

Abbildung 2.1: Visualisierung von 2D Faltung

## 2.3 Maschinelles Lernen

Beim ML wird das System so programmiert, damit das aus eingegebenen Daten automatisch lernt und sich mit der ‘‘Erfahrung’’ verbessert. Das heißt, das System wird mit Trainingsdaten trainiert. Hier bedeutet das Lernen: die eingegebenen Daten zu erkennen, verstehen und auf der Grundlage der gelieferten Daten eine sinnvolle Entscheidung zu treffen. Aber nicht alle Entscheidungen können auf der Grundlage aller möglichen Eingaben berücksichtigt werden. Um dieses Problem zu lösen, werden hier Algorithmen entwickelt, die das maschinelles Lernen optimieren und schneller machen.

## 2.4 Lernphase

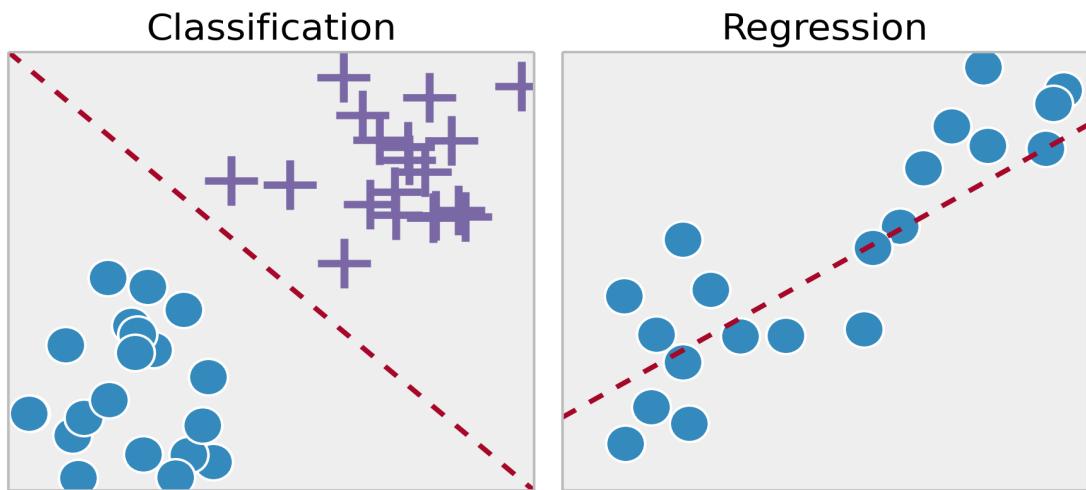
### 2.4.1 Supervised Learning

Beim supervised Learning brauchen die Trainingsdaten Label (Lösungen). D.h. man muss als Entwickler dem Modell vorher sagen, was die Lösung ist. Wenn man beispielsweise einem Bilderkennner beibringen will, Hunde- und Katzenbilder zu unterscheiden, muss vorher ein Mensch alle Trainingsbilder anschauen und notieren was zu sehen ist. Hier ist der Entwickler als "Lehrer" vorgesehen, der das ML-Modell beibringt. Ansonsten weiss der Algorithmus nicht, ob er falsch oder richtig entscheidet.

#### Anwendungen

**Bildklassifizierung** In der Zukunft wird von dem System erwartet, dass es ein neu gegebenes Bild erkennt.

**Marktvorhersage** Das System wird von Werten aus der vergangenen Marktdaten trainiert und wird erwartet, dass es den Preis für die Zukunft vorhersagt.

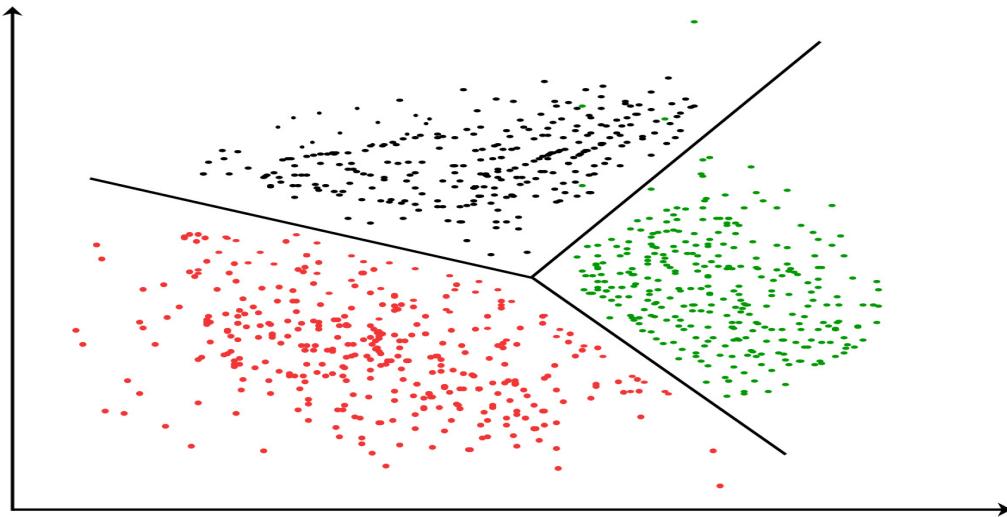


### 2.4.2 Unsupervised Learning

Beim Unsupervised Learning soll das ML-Modell aus Daten "lernen", die Bedeutungen von denen noch unbekannt sind. Hier wird dem Algorithmus die Daten ohne Zielvorgabe eingespeist und trainiert. Ohne Daten gibt es natürlich nichts zu "lernen". Unsupervised Learning wird verwendet, um einen bestimmten Datensatz in verschiedene Gruppen zu gruppieren. Dies wird häufig verwendet, um Kunden für bestimmte Eingriffe in verschiedene Gruppen zu segmentieren.

## Anwendungen

**Clustering** Ähnliche Daten werden in Cluster bzw. Gruppen eingeteilt, um die Gruppierung für ein Zweck zu nutzen. Dies ist in Forschung und Wissenschaft eingesetzt.



### 2.4.3 Reinforcement Learning

Bei der reinforcement Learning geht es darum, dass der Agent (hier: Software-Bot) in einer unbekannten und komplexen Umgebung sein Ziel erreicht. Dabei wird der Agent für richtiges Aktion belohnt und für falsches Aktion bestraft. Sein Ziel ist es, die Gesamtbelohnung zu maximieren bzw. Strafen zu minimieren. Der Unterschied zwischen Supervised-und Reinforcement Learning ist es, dass der Agent keine Hinweise oder Vorschläge zur Lösung bekommt. Die Lösung ist unbekannt und er versteht nur aus seiner Aktion, ob er falsch oder richtig liegt.

## 2.5 Kuenstliche neuronale Netze

Kuenstliche neuronale Netze sind eine Technik des ML, die den Mechanismus des Lernens in biologischen Organismen simulieren. [Hay94] Das menschliche Nervensystem enthält Zellen, die als Neuron bezeichnet werden. Jedes Neuron besteht aus drei Teilen: Es sind die Dendriten, der Zellkörper und das Axon. Abbildung 2.2 Aus dem Zellkörper verzweigen sich eine Reihe von Fasern ab, die Dendrite genannt werden. Die lange Faser wird Axon genannt und streckt sich nach großen Entfernung. Es kann von 1 cm bis 1 Meter lang sein.

Ein Neuron kann mit anderen Neuronen 10 bis 100.000 Verbindungen herstellen, die als

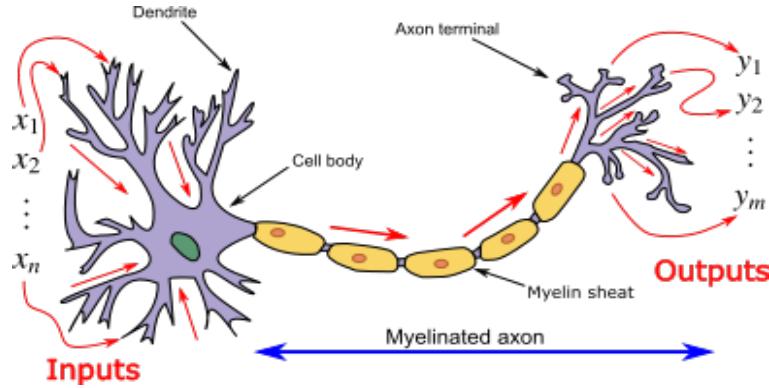
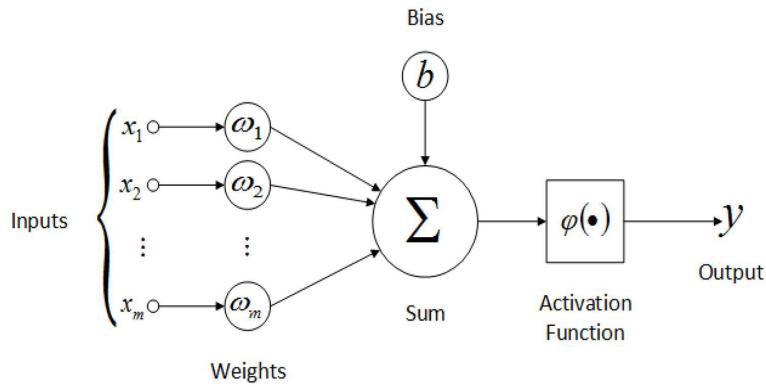


Abbildung 2.2: Aufbau eines Neurons

Abbildung 2.3: Die Eingänge  $x_1, x_2, \dots, x_m$  sind mit Gewichten  $w_1, w_2, \dots, w_m$  multipliziert und diese Werte werden durch Übertragungsfunktion summiert und Bias hinzugefügt. Die Aktivierungsfunktion hat dann den Ausgangswert bestimmt.

Synapsen bezeichneten werden. Die Kommunikationen zwischen Neuronen erfolgen mit elektrochemischen Signalen.

Dieser biologische Mechanismus wird in kuenstlichen neuronalen Netzen übertragen. Ein Neuron ist eine Einheit für die Informationsverarbeitung, das sich die Grundlage einer neuronaler Netze bildet.

**Gewichtung  $w_i$ :** Sie bestimmen den Grad, wie stark die Eingänge der Entscheidung des Neurons beeinflusst. Ein Gewicht mit dem Wert 0 bedeutet, dass es keine Verbindung zwischen Neuronen existiert.

**Übertragungsfunktion:** Die Netzeingabe wird durch die Übertragungsfunktion  $\sum$  von einzelnen Eingangs値, die mit Gewichte multipliziert sind, berechnet.

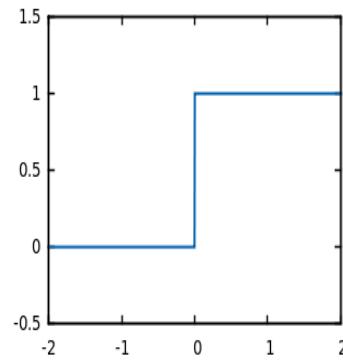
**Bias**: Die Übertragungsfunktion enthält noch ein externer Wert, die mit Bias  $b$  bezeichnet ist. Das kann auch positive und negative Werte einnehmen und hat Einfluss auf den Eingangswert der Aktivierungsfunktion. Damit wird dieser Wert erhöht oder auch gesenkt werden.

**Aktivierungsfunktionen:** Eine Aktivierungsfunktion ist eine mathematische Funktion und entscheidet, ob ein Neuron aktiviert werden soll oder nicht, indem sie die gewichtete Summe berechnet und eine Bias hinzufügt. Diese Funktionen sind:

### 2.5.1 Schwellenwertfunktion

Die Schwellenwertfunktion (engl. heaviside step function oder auch binary step function) nimmt nur die Werte 0 und 1 an. Liegt der Eingabewert über oder unter einem bestimmten Schwellenwert, wird das Neuron aktiviert und sendet das Eingangssignal. Für die Eingabe  $e \geq 0$  nimmt der Ausgangswert 1, ansonsten 0.

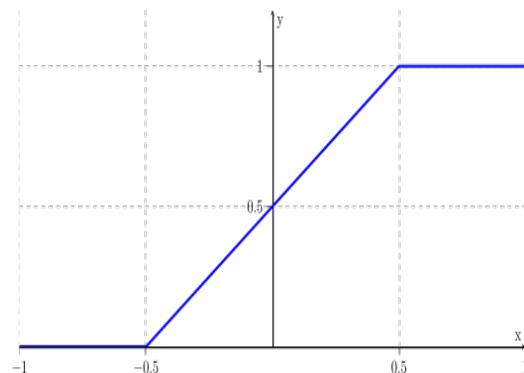
$$\varphi(e) = \begin{cases} 0 & \text{wenn } e < 0 \\ 1 & \text{wenn } e \geq 0 \end{cases}$$



### 2.5.2 Stückweise lineare Funktion

Stückweise lineare Funktion läuft in einem begrenzten Intervall linear ab. Außerhalb des Intervalls besitzt die Funktion einen konstanten Wert.

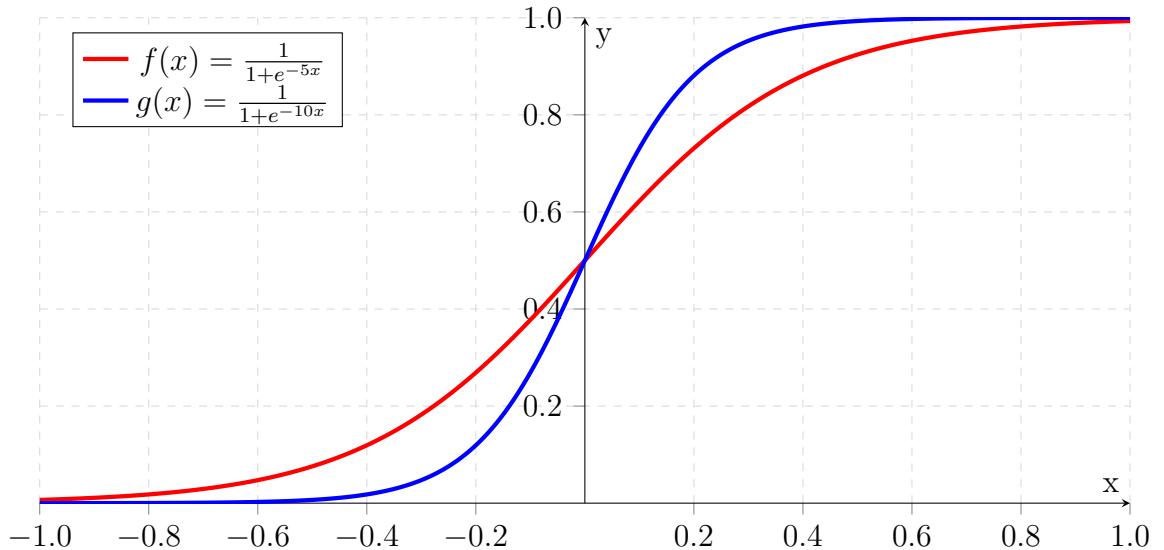
$$\varphi(e) = \begin{cases} 0 & \text{wenn } e \leq -\frac{1}{2} \\ e + 1/2 & \text{wenn } -\frac{1}{2} < e < \frac{1}{2} \\ 1 & \text{wenn } e \geq \frac{1}{2} \end{cases}$$



### 2.5.3 Sigmoidfunktion

Sigmoide Funktionen sind sehr häufig verwendete Funktionen. Sie besitzen ein variables Steigungsmaß  $\alpha$ , dass die Krümmung des Funktionsgraphen beeinflusst. Differenzierbarkeit der Sigmoidfunktion ist für einige ML-Verfahren wie Backpropagation-Algorithmus ein Vorteil. Der Nachteil ist ein starker negativer Eingangssignal, damit kann es während des Trainings hängen bleiben.

Sie ist definiert durch:

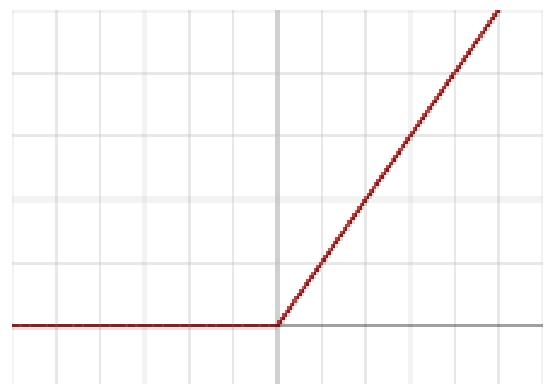


$$\varphi_{\alpha}^{sig}(e) = \frac{1}{1 + \exp(-\alpha e)}$$

### 2.5.4 Rectifier (ReLU)

Es wird besonders in Deep-Learning Modellen wie CNN( Convolutional Neural Networks) eingesetzt. Dies ist auch als Rampenfunktion bekannt. ReLU Funktion ist definiert durch:

$$\varphi(e) = \max(0, e)$$



**Kurze Zusammenfassung:** Ein Neuron Abbildung 2.4 besitzt viele Eingaben und jede Eingabe des Neurons wird mit einem Gewicht versehen. Die gewichteten Eingaben werden dann durch eine Übertragungsfunktion akkumuliert und daraus resultierende Netzeingabe in eine Aktivierungsfunktion gegeben, die die Ausgabe  $a_j = g(\sum_{i=0}^n w_{m,j} a_i)$  bestimmt.

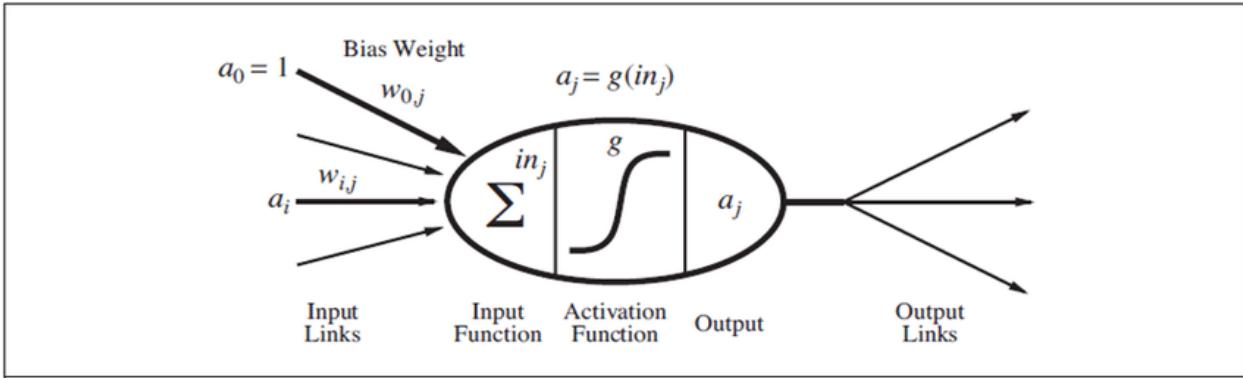


Abbildung 2.4: Ein vereinfachtes mathematisches Modell für ein künstliches Neuron

## 2.6 Strukturen der kuenstlichen neuronalen Netzwerke

Modelle der künstlichen neuronalen Netzwerke können als eine Reihe von grundlegenden Verarbeitungseinheiten verstanden werden, die miteinander eng verbunden sind. Das Ziel ist, die eingegebenen Daten ins neuronale Netz so verarbeiten, um gewünschten Ausgaben zu erzeugen.

In diesem Abschnitt wird die grundlegende Architektur von neuronalen Netzwerken diskutiert. Es gibt einschichtiges- (**single-layer**) und mehrschichtiges (**multi-layer**) neuronales Netz. In einem einschichtigen Netz wird der Ausgang direkt über die Aktivierungsfunktion mit den Eingängen verbunden. Dies wird in der Fachsprache auch als Perzeptron genannt. Das mehrschichtige Netz besteht aus Eingangs- und Ausgangsschicht engl. (**input- and output layers**) und zwischen diesen Schichten befinden sich die verdeckten Schichten engl. (**hidden-layer**). Die Informationen, die im neuronalen Netz verbreiten, können in zwei Kategorien eingeteilt werden.

**feedforward-Netze** Der Informationsfluss in einem feedforward-Netz erfolgt nur in eine Richtung. Wenn man das Netz als Graph und Neuronen als Knoten betrachtet, dann darf das Graph keine Schleifen oder Zyklen enthalten. Mehrschichtiges Netz und convolutional neural Network (CNN) sind gute Beispiele für feedforward-Netze. [SKB18]

**feedback-Netze** Die feedback-Netze können Zyklen oder auch Schleifen enthalten. Die Auszeichnung des feedback-Netzes ist die Errinnerungsfähigkeit und kann Informationen und Daten speichern. Beispiele für solche Architekturen sind Recurrent neural Network und Long-Short-Term-Memory (LSTM).

### 2.6.1 Einlagiges Perzeptron

Ein einlagiges Perzeptron enthält eine einzelne Eingabeebene und einen Ausgabeknoten. Die Eingabewerte  $x_1, x_2, x_3, \dots, x_n$  werden mit ihren jeweiligen Gewichten  $w_1, w_2, w_3, \dots, w_n$  multipliziert. Dieser gesamte Ausdruck wird innerhalb des Neurons summiert. Nach der Berechnung der Gesamtsumme kann auch ein Biaswert  $b$  addiert werden. Der Ausgangswert ist  $y = [\sum_{i=1}^n w_i x_i + b], y \in (-1, +1)$ . [Agg18] Das Ziel eines Perzeptrons ist es, mit den extern angelegten Werten  $x_1, x_2, x_3, \dots, x_n$  in eine von zwei Klassen,  $c1$  oder  $c2$ , richtig zu klassifizieren. Die Entscheidungsregel für die Klassifizierung erfolgt bei eingegebenen Werten  $x_1, x_2, x_3, \dots, x_n$  dann, wenn der Ausgangswert  $y = 1$  ist, Klasse C1 einordnen, wenn aber  $y = -1$  ist, dann Klasse C2 einordnen. Die Entscheidungsregionen werden durch Hyperebene eng. *hyperplane* getrennt, die mit  $\sum_{i=1}^n w_i x_i + b = 0$  Abbildung 2.5 definiert ist. [Hay94]

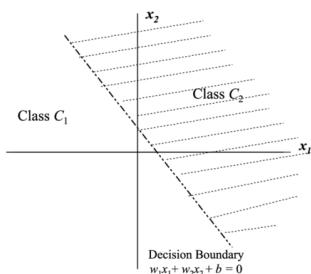
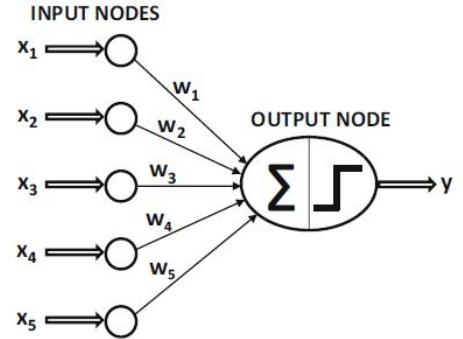
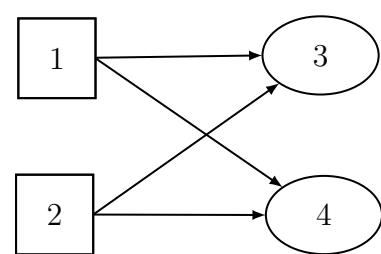


Abbildung 2.5: Entscheidungsregionen



Einschichtige Netze sind die einfachsten Strukturen der künstlichen neuronaler Netze, die auch als Rosenblatts Perzeptron bezeichnet wird. Die Arbeit von Frank Rosenblatt [Ros] wurde im Jahr 1958 veröffentlicht und sein Ziel war den Prognosefehler des Perzeptrons zu minimieren. Der Algorithmus wurde heuristisch entworfen, um die Anzahl von der falschen Klassifizierung zu minimieren.

$x_1$	$x_2$	$y_3$ (carry)	$y_4$ (sum)
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0



Wir wollen jetzt ein Perzeptron "beibringen", dass das 2-Bit addieren lernt. Oben links steht die Wertetabelle für ein Halbaddierer. Es gibt zwei Eingänge und 2 Ausgänge. Ein

Ausgang steht für ***carry*** und der zweite Ausgang für Summe ***sum***. Auf der rechten Seite sieht man die Einheiten. Die Einheit **3** ist für ***carry*** und die **4.** Einheit ist für die Summe ***sum***. Die **Einheit 3** lernt die carry-Funktion leicht, weil es hier lediglich um **AND**-Operator geht. Aber die **Einheit 4** scheint nicht zu funktionieren. Die Einheit soll **XOR** (*exklusive ODER*) lernen. Das Perzeptron kann aber **XOR** (*exklusive ODER*) nicht lernen, weil die nicht linear trennbar ist. Abbildung 2.6 [RN16]

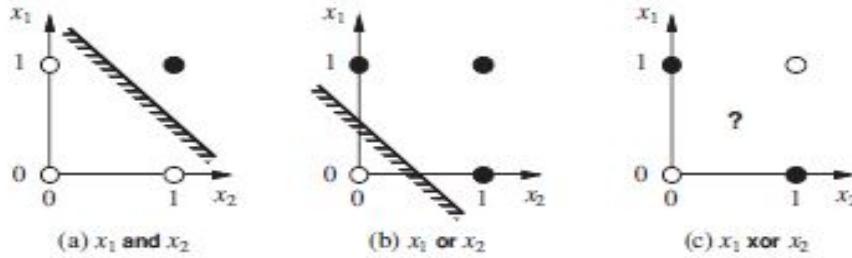


Abbildung 2.6: Entscheidungsgrenzen von logischen Gattern

## 2.6.2 Mehrschichtiges feedforward-Netz

Das einlagige Perzeptron oder auch Rosenblatts Perzeptron enthält keine versteckten Neuronen somit kann nicht linear trennbare Eingabemuster klassifizieren. siehe Abbildung 2.6 **XOR-Problem** Das XOR-Problem oder allgemeine Beschränkung des einlagigen Perzeptrons konnte mit dem mehrlagigen Perzeptron gelöst werden, bei dem es neben der Ausgabeschicht auch noch weitere Schicht verdeckter Neuronen engl. *hidden layers* gibt. Beim feedforward-Netz verläuft der Informationsfluss nur in einer Richtung. Ein rekurrentes neuronales Netz ist auch möglich, falls die Neuronen im Netz, die mit Neuronen der vorangegangenen Schicht verbunden sind. Es gibt:

**Fully connected** Die Neuronen einer Schicht werden mit allen Neuronen der nächsten folgenden Schicht verbunden. Abbildung 2.7

**Short-Cuts** Einige Neuronen einer Schicht werden nicht mit allen Neuronen der nächsten folgenden Schicht verbunden, sondern mit Neuronen im übernächsten Schichten.

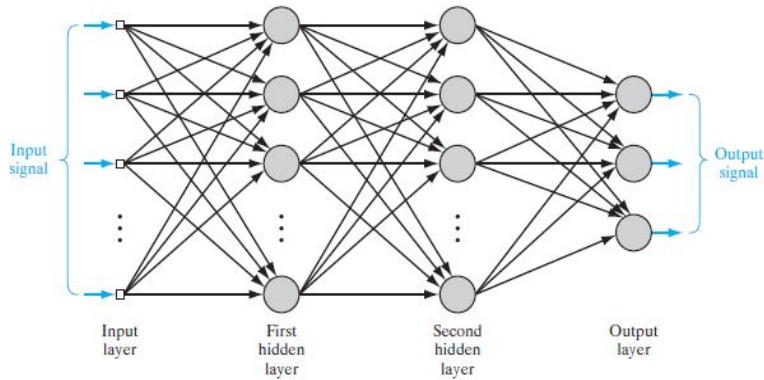


Abbildung 2.7: Struktur eines mehrschichtigen feedforward-Netzes

### 2.6.3 Convolutional Neural Network

Convolutional Neural Network (**CNN** oder **ConvNet**) ist eine beliebteste Technik der künstlichen neuronalen Netzwerke (KNN) für hochdimensionale Daten wie z.B. Bilder und Videos. Ein wesentlicher Unterschied zwischen CNN und allgemeine neuronalen Netzen besteht darin, dass jede Einheit in einer CNN-Schicht ein 2-dimensionales oder hochdimensionales Filter gibt, das mit der Eingabe dieser Schicht gefaltet wird. D.h. ein CNN kann als Eingang in Form einer Matrix verarbeiten, aber MLP siehe Unterabschnitt 2.6.2 ist nicht in der Lage eine eingegebene Matrix zu verarbeiten, da MLP einen Vektor als Input benötigt, um die Matrix zu verarbeiten. Außerdem die Pixelwerte des Bildes müssen hintereinander ausgerollt werden (Flattening). Dadurch sind normale neuronale Netze sind nicht in der Lage, Objekte in einem Bild unabhängig von der Position zu erkennen. [SKB18]

**Ein-und Ausgänge** Wenn ein Computer ein Bild sieht, dann sieht er nicht wie Menschen Abbildung 2.8, sondern einer Reihe von Pixelwerten und zwar eine Matrix. Abbildung 2.9 Jede dieser Zahlen von Matrix erhält einen Wert zwischen 0 und 255. Die einzelne Zahlen beschreiben die Intensität des Pixels. Diese Zahlen haben keine Bedeutung bei der Bildklassifizierung. Wofür werden dann Pixelwerte gebraucht? Die Idee ist, dass der Computer aus den Pixelwerten die Wahrscheinlichkeit berechnet und entscheidet mithilfe von Wahrscheinlichkeitswert, was in dem Bild zu sehen ist. (z.B. 0.80 Hund, 0.15 Katzen 0.05 Vogel) In diesem Beispiel die Wahrscheinlichkeit bei Hunden ist größer als die anderen und entscheidet das CNN für "Hunden".



Abbildung 2.8: Was wir sehen [Des]

```

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 60 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48

```

Abbildung 2.9: Was der Computer sieht [Des]

### 2.6.3.1 Struktur von Convolutional Neural Network

Ein CNN besteht aus mehreren Schichten (*Convoutional Layer*) Abbildung 2.10, die grundlegende Funktionen wie Normalisierung, Pooling und Faltung besitzen. Die schichten wiederholen sich abwechselnd und am Ende der CNN-Schichten befindet sich ein fully connected Neuronen siehe Unterabschnitt 2.6.2

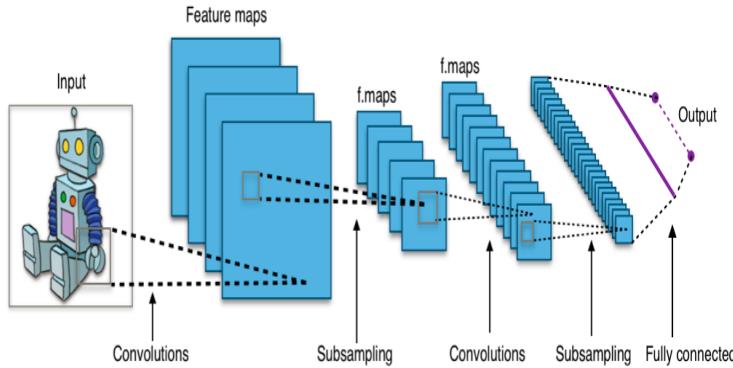


Abbildung 2.10: Struktur von Convolutional Neural Network CNN [Bec19]

### 2.6.3.2 Filter in Convolutional Layer

Eine Convolution Layer ist die wichtigste Komponente eines CNN. Die Eingabe einer Matrix (Array von Pixelwerte eines Bildes) wird zunächst von Filter in Convolutional Layers analysiert, die eine feste Pixelgröße (*Kernel-size*) besitzen. Über die Pixelwerte des Bildes bzw. Matrix wandern die Filter mit einem konstanten Schrittweite von Links nach Rechts. Am Rand der Zeile von Pixelwerte springen die Filter in die nächste tiefere Zeile. Mit dem sogenannten *Padding* wird festgelegt, wie sich die Filter am Rand der Matrix verhalten soll. Im Unterabschnitt 2.2.2 wird die Berechnungsmethode von Faltung erläutert.

In der ersten Ebene eines CNN befinden sich 16 oder 32 Filter und nach der Faltung entsteht eine neue Matrix als Output. Danach folgt ein Pooling layer, die auch in TPU Unterabschnitt 3.2.4 Abbildung 3.11 verwendet wird.

### 2.6.3.3 Pooling Layer

Es gibt zwei Arten von Pooling. Die sind Max-und Average (*Mittelwert*) Pooling. In CNN wird Max-Pooling angewendet. Der höchste Wert von der Kernel-Matrix weitergegeben und Rest ignoriert. Somit werden die relevantesten Signale für die nächsten Schichten ausgewählt und die Anzahl der Parameter eines Netzes reduziert. Abbildung 2.11

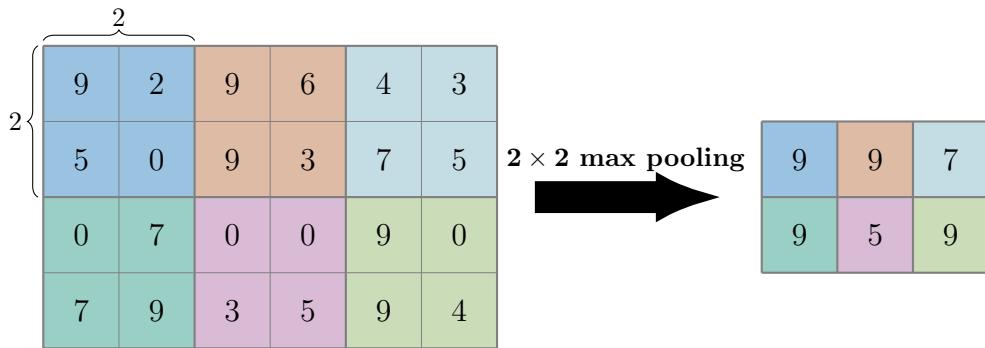
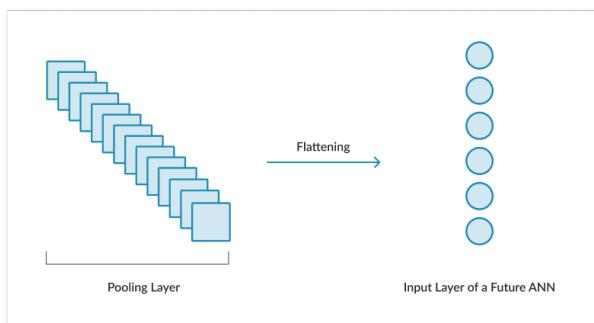


Abbildung 2.11: Hier ist die Kernel-Matrix 2x2, wird Maximum Wert aus dem Bereich ausgewählt und zur nächste 2x2-Matrix verschoben.

### 2.6.3.4 Fully connected Layer / Dense Layer

Es handelt sich hier um eine normale neuronale Netzstruktur. Die Neuronen sind fully-connected siehe Unterabschnitt 2.6.2. Alle Inputs und Outputs sind mit Neuronen verbunden. Um die Matrix aus dem Pooling Layer ins Netz speisen zu können, müssen sie zunächst ausgerollt werden (flatten). Fully Connected Layer kann die Matrix nicht verarbeiten, sondern nur die Werte der Matrix.



Zwischen convolutional layer und fully-connected layer befindet sich eine "flatten"-layer. Eine zweidimensionale Eingangsmatrix wird durch Abflachung in einen Vektor umgewandelt, damit sie ins neuronale Netz eingespeist werden können.

### 2.6.3.5 Zusammenfassung CNN

Ein CNN akzeptiert als Eingang eine Matrix, die reine Intensitätswerten eines Bildes bestehen. Die Verarbeitung verläuft durch mehreren Schichten, um Strukturen, wie Linien, Kanten, Kurven etc. eines eingegebenes Bildes zu erkennen. Die Filter in Schichten von CNN sind nicht vorgegeben, sondern vom Netz gelernt. In jeder höheren Filterebene erhöht sich die Abstraktions-Level des Netzes. Es ist sehr interessant, die Muster zu visualisieren, welche jeweils auf verschiedenen Ebenen zur Aktivierung der Filter führen. Abbildung 2.12

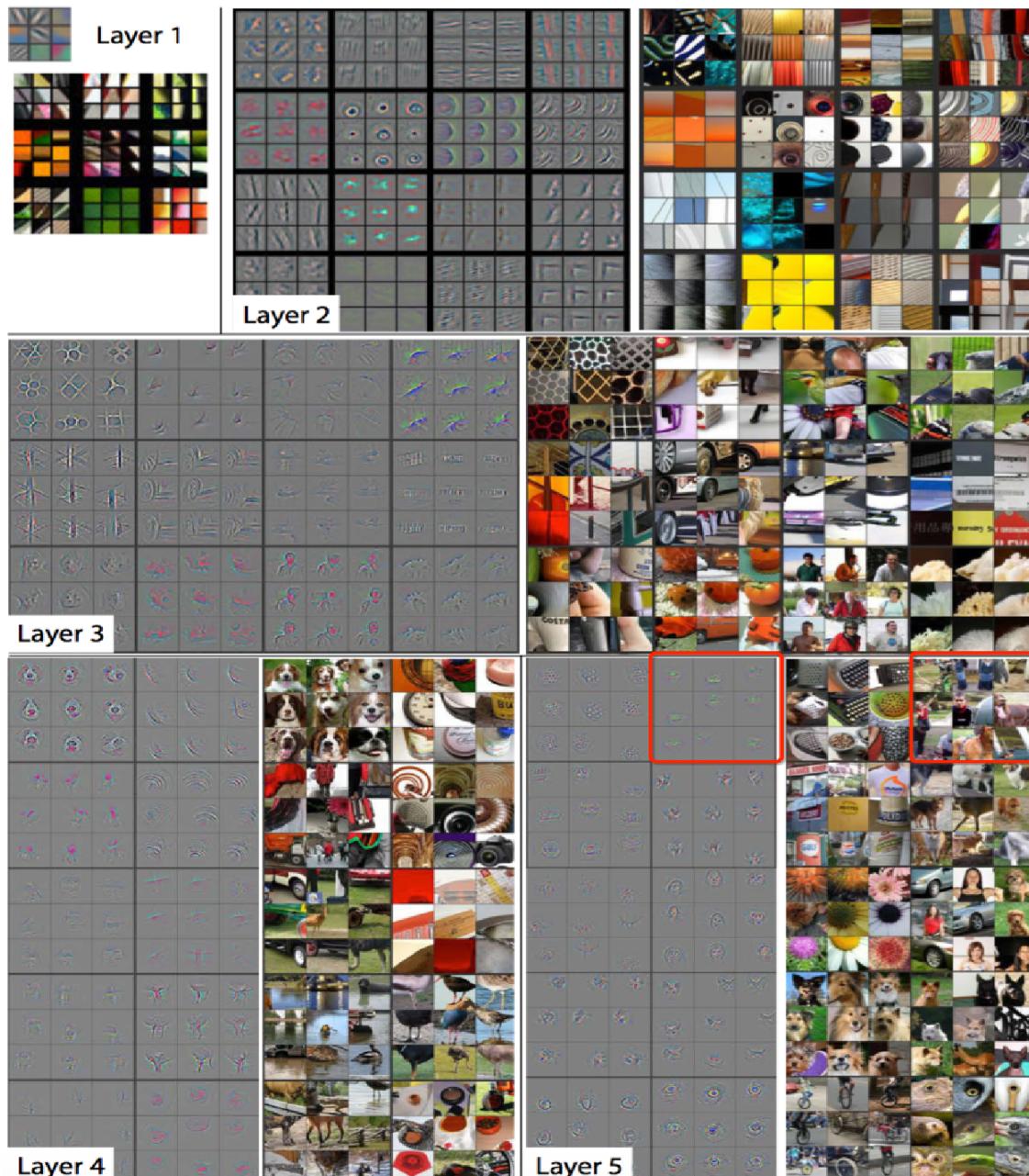


Abbildung 2.12: Visualisierung von Convolutional Network [ZF14]

### 2.6.4 Beispiele von CNN Architekturen

Hier werden einige erfolgreiche CNN-Entwürfe vorgestellt, die unter Grundlagen von CNN erstellt wurden. Die einige Grundlagen sind wie Faltungen, Max-Pooling, Funktion von Fully connected layer, flatening etc. schon oben erläutert. Zuerst möchte ich mit früheren Entwurf vorstellen und anschließend folgen dann die neue davon, sodass man den Trend bei der Entwicklung von CNN-Architekturen einen Überblick bekommt.

**LENET** Die LeNet-Architektur [LBBH98] ist eine der frühesten und grundlegendsten Formen von CNNs, die für die handschriftliche Ziffernidentifikation entwickelt wurde. [SKB18] Die erfolgreichste Variante dieser Architektur ist die LeNet-5 Modell, da es insgesamt 5 Gewichtsschichten umfasst. LeNet besteht aus zwei convolutional layer, denen jeweils eine (Max-Pooling)-Schicht folgt, um Merkmale zu extrahieren. Anschließend folgt eine einzelne convolution layer. Am Ende des Modells befinden sich dann fully-connected layer. Die Modellarchitektur ist in Abbildung 2.13 dargestellt.

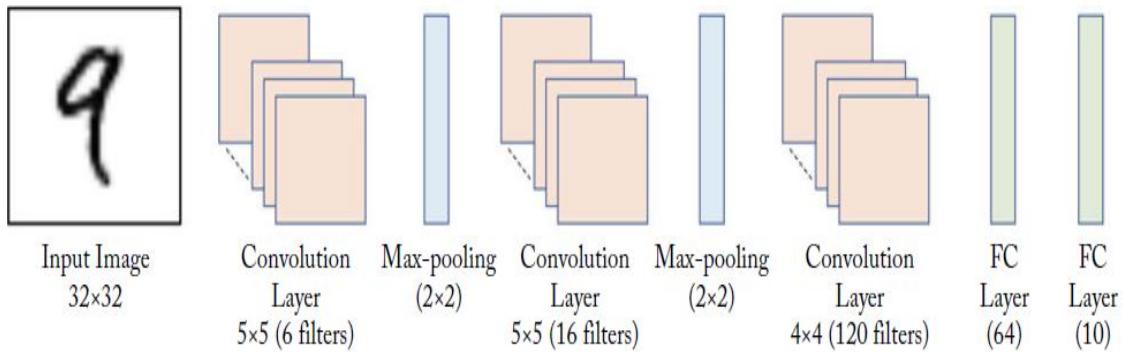


Abbildung 2.13: LeNet-5 Architektur [LBBH98]

**AlexNet** Im Jahr 2012 gewann AlexNet [Kri17] Architektur mit einem besonderen Erfolg im Vergleich zu den anderen Architekturen die ImageNet Large-Scale Visual Recognition Challenge (ILSVRC).<sup>1</sup> AlexNet besteht aus insgesamt acht Parameterschichten, von denen die fünf convolution layer und drei Fully connected Layer sind. Die letzte Fully-connected Layer klassifiziert das Eingangsbild in einer der 1000 Klassen von ImageNet-Dataset. Die Filtergrößen und die Position der Max-Pooling layer sind in Abbildung 2.14 dargestellt. Die Verwendung der nichtlineare Funktionen nach jeder convolutional layer verbessert die Trainingseffizienz im Vergleich zur traditionell verwendeten tanh-Funktionen.

<sup>1</sup><http://www.image-net.org/>

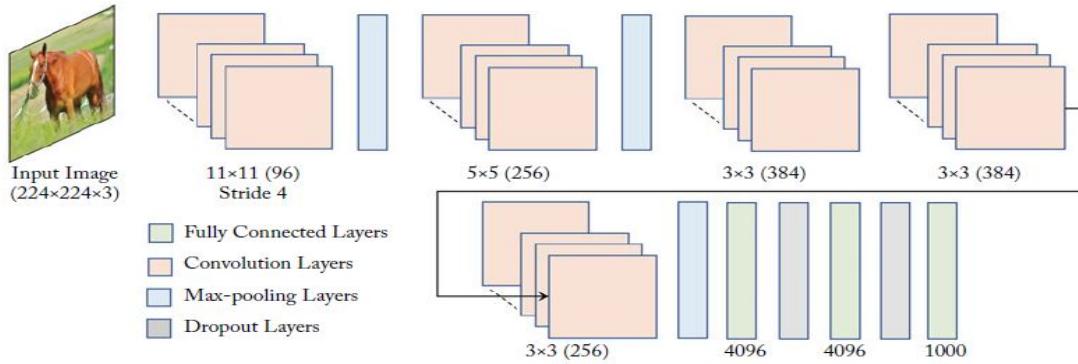
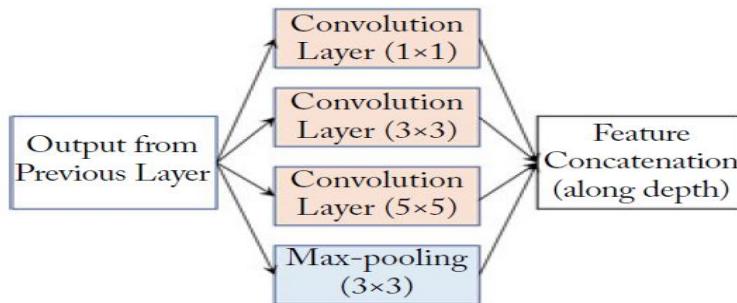


Abbildung 2.14: AlexNet Architektur [SKB18]

**GoogleNet** Bisher diskutierten Netzwerke bestehen aus einer sequentiellen Architektur in einer Richtung. Entlang dieser Richtung befinden sich nach Reihen geordnet Schichten wie convolution layer , Max-pooling, ReLu und Fully-connected layer etc. Die GoogleNet-Architektur [SLJ<sup>+</sup>15] ist eine komplexe Architektur mit mehreren Netzwerkzweigen und gewann den ILSVRC (Large Scale Visual Recognition Challenge) im Jahr 2014 mit der Fehlerquote von 6,7% bei der Klassifizierungsaufgabe. GoogleNet Abbildung 2.15 besteht aus insgesamt 22 Gewichtsschichten und das Grundkonzept ist das "Inception-Module". Die Verarbeitung dieses Moduls erfolgt im Vergleich bisher vorgestellten Architekturen nicht sequentiell, sondern parallel. Eine einfache Version dieses Moduls ist in Abbildung 2.15 dargestellt.

Die Hauptidee von GoogleNet ist, alle grundlegenden Verarbeitungsblöcke, die in einem regulären sequentiellen CNN auftreten, parallel zu platzieren und die Ausgaben davon kombinieren. [SLJ<sup>+</sup>15] Der Vorteil von GoogleNet ist, dass mehrere "Inception-Module" zusammen gestapelt werden, um ein riesigen Netzwerk zu erstellen.

Abbildung 2.15: Grundkonzept von GoogleNet Inception Module [SLJ<sup>+</sup>15]

Ein großes Problem beim "Inception Module" ist, dass die Faltungen wie 5x5 auf einer convolutional layer mit einer großen Anzahl von Filtern sehr ineffizient sein kann. Das wird noch deutlicher, wenn dazu Pooling-Einheiten hinzugefügt werden.

Die Anzahl der Ausgangsfilter entspricht der Anzahl der Filter in der vorherigen Stufen. Beim Mischen der Ausgabe der pooling-Unit mit den Ausgaben der convolutional layer würde die Anzahl von Ausgaben von Stufe zu Stufe extrem zunehmen. Innerhalb wenige Stufen könnte dies zu einer rechnerischen Explosion führen und die Ausgaben können höhere Dimensionen haben. Um dieses Problem zu überwinden, müssen die Dimension der Ausgaben reduziert werden. Abbildung 2.16 siehe mehr Details zur Lösung [SLJ<sup>+</sup>15]

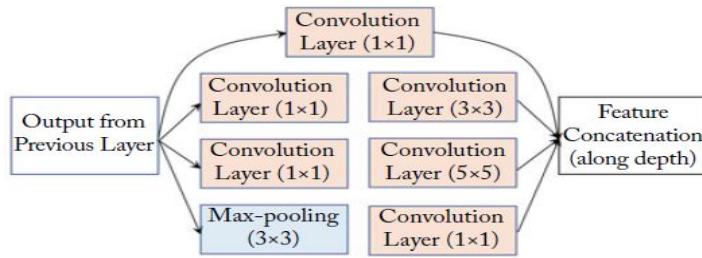


Abbildung 2.16: Neues Inception-Modul, um Dimensionen von Ausgaben zu reduzieren [SLJ<sup>+</sup>15]

# Kapitel 3

## Konzept

### 3.1 Einfuehrung der Systolic Array Architektur

Machine Learning basiert stark auf Berechnungen z.B. von Faltungen, Matrix-Matrix-und Matrix-Vektor Multiplikationen, Berechnung von Verlustfunktionen, Max-Pooling usw. siehe Kapitel 2

Die künstliche neuronale Netze bestehen aus vielen kuenstlichen Neuronen, indem viele Berechnungen stattfinden, die den Ausgabewert des ML-Modells beeinflussen. Das wesentliche Ziel ist die Entwicklung neuer Rechnerarchitektur und der effizienten Nutzung von modernen Systemen, um solchen komplizierten ML-Anwendungen schneller ausführen zu können. Die wachsende Nachfrage nach mehr Rechenleistung führte in den 80er Jahren zur Entwicklung parallel skalierbaren Multiprocessing-Systemen. Das System soll durch paralleles Rechnen in der Lage sein, dass ein kompliziertes großes Rechenproblem in kleinere Stück unterteilt und diese kleine Rechenprobleme gleichzeitig löst. [SV18]

**Problem der sequentielle Berechnungsmethode:** Bei der herkömmlichen Berechnungsmethode der Matrix-Matrix Multiplikationen Abbildung 3.1 wird zuerst überprüft, ob die Matrizen miteinander multiplizierbar sind. D.h. wenn die Spaltenanzahl von A-Matrix mit der Zeilenanzahl von B-Matrix übereinstimmen, dann sind sie miteinander multiplizierbar. Der Nachteil dieser Berechnungsmethode ist die sequentielle Ausführung von Rechenoperationen, die sehr zeitaufwändig ist. Es findet keine zeitgleiche bzw. parallele Berechnung statt. Das Ziel ist aber ein Algorithmus zu entwickeln, dass parallele Berechnung und sogar mehrere parallele Berechnungen durchführt, sodass die Rechenleistung erhöht wird.

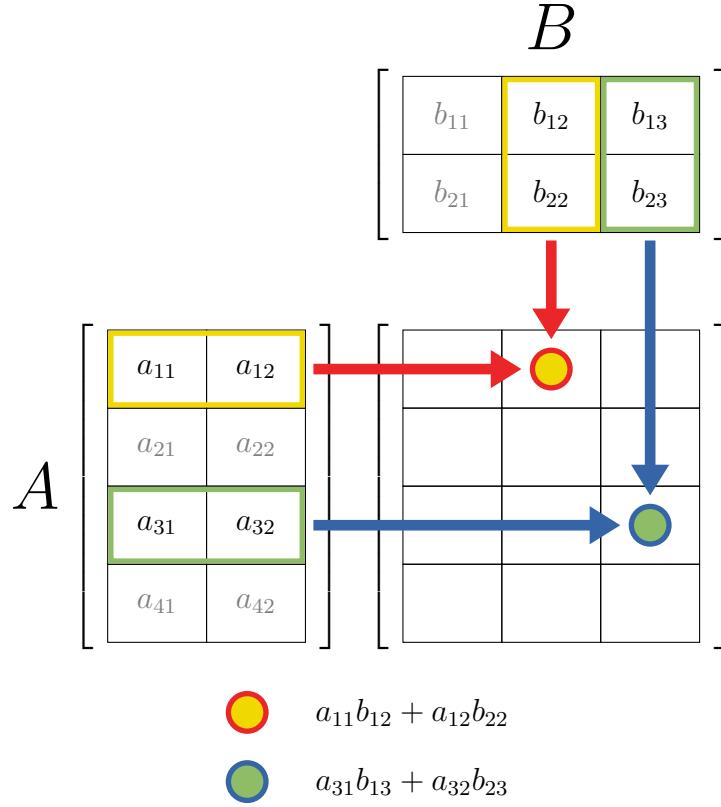


Abbildung 3.1: Matrix Multiplikation

## 3.2 Why Systolic Architectures?

### 3.2.1 Grundidee

Die Grundidee des systolischen Arrays [Kun82] ist, dass die Daten rhythmisch aus dem Computerspeicher fließen und laufen durch viele PEs (processing elements), bevor sie in den Speicher zurückkehren. Die Idee ist vergleichbar mit dem Blutkreislauf, dass der Herz zur Zellen Blut pumpt.

Das in Abbildung 3.2 Verarbeitungsschema ist ineffizient und zeitaufwändig, wenn es nur ein einziges PE gibt, das aus dem Speicher ein Data holt, verarbeitet und das Ergebnis

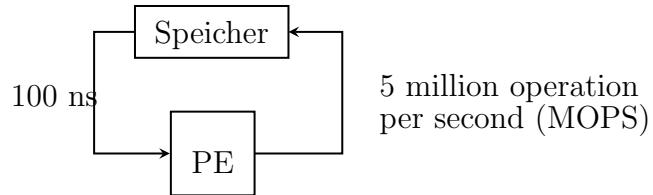


Abbildung 3.2: Das Grundprinzip des systolischen Systems

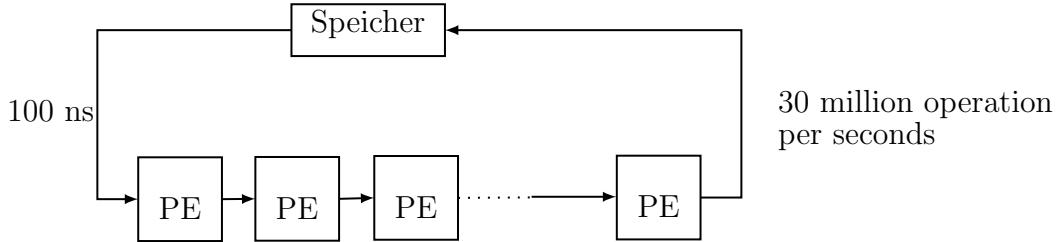


Abbildung 3.3: Das Grundprinzip des systolischen Systems

in den Speicher ablegt. In einer Sekunde werden nur 5 Millionen Operationen **MOPS** (*Millions of operations per seconds*) durchgeführt.

Um **MOPS** zu erhöhen soll mehrere PEs geben, die Daten mehrmals verarbeiten (wie Pipeline System) und am Ende das Ergebnis wird von dem letzten PE in den Speicher abgelegt. Abbildung 3.3 Das ist die Idee, die das System beschleunigen soll. Damit werden 30 Millionen Operationen durchgeführt.

### 3.2.2 Systolic Array Designs für Faltung Berechnungen

**Design1:** Gegeben sind hier die Eingangswerte  $[x_1, x_2, \dots, x_n]$  und die Gewichte  $[w_1, w_2, \dots, w_k]$ .

Die Ausgangswerte sind  $y_i = [w_1 x_i + w_2 x_{i+1} + \dots + w_k x_{i+k-1}]$ .

Die Gewichte sind schon in jeder Zelle vorgeladen. Die Eingangswerte  $x_i$  verbreiten sich zu jeder Zelle, wird multipliziert mit der Gewichte und Teilsummen  $y_i$  bewegen sich systolisch durch jede Zelle. Abbildung 3.4

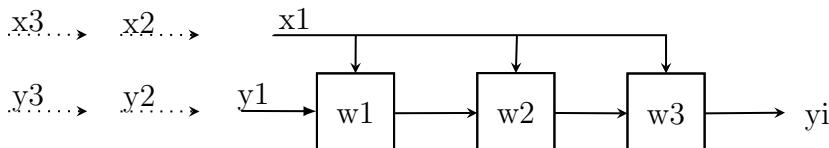


Abbildung 3.4: In PE gespeicherten Gewichten werden jeweils mit der Eingabe multipliziert und das Ergebnis zur Teilsumme vom vorherigen Element addiert. Beim nächsten Takt wird ein Eingangselement ausgelesen und die Teilsummen von PE wird zum Nachbar PE gesendet. Somit werden alle Daten trotz minimaler Bandbreite des Speichers verarbeitet.

**Design2:** Die Eingangswerte  $x_i$  verbreiten sich in Zellen und die Ausgangswerte  $y_i$  werden in Zellen berechnet und akkumuliert. Die Gewichte  $w_i$  bewegen sich in einer Schleife systolisch durch jede Zelle. Am Ende der Berechnung müssen die Ergebnisse  $y_i$  aus den Zellen geholt werden. Dafür ist eine separate BUS-Leitung nötig, um die Ergebnisse auszugeben, was im **Design1** Abbildung 3.4 nicht der Fall war. Das Ziel vom **Design2** Abbildung 3.5

ist die Erhöhung der Genauigkeit von Ergebnissen, da die  $y_i$ -Werte während der Berechnung wie im **Design 1** Abbildung 3.5 in jedem Takt sich ändern oder die Genauigkeit kann bei Gleitkommazahlen sinken und das Ergebnis kann damit gefälscht werden.

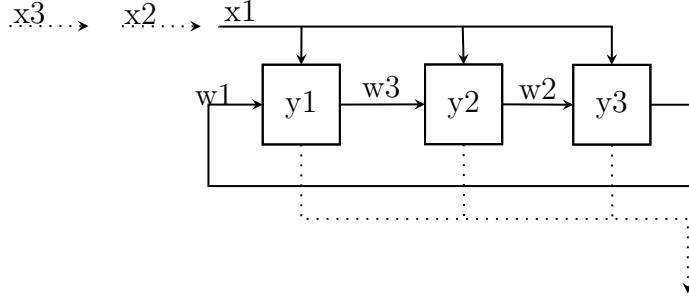


Abbildung 3.5: Die Eingangswerte  $x_i$  verbreiten sich und  $y_i$  bleiben in Zellen (PE) und die Gewichte bewegen sich systolisch durch jede Zelle.

$$y_{out} \leftarrow y + w_{in} \cdot x_{in}$$

$$w_{out} \leftarrow w_{in}$$

**Design3:** Die Gewichte  $w_i$  sind in Zellen vorgeladen. In jedem Takt wird ein Eingangswert ausgelesen und die Eingangswerte  $x_i$  fließen systolisch durch die Zellen. In PE werden die Eingangswerte mit Gewichten multipliziert und das Ergebnis zum globalen Adder (Addierer) weitergegeben. Im Addierer werden die Ergebnisse aus den Zellen gesammelt und addiert. Ein Nachteil dieser Entwurf ist der globale Akkumulator bzw. Addierer. Dafür ist eine separate BUS-Leitung nötig. Abbildung 3.6

**Anwendung:** Diese Methode wird in musterbasierte Suche (Pattern matching) oder auch in der digitale Signalverarbeitung eingesetzt.

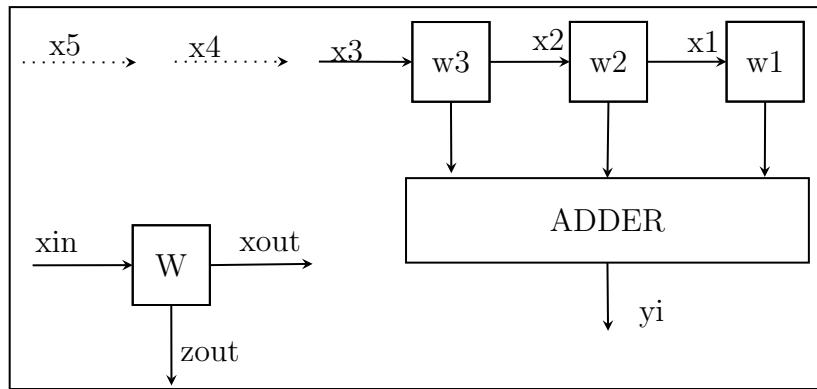


Abbildung 3.6: Die Gewichte  $w_i$  bleiben in Zellen (PE), die Eingangswerte bewegen sich systolisch durch jede Zelle und  $y_i$  werden in den Zellen berechnet und in der Komponente Adder akkumuliert.

$$z_{out} \leftarrow w \cdot x_{in}$$

$$x_{out} \leftarrow x_{in}$$

**Design4:** Abbildung 3.7 Die Ausgangswerte  $y_i$  bleiben fest in den Zellen. Die Gewichte  $w_i$  und Eingangswerte  $x_i$  bewegen sich systolisch und entgegengesetzte Richtungen durch die Zellen. Die Berechnungen finden in PE statt und das resultierende Produkt zu dem Teilergebnis werden dort gespeichert. Dieser Entwurf braucht keine BUS-Leitung oder externe Adder wie vorherigen Entwürfe. Die PEs haben eigene Akkumulator und das Ergebnis wird vom Akkumulator geholt. Der Nachteil hier ist die zusätzliche Logik, damit den Akkumulator in jedem PE am Ende der Berechnung zurücksetzt.

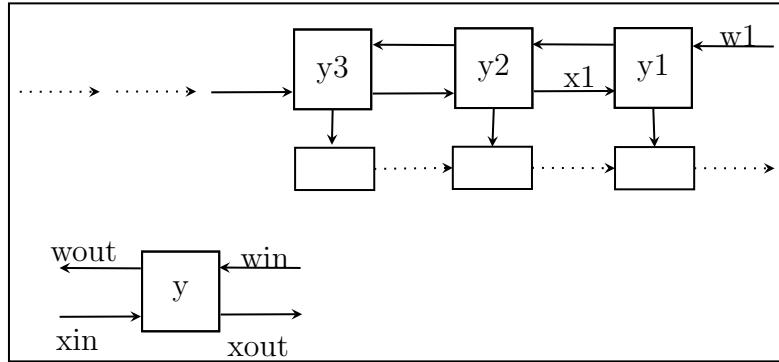


Abbildung 3.7:  $y_i$  bleiben in Zellen (PE) und  $x_i$  und  $w_i$  bewegen sich systolisch durch die Zellen in entgegengesetzte Richtung

$$y \leftarrow y + w_{in} \cdot x_{in}$$

$$x_{out} \leftarrow x_{in}$$

$$x_{out} \leftarrow w_{in}$$

### 3.2.3 Systolic Array Matrix Multiplikation

Bei der Matrix Multiplikation sind mehrere PEs (Processing elements) Abbildung 3.8 nötig, die miteinander verbunden sind. Der Datenfluss erfolgt von links z.B. Matrix A-Werte und von oben transponierte Matrix B-Werte. Die Aufgabe von PEs sind Multiplikationen von Eingangswerten und Akkumulation der Teilsumme in ihren eingebauten Register. Abschließend werden dann die Eingangswerte (Matrix A - Wert) nach rechts und (Matrix B - Wert) nach unten zur Weiterberechnung gesendet. Mehr Details über PEs finden Sie in Kapitel Kapitel 4 *Implementierung*.

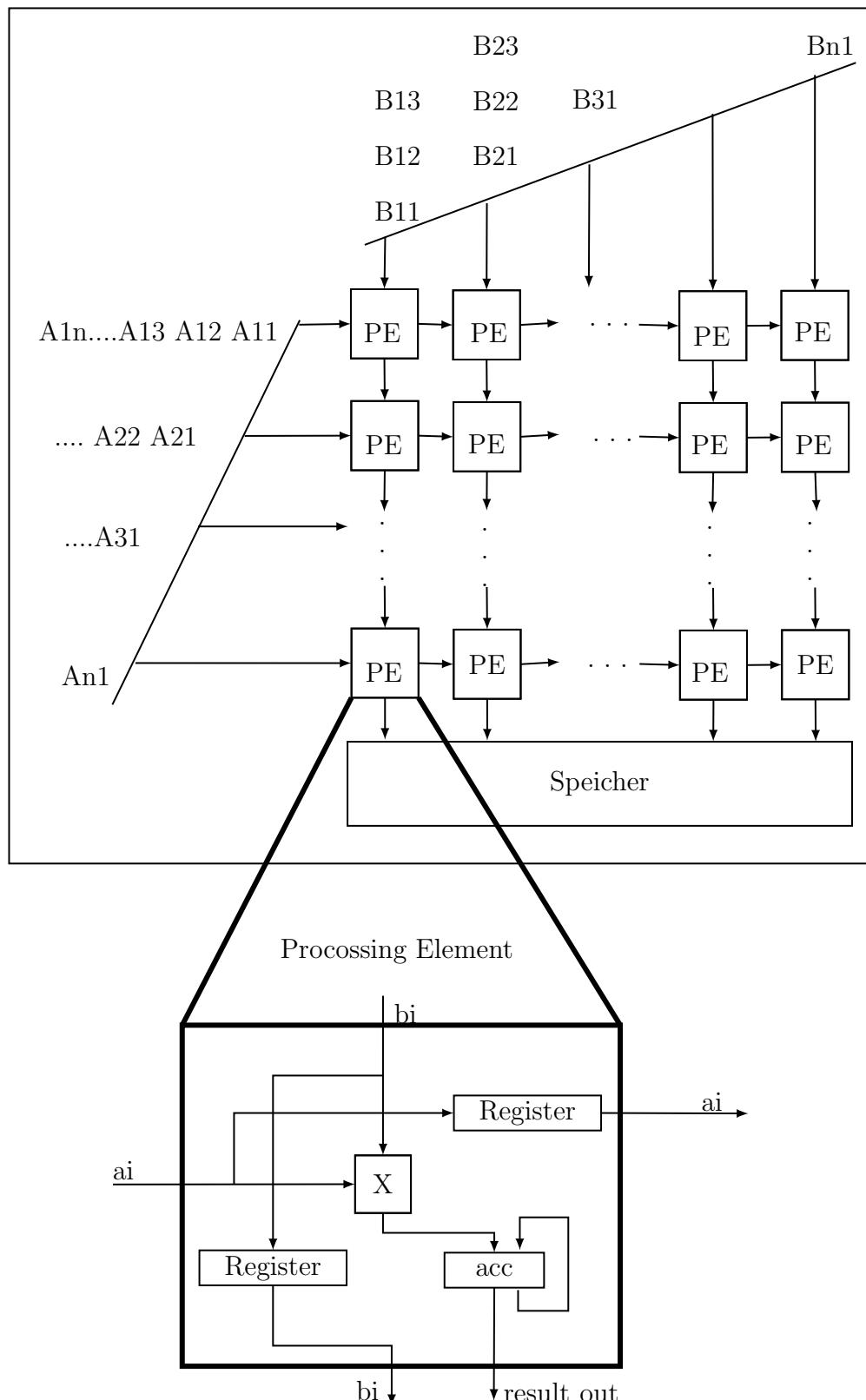


Abbildung 3.8: Systolic Array Architektur mit PE (Processing element)

### 3.2.4 Architektur der Tensor Processing Unit

Die Computer-Technologien der 80er Jahren war nicht genug, um die revolutionäre Idee "Systolic Array" von H.T. Kung in die Praxis umzusetzen. Von 80er Jahren bis heute (2020) die Computer Technologie hat sich ständig entwickelt und ist nicht mehr vergleichbar mit 80er Jahren. Aber die Idee von damals ist noch nutzbar.

Tensor Processing Unit (TPUs) [JY17] Abbildung 3.9 sind anwendungsspezifische Chips, die von Google entwickelt wurden, um ML-Anwendungen zu beschleunigen.

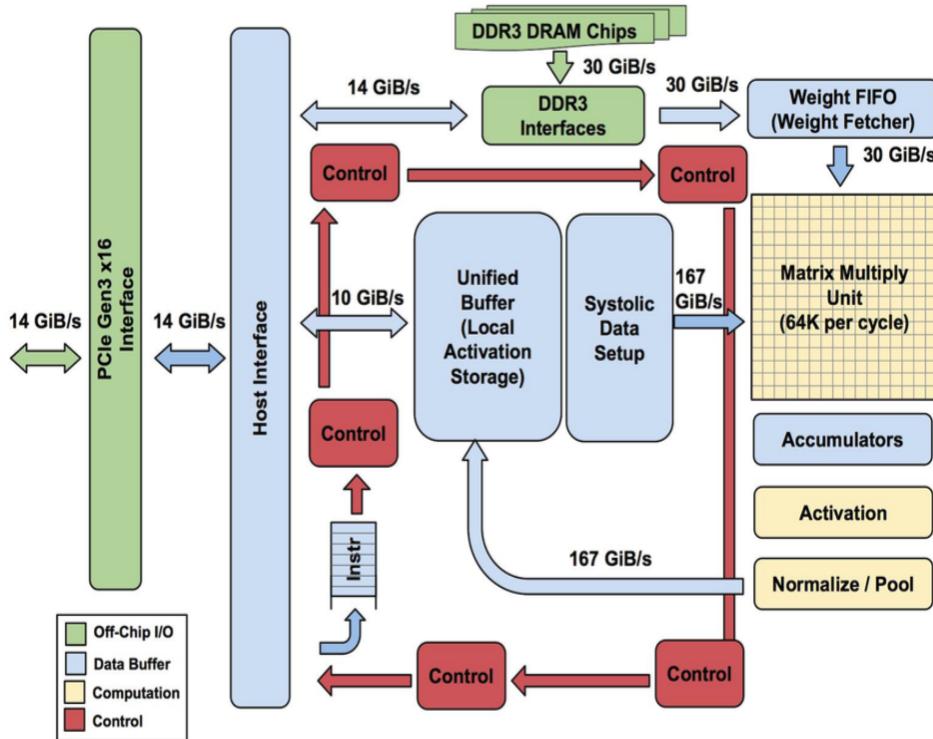


Abbildung 3.9: Architektur einer Tensor Processing Unit Design [JY17]

In dem TPU-Blockdiagramm befindet sich die Matrix-Multiplikationseinheit auf der rechten Seite. Seine Eingänge sind FIFO-Gewichten und der Unified Buffer (UB) und Ausgänge ist der Akkumulator (ACC). Der gelbe Bereich Aktivierung führt die nichtlinearen Funktionen aus und sendet über BUS-Leitung zu Unified Buffer UB. Es ist hier bemerkenswert, dass die Bandbreite von Unified Puffer (UB) zur MM-Einheit ist viel höher als die Bandbreite des Puffers von Gewichten, da die Gewichte viel seltener gelesen werden.

Das Herz des TPUs ist die Matrix-Multiplikationseinheit und enthält 256 x 256 MAC (Multiply-Accumulate). Die MACs können 8-Bit-Multiplikationen und Additionen von positiven und negativen Ganzzahlen ausführen.

Die Gewichte werden zuerst vorgeladen und die Aktivierungen aus dem Aktivierungsspeicherpuffer werden in die MM-Einheit eingelesen. Die Aktivierungen bewegen sich von links nach rechts systolisch und die Teilsummen bewegen sich vertikal von oben nach unten.

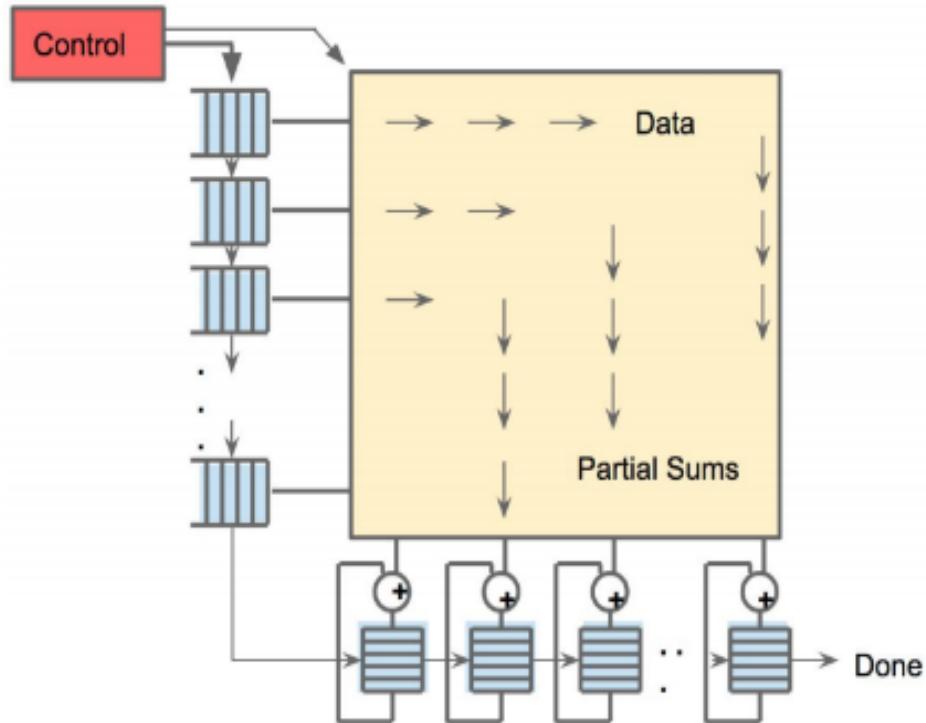


Abbildung 3.10: Systolischer Datenfluss der Matrix Multiply Unit. Die Gesamtsumme werden in den Akkumulatoren gebildet. [JY17]

**Systolic Data Setup:** Hier werden die Daten aus dem Unified Buffer so eingespeist, sodass die Eingänge jeder Zeile um einen Takt zur vorherigen Zeile verzögert werden. Dadurch entsteht eine Latenz von 256-Taktzyklen.

**Akkumulatoren:** In MM-Einheit in MACs finden 8-Bit-Multiplikationen statt und das Ergebnis ist 16-Bit-Produkte. Diese werden in den Akkumulator gesammelt und die Teilsumme dazu addiert. Es werden pro Taktzyklus 256 Teilsumme gebildet. Die hier verwendeten Akkumulatoren sind 32-Bit Akkumulator und befindet sich unter der MM-Einheit.

**Aktivierung:** hier werden die Funktionen des künstlichen Neuronen wie Sigmoid, ReLu, usw. ausgeführt. Es kann auch hier die Pooling Operationen Abbildung 3.11 für Faltungen ausgeführt werden.

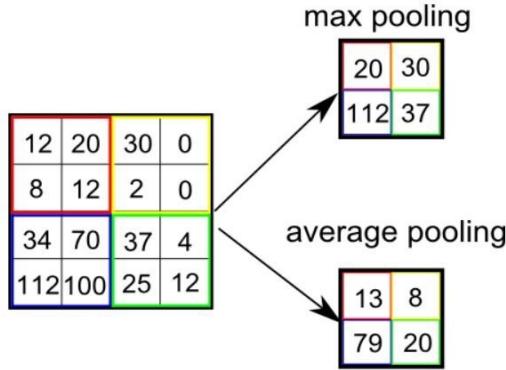


Abbildung 3.11: Average und Max - Pooling

**Read\_Host\_Memory:** liest Daten aus dem CPU\_Hostspeicher in den Unified Buffer (UB).

**Read\_Weights:** liest Gewichte aus dem Gewichtsspeicher-FIFO (Weight-Memory) und im Gewichtsspeicher in die MM-Einheit als Eingabe bereit gestellt.

**MatrixMultiply/Convolve:** mit den geladenen Gewichten und Daten aus dem Unified Buffer wird in MM-Einheit eine Matrixmultiplikation oder eine Faltung ausgeführt werden. Hier kann die eingegebene Matrix variabel sein. Die Matrix mit der Dimension  $B \times 256$  wird in MM-Einheit mit den geladenen Gewichten  $256 \times 256$  multipliziert werden. Die Ausgabe kann dann mit der Größe von  $B \times 256$  in den Akkumulator gespeichert werden, wobei B Pipeline-Zyklen abgeschlossen werden.

**Write\_Host\_Memory:** schreibt Daten aus dem Unified Buffer in den CPU-Hostspeicher und andere Anweisungen sind: alternate host memory read/write, set configuration, zwei Versionen der Synchronisierung, Interrupt-Host, debug-tag, nop und halt.

### 3.2.5 Vorteile des Systolic Arrays

Der Hauptvorteil des TPUs mit Systolic Array Architektur ist seine Unkompliziertheit und es ist nur konzentriert auf Matrix Multiplikation. [tel] Das TPU Design Abbildung 3.12 ist einfach und der Stromverbrauch ist gering gehalten. Unified Buffer und MM-Einheit nimmt die maximale Fläche in dem Chip. Die Control Unit nimmt die geringste Fläche, wobei die sehr schwer zu designen war.

### 3.2.6 Nachteile des Systolic Arrays

Die Multiplikation aus 2 Matrizen  $N \times N$  dauert  $2*N-1$  Taktzyklen. Für einen Batch der Größe B beträgt dann die Latenz  $N*(B+1)-1$ . Die Latenz hängt von den Matrixdimensionen ab. Das ist für Anwendungen mit konstanter Latenz nicht wünschenswert.

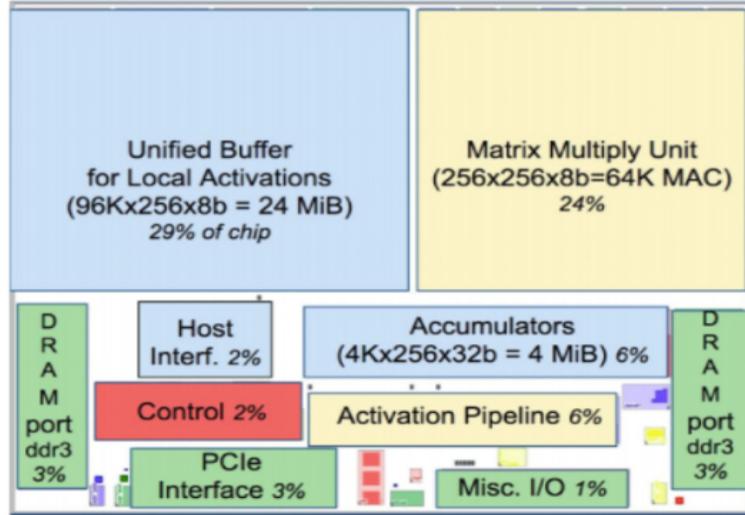


Abbildung 3.12: Diese Tabelle zeigt ein Vergleich mit anderen ähnlichen Produkten und der Stromverbrauch ist deutlich geringer als die anderen Geräten.

Model	Die									Benchmarked Servers					
	$mm^2$	nm	MHz	TDP	Measured		TOPS/s		GB/s	On-Chip Memory	Dies	DRAM Size	TDP	Measured	
					Idle	Busy	8b	FP						Idle	Busy
Haswell E5-2699 v3	662	22	2300	145W	41W	145W	2.6	1.3	51	51 MiB	2	256 GiB	504W	159W	455W
NVIDIA K80 (2 dies/card)	561	28	560	150W	25W	98W	--	2.8	160	8 MiB	8	256 GiB (host) + 12 GiB x 8	1838W	357W	991W
TPU	NA*	28	700	75W	28W	40W	92	--	34	28 MiB	4	256 GiB (host) + 8 GiB x 4	861W	290W	384W

Table 2. Benchmarked servers use Haswell CPUs, K80 GPUs, and TPUs. Haswell has 18 cores, and the K80 has 13 SMX processors. Figure 10 has measured power. The low-power TPU allows for better rack-level density than the high-power GPU. The 8 GiB DRAM per TPU is Weight Memory. GPU Boost mode is not used (Sec. 8). SECDEC and no Boost mode reduce K80 bandwidth from 240 to 160. No Boost mode and single die vs. dual die performance reduces K80 peak TOPS from 8.7 to 2.8. (\*The TPU die is  $\leq$  half the Haswell die size.)

Abbildung 3.13: Diese Tabelle zeigt ein Vergleich mit anderen ähnlichen Produkten und der Stromverbrauch von TPU ist deutlich geringer als die anderen Geräten.

Ein anderes Problem ist, dass die Eingangsmatrix nicht ein Vielfaches von der PE-Array bzw. MACs in MM-Einheit ist. Bei so einem Fall werden nicht alle verfügbaren MACs verwendet und entsteht damit ungenutzte bzw. verschwendeter Arbeitsspeicher bei nicht standardmäßige Matrixdimensionen.

TPU ist allgemein für Matrixmultiplikationen besser geeignet als Faltungsberechnungen. Die Faltung Operationen können auch ausgeführt werden, indem sie in Matrixmultiplikationen umgewandelt werden. Das ist für große Faltungskerngrößen nicht optimal, da Faltungen spezifische Datenflussmuster aufweisen sollen. Ein gutes Beispiel ist der Eyeriss-Beschleuniger (**An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks**), der speziell für die Faltungsoperation entwickelt wurde.

# Kapitel 4

## Implementierung

### 4.1 Implementierungsplan

Vorgehensweise Zur Realisierung der Systolic Array Architektur wird die Programmiersprache VHDL genutzt. Das Kürzel VHDL setzt sich zusammen aus: V steht für VHSIC (Very High Speed Integrated Circuit) und HDL (Hardware Description Language), die im Auftrag der US-Regierung anfangs der 80er entwickelt. Systolic Array Architektur besteht aus RAM (Random-Access Memory), MACs, in dem mehrere Processing Elemente stehen und Delay Control. Abbildung 4.1 Außerhalb der Block befindet sich ein Clockgenerator, der mit konstanten Frequenz takten soll.

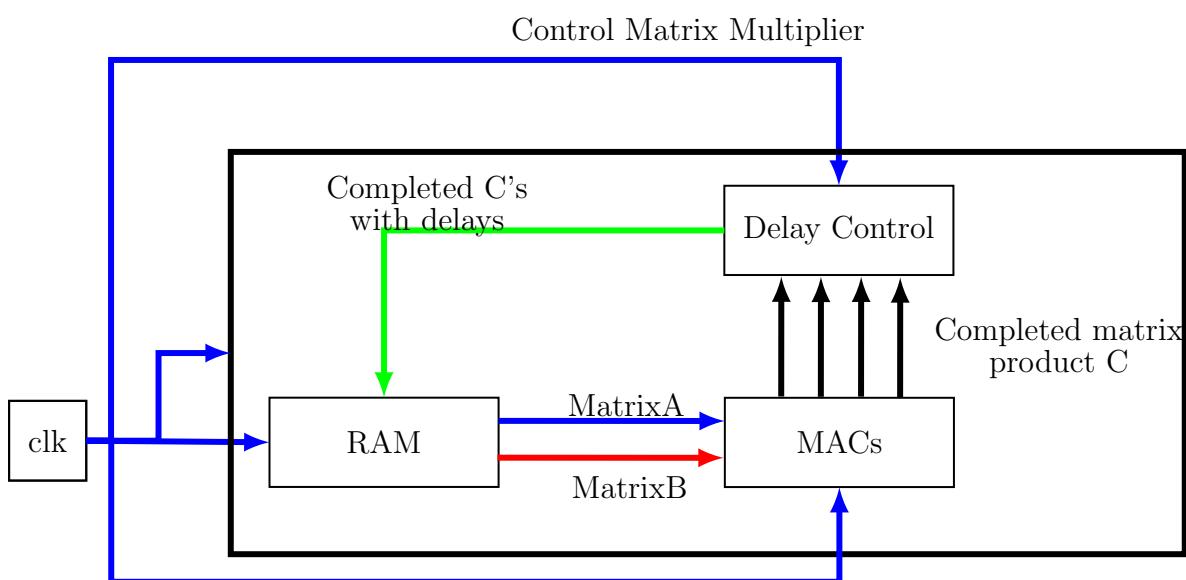


Abbildung 4.1: Übersicht des Codes für die Matrixmultiplikation

### 4.1.1 MAC

In dem Block MACs befinden mehrere PE (Processing Elements) Abbildung 4.2 und sie sind miteinander verbunden, sodass die Daten systolisch durch die Elemente bewegen. Ein PE hat zwei Eingänge und zwei enable Signale. Es findet die Berechnung dann statt, wenn enable Signale auf HIGH gesetzt sind. Das Ergebnis wird in dem Akkumulator von PE zur Teilsumme addiert. Dabei wird die Berechnungen durch ein Zähler bis zu einem bestimmten Wert gezählt. Wenn der Zähler fertig ist, dann wird die Gesamtsumme zum Speicher gesendet.

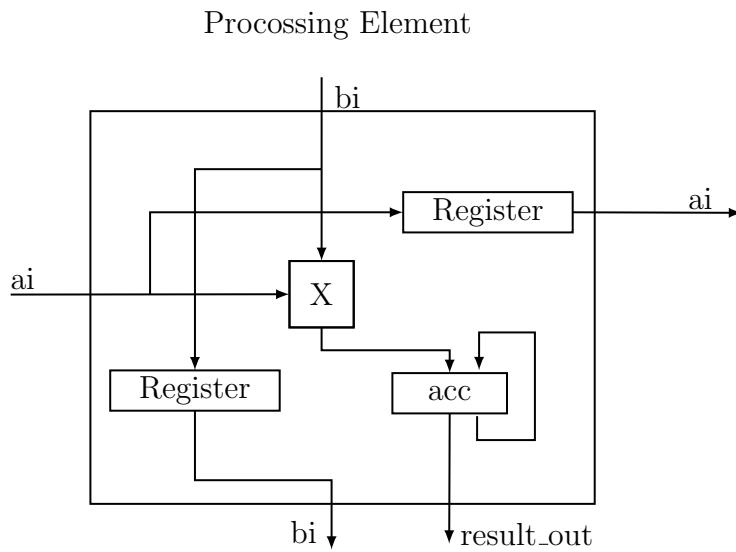


Abbildung 4.2: Innenleben der einzelnen Verarbeitungseinheit (PE)

### 4.1.2 RAM

In RAM werden die Eingangsmatrizen A-und B-Matrix gespeichert und fließen die Matrixelemente in die PE rein. Damit die richtige Matrixelemente in die richtige PE fließen kann, wird hier die logische Signale wie  $A\_enable$  und  $B\_enable$  genutzt.

$A\_enable$  bzw.  $B\_enable$  sind 3 Bit groß und wird im VHDL-Code so angewendet.

Wenn  $A\_enable := 001$  und  $A\_enable := 001$  sind, dann werden nur die erste Zeile von Matrizen eingespeist und in PE berechnet. Nachdem alle Elemente aus der ersten Zeile eingespeist sind dann werden die  $A\_enable := 010$  und  $A\_enable := 010$ , um die zweite Zeile zu berechnen usw.

### 4.1.3 Delay Controller

Innerhalb der MACs-Komponente in Register von PEs werden die Teilsumme zur Gesamtsumme akkumuliert, sobald die Gesamtsumme bereit sind, werden sie dann zu Delay

Controller gesendet. Jede Elemente der Produktmatrix hat eigenen Eingang und die Elemente werden dann zu RAM Adresse zugewiesen. Innerhalb der Delay Controller werden dann jede Elemente von Produktmatrix verzögert abgebildet und zum RAM zum speichern gesendet.

#### 4.1.4 Register in MAC

Die Register Blöcke siehe Abbildung 4.3 verzögern die Daten für einen Taktzyklus. Dies ist entscheidend für die Implementierung des systolischen Architektur. Ohne die Register wäre es nicht möglich den Datenfluss in MM-Unit zu kontrollieren.

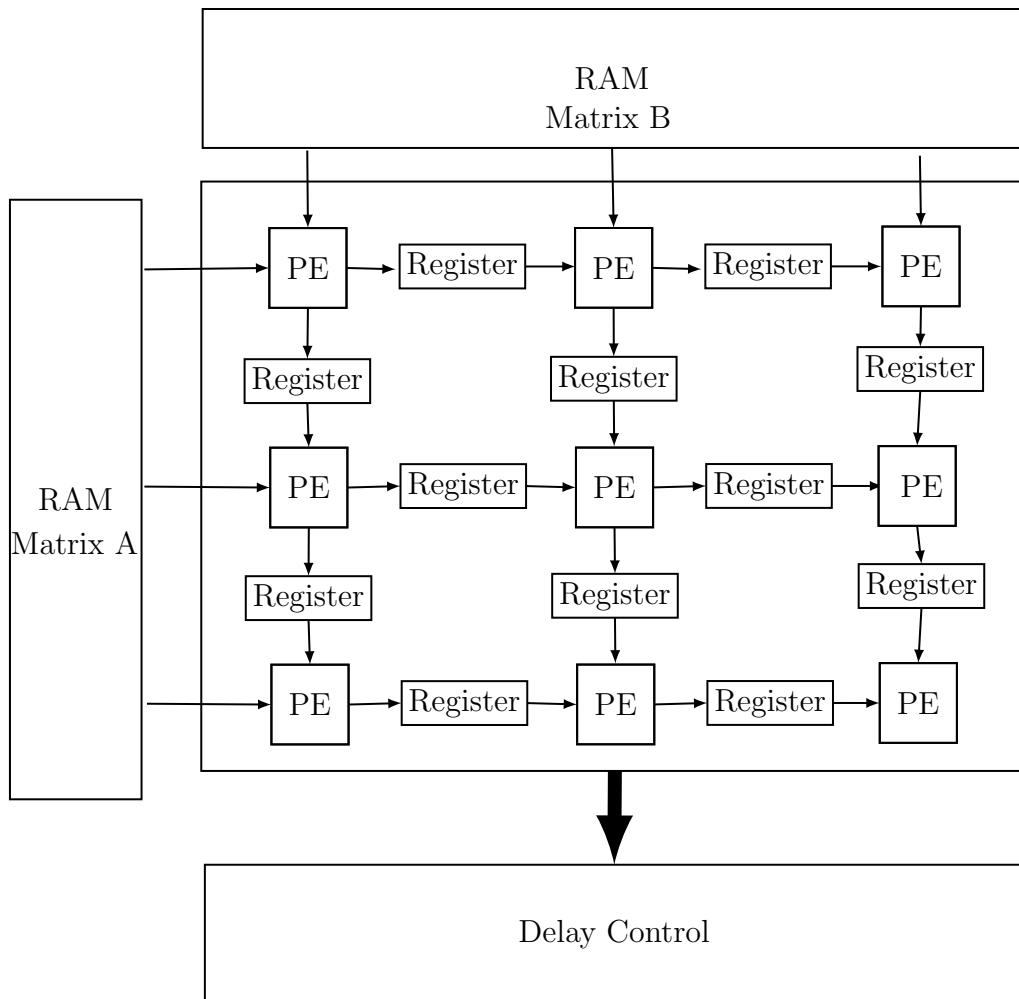


Abbildung 4.3: Innenleben der MAC

### 4.1.5 Simulationsergebnisse

Hier werden zwei gleiche Matrizen siehe Gleichung 4.1 miteinander multipliziert. Bei der Multiplikation wird die in Abschnitt 3.1 vorgestellte systolic Array Architektur angewendet. Um die Elemente von Produktmatrix diagonal ausgeben zu können, müssen jeweils die Eingaben diagonal in die MAC-Unit Unterabschnitt 4.1.1 eingespeist werden. Außerdem werden *enable-Signale* wie `to_C11_en` gebraucht, um die einzelne Werte der Produktmatrix in RAM speichern zu können.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix} \quad (4.1)$$

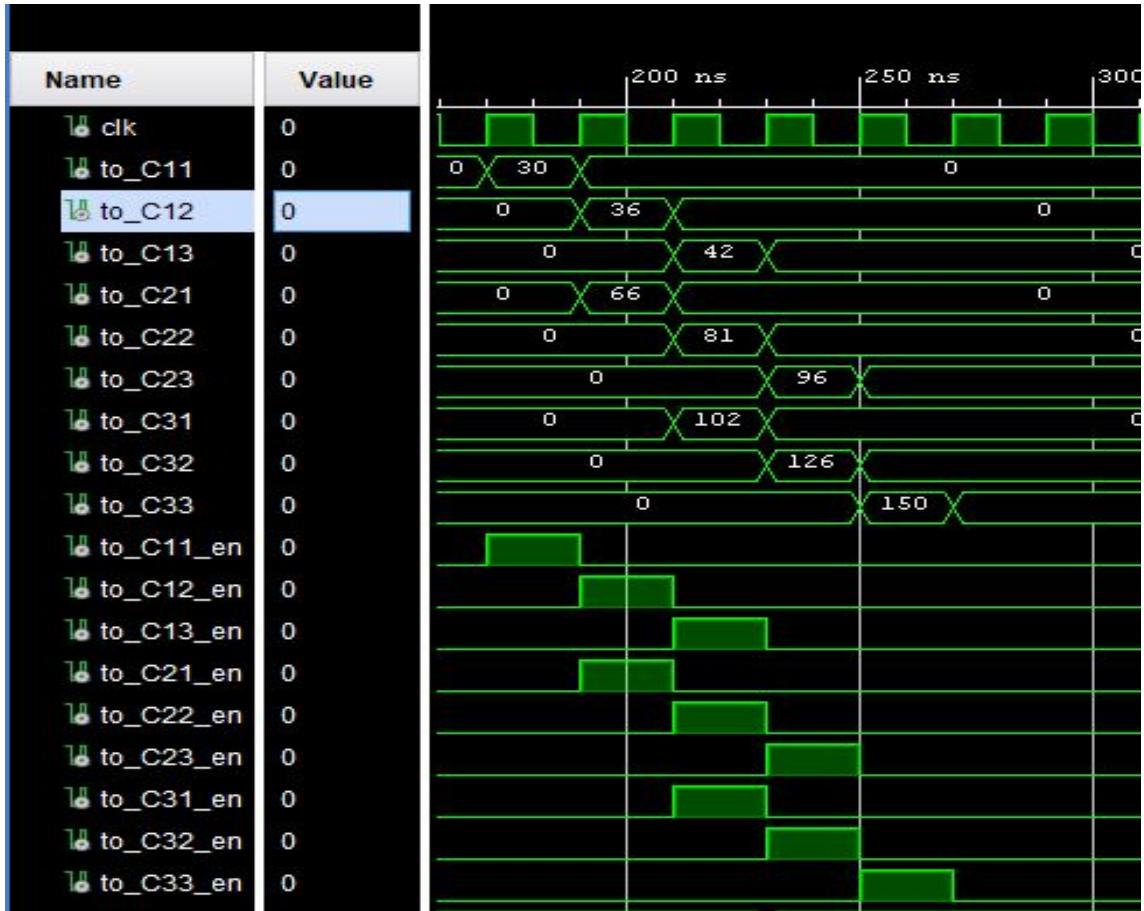


Abbildung 4.4: Simulationsergebnis einer Matrixmultiplikation

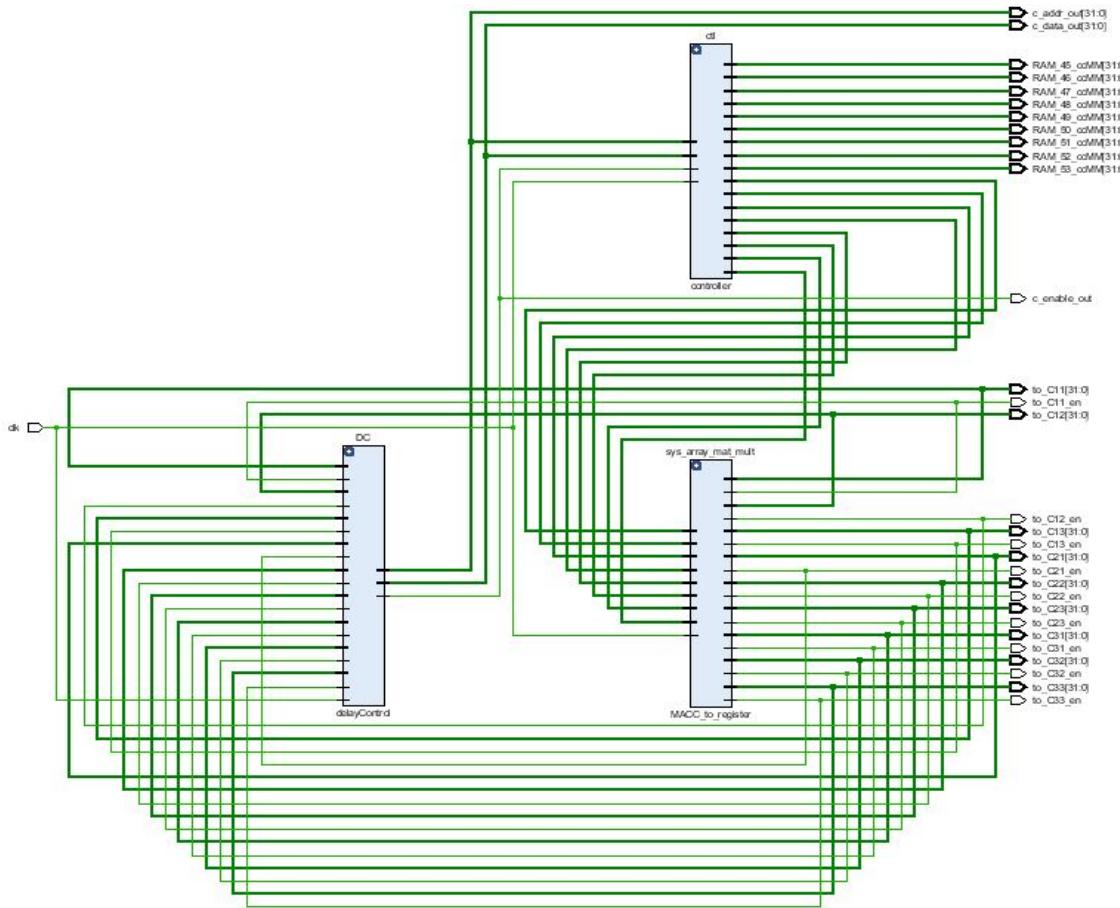


Abbildung 4.5: Es ist wegen Leitungen schwer zu sehen. In dem Design sind die oben vorgestellten Komponenten wie MAC (PE in MAC), RAM und Delay Controller.

#### 4.1.6 Zusammenfassung

In diesem Projekt wurde ein systolisches Array zur Berechnung der Matrixmultiplikation entworfen und implementiert. Die Vorteile des systolischen Arrays sind einfaches Design und zeitgleiche Operationen (Parallelität). Einige Nachteile sind, dass es nicht für allgemeine Anwendungen anwendbar ist, sondern eher für spezielle Zwecke geeignet ist, da die Organisation von PE (Processing element) zur Berechnung der Matrixelementen angepasst muss. Außerdem die Implementierung des systolischen Arrays ist beschränkt auf 3x3-Matrizen und die Dimension der Matrizen war nicht dynamisch, dass man am Anfang der Berechnung beliebig groß wählen darf.

# Kapitel 5

## Evaluation

Das Bilderkennungssystem ist ein sehr breites Forschungsgebiet und es geht nicht nur um Bilder zu erkennen, sondern auch deren Inhalt zu analysieren und zu verstehen. Heutzutage es ist ein sehr aktives Forschungsgebiet für Anwendungen von **Convolutional Neural Network (CNN)**. Die Aufgaben von solchen Anwendungen sind Klassifizierungen von Bildern, Segmentierung, Erkennung und Analyse von Szenen etc. [SKB18]

Die meisten CNN Architekturen sind eingesetzt, um ein Objekt oder eine Person in einem Bild zu identifizieren oder das Label eines Objekts auszugeben. Die in Unterabschnitt 2.6.4 vorgestellten Architekturen und auch andere verbesserten Versionen sind rechen- und speicherintensiv. Die Größe der CNN nehmen ständig zu. Dabei ist es wichtig, die Energieeffizienz und die Leistung bei gleichzeitiger Bewahrung der Genauigkeit zu verbessern. [DLW<sup>+</sup>17] Die ähnlichen Arbeiten werden hier vorgestellt, die die Matrixmultiplikationen in CNN beschleunigen.

### 5.0.1 Gemini: Ein agiles systolic Array Generator

Gemini ist ein agiles systolic Array Generator, der systematische Auswertungen von Deep-Learning Architekturen ermöglicht. Das ist ein Open-Source-Code und ist in Github <sup>1</sup> veröffentlicht.

**Funktion:** Das Gemini-Projekt ist ein Matrix-Multiplikation-Beschleuniger und basiert auf Systolic Array Architektur, der einen benutzerdefinierten ASIC-Beschleuniger generiert. Gemini ist in der Chisel-Programiersprache <sup>2</sup> geschrieben. Eine Systemübersicht ist in Abbildung 5.1 dargestellt. Im Bereich von Systolic Array wird eine 2D-Array Matrix-

---

<sup>1</sup><https://github.com/ucb-bar/gemmini>

<sup>2</sup>Chisel ist eine Hardware-Design-Sprache basiert auf Scala, die die Erzeugung und Wiederverwendung von Schaltkreisen für digitale ASIC- und FPGA-Logikdesigns erleichtert.

multiplikation Gleichung 5.1 durchgeführt und das Innenleben von Gemini-Systolic Array ist in Abbildung 5.2 dargestellt.

$$C = A * B + D \quad (5.1)$$

A und B sind Matrizen, die miteinander multipliziert werden. D ist ein Bias-Vektor

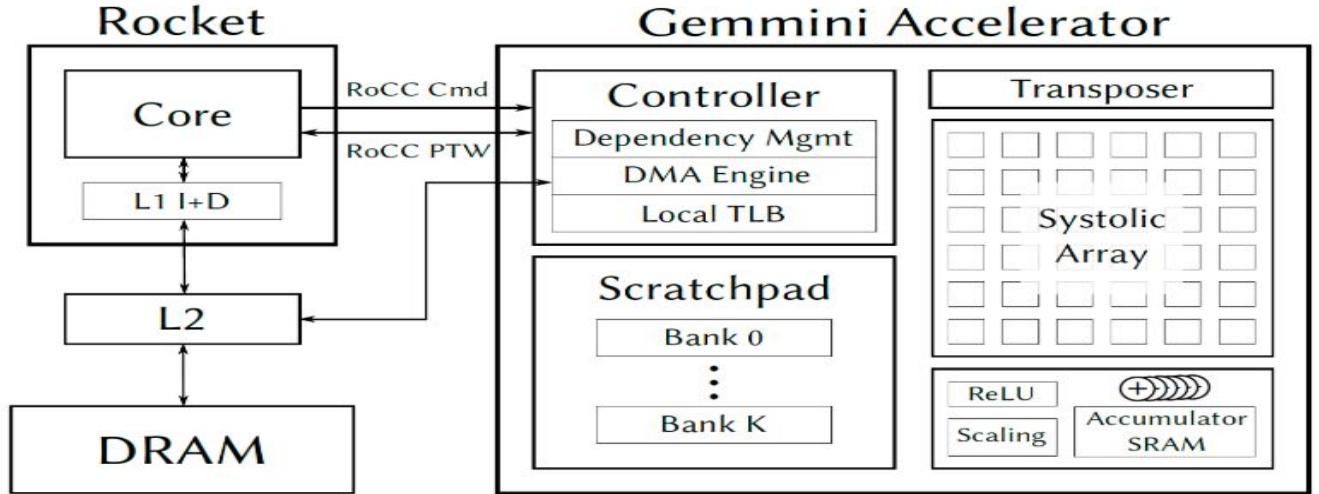


Abbildung 5.1: Eine Systemübersicht der Gemmini-basierten Systolic Array Generator [GHAI<sup>+</sup>19]

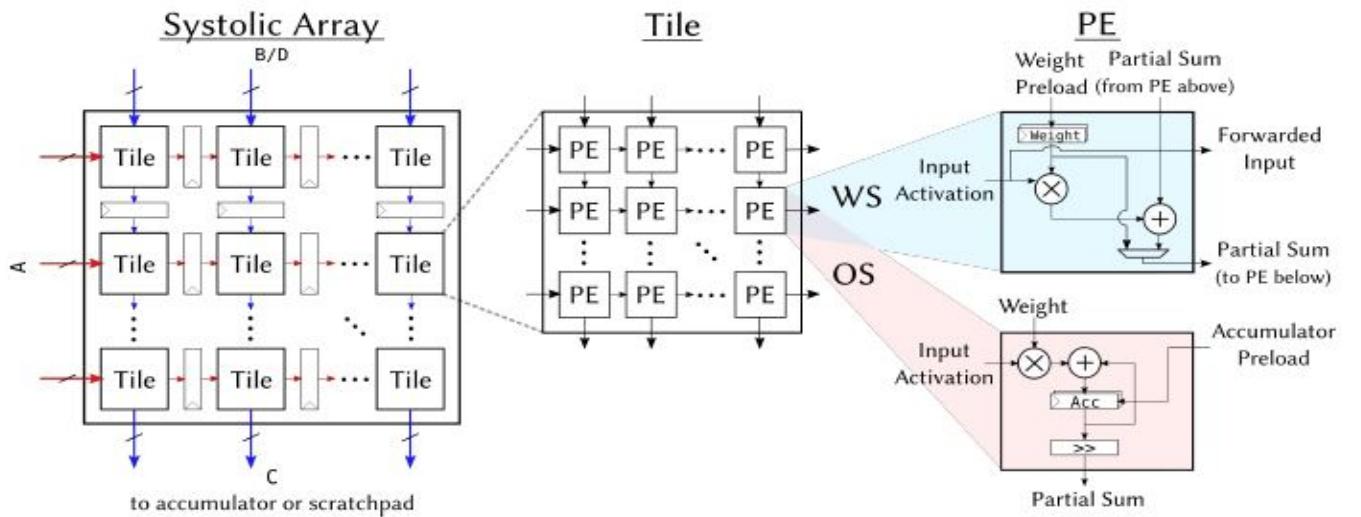


Abbildung 5.2: Hier sieht man das Innenleben von Gemini Systolic-Array mit seinen Verarbeitungseinheiten (PEs). WS steht für *weight-stationary* (d.h. die Matrizen B und D schon vorgeladen.) und OS für *output-stationary* [GHAI<sup>+</sup>19]

### 5.0.2 Die Fläche und der Stromverbrauch von Gemini-Design

Die Fläche und der Stromverbrauch sind stark korreliert. Die Evaluation erfolgte mit der Taktfrequenz von 500 MHz. In diesem Design wird  $0.467 \text{ mm}^2$  Fläche gebraucht und der Stromverbrauch war 611mW . Die andere Evaluation erfolgte mit der Taktfrequenz von 1GHz und dafür wird  $0.541 \text{ mm}^2$  Fläche gebraucht und 1.4W nötig. Abbildung 5.3

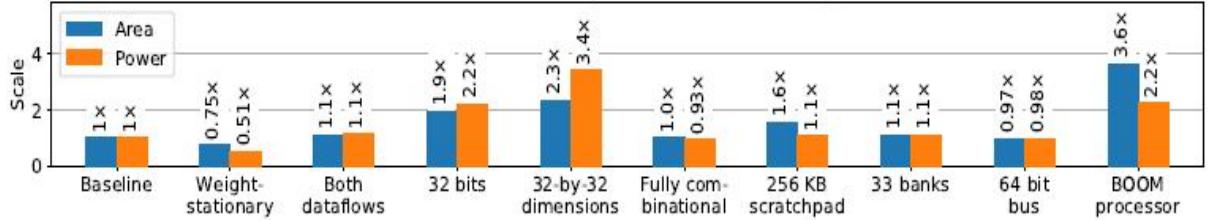
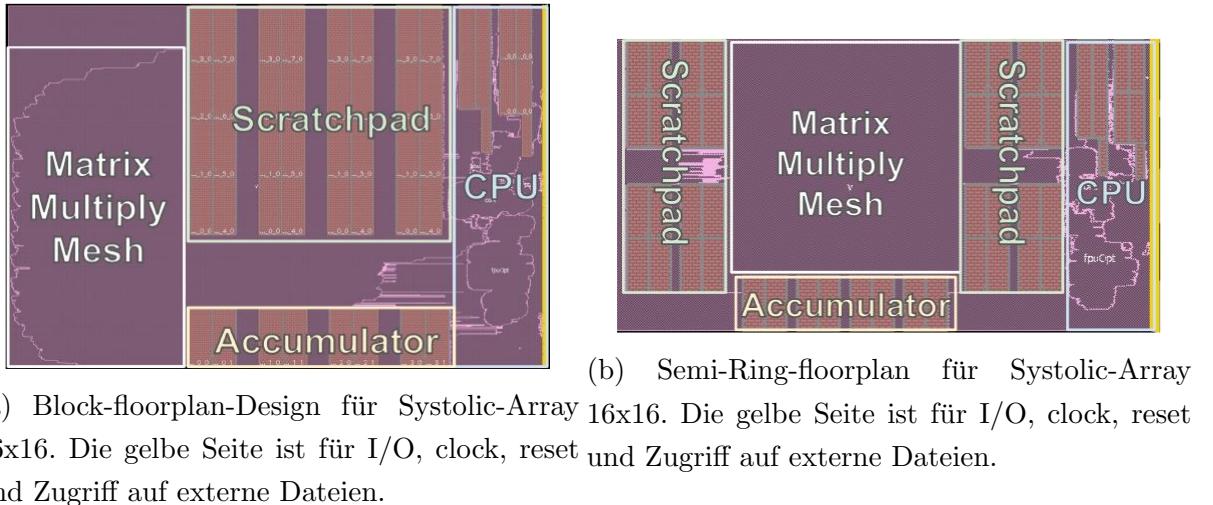


Abbildung 5.3: Fläche und der Stromverbrauch von Gemini-Design [GHAI<sup>+</sup>19]

Es wird nicht nur auf die Fläche und dafür benötigten Stromverbrauch untersucht, sondern auch mit unterschiedlichen physikalischen Designs *siehe* Abbildung 5.4a,b untersucht.



Die 16x16-Arrays mit der Frequenz 500 MHz und 1GHz verglichen. Man kann mithilfe der Tabelle feststellen, dass allgemein bei 1GHz Frequenz mehr Strom verbraucht wird. Außerdem Semi-Ring-floorplan im Vergleich zum Block-Floorplan verbraucht weniger Strom aber dafür benötigt mehr Fläche als Block-Floorplan. Die Tabelle Abbildung 5.5 zeigt, dass Semi-Ring Floorplan bei höchsten Taktfrequenz 1GHz deutlich weniger Strom als Block-Floorplan verbraucht.

Design	Freq (MHz)	Floorplan	Area (mm <sup>2</sup> )	Power (mW)
16×16	500	Block	1.34	321.71
16×16	500	Semi-Ring	1.21	312.41
16×16	1000	Block	1.34	773.70
16×16	1000	Semi-Ring	1.21	766.12
32×32	500	Block	2.81	1058.24
32×32	500	Semi-Ring	3.01	1078.71
32×32	1000	Block	2.81	2796.51
32×32	1000	Semi-Ring	3.01	2683.01

Abbildung 5.5: Tabelle für Fläche und der Stromverbrauch von Block- und semi-ring floorplan [GHAI<sup>+</sup>19]

# Kapitel 6

## Ausblick

In meiner Thesis werden zuerst die Grundlagen zum maschinellen Lernen erläuert und im folgenden die Strukturen der künstlichen neuronalen Netze betrachtet. Besonders wird die CNN-Architektur (*convolutional neural network*) und ihr Entwicklungstrend betrachtet. Die Gemeinsamkeit von dieser vorgestellten Architekturen sind, dass die erste Entwurf in CNN sind wie z.B. LeNET oder die besondere Erfolge in ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) bekommen haben. siehe Abbildung 2.15 Abbildung 2.14 In solchen ML-und Deep-Learning Anwendungen finden ständig Berechnungen (Faltung, Matrixmultiplikation etc.) statt, um von eingespeisten Daten sinnvolle Ausgabewerten zu erhalten. Im Jahr 1982 veröffentlichte Idee *Systolic-Array Architektur* wird eingesetzt, um die Berechnungen zu beschleunigen. Das Ziel dieser Arbeit ist die Umsetzung diese Idee in der Praxis. Um das Ziel zu erreichen, wird eine Programmiersprache VHDL für die Implementierung dieser Idee benutzt.

# Literaturverzeichnis

- [Agg18] AGGRAWAL, CC: *Neural Networks and Deep Learning. A Textbook.* 2018
- [Bec19] BECKER, Roland: CONVOLUTIONAL NEURAL NETWORKS – AUFBAU, FUNKTION UND ANWENDUNGSGEBIETE. (Februar, 2019)
- [Des] DESHPANDE, Adit: *A Beginner's Guide To Understanding Convolutional Neural Networks*
- [DLW<sup>+</sup>17] DING, Caiwen ; LIAO, Siyu ; WANG, Yanzhi ; LI, Zhe ; LIU, Ning ; ZHUO, Youwei ; WANG, Chao ; QIAN, Xuehai ; BAI, Yu ; YUAN, Geng u. a.: Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, S. 395–408
- [GHAI<sup>+</sup>19] GENC, Hasan ; HAJ-ALI, Ameer ; IYER, Vighnesh ; AMID, Alon ; MAO, Howard ; WRIGHT, John ; SCHMIDT, Colin ; ZHAO, Jerry ; OU, Albert ; BANISTER, Max ; SHAO, Yakun S. ; NIKOLIC, Borivoje ; STOICA, Ion ; ASANOVIC, Krste: *Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures.* 2019
- [Hay94] HAYKIN, Simon: *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994
- [JY17] JOUPPI, N. P. ; YOUNG, C.: In-datacenter performance analysis of a tensor processing unit. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017. – ISSN null, S. 1–12
- [Kri17] KRIZHEVSKY, Alex: ImageNet Classification with Deep Convolutional Neural Networks. (2017)
- [Kun82] KUNG, Hsiang-Tsung: Why systolic architectures? In: *Computer* (1982), Nr. 1, S. 37–46

- [LBBH98] LECUN, Yann ; BOTTOU, Léon ; BENGIO, Yoshua ; HAFFNER, Patrick: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE* 86 (1998), Nr. 11, S. 2278–2324
- [RN16] RUSSELL, Stuart J. ; NORVIG, Peter: *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016
- [Ros] ROSENBLATT, Frank: THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN.
- [SCP<sup>+</sup>19] SONG, Jinook ; CHO, Yunkyo ; PARK, Jun-Seok ; JANG, Jun-Woo ; LEE, Sehwahn ; SONG, Joon-Ho ; LEE, Jae-Gon ; KANG, Inyup: 7.1 An 11.5 TOPS/W 1024-MAC butterfly structure dual-core sparsity-aware neural processing unit in 8nm flagship mobile SoC. In: *2019 IEEE International Solid-State Circuits Conference-(ISSCC)* IEEE, 2019, S. 130–132
- [SKB18] SALMAN KHAN, Syed Afaq Ali S. Hossein Rahmani R. Hossein Rahmani ; BENNAMOUN, Mohammed: *A Guide to Convolutional Neural Networks for Computer Vision*. 2018
- [SLJ<sup>+</sup>15] SZEGEDY, Christian ; LIU, Wei ; JIA, Yangqing ; SERMANET, Pierre ; REED, Scott ; ANGUELOV, Dragomir ; ERHAN, Dumitru ; VANHOUCKE, Vincent ; RABINOVICH, Andrew: Going deeper with convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, S. 1–9
- [SV18] SHILPA V, Suma V S.: Design of Systolic Architecture Using Evolutionary Computation. (Mai, 2018), S. 1–17
- [tel] TELESENS: *Understanding Matrix Multiplication on a Weight-Stationary Systolic Architecture*
- [Wik] WIKIPEDIA: *Mooresches Gesetz*
- [ZF14] ZEILER, Matthew D. ; FERGUS, Rob: Visualizing and understanding convolutional networks. In: *European conference on computer vision* Springer, 2014, S. 818–833

# Abbildungsverzeichnis

1.1	Mooresches Gesetz von 1971-2018 [Wik]	3
2.1	Visualisierung von 2D Faltung	6
2.2	Aufbau eines Neurons	9
2.3	Künstliches Neuron	9
2.4	Ein vereinfachtes mathematisches Modell für ein künstliches Neuron	12
2.5	Entscheidungsregionen	13
2.6	Entscheidungsgrenzen von logischen Gattern	14
2.7	Struktur eines mehrschichtigen feedforward-Netzes	15
2.8	Was wir sehen [Des]	16
2.9	Was der Computer sieht [Des]	16
2.10	Struktur von Convolutional Neural Network CNN [Bec19]	16
2.11	Max Pooling	17
2.12	Visualisierung von Convolutional Network [ZF14]	18
2.13	LeNet-5 Architektur [LBBH98]	19
2.14	AlexNet Architektur [SKB18]	20
2.15	Grundkonzept von GoogleNet Inception Module [SLJ <sup>+</sup> 15]	20
2.16	GoogleNet Inception Module	21
3.1	Matrix Multiplikation	23
3.2	Einzelne Processing Elemente PEs	23
3.3	Mehrere Processing Elemente PEs	24
3.4	Design1	24
3.5	Design2	25
3.6	Design3	25
3.7	Design4	26
3.9	Architektur einer Tensor Processing Unit Design [JY17]	28
3.10	Systolischer Datenfluss der Matrix Multiply Unit. Die Gesamtsumme werden in den Akkumulatoren gebildet. [JY17]	29
3.11	Pooling	30

3.12 TPU-Design . . . . .	31
3.13 TPU-Verbrauch . . . . .	31
4.1 Übersicht des Codes für die Matrixmultiplikation . . . . .	32
4.2 Innenleben der einzelnen Verarbeitungseinheit (PE) . . . . .	33
4.3 Innenleben der MAC . . . . .	34
4.5 Block Design von Matrixmultiplikation mit Systolic Array Architektur . . .	36
5.1 Geminimi . . . . .	38
5.2 Gemini Systolic Array (PEs) . . . . .	38
5.3 Fläche und der Stromverbrauch von Gemini-Design . . . . .	39
5.5 Tabelle für Fläche und der Stromverbrauch . . . . .	40