technische universität
dortmund

Masterarbeit

# Effizente Bildverarbeitung hexagonaler Strukturen mittels Deep Convolutional Neural Networks

**Michael May**

**7. September 2018**

Gutachter: Prof. Dr. Katharina Morik
Jens Buß

Prof. Dr. Katharina Morik
Lehrstuhl 8 Künstliche Intelligenz
Fakultät Informatik
Technische Universität Dortmund
Otto-Hahn-Straße 12
44227 Dortmund

Michael May
michael.may@tu-dortmund.de
Matrikelnummer: 148416
Studiengang: Master Informatik
Prüfungsordnung: MPO Inf 2013

Masterarbeit
Thema: Effizente Bildverarbeitung hexagonaler Strukturen mittels Deep Convolutional Neural Networks

Eingereicht: 7. September 2018

Betreuer:

Prof. Dr. Katharina Morik
Lehrstuhl 8 Künstliche Intelligenz
Fakultät Informatik
Technische Universität Dortmund
Otto-Hahn-Straße 12
44227 Dortmund

# Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dortmund, den 7. September 2018

Michael May

# Abstrakt

Die klassische Bildverarbeitung basiert auf den Konzept quadratischer Pixel, welche auf einem Gitter aufgespannt werden und über ihre Breite und Höhe definiert sind. Allerdings bieten hexagonale Pixel Vorteile, wie unter anderem einer höheren Dichte und stärkere Radialsymmetrien. In dieser Arbeit eine hexagonale Convolution entwickelt, die eine direkte Darstellung auf hexagonalen Gitter in Convolutional Neural Networks ermöglicht. Dazu werden simulierte astrophysikalische Daten aus dem FACT (First G-APD Cherenkov Telescope) genutzt, um die Effektivität dieser Convolution zu testen. Das HIP (Hexagonal Image Processing) Konzept wird übernommen, um eine effiziente Adressierung der Bilder zu ermöglichen. Gleichzeitig ermöglicht dieses die Verwendung eines eigenen Rechensystem auf Basis des kartesischen Koordinatensystem zur Manipulation der Bildadressen.
Aus den Experimenten sind keine markanten Verbesserungen mit hexagonaler Convolution festzustellen. Getestete Modelle auf dem FACT Datensatz zeigten keine statistisch relevanten Abweichungen zwischen beiden Verfahren. Die Anzahl der zu trainierenden Parameter konnte allerdings reduziert werden, aufgrund der hexagonalen Strukturen. Dies führt bei guter Optimierung zu einem geringeren Speicheraufwand und kürzeren Trainigszeiten.

Common image analysis tasks work with square pixels on an rectangular lattice, which is bound by its height and width. But hexagonal pixel structures on a hexagonal lattice would bring beneficial advantages such as a higher pixel density and stronger rotational invariance. This works presents a method of real hexagonal convolution, which can then be used inside convolutional neural neworks. An evaluation of this concept is done using simulated astrophysical data of the FACT (First G-APD Cherenkov Telescope) project. Furthermore the HIP (Hexagonal Image Processing) framework will be used to create an efficient index schema for hexagonal images. By using its own arithmetic, image addresses can then be manipulated to translate and rotate the image space efficiently.
Experiments on FACT could not show any striking improvements of this works hexagonal convolution compared to the known convolutional strategy. Because of the hexagonal structure, the amount of trainable parameters has succesfully been reduced. This can lead to improvements regarding memory consumption and overall performance of the algorithm, if its optimized well enough.

# Inhaltsverzeichnis

**1 Einleitung** **1**
1.1 Ziele dieser Arbeit . . . . 4
1.2 Aufbau dieser Arbeit . . . . 5

**2 Adressierung hexagonaler Gitterstrukturen** **7**
2.1 Mehrdimensionale Adressierung . . . . 8
2.2 HIP System . . . . 10
2.2.1 Einführung in das Spiral-Rechensystem . . . . 13
2.3 Eingrenzung mit Ring-Indexierung . . . . 16

**3 Hexagnale Convolution** **19**
3.1 Grundlagen künstlicher neuronaler Netzwerke . . . . 19
3.2 Convolutional Neural Networks . . . . 20
3.3 Vorhersage und Training . . . . 22
3.4 Übergang zur HexConv . . . . 23
3.4.1 Hexagonaler Filter und Nachbarschaften . . . . 26
3.5 HexPooling . . . . 27

**4 Framework-Integration** **29**
4.1 Tensorflow-Operatoren . . . . 30
4.2 Python-API . . . . 33

**5 Experimentelle Daten und Netzwerke** **35**
5.1 FACT Datensatz . . . . 35
5.1.1 Datenextraktion mit `streams` . . . . 36
5.2 Konvertierung in das TFRecords-Format . . . . 37
5.3 Sampling . . . . 38
5.4 Testnetzwerk . . . . 40

**6 FACT: HexConv und Conv im direkten Vergleich** **43**
6.1 Performance . . . . 46

**7 Zusammenfassung** **49**
7.1 Ausblick . . . . 50

**A HexConv** **53**

**Literaturverzeichnis** **59**

# 1 Einleitung

Maschinelles Lernen, als Teilgebiet der Informatik, wird zur computerunterstützten Untersuchung vieler Anwendungsgebiete genutzt. Dabei werden Muster in Daten erkannt, die zur Vorhersage unbekannter Ereignisse genutzt werden. Dies kann auf einem breiten Spektrum verschiedener Bereiche angewendet werden. Beispielshalber siegte 2016 AlphaGo [33] im Spiel Go gegen den professionellen Spieler Lee Seedol. AlphaGo nutzte dabei eine Kombination aus tiefen Neuronalen Netzwerken und Baumsuche. Das Spiel Go galt lange Zeit als sehr anspruchsvoll zu Lernen aufgrund der breiten Anzahl möglicher Züge. Dies gelang AlphaGo allerdings unter anderem mit Hilfe einer sehr großen Datenbank aus vorbereiteter Spiele professioneller Spieler.
Weitere Anwendungsgebiete umfassen Sprach- und Texterkennung, Krankheitsdiagnosen, sowie Bild- und Videoanalysen. Letzteres findet in der jährlich ausgestellten ImageNet Large Scale Visual Recognition Challenge (ILSVRC) statt, in welcher sich Teams in Aufgaben zur Klassifizierung und Segmentierung von Bildern messen. Dabei werden häufig spezialisierte Neuronale Netzwerke eingesetzt, sogenannte Convolutional Neural Networks. AlexNet [24] nutzte diese 2012 und erreichte damit den ersten Platz. In den darauf folgenden Jahren wurden neue Konzepte vorgestellt und bestehende weiter optimiert, sodass beispielsweise 2014 VGG Net [34] und 2015 GoogLeNet[35] verbesserte Ergebnisse erzielen konnten.
Bilder bestehen aus einer Menge quadratisch geformter Pixel aufgespannt auf einem Gitter, das über eine Breite und Höhe definiert ist. Es existieren auch weitere Bildstrukturen, wie beispielsweise durch hexagonale Reflektoren aufgezeichnete Bilddaten. Diese besitzen einige vorteilhaften Eigenschaften gegenüber den quadratisch geformten Gitterstrukturen. Darunter fällt die Isoperimetrie, eine äquivalente Distanz benachbarter Pixel, sowie eine eindeutige Nachbarschaft, bestehend aus genau sechs direkt an Kanten angrenzenden Pixeln. Aus der isometrischen Ungleichung kommt hervor, dass Hexagone mehr Fläche abdecken als Quadrate und somit eine höhere Dichte besitzen. Da jeder hexagonale Pixel sechs Nachbarn besitzt, die jeweils um 60° voneinander abstehen, können Kurven besser als in rechteckigen Gittern dargestellt werden. Diese besitzen zum Vergleich nur vier solcher Nachbarn. Zuletzt sind Ränder und Bildregionen besser abgrenzbar, da eine eindeutige Nachbarschaft existiert. Eine Nachbarschaft ist dabei als die Menge der direkt angrenzenden Pixel definiert. In rechteckigen Gittern können Nachbarschaften aus vier, also nur den über Kanten miteinander verbundenen, oder acht Pixeln geformt werden. Hexagonale Nachbarschaften sind eindeutig definiert, da alle angrenzenden Nachbarn direkt über Kanten miteinander verbunden sind. Es existieren also keine Grenzfälle, die mit einbezogen werden müssen. Weiterhin ist eine Ähnlichkeit zum menschlichen visuellen System bemerkbar, im welchem Stäbchen und Zäpfchen hexagonaler Form zur Aufnahme der Lichtimpulse vorkommen und auf der Fovea in hexagonalen Gittern aufgereiht werden [5].
Dennoch besteht wenig Interesse an hexagonale Gitter in der Praxis. Ein Grund

dafür liegt womöglich am notwendigen Sampling der rechteckigen Gitter zu Hexagonalen. Denn bestehende Aufnahmesysteme wie Kameras nutzen in der Regeln quadratische Pixel zur Generierung der Bilddaten. Doch gerade aufgrund der eben erwähnten Eigenschaften sollten vergleichbare bis hin zu besseren Ergebnisse mit hexagonalen Gittern erzielt werden können. Mit weniger Datenpunkte sollten also vergleichbar gute Ergebnisse erbracht werden.
Neben bekannten Bilddatensets, wie CIFAR-10/100 [23], MNIST [25] und ImageNet [12], welche allesamt rechteckig gesampelte Bilder enthalten, sammelt das FACT Project [4, 3] astrophysikalische Daten. Ein in La Palma stationiertes Teleskop zeichnet dabei Gammaschauer im Bereich einiger hundert GeV bis hin zu 10 TeV auf. Hoch energetische Gammateilchen lösen in der Atmosphäre durch Aufprallen mit Partikel sogenannte Tscherenkow-Strahlung aus, die die eben erwähnten Gammaschauer verursacht. Neben diesen Schauern wird auch Hintergrundrauschen aufgezeichnet, welches ebenfalls dem Weltraum entstammt und mit Partikeln der Atmosphäre interagiert, allerdings nun aufgrund geladener Teilchen. Wegen ihrer Ladung interagieren diese bei ihrem Weg zur Erde mit elektromagnetischen Feldern und kommen so von einem direkten geradlinigen Weg ab. Von Interesse sind daher nur Gammaschauer, welche aufgrund ihrer fehlenden Ladung geradlinig auf die Erde treffen. Aus der aufgezeichneten Tscherenkow-Strahlung kann dann die ursprüngliche Position ermittelt werden. Dazu müssen zunächst beide Events voneinander differenziert werden. Die Differenzierung beider Events ist eine typische Aufgabe des maschinellen Lernens. Gammaschauer treten jedoch sehr viel seltener auf als Hintergrundrauschen, mit Verhältnissen von 1:1000 bis 1:10000 [8, 4], weshalb die korrekte Identifizierung Ersterer von Bedeutung ist. In einer Arbeit von Bock et al. [6] werden verschiedene Verfahren zu dieser Problemstellung getestet und miteinander verglichen. Enthalten sind unter anderem Random Forest, Neuronale Netzwerke und Support Vector Machines. Sie ermittelten, dass alle Verfahren ähnliche Ergebnisse erbringen konnten. Neuronale Netzwerke schienen allerdings Schwächen zu zeigen, wenn die gewählten Parameter nicht optimiert sind.
Algorithmisch lassen sich Trainigsverfahren in zwei Bereiche einteilen, dem überwachten und unüberwachten Lernen. Beim unüberwachten Lernen werden keine Klassenzugehörigkeiten in den Trainingsdaten benötigt. Dazu gehören größtenteils Clustering-Verfahren, welche Datengruppen aus den Messwerten erschließen und so eine Klassifikation derer ermöglichen. Überwachtes Lernen dagegen benötigt Trainingsdaten mit Klassenzugehörigkeit. Anhand der Daten und Klassen kann eine Kostenfunktion optimiert werden, welche den Fehler falscher Vorhersagen versucht zu minimieren. Dazu werden Gradientenverfahren benötigt, welche den erschlossenen Wert wieder zurück in die Funktion einführen und so die Kostenfunktion optimieren. Diese Arbeit fokussiert sich auf derartige Verfahren.

Das Training von FACT erfolgt mittels gelabelten Daten aus Monte-Carlo Simulationen. Diese erzeugen physikalisch korrekte Zufallsergebnisse, welche die Energie des ausgelösten Teilchen und weitere Merkmale beinhalten. Zudem können diesen Ereignissen eine Klasse zugeordnet werden, die für das Training benötigt wird. Die Daten selbst liegen als Viedeos bestehend aus 300 Einzelbildern in einer Auflösung von jeweils 1440 Pixeln vor. Der Zeitraum dieser Aufnahme umspannt lediglich ei-

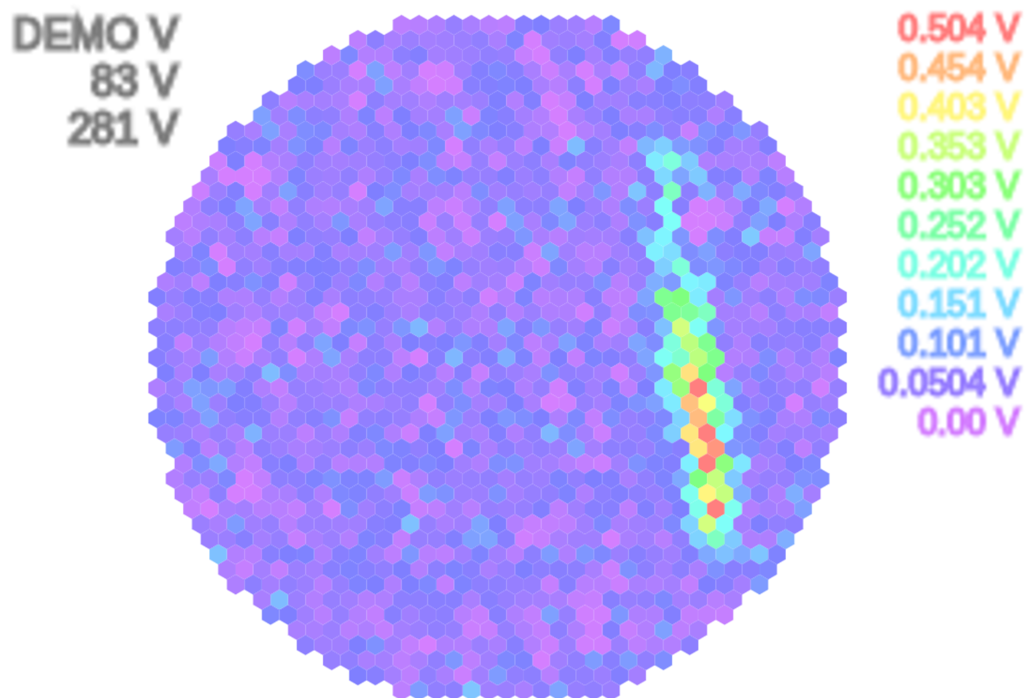



**Abbildung 1.1:** Auszug einer Aufzeichnung aus dem Monitoring-Tool `smartfact`[1]. Die Legenden wurden zur Lesbarkeit vergrößert. Gezeigt sind die einzelnen Pixel einer FACT-Aufnahme. Die Farbwerte entsprechen den Spannungswerten.

nige hundert Nanosekunden [8], weswegen Daten nur bei erreichen voreingestellter Grenzwerte gespeichert werden. So aufgezeichnete Events müssen dann weiter verarbeitet und anschließend in eine der beiden Klassen, Gamma oder Hadron, eingeteilt werden. Deswegen ist diese Aufgabe auch als Gamma-Hadron-Separation bekannt. Das Teleskop selbst besteht aus hexagonal geformten Reflektoren, weshalb aufgezeichnete Daten ebenfalls einem hexagonalen Gitter zugeordnet werden, wie in Abbildung 1.1 zu erkenne ist. Es wird eine farblich hervorgehobene Region gezeigt, die eine elliptische Form aufweist. Gamma und Hadron Events erzeugen beide diese Formen, jedoch unterscheiden sich beide Klassen in der Ausprägung dieser Ellipse.

Convolutional Neural Networks (CNN) können hier eingesetzt werden, um die elliptischen Muster zu erlernen, um so eine mögliche Differenzierung beider Klassen zu ermöglichen. In den geläufigen Frameworks werden CNNs zwar bereitgestellt, nutzen aber nur rechteckige Filter und basieren Bildern mit rechteckigen Gittern. FACT-Aufnahmen müssten daher zunächst auf rechteckige Gitter transformiert werden. Alternativ kann daher auch ein hexagonales Gitter mit hexagonalen Filtern erprobt werden. Dies verlangt allerdings eine Anpassung des bekannten Convolution Verfahren durch Umstellungen der Strukturen auf hexagonale Formen.

[1] `https://www.fact-project.org/smartfact`

Asharindavida, Hundewale und Aljahdali [5] erläutern die verschiedenen Methoden zum Sampling hexagonaler Bilder und ihre Adressierung. Da durch den Samplingprozess häufig auch geometrische Eigenschaften verloren gehen, erarbeiteten Wu, He und Hintz [38] ein virtuelles System, welches die Spirale Architektur zur Adressierung nutzt. Mit diesen System erhalten die gesampelten Bilder ihre hexagonalen Eigenschaften.
Philip Sheridan [32] erarbeitete ein Adressierungsverfahren, welches effektiv Translationen und Rotationen des Bildes umsetzen kann. Aufgrund der sogenannte Spiral Honeycomb Lie Algebra (SHLA) können durch Addition und Multiplikation Translation und Rotation berechnet werden. Middleton und Sivaswamy [27] vertiefen in ihren Werk diese Spirale Architektur und thematisieren hexagonale Bildanalyse ganzheitlich. Eine detailliertere Beschreibung zu diesem Verfahren befindet sich in Kapitel 2.
Scotney, Coleman und Gardiner [10, 15, 31] erarbeiten Operatoren zur hexagonalen Convolution zur effektiven Kantenerkennung. Das Verfahren nutzt die Spirale Architektur und Integralbilder, nach Vorbild von Crow [11], um eine schnelle und oberflächliche Kantendetektion zu ermöglichen. In nur sieben Subtraktionen und Multiplikationen sollen so Ergebnisse berechnet werden können. Das Verfahren soll vor allem in Echtzeitanwendungen ihren Einsatz finden.
Mit HexaConv stellen Hoogeboom et al. [20] ein hexagonales Verfahren vor, welches hexagonale Radialsymmetrien unter Ausnutzung von sogenannten *group convolutions* beinhaltet. Aufgrund der höheren Radialsymmetrie von Hexagonen gegenüber Quadraten erzielte ihre entwickelte HexaConv bessere Ergebnisse auf einem Testdatensatz gegenüber einer klassischen Convolution.

## 1.1 Ziele dieser Arbeit

Im Umfang dieser Arbeiten sollen tiefe Convolutional Neural Networks anhand eines FACT-Datensatz zur Klassifizierung von Gamma und Hadron Events trainiert werden. Dabei soll das Verhalten hexagonalen Gitter in Verbindung mit CNNs näher untersucht werden. Motiviert durch die erwähnten vorteilhaften Eigenschaften dieser Gitter werden bessere Resultate als mit rechteckigen Verfahren erwartet.
Zum Erreichen dieser Netzwerke müssen zunächst unterschiedliche Aspekte zur hexagonalen Bildanalyse näher untersucht werden.

**Adressierung hexagonaler Bilder.** Es existieren unterschiedliche Adressierungsarten hexagonaler Bilder. In dieser Arbeit werden verschiedene Verfahren auf ihre Effektivität bezüglich Speicheraufwand und Nutzbarkeit untersucht. Es wird eine Spiral-Adressierung am Vorbild vom HIP-System [27] implementiert, da durch dieses Bildtransformationen schnell und effizient berechnet werden können.

**Hexagonale Convolution.** Aufgrund der Unterschiede zwischen hexagonalen und rechteckigen Gittern muss eine neue Art der Convolution basierend auf Hexagone erstellt werden. Ziel ist es darauf zu achten, keine hexagonale Approximation in rechteckigen Formen zu erarbeiten, sondern die ursprüngliche Form beizubehalten.

Die erforderlichen Umstellungen des bekannten Verfahrens auf hexagonale Formen werden hier erarbeitet. Dazu gehören die Veränderungen von Padding, Schrittweite und Dilation, sowie die effiziente Berechnung von Nachbarschaften, welche Eingaben und Filter miteinander verbinden.

**Vergleich beider Ansätze.** Ein weiteres Interesse dieser Arbeit gelten den vermuteten Verbesserungen von hexagonalen CNNs gegenüber rechteckiger Verfahren. Anhand des FACT-Datensatzes werden beide Ansätze miteinander verglichen. Dazu wird ein Testnetzwerk erstellt, welches sowohl für hexagonale als auch rechteckiger Convolution anwendbar ist. Untersucht werden hier die erreichte Accuarcy, Precision, Recall und AUC aus mehreren Trainingseinsätzen.

## 1.2 Aufbau dieser Arbeit

Das folgenden Kapitel 2 stellt die Mehrdimensionale und Spirale Adressierung hexagonaler Bilder vor. Es wird das Spiral-Rechensystem vorgestellt, welches dem HIP-System entstammt. Eine Ring-Indexierung wird eingeführt, welche die Größe und Form der Bildstrukturen eingrenzen soll.
Kapitel 3 führt die Grundlagen zu Funktion der CNNs ein. Weiterhin wird die hexagonale Convolution, im folgenden auch HexConv genannt, vorgestellt, welche zur nativen Nutzung hexagonaler Strukturen eingesetzt werden kann. Dazu werden Funktionsänderungen durch angepasste Padding, Schrittweite und Dilation vorgestellt.
Ein Überblick zur Integration von HexConv ins TensorFlow [2] werden in Kapitel 4 vorgestellt und dessen Nutzung näher beschrieben. Eine Python-API steht TensorFlow zur Verfügung, welche um der in dieser Arbeit implementierten HexConv erweitert wurde.
Mit Kapitel 5 wird der FACT-Datensatz und die Feature-Extraktion für weitere Versuche beschrieben. Dazu werden verschiedene Sampling-Verfahren näher untersucht und über eine mögliche Nutzung dieser diskutiert. Das hier genutzte Testmodell, ein einfaches sequentiell verkettetest Modell bestehend aus Convolution-, Normalisierung und Aktivierungsebenen wird ebenfalls eingeführt.
Kapitel 6 stellt die Resultate aus Testanwendungen von HexConv und normaler Convolution anhand des FACT-Datensatzes vor. Es wird ein Vergleich beider Verfahren hergeleitet und ihre Unterschiede sowie Gemeinsamkeiten ausgeführt. Zudem wird ein Blick auf die Performance der entwickelten HexConv geworfen, indem sie wieder mit der vorhandenen Methode verglichen wird.
Ein abschließendes Resümee gezogen, welches die erreichten Aspekte der Arbeit noch einmal zusammenfasst und einen Ausblick auf mögliche Erweiterung gibt.

# 2 Adressierung hexagonaler Gitterstrukturen

Bilder bestehen aus einer Menge geordneter Pixel, welche meist auf rechteckige Gitter angeordnet und so in zweidimensionalen Arrays gespeichert werden. Jeder Pixel $p$ kann eine Nachbarschaft $N$ zugeordnet werden. Eine Nachbarschaft ist dabei die Menge der angrenzenden Pixel. Diese kann in mehreren Größen aufgeteilt werden. Betrachtet man den Spezialfall der rechteckigen Gitter, dann ist es offensichtlich, dass einem Pixel mehrere Nachbarschaften zugewiesen werden können. Beispielshalber besteht die Nachbarschaft $N_4$ aus den über Kanten verbundenen Nachbarn. Anhand von Abbildung 2.1 kann dies direkt abgelesen werden. Dementsprechend ergibt sich die Nachbarschaft als $N_4(p) = \{p - m, p - 1, p + 1, p + m\}$. Dabei sei $m$ als Breite des Gitters und $n$ als Höhe zu vermerken. Aufgrund der quadratischen Struktur lässt sich allerdings eine weitere Nachbarschaft definieren. Diese besteht aus insgesamt acht Pixeln und beinhaltet neben den Kantennachbarn auch die Diagonalen. Mit der selben Herangehensweise lässt sich diese neue Nachbarschaft nun auch festlegen. Die Menge der acht Nachbarn ist gegeben als $N_8(p) = N_4 \cup \{p - m - 1, p - m + 1, p + m - 1, p + m + 1\}$.

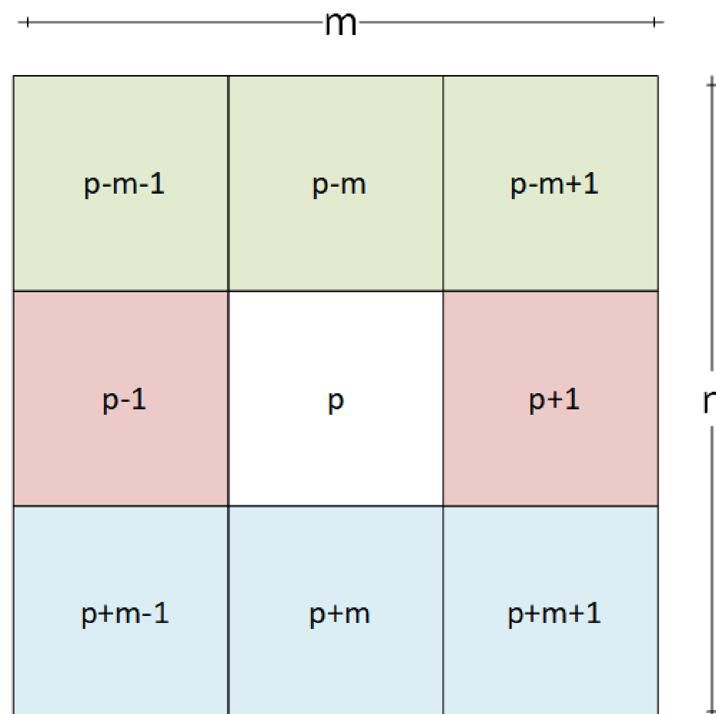



**Abbildung 2.1:** Nachbarschaft für Pixel auf rechteckigen Gittern mit zeilenweiser Indexierung

Insgesamt ist dies nur die Nachbarschaft der ersten Stufe. Eine Stufe meint hier den Umfang der Umrandung des ursprünglichen Pixels. Die Berechnung einer Stufe $n$ Nachbarschaft ist eine Notwendigkeit im Bereich der Convolution, da Filterfunk-

tionen auf Teilbereiche des Bildes anwenden. Eben solche Bereiche werden durch Nachbarschaften beschrieben.

In hexagonalen Gittern sind Nachbarschaften eindeutig definiert, da sie keine diagonal anliegenden Nachbarn besitze. Eine Nachbarschaft kann daher eindeutig definiert werden aus den über Kanten verbundenen Pixeln. Jedoch birgt eine Adressierung wie in Abbildung 2.1 Probleme auf hexagonalen Gittern mit sich, weshalb eine Definition über Breite und Höhe überdacht werden muss.
Im hexagonalen Gitter besteht ein kleiner Abstand zwischen den einzelnen Spalten und Zeilen, welcher im folgenden als Offset bezeichnet wird. Dies stört zunächst nicht für eine Adressierung selbst, allerdings besteht dadurch die Gefahr die räumliche Position der Pixel falsch einzuschätzen. Beispielshalber liegt ein Pixel der dritten Zeile und zweiten Spalte nicht auf der selben Vertikalen wie ein Pixel der zweiten Zeile und dritten Spalte. Analog kann das Gitter rotiert werden, sodass Spalte und Zeile im Beispiel ausgetauscht werden können. In dem Fall würden die Pixel allerdings nicht mehr auf der gleichen Horizontalen liegen.
Abbildung 2.2 illustriert das Problem noch einmal anhand der eingezeichneten Geraden. In hexagonalen Gitter, identisch zum rechteckigen, können Geraden über horizontale, vertikale und diagonale Linien definiert werden. Im aufgezeigten Gitter besteht ein Offset zwischen den Spalten des Gitters. Dies kann allerdings auch zeilenweise verlaufen. Eine Rotation um 90° ermöglicht dies.

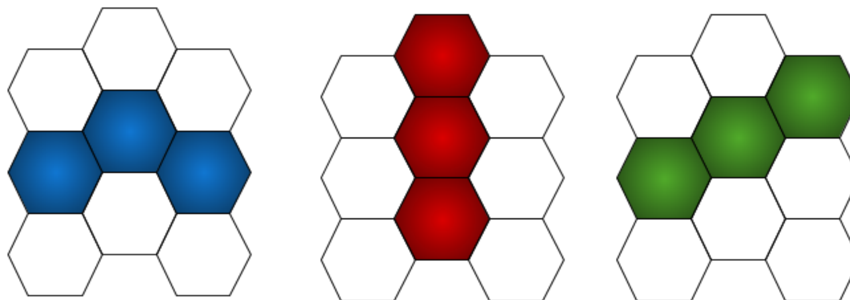


**Abbildung 2.2:** Horizontale (Links), vertikale (Mitte) und diagonale (Rechts) Geraden im hexagonalen Gitter.

Eine Definition über Spalten und Zeilen, beziehungsweise Breite und Höhe, enthält daher nicht die Information, ob ein beliebiger Pixel nun auf einer Offset-Geraden liegt oder nicht. Dies führt zu Problemen, wenn ein Bild gespiegelt, rotiert oder auch nur verschoben werden soll. Diese Aufgaben sind aber ein wichtige Aspekte von Bildanalysen, weswegen alternative Adressierungen notwendig erscheinen.

## 2.1 Mehrdimensionale Adressierung

Die Adressierung über Spalten und Zeilen lässt sich weiter verallgemeinern. Anstelle dieser Grenzen kann ein Bild beliebiger Größe auf eine Ebene projiziert werden. In dieser Art der Adressierung spannt eine Ebene aller möglichen Werte mit zwei Ach-

sen in einem 90° Winkel auf. Abbildung 2.3 zeigt einen Ausschnitt dieser Methode anhand eines hexagonalen Gitters.

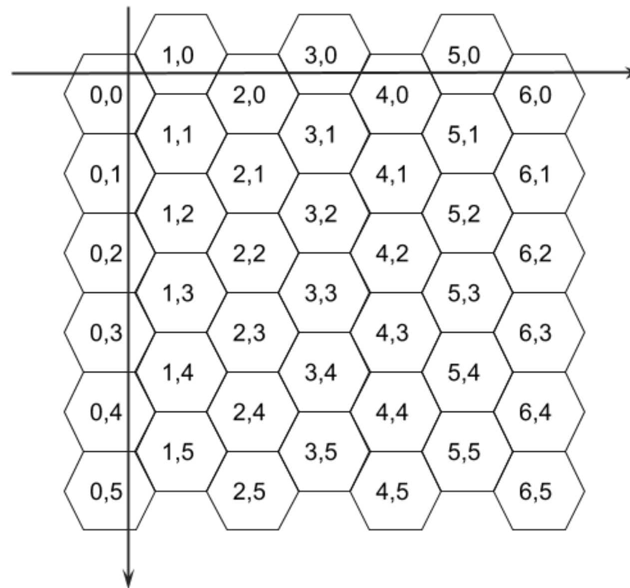



**Abbildung 2.3:** Zweidimensionale Adressierung über 90° versetze Achsen.

Dies kann als zweidimensionale Adressierung bezeichnet werden, da jeder Pixel über zwei Koordinaten eindeutig festgelegt ist. Ein Problem dieser Methode ist allerdings, dass aufgrund des bereits erwähnten Offsets keine getreue Darstellung der räumlichen Position von Pixeln dargestellt wird. Stellt man sich das beschriebenen Verfahren in einem kartesischen Koordinatensystem vor, so würde jeder Pixelposition ein Vektor beschreiben. Die Position dieser Vektoren ist dann allerdings nicht der Realität entsprechend. Anhand Abbildung 2.3 ist offensichtlich zu erkennen, dass die Koordinaten keinen Vektorraum beschreiben. Beispielsweise liegen die Mittelpunkte der hexagonalen Pixel $p_1 = (1, 0)^\intercal$ und $p_2 = (2, 0)^\intercal$ nicht auf der selben horizontalen Achse. Die würde man allerdings von einem Vektorraum erwarten.
Dieses Problem kann umgangen werden, indem anstelle des rechten Winkels die Achsen um 60° oder 120° versetzt zueinander angelegt werden, wie in Abbildung 2.4 illustriert.
Nach Vorbild von Her [19, 18] kann dieses um eine dritte Achse erweitert werden, welche jedoch aus der Linearkombination der beiden Anderen berechnet werden kann. Das drei-Achsen System entspricht einer Projektion des dreidimensionalen Kartesisches Koordinatensystem $\mathbb{R}^3$ und besitzt folglich dessen Eigenschaften.
Diese Art der mehrdimensionale Adressierung ist sehr dynamisch, weil von keiner festen Struktur ausgegangen wird. Durch die Achsen wird eine Ebene aufgespannt, in welcher jede zweidimensionale Form adressiert werden kann. Die Abspaltung vom orthogonalen System war ebenfalls Notwendig, um die Verbindung zum Kartesischen Koordinatensystem wieder herzustellen.

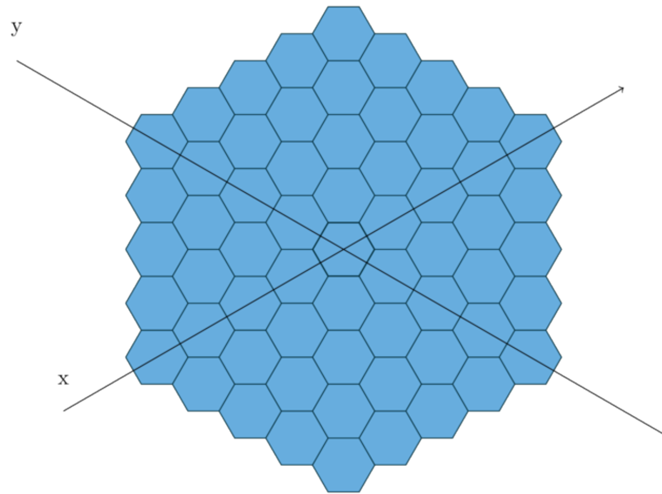



**Abbildung 2.4:** Beispielhafte Platzierung der um 60° verzerrten Achsen in einer zweidimensionalen Adressierung.

## 2.2 HIP System

Am Vorbild von Middleton and Sivaswamy [27] kann ein hexagonales Gitter auch über einen einzelnen Index adressiert werden. Sie entwickelten dazu in ihrer Arbeit das Hexagonal Image Processing (HIP) System, welches ein Verfahren zur effizienten Adressierung hexagonaler Bilder beschreibt. Die folgenden Definitionen entstammen dem Buch und wurden zu illustrativen Zwecken um Beispiele erweitert.
Das Verfahren beginnt im Mittelpunkt des Bildes und nummeriert Nachbarn in Clustern selber Größe. In Abbildung 2.5 ist ein sogenannter Level 1 Cluster zu erkennen, welcher aus dem Mittelpunkt, einem Level 0 Cluster, und den sechs benachbarten Level 0 Clustern aufgebaut ist.

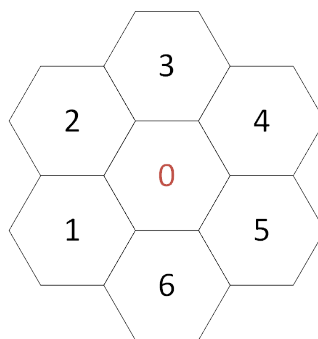



**Abbildung 2.5:** Nummerierung eines Level 1 Cluster im HIP System. Die Nummerierung startet im Zentrum (0) und setzt sich in einer spiralen Bewegung fort (1-6).

Das Verfahren setzt sich iterativ fort, sodass zunächst ein Cluster aus einen einzelnen Pixel gebildet wird und alle Nachbarn der gleichen Größe weiter nummeriert werden. Dies bildet einen neuen Cluster eines höheren Levels. Im nächsten Schritt werden wieder die 6 benachbarten Cluster des neuen Levels gesucht, welche wiederum den selben Prozess durchlaufen. Die Nummerierung erfolgt dabei auf einem Basis 7 System.

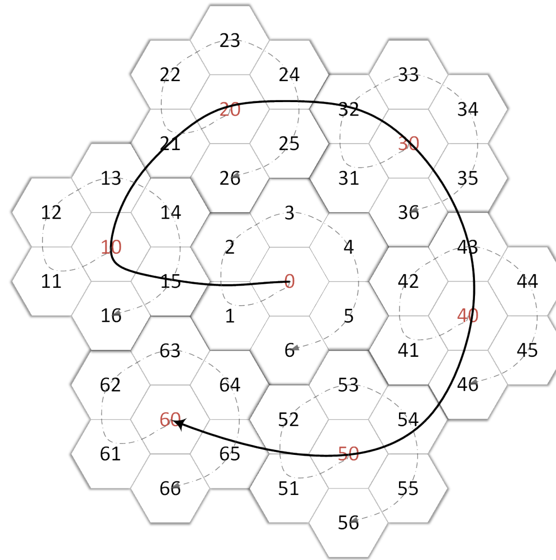



**Abbildung 2.6:** Rekursive Adressierung eines Level 2 Clusters

Wie in Abbildung 2.6 illustriert ist, startet der erste benachbarte Level 1 Cluster nun mit Index 10. Innerhalb dieses Clusters wird die Nummerierung für einen Cluster niedrigeren Levels fortgesetzt. Die Cluster gleichen Levels starten aufgrund der Basis 7 Notation daher mit den Indizes $20, 30, 40, 50$ und $60$. Dieses Verfahren kann solange fortgesetzt werden, bis alle Bildpixel von der Adressierung abgedeckt werden.
Die resultierenden Bilder bestehen immer aus $7^{\lambda}$ Pixel, wobei $\lambda$ das Cluster-Level beschreibt. Ein Level $\lambda$ Bild besteht aus sieben Level $\lambda - 1$ Cluster und vergrößert sich für jedes Level um $6 \cdot 7^{\lambda}$ Pixel. Die Stellenanzahl gibt zudem Aufschluss über das zugehörige Level. So ist eindeutig bestimmt, dass beispielsweise die Adresse ***42*** in einem Level 2 Cluster vorliegt, da sie aus zwei Ziffern zusammengesetzt wird.

Alle Adressen des HIP Systems werden in Basis 7 Notation verfasst, entspringen aber ursprünglich einer zweidimensionaler Adressierung mit zwei schrägen Achsen, wie in Abbildung 2.4 gezeigt. Der Basisvektor dieser Ebene sei festgelegt als:

$$B = \{\begin{pmatrix}1\\0\end{pmatrix}, \frac{1}{2}\begin{pmatrix}-1\\\sqrt{3}\end{pmatrix}\}. \tag{2.1}$$

Mit einem so definierten Basisvektor $B$ wird nun eine Ebene aufgespannt, welche alle möglichen Adressen beinhaltet. Sei die Menge aller ganzzahliger Punkte auf dieser Ebene definiert als $A$ und sei diese aufteilbar in $\lambda$ weitere Untermengen $A_i$

mit $i \leq \lambda$, so kann diese Ebene als hierarchische Kachelung aus $A_\infty$ Untermengen umgesetzt werden. Zusätzlich können die Eigenschaften des Koordinatenraumes weiterverwendet werden, insbesondere die Vektoraddition, die Grundlage zur Arithmetik im hexagonalen Raum ist.
Wie bereits beschrieben, befindet sich der Ursprung am Punkt $(0,0)$ relativ zum Basisvektor $B$, der gleichzeitig den ersten Cluster $A_0$ definiert. Das erste Level bildet sich dann aus den Punkten der benachbarten Level 0 Cluster. Gegeben einer linksdrehenden Spirale um das Zentrum herum, wie es in Abbildung 2.5 bereits gezeigt wurde, definieren diese sieben Punkte den Cluster $A_1$ wie folgt:

$$A_1 = \{A_0, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}-1\\0\end{pmatrix}, \begin{pmatrix}-1\\-1\end{pmatrix}, \begin{pmatrix}0\\-1\end{pmatrix}\}.$$

Es ergibt sich folgenden Mengenhierarchie für die Cluster eines beliebigen Levels $\lambda$:

$$A_0 \subset A_1 \subset A_2 \cdots \subset A_\lambda, \qquad \lambda \in \mathbb{N}_0 \tag{2.2}$$

Folgende Level $\lambda$ Cluster lassen sich aus den bereits gebildeten Level $(\lambda-1)$ Clustern berechnen. Dazu kann folgende Translationsmatrix genutzt werden:

$$N_{\lambda-1} = \begin{pmatrix}3 & -2\\2 & 1\end{pmatrix}^{\lambda-1}. \tag{2.3}$$

Ein Level $\lambda$ Cluster berechnet sich dann mithilfe von Gleichung 2.3 aus:

$$A_\lambda = N_{\lambda-1}A_1 \cup \cdots \cup N_1 A_1 \cup A_1. \tag{2.4}$$

Die Multiplikation einer Menge $A$ mit der Translationsmatrix $N$ definiert sich hier als das Produkt aus jedem Element und der Matrix.
Es fällt auf, dass die Elemente innerhalb dieser Mengen immer in Gruppen der Größe sieben unterteilt werden können, was auf dem hierarchischem Aufbau zurückführen lässt. Zudem liegt jeder Menge $A_i$ eine festgeschriebene Reihenfolge vor, die bei der kleinsten Teilmenge $A_0$ startet. Aufgrund dieser Beschaffenheit bietet sich eine Indexierung mit Basis 7 an, die mit 0 startet und der Reihenfolge entsprechend fortgeführt wird. Die Menge $A_1$ kann also spiral-indiziert werden mit:

$$S_1 = \{0,1,2,3,4,5,6\}. \tag{2.5}$$

Sei die Menge aller möglichen Spiraladressen im euklidischen Raum festgelegt durch $\mathbb{S}^\infty$ und beschränkt durch ein Level $\lambda$ mit $\mathbb{S}^\lambda$, dass heißt die Menge beschränkt sich auf einen Level $\lambda$ Cluster, so kann eine beliebige Adresse $\mathbf{s}$ eines Pixels $p$ definiert werden mit:

$$\mathbf{s} = s_{\lambda-1} \cdots s_1 s_0, \qquad s \in \mathbb{S}^\lambda. \tag{2.6}$$

Eine Umrechnung in das Dezimalsystem erfolgt durch ein Basiswechsel mittels:

$$(s_{\lambda-1} \cdots s_1 s_0)_7 = s_{\lambda-1}7^{\lambda-1} + \cdots + s_1 7 + s_0, \qquad 0 \leq s_k < 7. \tag{2.7}$$

Dies erlaubt die Speicherung und Verwaltung von Spiraladressen in einem eindimensionalen Daten-Array mithilfe einer einzelnen Adresse.

### 2.2.1 Einführung in das Spiral-Rechensystem

Durch die Herleitung des HIP Systems aus dem Kartesischen lassen sich einige Eigenschaften von diesem mit übernehmen. Im Spezifischen ist hier Rotation, Translation, Reflexion und Gleitspiegelung von Interesse, da diese wichtige Aspekte von Bildanalysen sind.
Hexagone, wie sie in dieser Arbeit und insbesondere im HIP System verwendete werden, bestehen ausschließlich aus regelmäßigen Sechsecken. Aus der Kantenlänge $a$ lassen sich weitere Größen ableiten, wie den Radius zwischen Mittelpunkt und Kante $s = a\frac{\sqrt{3}}{2}$. Wird nun ein hexagonales Gitter im HIP System aufgespannt und besteht ein Offset zwischen den horizontalen Zeilen, wie in Abbildung 2.7 gezeigt, so beträgt der horizontale Abstand zwischen zwei benachbarten Punkten genaue $s_n = 2s$. Der Abstand zu den vertikal verschobenen Nachbarn beträgt in vektorisierte Schreibweise $\vec{d} = (a, a\frac{3}{2})^\intercal$.

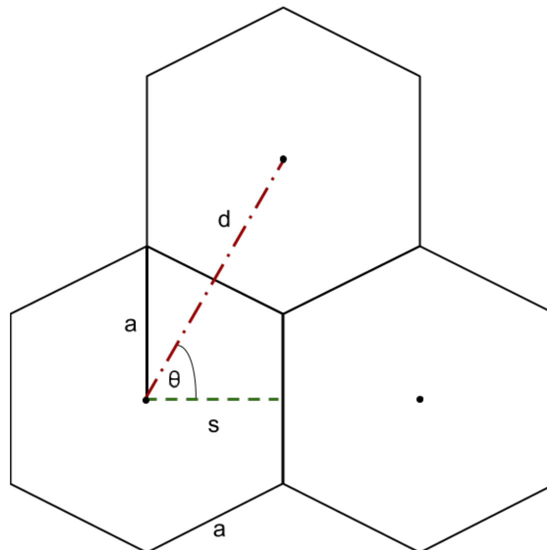



**Abbildung 2.7:** Skizze der Abstandsmaße im hexagonalen Gitter mit regelmäßigen Sechsecken. Dargestellt ist die Kantenlänge $a$, der innere Radius $s$ und der Abstand zweier vertikal versetzter Hexagone $d$.

Der dargestellte Winkel $\theta$ beträgt $60° = \frac{\pi}{3}$rad. Alle umliegenden Nachbarn werden in einem Vielfachen dieses Winkels um das Hexagon angelegt.

#### Addition

Die Spiraladdition kann nun mittels bekannter Vektoraddition definiert werden. Wenn die Spiraladressen für das erste Level auf ein Gitter aufgetragen werden, verfügt jede Adresse über einen eindeutigen Vektor, der gleichzeitig die räumliche Lage

angibt. Für eine festgelegte Kantenlänge $a = \frac{1}{\sqrt{3}}$ lassen sich die Ortsvektoren des ersten Levels in kartesischen Koordinaten wie folgt darstellen:

$$
\begin{aligned}
\boldsymbol{0} &\equiv \begin{pmatrix} 0 \\ 0 \end{pmatrix} & & \\
\boldsymbol{1} &\equiv \begin{pmatrix} -1 \\ 0 \end{pmatrix} & \boldsymbol{2} &\equiv \frac{1}{2}\begin{pmatrix} -1 \\ \sqrt{3} \end{pmatrix} \\
\boldsymbol{3} &\equiv \frac{1}{2}\begin{pmatrix} 1 \\ \sqrt{3} \end{pmatrix} & \boldsymbol{4} &\equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
\boldsymbol{5} &\equiv \frac{1}{2}\begin{pmatrix} 1 \\ \sqrt{3} \end{pmatrix} & \boldsymbol{6} &\equiv \frac{1}{2}\begin{pmatrix} -1 \\ -\sqrt{3} \end{pmatrix}
\end{aligned}
$$

Durch Vektoraddition können alle weiteren Adressen mit einer paarweisen Addition berechnet werden. Das Ergebnis aus ***1*** + ***2*** wird wie folgt berechnet:

$$
\begin{pmatrix} -1 \\ 0 \end{pmatrix} + \frac{1}{2}\begin{pmatrix} -1 \\ \sqrt{3} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} -3 \\ \sqrt{3} \end{pmatrix}.
$$

Dieser Vektor ist die exakte Position der Spiraladresse ***15***. Nach diesem Prinzip müssen noch die Adresse ***14***, ***26***, ***25***, ***31***, ***36***, ***42***, ***41***, ***53***, ***52***, ***64*** und ***53*** ausgerechnet werden. Aus diesen Adressen lässt sich eine Lookup-Tabelle generieren und mit Spiraladressen ersetzen:

| $+_\circ$ | ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** |
|---|---|---|---|---|---|---|---|
| ***0*** | ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** |
| ***1*** | ***1*** | ***63*** | ***15*** | ***2*** | ***0*** | ***6*** | ***64*** |
| ***2*** | ***2*** | ***15*** | ***14*** | ***26*** | ***3*** | ***0*** | ***1*** |
| ***3*** | ***3*** | ***2*** | ***26*** | ***25*** | ***31*** | ***4*** | ***0*** |
| ***4*** | ***4*** | ***0*** | ***3*** | ***31*** | ***36*** | ***42*** | ***5*** |
| ***5*** | ***5*** | ***6*** | ***0*** | ***4*** | ***42*** | ***41*** | ***53*** |
| ***6*** | ***6*** | ***64*** | ***1*** | ***0*** | ***5*** | ***53*** | ***52*** |

**Tabelle 2.1:** Lookup-Tabelle der Spiraladdition. Addiert werden Elemente der äußeren Ränder immer paarweise. So kann das Ergebnis der Spiraladdition jeder einstelliger Spiraladressen direkt abgelesen werden.

Eine Spiraladdition berechnet sich ziffernweise zusammen mit Übertrag. In Tabelle 2.1 ist erkennbar, dass einige Summen in höheren Spiral-Level liegen, weshalb der Übertrag benötigt wird. Die Adressen ***25*** und ***36***, wie sie in Abbildung 2.8 orientiert sind, addieren sich mit dieser Tabelle wie folgt zusammen:

$$
\begin{array}{ccc}
 & {\scriptstyle \boldsymbol{5}} & \\
 & \boldsymbol{2} & \boldsymbol{5} \\
+ & \boldsymbol{3} & \boldsymbol{6} \\
\hline
 & \boldsymbol{3} & \boldsymbol{3}
\end{array}
$$

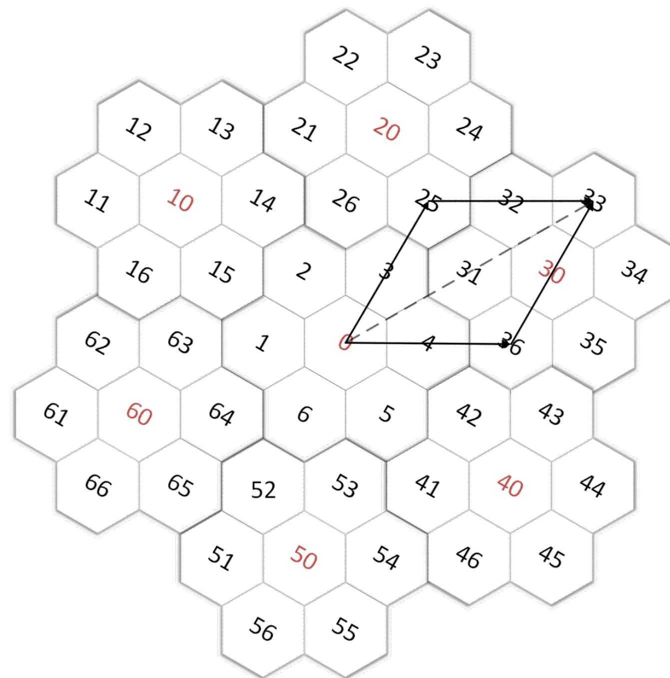



**Abbildung 2.8:** Spiraladdition in Vektordarstellung. Dargestellt wird die Addition der Element ***25*** und ***36***. Das Ergebnis ***33*** entstammt der Vektoraddition beider Adressen.

Es spielt hierbei keine Rolle, ob der Übertrag zuerst addiert wird, oder zum Schluss. Die Addition von ***5*** und ***2*** benötigt in diesem Fall allerdings weniger Schritte, da kein höheres Spiral-Level erreicht wird. Die Reihenfolge der Addition ist irrelevant, da weiterhin die Kommutativität gilt.
Aus Tabelle 2.1 lassen sich noch weitere bekannte Muster erkennen. So ist ***0*** ein neutrales Element und für jede Adresse existiert genau ein Inverses. Das HIP System erfüllt sogar noch weitere bekannte Eigenschaften, namentlich Abgeschlossenheit und Assoziativität.
Mittels der Inversen lässt sich auch eine Subtraktion definieren. Wie aus der Vektorarithmetik bekannt, lässt sich die Subtraktion auch als Addition mit dem inversen Element ausdrücken. Das Inverse von ***25*** ist ***52***, was sich aus der Lookup-Tabelle leicht ablesen lässt. Die Subtraktion $\boldsymbol{33} - \boldsymbol{25} = \boldsymbol{36}$ ist also äquivalent zur Addition $\boldsymbol{33} + \boldsymbol{52} = \boldsymbol{36}$. Auch dies ist eine logische Schlussfolgerung der Vektorschreibweise.
Die Verfügbarkeit von bekannten Addition- und Subtraktion-Operatoren hat einen großen Einfluss auf Bildoperatoren. Auf diese Weise lässt sich beispielsweise ein Bildausschnitt schnell berechnen, indem eine beliebige Adresse als neues Zentrum ausgewählt und ihr Inverses auf alle Adressen addiert wird. Intuitiv verschiebt sich das Bild im Zentrum um diesen Wert. Addition und Subtraktion simulieren daher eine Verschiebung des Bildes.
Während der Convolution muss die Filter-Nachbarschaft jeder Adresse gefunden werden. Dies ermöglichen Addition und Subtraktion effizient zu berechnen.

### Multiplikation

Eine Multiplikation von Spiraladressen ist ebenfalls möglich und baut auf den selben Prinzipien auf wie die Addition. Durch Polarkoordinaten in Exponentialform lässt sich dies einfach herleiten. Die direkten Nachbarn eines Hexagons lassen sich über den Abstand ihrer Mittelpunkte $r_x$ und dem angrenzenden Winkel $\theta_x$ festlegen. Eine beliebige Spiraladresse ist daher eindeutig ablesbar aus:

$$\boldsymbol{a} \equiv r_a \cdot e^{i\theta_a}, \qquad \boldsymbol{a} \in \mathbb{S}^\infty \tag{2.8}$$

Die Multiplikation zweier Adressen ist damit:

$$\boldsymbol{a} \times_\circ \boldsymbol{b} = r_a r_b \cdot e^{i(\theta_a + \theta_b)} \tag{2.9}$$

Dies führt daher zu einer Rotation der Adresse $\boldsymbol{a}$ um $\theta_b$ und eine Skalierung um $r_b$. Für beliebige Multiplikation muss wieder eine Lookup-Tabelle gefunden werde, um ein ziffernweises Multiplizieren in Spiraldarstellung zu ermöglichen. Diese setzt sich analog zur Addition aus der paarweisen Multiplikation mit den einstelligen Adressen zusammen.

| $\times_\circ$ | ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** |
|---|---|---|---|---|---|---|---|
| ***0*** | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** |
| ***1*** | ***0*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** |
| ***2*** | ***0*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***1*** |
| ***3*** | ***0*** | ***3*** | ***4*** | ***5*** | ***6*** | ***1*** | ***2*** |
| ***4*** | ***0*** | ***4*** | ***5*** | ***6*** | ***1*** | ***2*** | ***3*** |
| ***5*** | ***0*** | ***5*** | ***6*** | ***1*** | ***2*** | ***3*** | ***4*** |
| ***6*** | ***0*** | ***6*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** |

**Tabelle 2.2:** Lookup-Tabelle der Spiralmultiplikation

Es ergibt sich, analog zur Addition, Tabelle 2.2 für die Multiplikation. Das neutrale Element der Multiplikation ist nun ***1***, während inverse Elemente vom Winkel $\theta$, als auch dem Skalar $r$ abhängen. Es muss also gelten:

$$\texttt{inv}_{\texttt{mul}}(\boldsymbol{a}) = \boldsymbol{b}, \qquad r_a r_b \cdot e^{i(\theta_a + \theta_b)} = \boldsymbol{1} \tag{2.10}$$

Eine Spiraldivision ist auch möglich, indem die selben Techniken wie zur Subtraktion in Spiraldarstellung genutzt werden. Im Wesentlichen wird die Division zur Multiplikation mit dem inversen Element umgeformt, mit ein paar weiteren Einschränkungen. Da sowohl Spiralmultiplikation als auch Spiraldivision im Bereich der Convolution wenig Gebrauch finden, wird für weitere Informationen auf die Hauptliteratur des HIP-Frameworks [27] verwiesen.

## 2.3 Eingrenzung mit Ring-Indexierung

Die bekannte Bildstruktur lässt sich in drei Dimensionen aufteilen. Breite, Höhe und Farbkanäle ($W \times H \times C$) sind die Geläufigsten. Für hexagonale Bilder bietet sich

die Anzahl der Ringe $r$ als Alternative für Breite und Höhe an. Aus ihnen kann die Anzahl der Bildelemente einfach berechnet werden. Sei ein Zentrum immer gegeben, das heißt ein Bild $I$ ist nur dann definiert, wenn mindestens ein Pixel vorhanden ist, so beschreibt ein Bild mit keinem Ring genau diesen Pixel.

$$I(0) = 1 \tag{2.11}$$

Jeder weitere Ring erhöht die Anzahl um Faktor sechs, gegeben dem Ringlevel. Das Erste enthält somit $I(1) = 1{+}6{\cdot}1 = 7$ Elemente. Das Zweite $I(2) = 1{+}6{\cdot}1{+}6{\cdot}2 = 19$, das Dritte $I(3) = 1{+}6{\cdot}1{+}6{\cdot}2{+}6{\cdot}3 = 37$ und jeder weitere Ring setzt dieses Muster fort. Zusammengefasst kann die Anzahl der Elemente im Bild aus der Ringanzahl berechnet werden durch:

$$\begin{aligned} I(r) &= 1 + 6 \cdot 1 + 6 \cdot 2 + ... + 6 \cdot r \\ &= 1 + 6 \cdot (1 + 2 + ... + r) \\ &= 1 + 6 \cdot \frac{r \cdot (r+1)}{2} && (\textit{Gaußsche Summenformel}) \\ &= 1 + 3 \cdot r \cdot (r+1). \end{aligned}$$

Für beliebige hexagonale Bilder $H$, welche aus Ringen zusammengesetzt sind, lassen sich folgendermaßen deren Pixelanzahl berechnen:

$$\mathbf{R}(r) = 3r^2 + 3r + 1, \qquad r \in \mathbb{N}_0 \tag{2.12}$$

Im Vergleich zur Spiraldarstellung, die ein exponentielles Wachstum der Level entsprechend besitzt, gilt hier nur ein quadratisches Wachstum anhand des Ringlevels. Allerdings ist diese Adressierung wieder nur über zwei Dimensionen direkt abrufbar, dem Ring $r$ und der Position innerhalb des Ringes $x$.
Dennoch bietet diese Adressierung die Möglichkeit dynamische Größeneinstellungen vorzunehmen, welche gerade in CNNs häufig eingesetzt werden. Zwischen den einzelnen Ebenen der Netzwerke findet häufig ein Downsampling statt, um so eine Parameterreduktion durchzuführen. Das Downsampling meint hier die Verkleinerung des Eingabebildes.
Eine Größenveränderung verändert hierbei auch nicht die Form des Bildes. Die Hinzunahme von Ringen führt weiterhin zu einem hexagonalen Umriss, während bei Spiraler Adressierung eine Schneeflockenform entsteht. Dies ähnelt den bekannten rechteckigen Verfahren, im welchen größere Bilder weiterhin eine rechteckige Form annehmen.
Um weiterhin die Vorteile des Spiral-Rechensystems nutzen zu können, sollte diese Adressierung lediglich genutzt werden, um die Daten im Array zu indexieren. Der benötigte Speicherplatz verringert sich zunächst, da nicht mehr exponentiell viele Bildpixel gespeichert werden müssen. Lediglich eine Lookup-Tabelle zum Zuordnen der Adressen wird benötigt. Die Ring-Indexierung begrenzt zu große Spiralbilder und vereinheitlicht dessen Form. Die Zuweisung der Spiral- und Spiraladressen beginnt wieder im Zentrum des Bildes. Anschließend werden in einer Kreisbewegung alle umliegenden Pixel durchlaufen, identisch zum adressieren eines Level 1 Cluster.

Dazu wird eine Bewegung in Richtung der Spiraladresse ***1*** durchgeführt. Dadurch wird ein neues Ringlevel $r$ erreicht, welcher nun komplett durchlaufen werden muss. Dieser Rundlauf enthält genau $6r$ Element, welche den Rand der hexagonalen Form bilden. Sobald der Ring komplett durchlaufen wurde und man wieder in der Startposition ***1*** steht, wird eine Bewegung zum nächsten Ringlevel gestartet, indem wieder eine Spiraladdition mit ***1*** durchgeführt wird. Die Startposition des neuen Ringlevels ist demnach $\boldsymbol{63} = \boldsymbol{1} +_{\circ} \boldsymbol{1}$. Der zugehörige Ringindex dazu ist 7. Die Fortsetzung dieses Verfahrens für eine vollständige Zuordnung eines Level 4 Clusters und einer Ringindexierung mit vier Ringen kann Abbildung 2.9 entnommen werden.

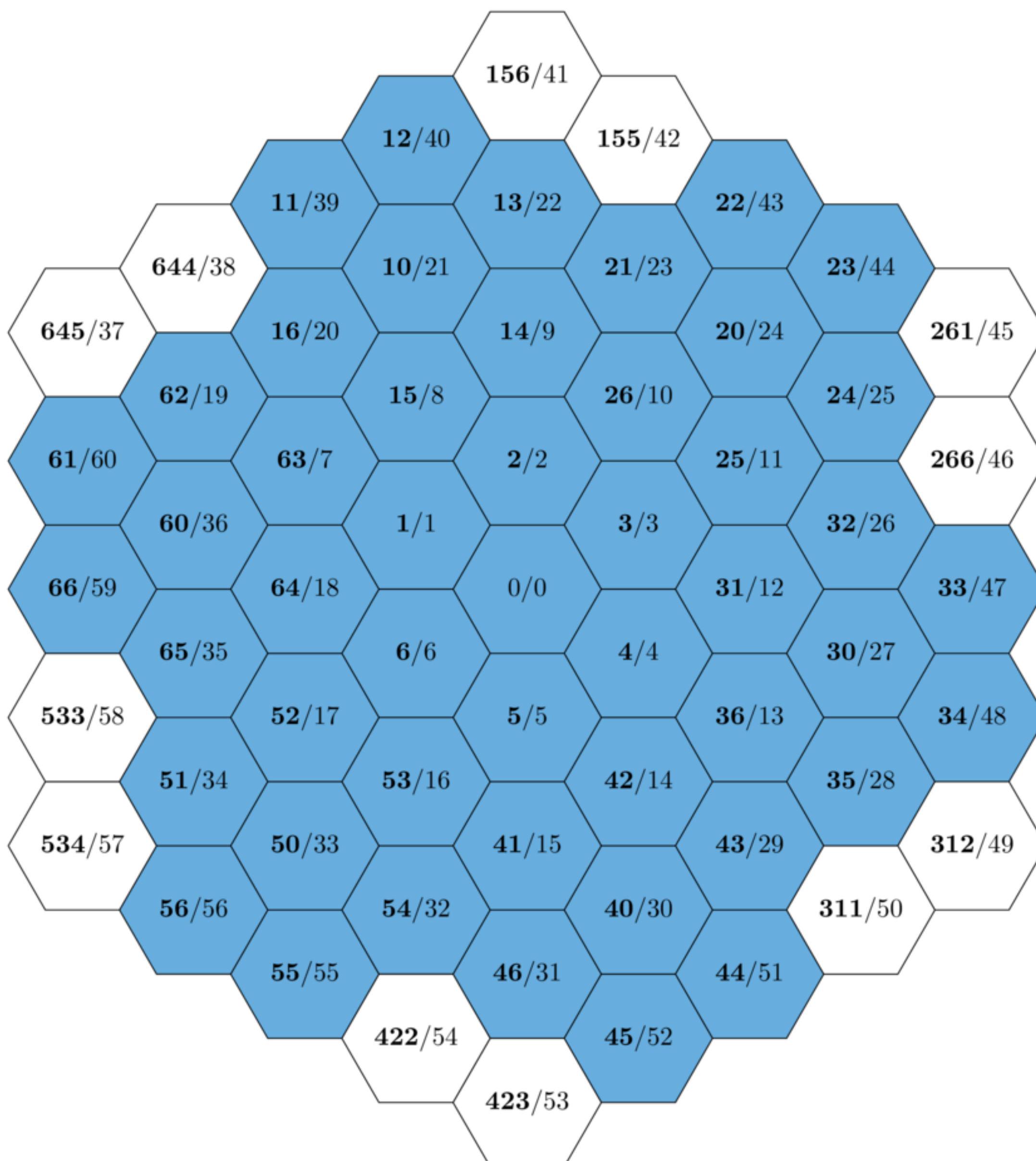



**Abbildung 2.9:** Ringindexierung mit vier Ringen und zugehörige Spiraladressen (Spiralaresse/Ringindex). In blau hervorgehoben sind die Spiraladressen aus $\mathbb{S}^4$.

# 3 Hexagnale Convolution

Die hexagonale Convolution, im weiteren als HexConv bezeichnet, ist eine Umsetzung des als Convolution bekannte Verfahren, welches hexagonale Bilder als Input verwendet. Die allgemeine Struktur der Convolution soll bestehen bleiben. Dabei entsteht intuitiv eine Ringconvolution, welche aufgrund des HIP Systems dynamisch vergrößert und verkleinert werden kann. Dafür müssen lediglich die Anzahl der Umrandungen mit einbezogen werden. Unter Berücksichtigung des HIP Systems und vorher generierten Lookup-Tabellen lässt sich dann die Indexierung effizient durchlaufen.
In diesen Kapitel werden die Grundlagen künstlicher neuronaler Netzwerke kurz wiederholt. Eine Einführung in Convolutional Neural Networks erfolgt ebenfalls, welche dann genutzt wird, um die Änderungen zum Übergang in eine hexagonale Convolution zu geben.

## 3.1 Grundlagen künstlicher neuronaler Netzwerke

Künstliche neuronale Netzwerke (KNN) bestehen aus miteinander verbundenen Knoten, genannt Neuronen, welche über Übertragungsfunktionen verknüpft sind. Sie bauen sich in Schichten auf, die nur eine direkte Verbindung zwischen zwei aufeinander folgenden Schichten erlauben. Im klassischen Netz wird jede Verbindung gewichtet, siehe auch Abbildung 3.1 für eine beispielhafte Visualisierung solcher Netzwerke.
Die Übertragung der Eingaben erfolgt durch summieren dieser gewichteten Werte. Dies führt zu folgender Definition der Übertragungsfunktion $o_j$ einer Schicht $j$ mit Eingaben $x$ und Gewichten $w$, sowie einem Bias $b_i$:

$$o_j(x, w) = \sum_i w_i x_i + b_i. \tag{3.1}$$

Dies verallgemeinert also den Informationsgehalt aus Bereichen des gesamten Eingabespektrums. Die Gewichtung verstärkt oder schwächt dann die Relevanz der jeweiligen Informationen.
Anschließend erfolgt eine Schwellwertabfrage mittels einer Aktivierungsfunktion $\varphi$. Eingehende Informationen, beispielsweise Bildpixel, werden also bei jeder Übertragung zwischen zwei Schichten auf ihre Aussagekraft hin überprüft. Übersteigt dieser Wert den angegeben Grenzwert, so werden die Informationen zur nächsten Schicht weitergeleitet.
In einer Trainingsphase können nun verschiedenen Methoden angewandt werden, um die Aktivierung der einzelnen Neuronen zu verändern. Dazu können neue Neuronen hinzugefügt werden, bestehende gelöscht, Gewichte angepasst oder Schwellwerte verändert werden.
Die Topologie des Netzwerkes kann daher nicht verallgemeinert werden, weswegen das selbe Netzwerk auf verschiedenen Aufgabenbereiche unterschiedlich gute Ergebnisse erzielen kann.

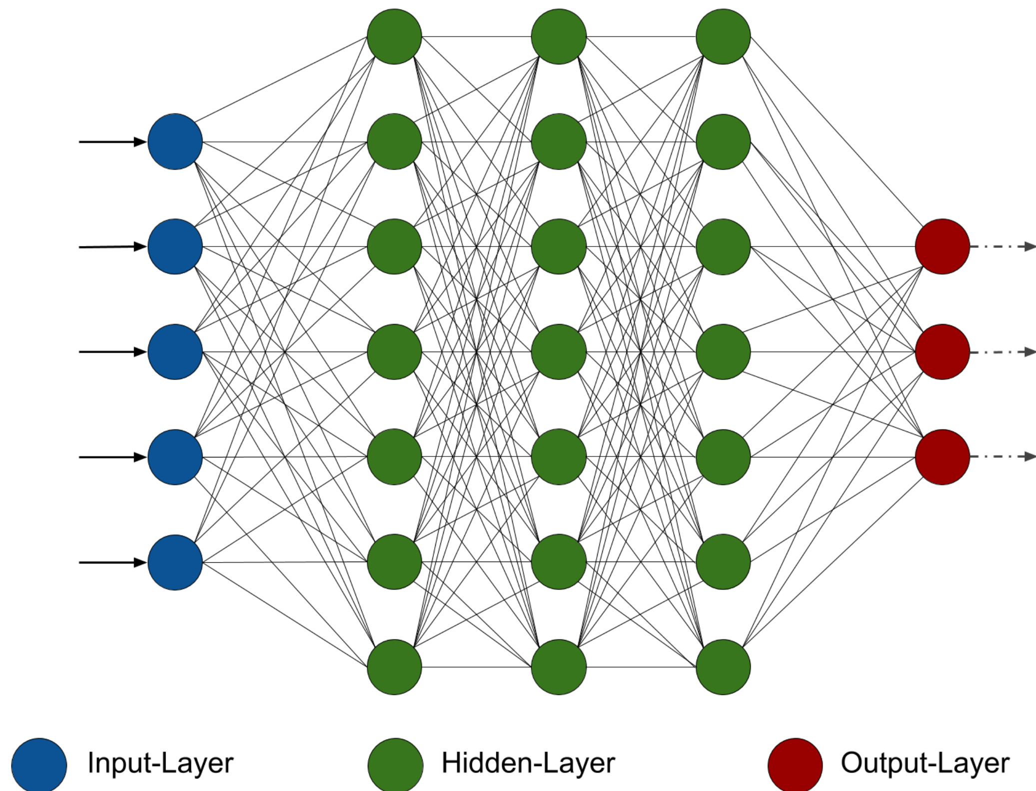



**Abbildung 3.1:** Vereinfachte Darstellung künstlicher neuronaler Netzwerke. Gezeigt sind Input-Layer (blau), Hidden-Layer (grün) und Output-Layer (rot). Kanten zwischen zwei Knoten stellen die gewichtete Übertragungsfunktion dar.

## 3.2 Convolutional Neural Networks

Eine Variante von KNNs findet sich in den Convolutional Neural Networks wieder. Dabei werden Eingabe als Bildpixel interpretiert und nur auf lokalen Regionen gewichtet. Diese Netzwerke finden ihren Einsatz in der Bildverarbeitung und Mustererkennung. Es lässt sich sagen, dass tiefere Schichten sehr spezifische Merkmale beinhalten, während die Ersten eher gröbere Informationen enthalten. Es hat sich gezeigt, dass vortrainierte Netzwerke auf den ersten Schichten eingefroren werden können, dass heißt es findet keine Neugewichtung dieser statt, um sich auf die Bild spezifischen Merkmale zu spezifizieren.
In der Convolutional Schicht (`CONV`) werden lokale Bildausschnitte gewichtet. Dabei bleiben die Gewichte für jeden Ausschnitt gleich, es findet also ein Parameteraustausch statt. Besteht ein Bild nun aus $(32 \times 32 \times 3)$ Pixel, dass heißt es besitzt eine Breite von 32 Pixel und eine Höhe von 32, welche auf 3 Farbkanäle verteilt sind, und sei ein Filter festgelegt mit $(3 \times 3)$, so lässt sich die Ausgabedimension direkt als $(30 \times 30 \times 3)$ bestimmen. Dies ist eine 2D-Convolution, da die Farbkanäle getrennt

voneinander durchlaufen werden. Die Ausgabedimension $\mathbf{O} = (W_O, H_O)$ berechnet sich aus einer Eingabegröße $\mathbf{I} = (W_I, H_I)$ und einem Filter $\mathbf{W} = (W_W, H_W)$ folgendermaßen:

$$\mathbf{O} = \mathbf{I} - \mathbf{W} + 1 \tag{3.2}$$

**Padding.** Mit jeder `CONV` Schicht schrumpft nach diesem Prinzip die Ausgabedimension um die Größe des Filters. Sollen nun sehr tiefe Netze erstellt werden, oder es besteht ein Bedarf von größeren Eingaben in tieferen Schichten, so kann Padding eingesetzt werden. Durch Padding, also einer Vergrößerung der Eingabe, kann dies umgangen werden. Gleichzeitig schützt ein Padding auch vor einem Verwaschen der Randwerte. Dabei können die Padding-Dimensionen unterschiedlich groß ausfallen, im Regelfall werden diese aber gleichgesetzt. In der Praxis sind diese aufgefüllten Werte nicht im Daten-Array abgespeichert, sondern werden frühzeitig erkannt. Ein zweidimensionales Padding $\mathbf{P} = (P_W, P_H)$ führt für Gleichung 3.2 zu folgenden Anpassungen:

$$\mathbf{O} = \mathbf{I} + 2\mathbf{P} - \mathbf{W} + 1 \tag{3.3}$$

**Stride.** Bisher betrachtet Gleichung 3.3 nur eine Schrittgröße von 1, dass heißt jedes Element wird betrachtet. In einigen Fällen ist es vorteilhaft die Ausgabedimension wieder zu reduzieren und gleichzeitig Randwerte mit einzubeziehen. Dafür kann die Schrittgröße angepasst werden, wodurch weniger Elemente abgetastet und Überlappungen reduziert werden. Die Schrittgröße $\mathbf{S} = (S_W, S_H)$ führt daher zu folgenden Änderungen in der Gleichung:

$$\mathbf{O} = \lfloor \frac{\mathbf{I} + 2\mathbf{P} - \mathbf{W}}{\mathbf{S}} \rfloor + 1 \tag{3.4}$$

**Dilation.** Relativ neu ist das Konzept der Dilation [39], also der Ausdehnung des Filters. Diese sollen besonders für Bildsegmentierung bessere Ergebnisse erzielen. Dabei wird ein Ausdehnungsfaktor $\mathbf{D}$ festgelegt, mit welchem die Filterposition verschoben werden. In der Praxis wird daher auch nicht der Filter selbst vergrößert, sondern nur die Positionen angepasst, an welchen dieser zum Einsatz kommt. Durch diese Ausdehnung wird eine höhere Auflösung erreicht, da mehr Elemente vom Filter abgedeckt werden können. Das Aufnahmefeld vergrößert sich dementsprechend mit höheren Ausdehnungsfaktor. Nur der Filter wird von diesen Faktor beeinträchtigt, entsprechend ergeben sich folgende Anpassungen:

$$\mathbf{O} = \lfloor \frac{\mathbf{I} + 2\mathbf{P} - \mathbf{W} - (\mathbf{W} - 1)(\mathbf{D} - 1)}{\mathbf{S}} \rfloor + 1 \tag{3.5}$$

Für einen tieferen Einblick in die Arithmetik der Convolution bietet sich die Arbeit von Vincent Dumoulin und Francesco Visin [13] an. Das verlinkte GitHub-Repository bietet die Möglichkeit verschiedene Konfigurationen der Convolution zu visualisieren. Dieses wurde auch als Grundlage für die in dieser Arbeit gezeigten Darstellungen zu Hilfe genommen. Neben der `CONV` Schicht und der Aktivierungsfunktion können einem Convolutional Neural Network vor allem Pooling-Schichten,

im folgenden mit `POOL` abgekürzt, helfen, um den Hyperparameterraum zu verringern. Auch hier werden lokale Bildregionen erfasst und mittels nicht-lineare Funktionen gesampelt. Meist wird dafür die `max`-Funktion genutzt, da diese in der Praxis gute Ergebnisse erzielt hat. Hauptziel von `POOL` ist die Parameterreduzierung und eine Vorbeugung gegen Overfitting.
Abgeschlossen werden die Netzwerke mit vollständig verknüpften Schichten (`FC`), welche jedem aktivierten Neuron aus der vorherigen Schicht eine Gewicht zuweist. Eine `CONV` Ebene kann als vollständig verknüpfte eingesetzt werden, indem die Filterdimension mit den Eingabedimensionen übereinstimmen, sodass eine einzige Matrixmultiplikation zur Berechnung genügt. Gezielt werden solche Ebenen, um extrahierte Features aus den `CONV` Schichten in Klassen einzubinden. Aber jeder beliebiger Klassifikator, beispielsweise eine SVM, kann auch eingesetzt werden.

## 3.3 Vorhersage und Training

Ein Training der Netzwerke erfolgt durch die Vorhersage der Klassenzugehörigkeit und der Rückführung fehlerhafter Vorhersagen ins Netzwerk durch Anpassung der Gewichte. Zunächst sollte dafür zwischen der Vorhersage, auch Feedforwardpropagation genannt, und der Fehlerrückführung, beziehungsweise Backpropagation, unterschieden werden.
Vorhersagen entstehen im Netzwerk durch die Berechnung von Gleichung 3.18 zusammen mit der Aktivierungsfunktion $\varphi$. Besteht ein Netzwerk aus $n$ Hidden-Layer, einer Eingabe $\mathbf{X}$, den Gewichten $\mathbf{W}_k$ mit $k = 1, ..., n$ aus den Hidden-Layern und dem Gewicht $\mathbf{W}_O$ der Übertragung zum Output-Layer, sowie den gewichteten Eingaben eines Hidden-Layers $\mathbf{H}_k = \varphi_k(o_k(\mathbf{H}_{k-1}, \mathbf{W}_k))$, dann berechnet sich die Vorhersage $\mathbf{P}(\mathbf{X})$ wie folgt:

$$\begin{aligned}\mathbf{P}(\mathbf{X}) &= \varphi_O(o_O(\mathbf{H}_k, \mathbf{W}_O)) \\ &= \varphi_O(o_O(\varphi_k(o_k(\mathbf{H}_{k-1}, \mathbf{W}_k)), \mathbf{W}_O)) \\ &\vdots \\ &= \varphi_O(o_O(\cdots(\varphi_1(o_1(\mathbf{X}, \mathbf{W}_1)), \cdots), \mathbf{W}_O)).\end{aligned}$$

Das Ziel der Backpropagation ist es den allgemeinen Fehler der Vorhersage zu minimieren. Dies erfolgt meist durch ein Gradientenverfahren mit dem Ziel die Kostenfunktion zu minimieren. Für die Fehlerrückführung kann nun eine beliebige Verlustfunktion $l$, häufig auch als Kostenfunktion bezeichnet, beispielshalber MSE oder Kreuzentropie, eingesetzt werden, um den Fehler zu bestimmen. Zur Optimierung dieser Funktion $l(\mathbf{P}(\mathbf{X}))$ wird also dessen Gradient benötigt. Dieser lässt sich mit Hilfe der Kettenregel leicht bestimmen. Bezüglich einer Eingabe $x$ is er definiert als:

$$l'(x) = l'(\varphi_O)\varphi_k'(o_O)o_O'(\varphi_k)\varphi_k'(o_k)\cdots o_1'(x). \tag{3.6}$$

Die Ableitung in einer bestimmten Schicht $i$ hängt daher nur von ihren Nachfolgern ab. Die Ableitung bezüglich einer Schicht $i$ ist also:

$$l'(\varphi_i) = l'(\varphi_O)\varphi_k'(o_O)o_O'(\varphi_k)\varphi_k'(o_k)\cdots o_{i+1}'(\varphi_i), \qquad \forall i < k. \tag{3.7}$$

Aufgrund dieser Struktur lassen sich bereits berechnete Ergebnisse wiederverwerten. Anstatt in jeder Schicht Gleichung 3.7 vollständig neu zu berechnen, können Teile der Berechnung aus den hinteren Layern nach vorne weiter gereicht werden. Der Fehler der äußersten Schicht, im folgenden Output-Layer-Error genannt, sei folgendermaßen definiert:

$$E_O = l'(\varphi_O) \cdot \varphi'_O(o_O(\mathbf{X}_O, \mathbf{W}_O)). \tag{3.8}$$

Analog dazu bestimmt sich der Fehler eines Hidden-Layer $k$, auch als Hidden-Layer-Error bezeichnet, aus dem Hidden-Layer-Error folgender Schichten.

$$E_k = E_{k+1} \cdot \mathbf{W}_{k+1} \cdot \varphi'(o(\mathbf{X}_k, \mathbf{W}_k)), \qquad 1 \leq k < n. \tag{3.9}$$

Im Spezialfall des letzten Hidden-Layers $n$ wird der Output-Layer-Error hinzugezogen:

$$E_n = E_O \cdot \mathbf{W}_O \cdot \varphi'(o(\mathbf{X}_n, \mathbf{W}_n)). \tag{3.10}$$

Für jede Schicht wird also der aktuelle Fehler berechnet und den vorherigen Schichten weitergereicht. Während dieser Schritte müssen ebenfalls die Gewichte aktualisiert werden. Dazu muss der Gradient der Kostenfunktion bezüglich der Gewichte bestimmt werden.

$$l'(\mathbf{W}_O) = E_O \cdot H_n \tag{3.11}$$

$$l'(\mathbf{W}_k) = E_k \cdot H_{k-1}, \qquad 1 < k < n \tag{3.12}$$

$$l'(\mathbf{W}_1) = E_1 \cdot X \tag{3.13}$$

Der Backpropagation-Algorithmus durchläuft also zunächst die Feedforwardprogation, um eine Vorhersage der aktuellen Eingabe durchzuführen. Aus dieser wird mit Hilfe der Kostenfunktion der aktuelle Fehler bestimmt. Dieser wird dem Netzwerk wieder zugeführt, indem Output-Layer-Error und Hidden-Layer-Error schichtweise von der hintersten zur vordersten Schicht durchgereicht werden.
All diese Verfahren lassen sich auf hexagonale Bildstrukturen übertragen. Im folgenden werden die notwendigen Schritte dafür ausgeführt, welche die Arithmetik der hexagonalen Convolution, sowie deren Anwendung beinhaltet.

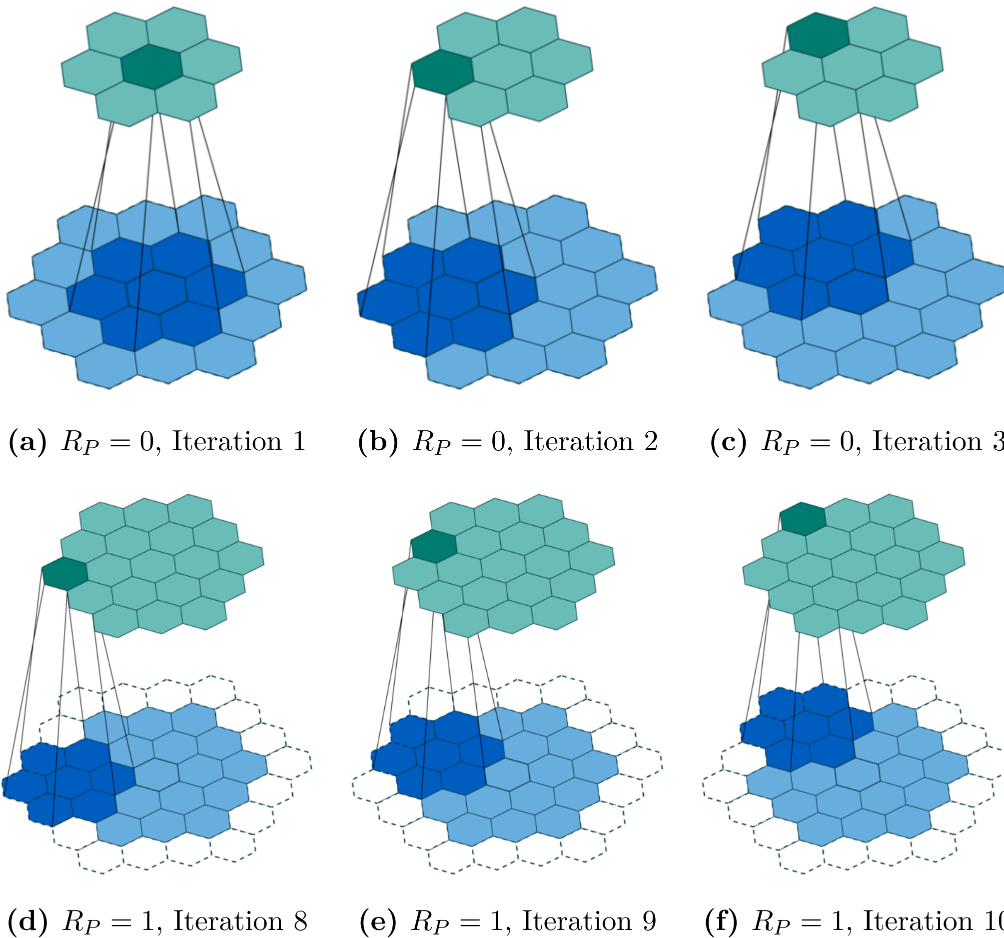
## 3.4 Übergang zur HexConv

Die einzelnen Schritte ändern sich beim Übergang zu einer hexagonalen Convolution wenig, bei welcher reale hexagonale Filter eingesetzt werden, um Bilder, bestehend aus hexagonalen Pixeln, zu behandeln. Diese realen Filtern bestehen aus regelmäßigen Hexagone und werden wie in Kapitel 2 aufgebaut, identisch zu den Eingabebilder. Da solche Hexagone nicht mehr auf einen rechteckigen Raum angepasst werden müssen, beispielsweise indem ein Approximieren mittels Hyperpixel stattfindet, kann die Parameteranzahl im Netzwerk reduziert werden. So müsste ein 7-Element Hexagon nicht mehr in ein 9-Elemente Rechteck eingeführt werden, wodurch gleichzeitig eine Verzerrung der Positionen vorgekommen wäre.
In dieser Arbeit wurden echte Hexagone mit gleichförmigen Filtern auf die bekannte

Convolution übertragen. Durch den Strukturwandel änderten sich einige Eigenschaften wie Padding, Schrittgröße und Filternachbarschaft. Da Hexagone besser über die Anzahl der Ringe beschrieben werden können, müssen die bekannten Gleichung insofern angepasst werden. Für ein hexagonales Bild bestehend aus der Eingabe $\mathbf{I_H} = (R_I)$ und einem Filter $\mathbf{W_H} = (R_W)$ berechnet sich die Ausgabe $\mathbf{O_H} = (R_O)$ wie folgt:

$$\mathbf{O_H} = \mathbf{I_H} - \mathbf{W_H} \tag{3.14}$$

Die Filteringe reduzieren sich hier nicht mehr, da nach Gleichung 2.11 immer mindestens ein Element in Bild vorhanden sein muss. Dies ist eine notwendige Vorkehrung, damit Gleichung 2.12 bestehen bleiben kann.



**(a)** $R_P = 0$, Iteration 1 **(b)** $R_P = 0$, Iteration 2 **(c)** $R_P = 0$, Iteration 3

**(d)** $R_P = 1$, Iteration 8 **(e)** $R_P = 1$, Iteration 9 **(f)** $R_P = 1$, Iteration 10

**Abbildung 3.2:** Visualisierung der Auswirkung des Paddings auf die Ausgaben. Abbildungen (a) - (f) zeigen Teilschritte der hexagonalen Convolution ohne Padding, und (d) - (f) mit Padding.

**Padding.** Die Effekte und Ziele des Paddings bleiben bestehen, nur muss nun die Eingabe mit Elementen umrandet werden. Dadurch entfällt der Unterschied zwischen rechter und linker Seite, sowie Oberer und Unterer, weshalb hexagonales

Padding symmetrisch auf allen Seiten existiert. Da nun neue Ringe hinzugefügt werden müssen, ändert sich das Padding zu $\mathbf{P_H} = (R_P)$.

$$\mathbf{O_H} = \mathbf{I_H} + \mathbf{P_H} - \mathbf{W_H} \tag{3.15}$$

Zur besseren Visualisierung kann die Convolution iterativ betrachtet werden. Dabei wird im Zentrum gestartet, von welcher mittels Spiral-Addition die nächsten Ausgangspunkte berechnet werden. Padding ermöglicht den Erhalt der ursprünglichen Form zu garantieren. Abbildung 3.2a - 3.2c wird auch VALID-Padding genannt. Hierbei werden die Filter nur bis zum äußersten Rand des Bildes angesetzt. Der Filterrand tritt nicht aus dem Bild aus. Abbildung 3.2d - 3.2f stellt dagegen ein sogenanntes SAME-Padding dar, welches immer den äußersten Bildrand mit einbezieht. Die Anzahl der Ringe des Paddings ist dann genau die Ringzahl des Filters.

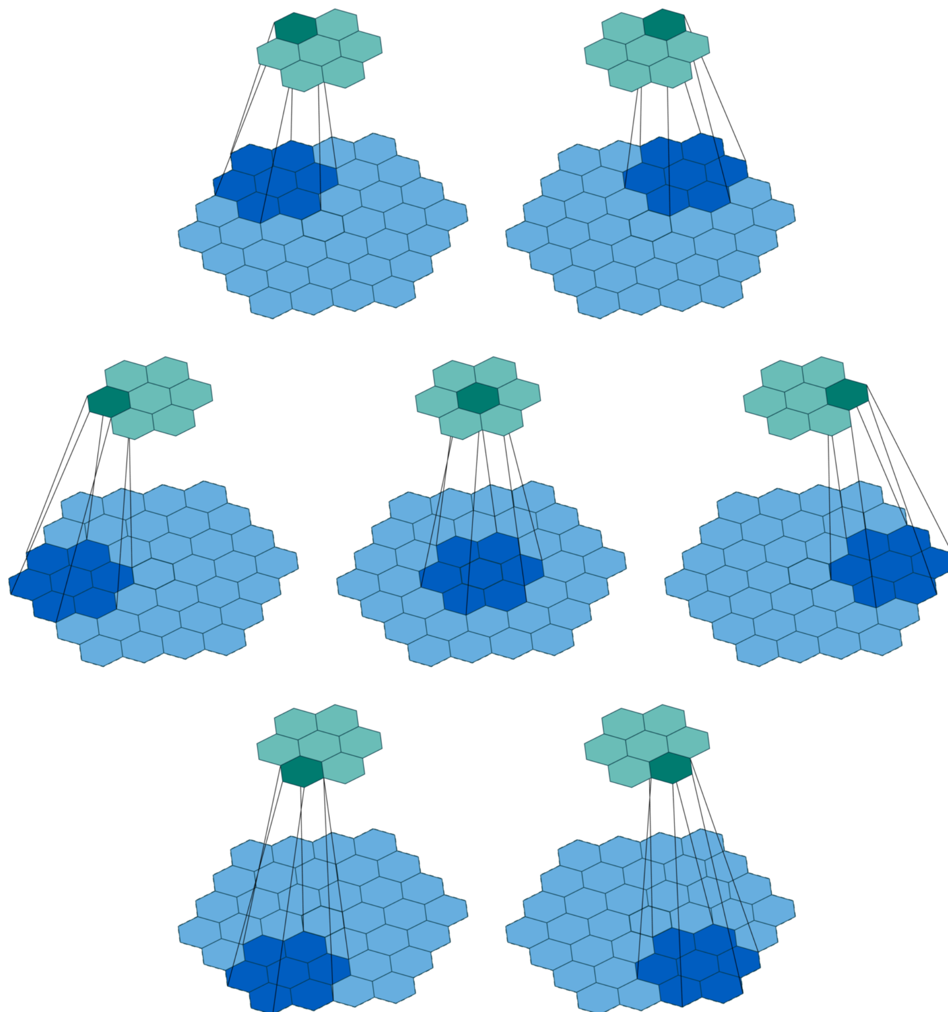
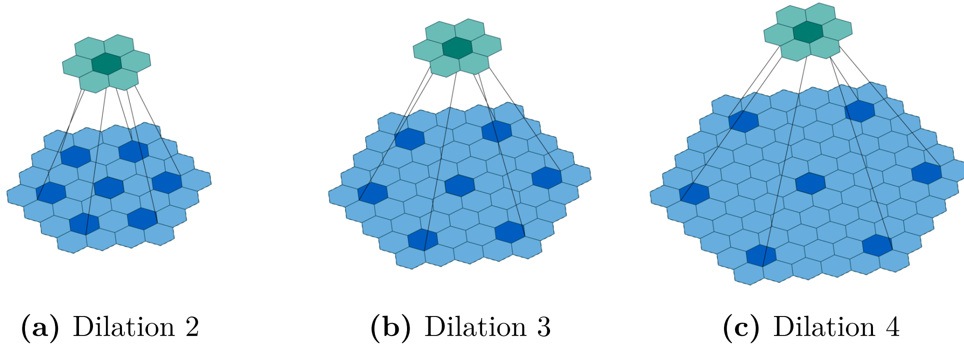


**Abbildung 3.3:** Überspringen von Ringen und Elemente durch höhere Schrittgrößen $R_S = 2$ mit $R_I = 3$, $R_P = 0$ und $R_W = 1$. Dargestellt sind die ersten sieben Schritte der iterativen Convolution.

**Stride.** Da Eingabe und Ausgabe ihre hexagonale Form beibehalten sollen, müssen in jeden Schritt $s$ Ringe übersprungen werden. Gleichzeitig darf nur jedes $s$-te Element im Ring erreicht werden. Da die Anzahl der Ringelemente ein Vielfaches von sechs entsprechen, ergeben nur so die Ausgaben wohlgeformte Hexagone. Abbildung 3.3 verdeutlicht den Effekt einer größeren Schrittweite. Gegebene einer Schrittgröße $\mathbf{S_H} = (R_S)$ ergibt sich folgendes:

$$\mathbf{O_H} = \lfloor \frac{\mathbf{I_H} + \mathbf{P_H} - \mathbf{W_H}}{\mathbf{S_H}} \rfloor \tag{3.16}$$

Die Schrittgröße lässt sich um ein weiteren Parameter erweitern, welcher das Startelement in einem Ring angibt. Dies führt zu keinen Änderungen in Gleichung 3.16, es bietet lediglich eine Form von Variation.



**(a)** Dilation 2 **(b)** Dilation 3 **(c)** Dilation 4

**Abbildung 3.4:** Visualisierung unterschiedlicher Ausdehnungsfaktoren für HexConv

**Dilation.** Analog zur bekannten Convolution verändert der Ausdehnungsfaktor $\mathbf{D_H}$ das Aufnahmefeld, wie in Abbildung 3.4 gezeigt. Folgende Änderung in Bezug auf Gleichung 3.16 findet statt:

$$\mathbf{O_H} = \lfloor \frac{\mathbf{I_H} + \mathbf{P_H} - (\mathbf{W_H} \cdot \mathbf{D_H})}{\mathbf{S_H}} \rfloor \tag{3.17}$$

Die Hinzunahme der Spiralarithmetik führt zu einigen Änderungen in Gleichung 3.1. Sei die Eingabe gegeben durch einen Vektor $\mathbf{X}$ bei welchem die Elemente $X_{c,s}$ die Werte im Farbkanal $c$ an Spiral-Adresse $s$ angeben. Der Filterkernel $\mathbf{W}$ mit den Werte $W_{d,c,s}$ beschreibt die Verbindung zwischen den Gewichten an Spiral-Adresse $s$ im eingehenden Farbkanal $c$ und ausgehenden Kanal $d$. Die Ausgabe $\mathbf{O}$ berechnet sich dann aus den Eingaben $\mathbf{X}$ und $\mathbf{W}$ in allen Farbkanälen $d$ und Positionen $s$ wie folgt:

$$O_{d,s} = o(\mathbf{X}, \mathbf{W})_{d,s} = \sum_{l,k} x_{l,s+_{\circ}k} \cdot w_{d,l,k} + b_{l,k} \tag{3.18}$$

### 3.4.1 Hexagonaler Filter und Nachbarschaften

Der Filter ist das Herz der Convolution. Der Filter, häufig auch als Kernel bezeichnet, gewichtet die Informationen der einzelnen Bildausschnitte. Dies ist wichtig für das

Training, da Neugewichtung dieser die getroffene Entscheidung verändern. Auf den ersten Schichten fokussieren sich diese eher auf gröbere Informationen, wie unter anderem Kanten, während tiefere Schichten stärker spezifiziert sind.
Bei HexConv werden auch hexagonale Filter eingesetzt, dass heißt die Gewichte sind nicht mehr in rechteckiger Form als zweidimensionale Tensor hinterlegt, sondern in einer Hexagonalen, wie auch die Bilder. Hexagonale Filter bieten sich an, da Kurven und Kreise besser aufgenommen werden.

**Algorithmus 3.1 :** SpiralNeighbours

**input** : spiral index $\boldsymbol{a}$, filter $w$
**output** : filter vicinity $\mathsf{N}$ around $\boldsymbol{a}$ with filter $w$

```
N ← {};
W ← HexSpirals(w);        // returns set of spiral addresses around center 0
for each w in W do
    c ← a +∘ w;           // spiral addition will reposition a
    add c to N;
return N;
```

Wie bereits beschrieben schieben sich diese Filter über die einzelnen Bildregionen, was nach Gleichung 3.1 dem Skalarprodukt entspricht. Die die Produkte selbst unabhängig voneinander sind, kann dieser Prozess leicht parallelisiert und optimiert werden. Für jeden Index muss bei jeden Durchlauf die Nachbarschaft bestimmt werden. In der bekannten rechteckigen Convolution können die direkt angrenzenden Indizes über die Breite und Höhe leicht berechnet werden. Die Filter basieren auf der bereits vorgestellten Spiralen Struktur, welche ein eigenes Rechensystem enthält. Dieses kann genutzt werden, um eine Nachbarschaft schnell zu bestimmen. Startet man bei einer beliebigen Adresse **a**, so kann die Nachbarschaft bestehend aus $n$ Elementen erreicht werden, indem wiederholt die ersten Adressen ***1-6*** addiert werden. Wie in Alogrithmus (3.1) beschrieben genügt die Spiral-Addition zur Neupositionierung des Filters um den Punkt $\boldsymbol{a}$ herum. Der Filter $w$ beinhaltet dabei die genaue Spezifikation seiner Struktur, die für `HexSpirals` benötigt wird. Nach Abschnitt 2.3 kann dies beispielsweise die Anzahl der Ringe sein. In der Methode muss dann ein Mapping der Ring-Indexierung zur Spiraldarstellung stattfinden. Ein Rechenbeispiel zu diesem Verfahren befindet sich in Abbildung A.1. Dies zeigt sowohl den genutzten Filter, dessen Nachbarschaft und die berechneten Ausgaben anhand der Filterwerte.

## 3.5 HexPooling

Pooling ist eine spezielle Form der Convolution, bei welcher anstelle der Gewichte eine nicht-lineare Funktion angewendet wird. Im Regelfall ist dies `MAX`, welche nur den maximalen Wert aus der Bildregion übernimmt. Ein Rechenbeispiel ist Abbildung 3.5 zu entnehmen.
Um möglich wenig Informationen zu zerstören, werden kleine Filter bevorzugt. Ein Beispiel für HexPooling wäre ein Filter mit einem Ring $R_F = 1$ und einer Schrittweite von $S_R = 2$. Dabei kommt es zu einigen Überlappungen zwischen mehreren

Bildausschnitten, was normalerweise unerwünscht ist, aber im hexagonalen Fall nicht zu vermeiden ist. Es kann zwar eine größere Schrittweite gewählt werden, aber dies würde Bildelementen zu Folge haben, die nie mit in den Berechnungen berücksichtigt werden würden.

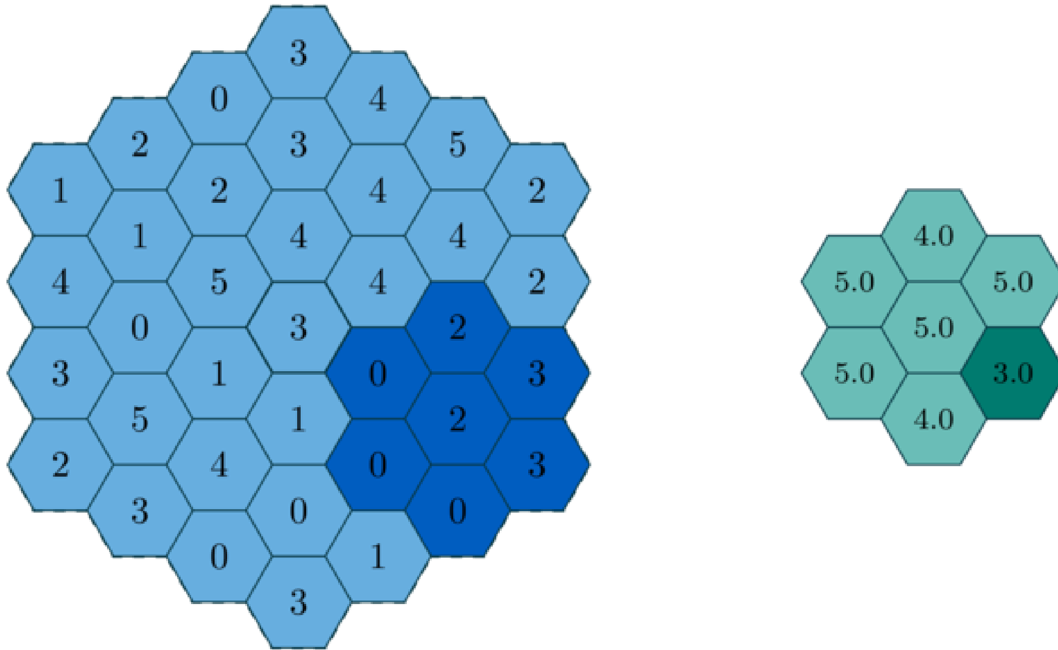



**Abbildung 3.5:** MAX-HexPooling mit $R_I = 3$, $R_P = 0$, $R_S = 2$ und $R_W = 1$ und Beispielwerten. Auf der linke Seite ist die Eingabe (Blau) und rechts die Ausgabe (Cyan) zu erkennen. Gewichte sind den Eingabewerten als Subskript angefügt.

Die Ausgabearithmetik bleibt weiterhin erhalten, da es sich nur um eine spezielle Version der Convolution handelt. Gleichung 3.17 wird weiterhin genutzt, um die Ausgabedimension zu bestimmen.
Das Pooling wird hauptsächlich zum Downsampling, also der Reduzierung von Parametern, genutzt. Dies ist erwünscht, um tiefe Netze zu generieren. Alternativ zum Pooling kann aber auch die normale Convolution eingesetzt werden durch geschickte Spezifikation der Schrittweite und Filtergröße, weswegen die Nutzung und Implementierung von HexPooling nicht im Fokus dieser Arbeit steht.

# 4 Framework-Integration

Die Auswahl eines geeigneten Frameworks ist nicht trivial. Für maschinelle Lernverfahren im Bereich des Deep Learnings bieten unter anderem TensorFlow [2, 1], Caffe [22], PyTorch [28] die Möglichkeit aufgrund ihrer Open-Source-Lizenen eigene Verfahren in ihnen einzubauen und vorhandene Strukturen anzupassen. Im Rahmen dieser Arbeit wurde TensorFlow als Framework ausgewählt. Es bietet einen Zugriff über eine Python-API, welche häufig für maschinelle Lernaufgaben eingesetzt wird. Mit TensorBoard verfügt TensorFlow auch über eine Web-Oberfläche zur Daten-Visualisierung.
TensorFlow ist eine von Google ins Leben gerufenes Framework, was gezielt im Bereich Deep Learning eingsetzt wird. Die Architektur unterstützt eine Vielzahl von Plattformen, darunter CPUs, TPUs und GPUs, sowohl auf Desktop- und Mobilegeräten, als auch Servercluster.
TensorFlow arbeitet im Unterschied zu PyTorch mit statischen Ausführungsgraphen. Dadurch können Modelle nicht während der Laufzeit verändert werden, dass heißt vor Ausführung eines Modells muss der gesamte Ausführungsgraph bereits aufgebaut sein. Daten können zu Beginn als Platzhalter hinzugefügt, während der Laufzeit allerdings nicht angepasst werden.

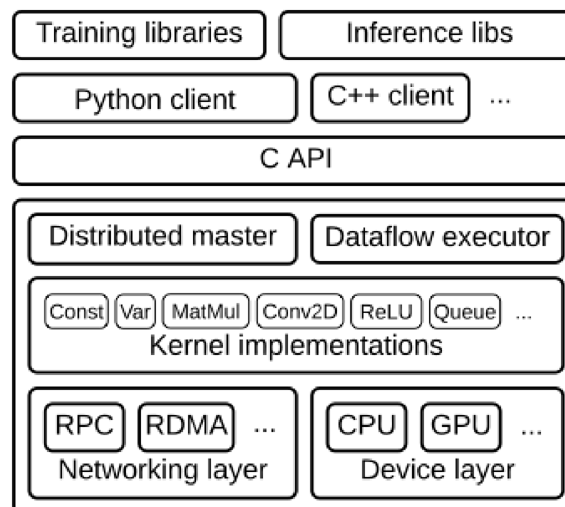



**Abbildung 4.1:** Architektur des TensorFlow-Frameworks. Illustriert sind die einzelnen Schichten der Architektur.

Die generelle Architektur ist in Abbildung 4.1 gezeigt. Zu erkennen ist die Plattformunabhängigkeit TensorFlows, da es unterschiedliche Programmiersprachen über eine C API zur Verfügung stellt.
In diesem Kapitel werden die Konzepte der Kapitel 2 und 3 angewandt. Dem TensorFlow Framework müssen Opertoren zur natürlichen hexagonalen Convolution hinzugefügt werden. Ziel hierbei ist eine effiziente Code-Basis zu erschaffen, welche mit den bereits vorhandenen Convolution-Operatoren mithalten kann. Dafür

müssen sowohl CPU- und GPU-Operatoren erstellt werden. Ein weiterer Teil dieses Kapitels beschäftigt sich mit der Beschleunigung durch GPU Kompatibilität.
TensorFlow unterstützt die Werkzeuge CMake[1] und Bazel[2] zum automatisierten Kompilieren des Source-Codes. Alle in dieser Arbeit getesteten Methoden basieren auf Source-Code Kompilationen mittels Bazel. TensorFlow unterteilt dabei offizielle Features und Erweiterungen in zwei Pakte, *core* und *contrib*. Alle offiziell unterstützten Features gehören zum *core*-Paket, alles weitere sollte *contrib* zugeordnet werden. Der Source-Code zur HexConv Erweiterung ist in einem öffentlichen Repository[3] als TensorFlow-Fork hinterlegt. Die Erweiterung ist im Branch `r1.9hex` vorzufinden.

## 4.1 Tensorflow-Operatoren

Diese Arbeit baut auf der TensorFlow-API in Version `r1.9` auf. Der hier beschriebene Source-Code ist daher nur mit diesem Release kompatibel.

**Spezifikation.** Um einen eigenen TensorFlow-Operator zu schreiben müssen diese in einer C++ Datei spezifiziert sein. Dabei muss Name, Eingabe, Ausgabe und weitere Attribute, sowie Ausgabedimension definiert werden. Über ein Makro `REGISTER_OP` werden Spezifikation und Operatordefinition zusammengebracht. Der HexConv Operator lässt sich folgendermaßen definieren:

```
1  REGISTER_OP("HexConv2D")
2     .Input("input: T")
3     .Input("filter: T")
4     .Output("output: T")
5     .Attr("T: {half, bfloat16, float, double}")
6     .Attr("strides: list(int)")
7     .Attr("dilations: list(int) = [1,1,1,1]")
8     .Attr(GetPaddingAttrString())
9     .Attr(GetConvnetDataFormatAttrString())
10    .SetShapeFn(shape_inference::HexConv2DShape);
```

**Abbildung 4.2:** Mögliche Spezifiktation eines HexConv Operators.

Es ist deutlich zu erkenne, welche Eingaben und Ausgaben dieser Operator nutzt und mit welchen Attributen er versehen werden kann. Es existiert eine Auswahl von möglichen Datentypen, welche von den Werten angenommen werden können. Zugleich erlauben Attribute eine optionale Eingabe, in diesem Fall werden Paddingverfahren, VALID oder SAME, zur Auswahl gestellt. Zugleich kann der Tensorrang in `GetConvnetDataFormatAttrString()` eingestellt werden. Dies ist das Datenformat, welches die Reihenfolge der eingehenden Dimensionen angibt. Die hier relevanten in TensorFlow vordefinierten sind `NHWC` und `NCHW` für Eingaben, sowie `HWIO` für Filter. Die Anzahl der Batches ist `N`, die Höhe des Bildes `H`, die Breite `W` und

[1] `https://cmake.org/`
[2] `https://bazel.build/`
[3] `https://bitbucket.org/maymic/tensorflow`

die Farbkanäle `C`. Es wird zwischen den Formaten *channels-last* und *channels-first* unterschieden, dass heißt ob die Position der Farbkanal Dimension vor oder nach der des Bildes gestellt wird. Dies spielt für GPU-Optimierungen eine Rolle, da die angebundene CUDA-Bibliothek laut der TensorFlow Dokumentation effizienter im *channel-first* Modus rechnet.
In Abschnitt 3.4 wurden eine Gleichung hergeleitet, um Form einer jeden Schicht aus der Ringanzahl ableiten zu können. Die bekannten Daten-Formate können wiederverwendet werden, indem die Größe der Eingabe, nach Berechnung durch Gleichung 2.12, in eine der Dimensionen `H` oder `W` eingebaut wird. Die andere Dimension muss dann nur auf 1 gesetzt werden und kann so weiterhin als Dummy beibehalten werden. Alternativ muss ein neues Datenformat in TensorFlow eingeführt, wodurch einen großer Testaufwand im Bezug auf Kompatibilität mit weiteren Operatoren folgt.
`HexConv2DShape` ist eine simple Funktion, welche aus Eingabegrößen, Filter und vorhandenen Attributen die Ausgabedimension gemäß Gleichung 3.17 berechnet.

**Implementierung.** Die Implementierung eines Operators erfordert eine Vererbung der `OpKernel` Klasse, für welche die `Compute` Funktion implementiert werden muss. Über Klassentemplates können Datentyp und Device festgelegt werden, wie Abbildung 4.3 illustriert.

```
1  template <typename Device, typename T>
2  class HexConv2DOp : public BinaryOp<T> {
3    public:
4      explicit HexConv2DOp(OpKernelConstruction* ctx) : BinaryOp<T>(
   ctx) {
5        ...
6      }
7
8      void Compute(OpKernelContext* ctx) override {
9        ...
10     }
11 }
```

**Abbildung 4.3:** Implementation eines binären Operator mit Klassentemplates für Datentyp und Ausführungsdevice

`BinaryOp` ist eine spezielle Version des `OpKernel`, welche genau zwei Eingaben und eine Ausgabe mit den gleichen Datentype verlangt.

**Registrierung.** Die Ausführung auf CPU oder GPU kann so mithilfe eines Klassentemplates festgelegt werden. Um die zuvor registrierte Spezifikation und die Implementierung zu verbinden benötigt es eines weiteren Makros. Die in TensorFlow erlaubten Datentypen können über ein `TF_CALL_ALL_TYPES` Makro für beliebige Templates dem Compiler bereit gestellt werden. Es gibt mehrere Abstufungen dieses Markos. So sind unter anderem alle GPU-kompatible Datentypen mittels

`TF_CALL_GPU_ALL_TYPES` und nur Numerische mit `TF_CALL_GPU_NUMBER_TYPES` abrufbar. Eine vollständige Sammlung dieser ist in der Klasse `register_types.cc` enthalten.

```
1  #define REGISTER_CPU(T)                                              \
2    REGISTER_KERNEL_BUILDER(Name("HexConv2D")                          \
3      .Device(DEVICE_CPU)                                              \
4      .TypeConstraint<T>("T"), HexConv2DOp<CPUDevice, T>);             \
5
6  TF_CALL_GPU_NUMBER_TYPES(REGISTER_CPU)
7
8  #undef REGISTER_CPU
```

**Abbildung 4.4:** Registierung der HexConv2DOp-Klasse mit Namen HexConv2D und CPU-Unterstützung

Ein weiteres Makro `REGISTER_KERNEL_BUILDER` wir benötigt, welches zur Registrierung des Operators mit den angegeben Namen benötigt wird. Anhand des Namens wird auch die Zugehörigkeit zur Spezifikation hergestellt. Mit Definition des Kernels kann dieser dem Ausführungsgraph hinzugefügt werden. Abbildung 4.4 zeigt die Registrierung des HexConv2D-Operators für CPU-Geräte.
Hier gezeigt ist nur die Registrierung eines `CPUDevice`, welches nur Ausführung auf CPU-Kernen erlaubt. Analog dazu gibt es das `GPUDevice`, welches für die Anwendung auf GPU-Geräten benötigt wird. Diese Arbeit unterstützt dabei nur CUDA-Operatoren, dass heißt es gibt keine Unterstützung für OpenCL[4].
Neben dem HexConv-Operator muss auch HexPooling eingepflegt werden. Dies verläuft analog zu den hier beschriebenen Verfahren und wird deswegen nicht gesondert beschrieben.

**GPU-Unterstützung.** Um einen GPU-kompatiblen Operator in TensorFlow einpflegen, müssen Konvertierungen in CUDA vorgenommen werden. Insgesamt werden drei GPU-Operatoren für HexConv und jeweils zwei weitere für beliebige Pooling-Verfahren benötigt.
Analog zu Abbildung 4.4 müssen Registrierungen für GPU-Operatoren vorgenommen werden, indem das Device auf `DEVICE_GPU` umgestellt wird. Die eigentliche Implementierung erfolgt dann in CUDA-Code. Dabei können Eingaben und Ausgaben direkt im Device-Memory angesprochen werden.
Die ersten Tests mit dieser Implementierung erbrachten nicht die gewünschten Beschleunigungen. Zwar konnten die Operatoren durch GPU-Unterstützung gegenüber der CPU-Variante deutlich beschleunigt werden, sind der TennsorFlow Convolution für rechteckige Bilder aber unterlegen. Dies könnte zum einem an dem fehlende Memory-Management liegen. Die ersten umgesetzten Versionen agieren aus den *global memory* CUDAs heraus und greifen nicht auf den performanteren *shared* oder *texture memory* zu. Die Hinzunahme von diesen und weitere Maßnahmen, wie es

[4]https://www.khronos.org/opencl/

beispielsweise in [29] beschrieben ist, sollten die allgemeine Performance verbessern können.

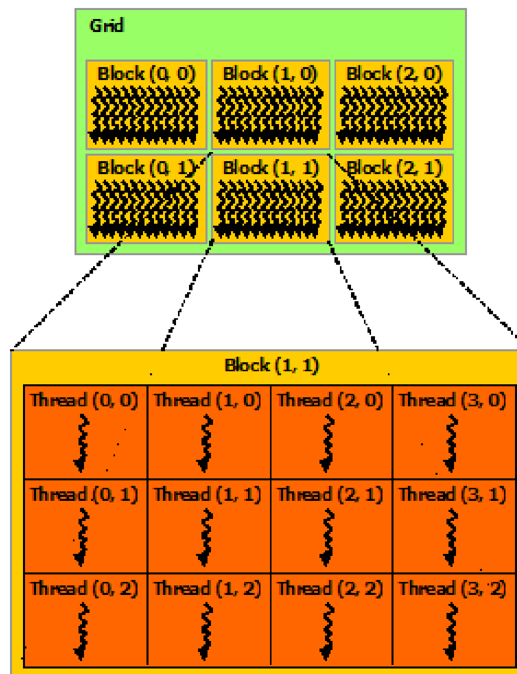



**Abbildung 4.5:** Verknüpfung der *blocks* und *threads* in CUDA. Ein verteiltes Programm wird in parallelisierte Blöcke unterteilt, welche jeweils eine festgelegte Anzahl von Threads ausführen können. Bild dem CUDA *programming guide* entnommen: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/grid-of-thread-blocks.png`.

In CUDA werden parallelisierbare Teilaufgaben in Blöcke (*block*) unterteilt. Diese bestehen wiederum aus einer Menge parallel laufender Threads (*threads*), wie es in Abbildung 4.5 dargestellt wird. Die maximale Anzahl von existierenden Blöcken und Threads wird dabei beschränkt. Die genaue Spezifikation hängt von der genutzten Hardware und CUDA Version ab und kann der offiziellen Dokumentation[5] entnommen werden.

## 4.2 Python-API

Ein großer Teil der Anbindung zu Python wird von TensorFlow automatisch beim Kompilieren ermöglicht. So werden die Interface-Methoden aus den registrierten Operatoren selbständig generiert und betten den Performanten C++ Code in Python ein.
Lediglich die Registrierung der Gradienten muss manuell vorgenommen werden. Wie aus Kapitel 3 bekannt ist, verläuft das Training neuronaler Netzwerke über Backpropagation. Mit Hilfe der transponierten Convolution lässt sich dies mittels

[5] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications`

Convolution-Operatoren realisieren, welche sowohl für Eingaben und Gewichte definiert werden müssen. Im Fall von HexConv, welche zwei Eingaben besitzt, müssen zwei Gradienten `HexConv2DBackpropInput` und `HexConv2DBackpropFilter` erstellt werden. Diese sind die Ableitung bezüglich Eingabe $\mathbf{X}$ und Filter $\mathbf{W}$. Die Registrierung wird mit dem Python Dekorator `@ops.RegisterGradient('OpName')` ermöglicht.

**TensorFlow-API.** Nach erfolgreichen Kompilieren des Projekts und Erstellen eines pip-Paketes, kann dieses in Python importiert werden. Der HexConv-Operator ist über

```
tensorflow.contrib.hexagonal.hexconv2d
```

abrufbar.

**Keras-API.** Keras [9] ist eine modulare, nutzerfreundliche High-Level-API für neuronale Netzwerke die Python zur Verfügung steht. TensorFlow bietet Keras-API Unterstützung über eigene Implementierungen an. Eine HexConv-Schicht kann nach Keras Spezifikation mit

```
tensorflow.contrib.hexagonal.HexConv2D
```

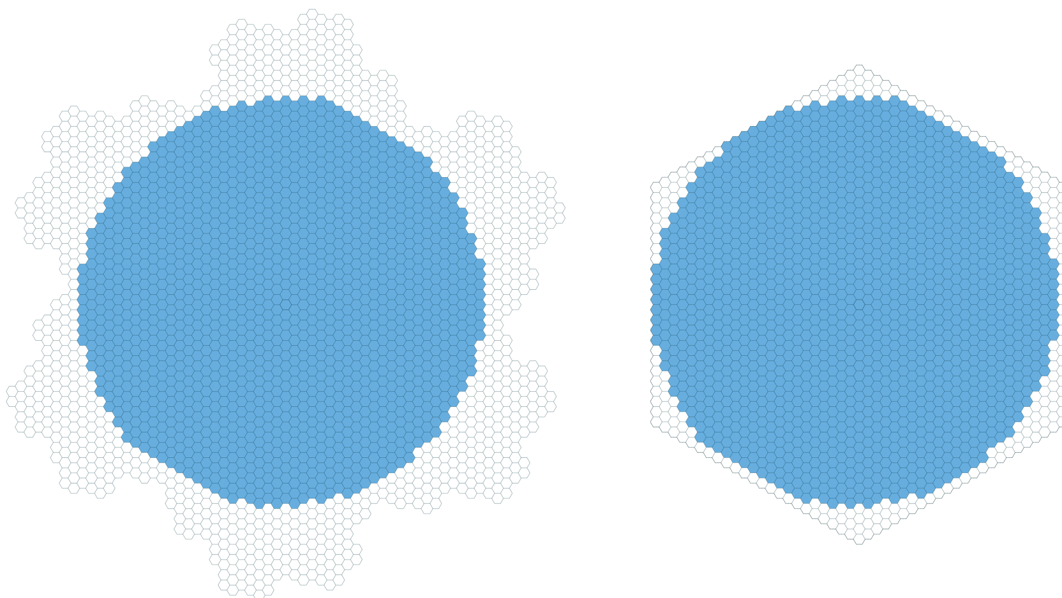
gebaut werden.

# 5 Experimentelle Daten und Netzwerke

Untersucht wurden mehrere Netzwerkstrukturen auf Performanz und Klassifikationsgüte. Ziel dieser Arbeit ist nicht ein ideal trainiertes Netzwerk vorzuzeigen, sondern Effekte der hier vorgestellten HexConv den klassischen Verfahren gegenüberzustellen.
Zu Grunde liegt ein FACT-Datensatz bestehend aus Monte-Carlo-Simulationen, welchem bestimmte Merkmale extrahiert worden sind.

## 5.1 FACT Datensatz

FACT-Aufnahmen setzen sich aus 1440 Pixel zusammen, welche insgesamt eine hexagonale Form einnehmen. Wie bereits erwähnt ist das mehrdimensionale Adressierungssystem sehr dynamisch, weshalb auch für FACT aus zwei, oder drei, gewählten Achsen das komplette Bild indiziert werden kann. Nach Her [19] ist dieses allerdings für nicht symmetrische Rechenaufgaben aufwändig zu berechnen.



**(a)** Overlay eines Level 4 Clusters **(b)** Overlay einer Ring-Indexierung

**Abbildung 5.1:** FACT-Aufnahmen auf Spiraler- und Ring-Adressierung mit einem Overlay der Gesamtgröße des jeweiligen Verfahrens.

Die Spirale Architektur ist durch die vorhandene Arithmetik gut berechenbar und effizienter in der Indizierung, da jeder Pixel durch einen einzelnen Index repräsentiert wird. Jedoch benötigt die Spirale Architektur Cluster einer festen Größen, abhängig vom Cluster-Level. Aufgrund dessen muss ein Level 4 Cluster, bestehend aus 2401 Punkten, über das Bild gelegt werden, wie in Abbildung 5.1a illustriert ist. Somit

existieren mindestens 961 unbesetzte Pixel im Bild, welche den Speicheraufwand erhöhen und zusätzlich während der Convolution berechnet werden müssen.
Mit einer Ring-Indexierung kann dies eingegrenzt werden, wie in Abbildung 5.1b zu sehen ist. FACT-Aufnahmen können in einer Ring-Indexierung mit 23 Ringen und insgesamt 1657 Pixel eingefangen werden. Im Vergleich zur direkten Spiral-Adressierung sind 744 Pixel weniger außerhalb der Aufnahme enthalten.
Bei näherer Betrachtung der einzelnen Aufnahmen ist zu erkenne, dass die Gamma-Klassen stärker zentriert und markanter erkennbar sind als Ellipsen. Hadron-Klassen fallen dagegen etwas breiter gefächert aus.

### 5.1.1 Datenextraktion mit `streams`

Die FACT-Aufnahmen, wie sie für dieser Arbeit vorliegen, können nicht direkt in CNNs eingebunden werden. Sie bestehen aus 300 Einzelaufnahmen, bei dem das Teleskop die elektrische Ladungen an den 1440 Pixeln aufzeichnet. Aufzeichnungen entstehen nur nach einen festgesetzten Auslöser, welcher sowohl für die positive und negative Klasse triggert. Für ein einzelnes Event werden also 432000 Einzelwerte aufgezeichnet.
Für diese Arbeit wurde ein einzelnes Merkmal extrahiert und für weitere Analysen aufbereitet. Dieses wird im weiteren als Lichtintensität bezeichnet und beschreibt die Intensität der Tscherenkow-Strahlung. Dabei können die Messwerte aller Events stark schwanken. Dieser liegt in etwa zwischen $-2$ und 1400. Deswegen sollten sie vor einer Anwendung normalisiert werden. Da sich CNNs mit Bilder befassen, bietet sich ein neuer Wertebereich von $0 - 255$ an.
In Abbildung 5.2 ist die XML-Struktur der Extraktion zu sehen. Die Rohdaten liegen dabei auf einem Hadoop [14] Cluster vor. Mit Hilfe des `streams`-Frameworks [7] und der `fact-tools`[1] wurden dann die benötigten Daten extrahiert. Um die Extraktion zu beschleunigen wurde Sparks [41] genutzt. Dies war mit einer `streams`-Erweiterung namens `streams-spark`[2] möglich. Die erstellte XML-Datei kann dann zusammen mit einer Sparks-Binary dem YARN [36] Ressourcen-Manager zugeschickt werden. Die `streams`-XML wird daraufhin als verteile Sparks Applikation auf den Hadoop Cluster ausgeführt.
Neben der Lichtintensität existieren weitere Merkmale, welche jedoch nicht für die hier durchgeführten Untersuchgen genutzt werden. Die bekannten Verfahren zur Klassifizierung der FACT-Aufnahmen nehmen unter anderem Größe der Ellipse und Eintrittswinkel mit auf. Ein jedoch benötigtes Attribut ist die Klasse selbst, da hier nur CNNs mit überwachten Lernen trainiert werden. Untersucht wird daher nur die Klassifizierung von FACT-Aufnahmen. Die Klassenzugehörigkeit wurde hier anhand der Quelldatei festgelegt. `AddBinaryDecisionKey` setzt die Klasse auf 1 für alle Events deren Quelldatei auf einem Pfad liegen, der den regulären Ausdruck *.*gamma.** positiv abgleicht. Alle weiteren Events werden der Klasse 0 zugeordnet.
Die extrahierten Daten werden dann als CSV-Dateien über die `PhotonchargeRDDSink` im Hadoop Cluster hinterlegt. Dabei werden nur der Pfad zur Quelldatei, die Messwerte selbst und die Klasse abgespeichert.

[1] `https://github.com/fact-project/fact-tools`
[2] `https://bitbucket.org/mbunse/streams-spark`

```
<container>
  <properties url="classpath:/default/settings_mc.properties"/>
  <property name="drsfile" value="hdfs://path/to/stdMcDrsFile.drs.fits.
    gz"/>
  <property name="integralGainFile" value="classpath:/default/
    defaultIntegralGains.csv"/>
  <service id="calibService" class="fact.calibrationservice.
    ConstantCalibService"/>

  <stream id="in" class="stream.io.multi.FitsStreamGenerator" url="hdfs
    ://path/to/source/files"/>

  <sink id="out" class="stream.io.PhotonchargeRDDSink" url="hdfs://path
    /to/export"/>

  <dProcess id="dp" input="in" output="out">
    <stream.flow.Skip condition="%{data.TriggerType} != 4"/>
    <!-- Calibration: -->
    <include url="classpath:/default/mc/calibration_mc.xml"/>
    <!-- Extraction -->
    <include url="classpath:/default/mc/extraction_mc.xml"/>

    <stream.processors.AddBinaryDecisionKey newKey="IsGamma" whenKey="
   @source" matchesRegExp=".*gamma.*"/>
    <!-- Only send actual results (i.e., features) over network -->
    <stream.io.FilterKeys keys="@source,photoncharge,IsGamma"/>
  </dProcess>
</container>
```

**Abbildung 5.2:** `streams`-XML zur Extraktion der Lichtintensität. Anwendung erfolgte mit einer `streams` Erweiterung mit Sparks Unterstützung, die von der Projektgruppe 594 der TU Dortmund entwickelt wurde. Ebenfalls wird `fact-tools` zur Extraktion der FACT-Daten genutzt.

## 5.2 Konvertierung in das TFRecords-Format

Nativ bietet TensorFlow Import von Textdaten, CSV-Formaten und TFRecords an. In der Praxis erwies sich das TFRecord-Format performanter als CSV-Imports, aufgrund der binären Struktur. Die eigentlichen Rohdaten aus FACT liegen zunächst im Flexible Image Transport System (**FITS**) vor, welches ein strukturiertes Datenformat für astronomische Bildaufnahmen ist. Im Unterschied zu JPEG oder GIF bietet FITS eine leserliche Header-Tabelle mit Metadaten an und strukturiert Daten in tabellarisch. Eingelesen kann dieses in verschiedenen Programmierspachen über vorhandene Bibliotheken. Es wäre möglich eine eigene Daten-Import Funktion in TensorFlow einzubinden, welches FITS-Dateien direkt lesen könnte, doch dieser Mehraufwand, welcher Performance und Stabilität beeinflusst, erscheint nicht realistisch.
Im Kontext dieser Arbeit wurden mehrere Datensets mit mehreren Millionen Datenpunkten zunächst in CSV-Dateien exportiert und anschließend mittels TensorFlow

ins TFRecord-Format umgewandelt. Bei der Extraktion wurden nur die Messwerte und Klasse der Events übernommen. Da in dieser Arbeit rechteckige Convolution den hier vorgestellten hexagonalen Verfahren gegenübergestellt werden, bot sich bei der Umwandlung direkt eine Speicherung beider Darstellungen an. Die benötigen Informationen beim Import der Dateien lassen sich dazu selektieren und filtern. Eine generierte TFRecord-Datei enthält folgende Informationen und lässt sich über die TensorFlow Python-API wie folgt einlesen:

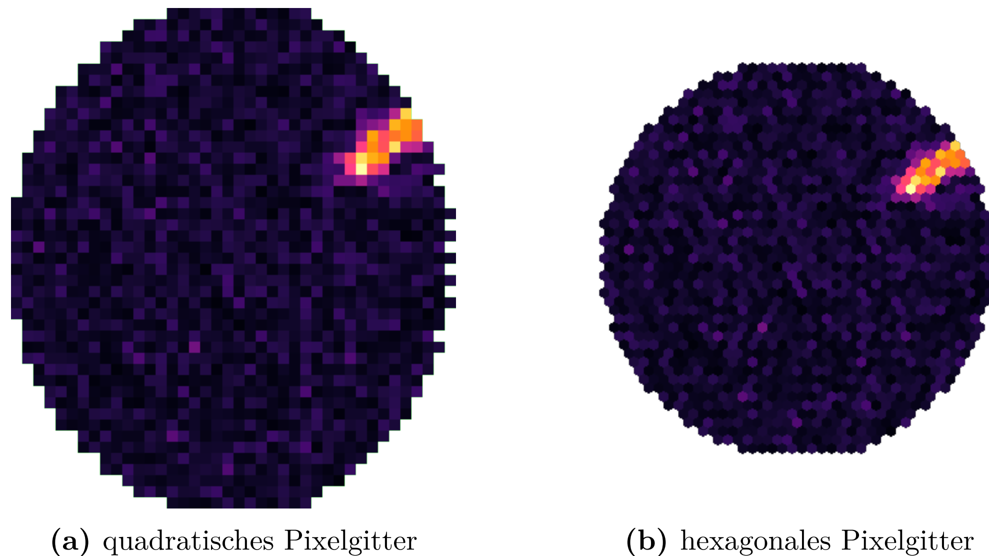
```
import tensorflow as tf
features = {
  'hex/features': tf.FixedLenSequenceFeature([], dtype=tf.float32,
 allow_missing=True),
  'fact/features': tf.FixedLenSequenceFeature([], dtype=tf.float32,
 allow_missing=True),
  'rec/features': tf.FixedLenSequenceFeature([], dtype=tf.float32,
 allow_missing=True),
  'hex/elements': tf.FixedLenFeature([], dtype=tf.int64),
  'fact/elements': tf.FixedLenFeature([], dtype=tf.int64),
  'rec/height': tf.FixedLenFeature([], dtype=tf.int64),
  'rec/width': tf.FixedLenFeature([], dtype=tf.int64),
  'rec/depth': tf.FixedLenFeature([], dtype=tf.int64),
  'label': tf.FixedLenFeature([], dtype=tf.int64)
}
```

**Abbildung 5.3:** Inhalte eines exportierten FACT-Exports im TFRecord-Format. Gezeigt wird die exportierten Elemente und deren Datentyp.

Die Originalform ist unter *'fact/features'* hinterlegt und beinhalten genau 1440 Datenpunkte. Die Werte zur Verwendung in hexagonalen CNNs sind unter *'hex/features'* zu finden und dessen Länge in *'hex/elements'*. Analog dazu wurden die Aufnahmen in ein rechteckiges Format gezwängt, welches unter *'rec/features'* aufrufbar. Abbildung 5.4 zeigt den Unterschied zwischen beiden Formaten. Die Verzerrung des Bildes ist dabei deutlich im rechteckigen Format zu erkennen.
TFRecord bietet sich zudem an, da neue Information den bereits bestehenden einfach hinzugefügt werden können. Soll beispielshalber ein weiteres Merkmal, wie der Energie, hinzugefügt werden, müssen die Dateien nur eingelesen und um den neuen Punkt erweitert werden. Die *label-value* Schreibweise von TFRecord erleichtert dieses Vorgehen.

## 5.3 Sampling

Der für diese Arbeite vorliegende Datensatz umfasst mehr als 2.4 Millionen Events. Insgesamt liegen knapp 1.9 Millionen Gamma und 500 Tausend Hadron Events vor. Es stellt sich die Frage, ob unausgeglichene Trainingsdatensätze zu Problemen im Training der Netzwerke führt. Obwohl dieses Verhältnis nicht als drastisch anzusehen ist, denn es existieren Datensätze mit schlechteren Verhältnissen von 1 : 100 und mehr [37], sollten dennoch die Auswirkungen auf diesem Datensatz untersucht werden.



**(a)** quadratisches Pixelgitter **(b)** hexagonales Pixelgitter

**Abbildung 5.4:** Visualisierung einer FACT-Aufnahme durch rechteckige und hexagonale Pixel. Die hexagonalen Pixel wurden hierbei durch eine Vielzahl Rechteckiger approximiert. Diese werden auch Hyperpixel genannt.

Ein ausgewogenes Trainingsset ist von Relevanz, da Bewertungsfunktionen wie Accuracy, Precision und Recall stark vom Klassengleichgewicht abhängen. Dies kann zu Fällen führen, in welchem nur eine einzelne Klasse trainiert wird, aber trotzdem eine sehr hohe Accuracy vorherrscht, wodurch ein gutes Ergebnis vorgetäuscht wird. Lösungsansätze können algorithmischen und datenbasierten Verfahren zugeordnet werden. In dieser Arbeit werden nur die Datenbasierten näher untersucht. Diese umfassen ein Resampling, um ein ausgeglichenes Verhältnis wieder herzustellen. Ein naiver Ansatz wäre beispielsweise ein randomisiertes Downsampling der vorherrschenden Klasse. Dabei wird ein Teil der Trainingsdaten gefiltert. Alternativ kann auch Oversampling genutzt werden, um die sekundäre Klasse zu vergrößern. Dies verlangt allerdings das Generieren neuer Datenpunkte, basierend auf den bereits Vorhandenen.
Anstelle von randomisierten Verfahren können Kostenfunkionen und Gewichte helfen ein besseres Verhältnis herzustellen. Dabei müssen zusätzliche Informationen zu den Daten gesammelt werden, welche die Relevanz dieser festlegen. Je relevanter ein Punkt dann ist, desto wahrscheinlicher kann er dem Datensatz zugehörig bleiben (Downsampling), oder beispielsweise dupliziert werden (Oversampling). Eine Kombination aus beiden Verfahren ist weiterhin auch möglich.
In dieser Arbeit werden zwei Datensätze aus den verfügbaren Daten generiert. Um die Auswirkung der Verhältnisse näher zu untersuchen werden diese auf jeweils 1 Millionen Punkte limitiert. Es wird hierbei lediglich ein randomisiertes Downsampling der vorherrschenden Klasse genutzt. Die Datensätze sind in Tabelle 5.1 aufgelistet.

| Datasets | Klassen | Größe | Verteilung |
|---|---|---|---|
| *fact-full* | 2 | 2.4M | $\sim 80/20$ |
| *fact-50* | 2 | 1.0M | $\sim 50/50$ |
| *fact-70* | 2 | 1.0M | $\sim 70/30$ |

**Tabelle 5.1:** Unterschiedlich gesampelte FACT-Datensätze mit ihrer Größe und Verteilung.

## 5.4 Testnetzwerk

Die Auswahl einer geeigneten Netzwerkstruktur ist keinesfalls trivial. Die korrekte Einstellung von Hyperparametern wie Anzahl der Schichten und Filter beeinflussen das Lernverhalten der Netzwerke. Da beide Verfahren, rechteckige Convolution, fortan zu Conv abgekürzt, und HexConv einer direkten Gegenüberstellung unterstellt werden, ist eine identische Netzwerk-Topologie notwendig.

**Simpel-Sequentiell (SimpSeq).** Ein erster Test erfolgt mit einem recht simplen Netzwerk, welches aus Convolution, Batch-Normalisierung [21], ReLU-Aktivierung [26] und einer Fully-Connected Schicht besteht. Die Filtergrößen sind in allen Schichten $\mathbf{W} = (3, 3)$ und $\mathbf{W_H} = (7)$ für jeweils Conv und HexConv. Die Struktur ist von ResNet [17, 40, 16] inspiriert, jedoch stark simplifiziert und kann in Bausteine und Etappen unterteilt werden. Es existieren zwei Bausteintypen, welche Convolution, Normalisierung und Aktivierung beinhalten, wie in Abbildung 5.5 illustriert ist.

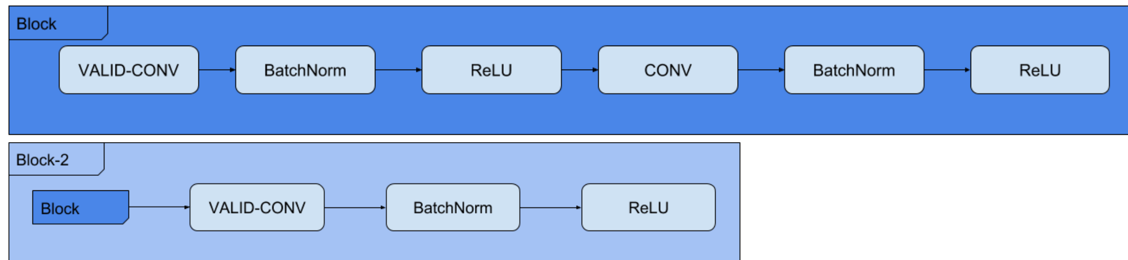



**Abbildung 5.5:** Die zwei Grundbausteine des SimpSeq Modells. Der obere Baustein enthält eine einzelne Downsampling-Schicht (VALID-CONV) und der Untere wird um eine zusätzliche erweitert.

Eine Etappe besteht aus jeweils drei solcher Bausteine.
Insgesamt setzt sich SimpSeq aus einem eingehen Baustein gefolgt von drei Etappen und einer abschließenden Fully-Connected Schicht zusammen. In Abbildung 5.6 ist dies noch einmal visualisiert worden.
Die Anzahl der Filter jeder Schicht vergrößern sich mit der Tiefe des Netzwerks. Im ersten Block werden die eingehenden FACT-Aufnahmen, welche einen einzigen Farbkanal besitzen, auf vier Kanäle erweitert. Während der ersten Etappe bleibt diese Anzahl bestehen, vergrößert sich dann auf acht im Wechsel zur Nächsten. Diese vergrößert sich dann erst wieder beim Erreichen von Etappe drei auf insgesamt zehn Filter. Für die Klassifizierung werden diese auf der Fully-Connected Ebene wieder auf zwei reduziert. Dies entspricht einer Kurzschreibweise von 4-4-8-10.

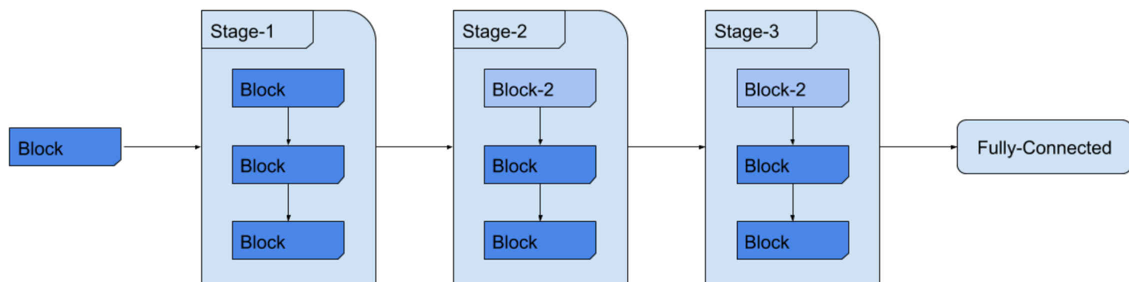



**Abbildung 5.6:** Gesamte Netzwerkstruktur des SimpSeq Modells.

# 6 FACT: HexConv und Conv im direkten Vergleich

Der in Kapitel 2 und Kapitel 3 beschriebene HexConv Algorithmus wird im folgenden mit einer konventionellen Convolution verglichen. Dazu werden mehrere SimpSeq Modelle erstellt und ausgewertet. Die Auswertung umfasst die Klassifikationsgüten Accuarcy, Precision, Recall und *area under the receiver operating characteristic curve* (AUC). Beide Modelle werden zunächst über 150.000 Iterationen hinweg trainiert. Das Training erfolgt auf einem separaten Datensatz zur Evaluierung. Insgesamt stehen zwei Drittel der gesamten Daten dem Training zur Verfügung und ein Drittel der Evaluation.

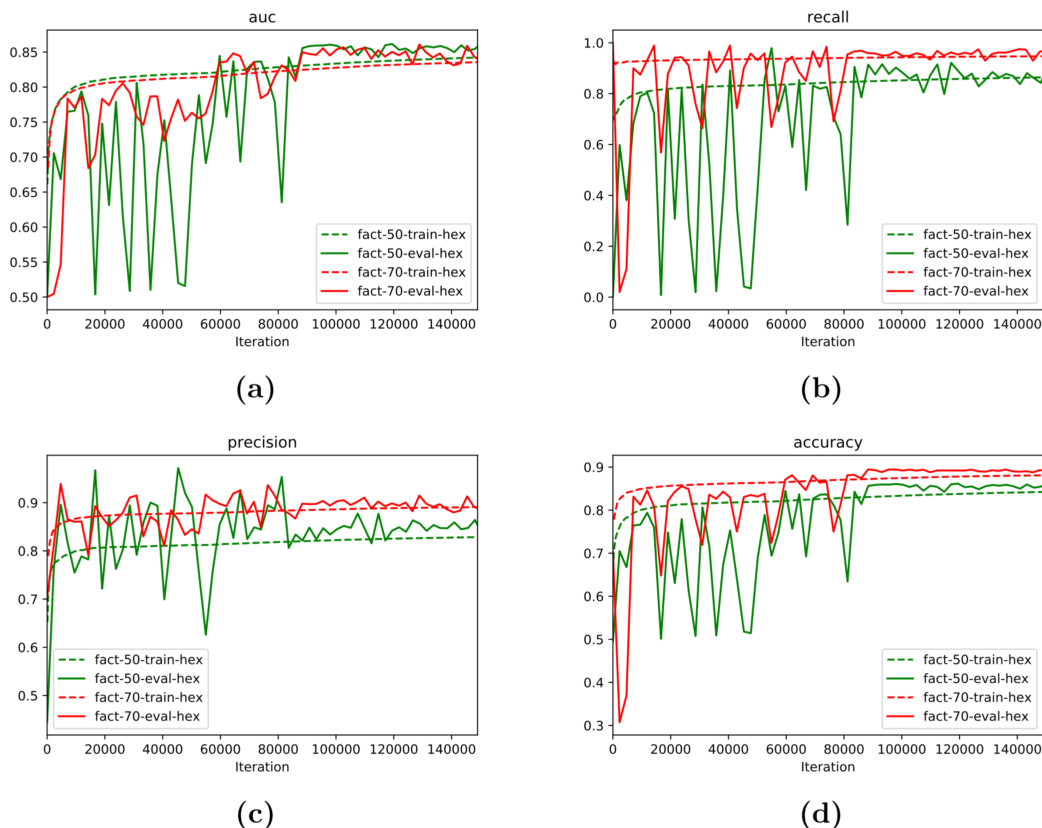



**Abbildung 6.1:** Graphiken zu AUC (6.1a), Recall (6.1b), Precision (6.1c) und Accuracy (6.1d) aus Training und Evaluierung des SimpSeq-Modells mit HexConv. Aufgetragen ist die erreichte Güte auf der y-Achse in Abständen von 100 Iterationen (x-Achse) für die Trainingsphase. Güten der Evaluationsphase werden in Abständen von zehn Minuten, die das Netzwerk trainiert, aufgezeichnet. Die Datensätze *fact-70* (rot) und *fact-50* (grün) sind farblich hervorgehoben. Die Trainingskurve gibt den allgemeinen Trend des gesamten Trainings wieder, während für Evaluierung Momentaufnahmen aus der jeweiligen Iteration mit dem trainierten Modell angezeigt werden. Die Legende setzt sich aus genutzten Datensatz, Training- oder Evaluationsphase und Art der Convolution zusammen; *dataset*-type-*conv*.

Während des Trainings wird pro Iteration ein Sample des Trainingsdatensatzes, bestehend aus insgesamt 90.000 Events, zufällig gezogen. Alle zehn Minuten wird das Training pausiert und eine Evaluierung des aktuellen Modells gestartet. Diese zieht ebenfalls ein Sample aus dem vom Training unabhängigen Evaluierungsdatensatzes, bestehend aus jetzt 30.000 zufälligen Events.

**HexConv Training auf balancierten und schiefen Klassenverhältnis.** Ein Vergleich der Precision, Accuracy und Recall zeigt, dass HexConv für den *fact-50* länger benötigt, um eine Konvergenz zu erreichen (vgl. Abbildung 6.1). Der Datensatz *fact-70* erzielte nach den vorgegebenen Iterationen leicht höhere Werte für Precision, Recall und Accuracy, sowohl für Training als auch für Evaluierung. Dies kann allerdings dem nun schiefen Klassenverhältnis zugeschrieben werden, da AUC beider Datensätze nahezu identisch verlaufen.
Des Weiteren ist zu erkennen, dass nach knapp 90.000 Iterationen sich die Evaluationsergebnisse stabilisieren. Ab diesem Zeitpunk haben die Netzwerkparameter eine Konvergenz erreicht, die durch weiteren Iterationen nur noch minimal verbessert wird.

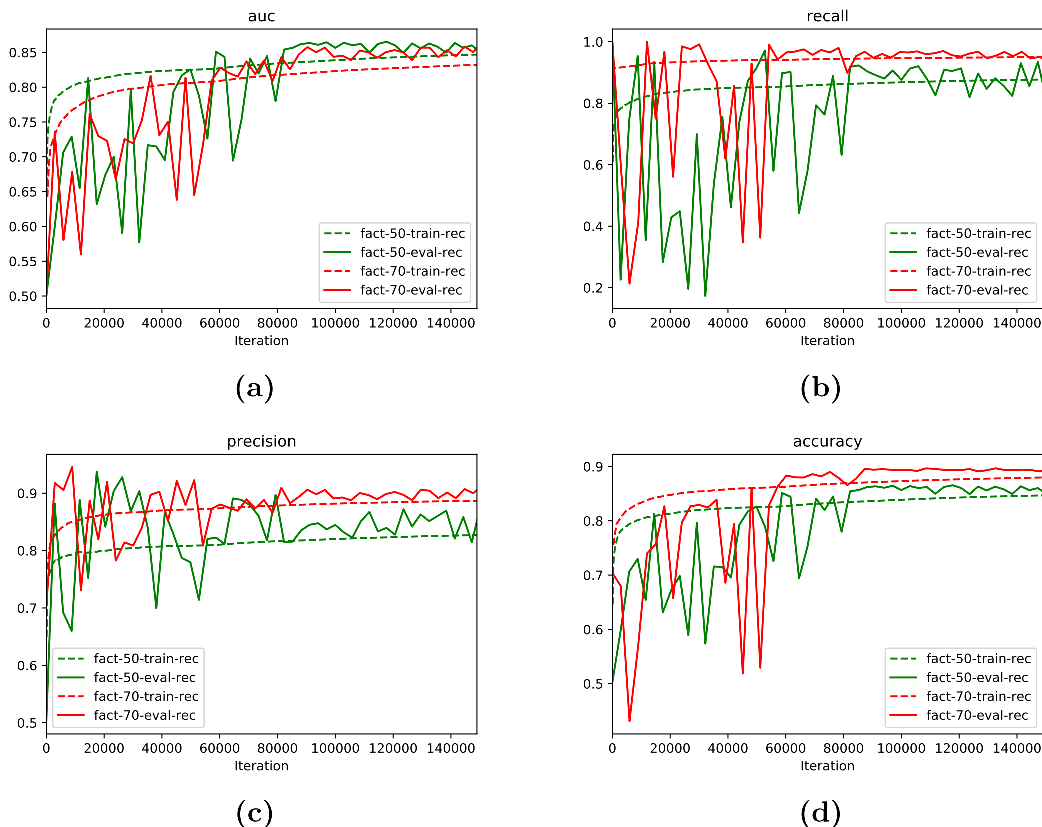



**Abbildung 6.2:** Graphiken zu AUC (6.2a), Recall (6.2b), Precision (6.2c) und Accuracy (6.2d) aus Training und Evaluierung des SimpSeq-Modells mit klassischer Convolution.

**Conv Training auf balancierten und schiefen Klassenverhältnis.** Das trainierte SimpSeq Modell mit klassischer Convolution zeigt wieder eine leichte Verbesserung für Precision, Recall und Accuarcy auf (vgl. Abbildung 6.2). Alle Güten erreichten nach etwas mehr als 90.000 Iterationen wieder einen stabile Phase. Auch hier ist eine Konvergenz der einzelnen Güten bereits zu erkennen. Ebenfalls verläuft die AUC beider Datensätze nahezu identisch.

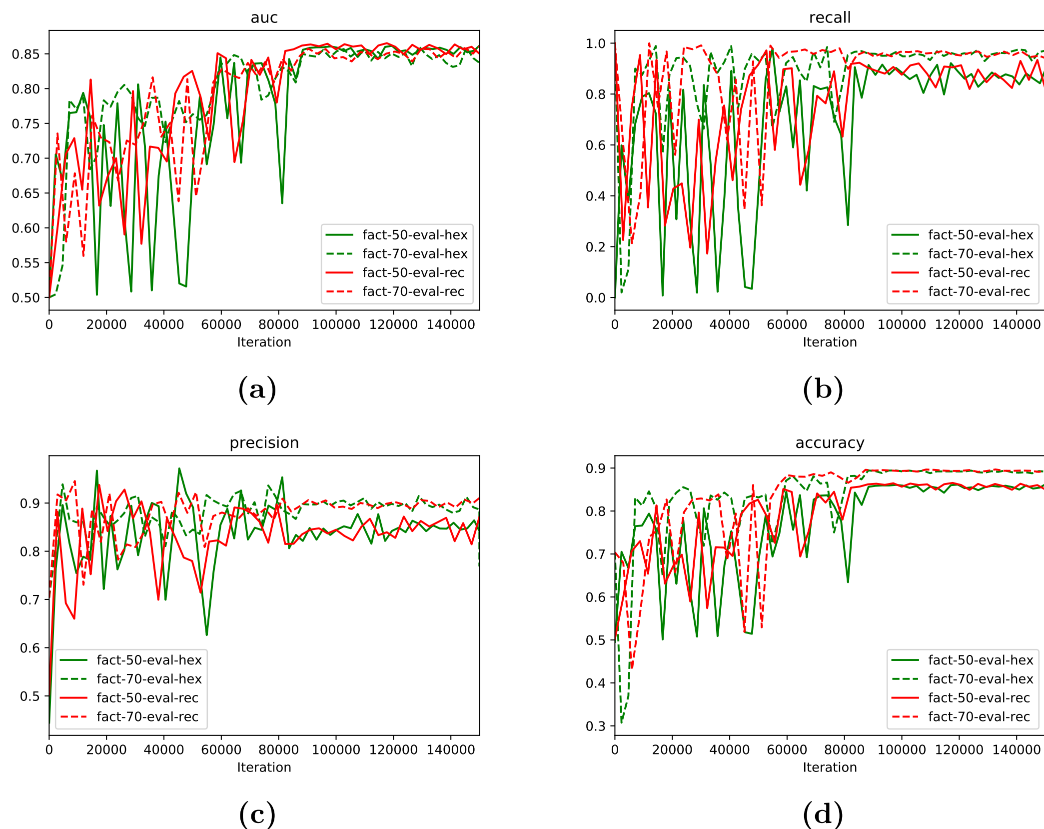



**Abbildung 6.3:** Überlagerung der Güten AUC (6.3a), Recall (6.3b), Precision (6.3c) und Accuracy (6.3d) beider Convolutionen Verfahren auf 150.000 Iterationen. HexConv (grün) und Conv (rot) werden farblich voneinander getrennt. Die Legende unterteilt sich in den genutzten Datensatz, das Evaluationsdaten genutzt werden und die Art der Convolution; *dataset*-type-*conv*.

**Vergleich beider Verfahren.** Vergleicht man beide Ansätze direkt miteinander, dass heißt werden die Evaluationskurven beider Variante überlagert, so erkannt man ein nahezu identischen Verlauf, siehe dazu Abbildung 6.3. Demnach kann kein relevanter Unterschied zwischen beiden Ansätzen festgestellt werden.
Dies kann mehreren Ursachen zu Grunde liegen. Zum einen könnten die Datensätze selbst nicht ausreichend gute Informationen enthalten, um eine bessere Trennung beider Klassen zu erlangen. Der erwartete Effekt, dass hexagonale Convolution besser Kurven aufgrund der höheren Radialsymmetrie identifizieren kann und so die elliptischen Formen beider Klassen besser differenziert, ist hier nicht erkennbar.

| Modell | AUC | Precision | Recall | Accuracy |
|---|---|---|---|---|
| SimpSeq-HexConv-50 | **0.86** ±0.001 | 0.85 ±0.003 | 0.87 ±0.007 | 0.85 ±0.001 |
| SimpSeq-HexConv-70 | 0.84 ±0.002 | 0.89 ±0.006 | **0.96** ±0.003 | **0.89** ±0.003 |
| SimpSeq-Conv-50 | **0.86** ±0.001 | 0.85 ±0.004 | 0.88 ±0.008 | 0.86 ±0.001 |
| SimpSeq-Conv-70 | 0.85 ±0.002 | **0.90** ±0.002 | 0.95 ±0.003 | **0.89** ±0.001 |

**Tabelle 6.1:** Gemittelte Messwerte mit Fehler der letzten 20 Evaluationsergebnissen aus 150.000 Trainingsschritten.

Die gemittelten Werte der letzten 20 Aufzeichnungen aus den Evaluierungsdaten sind Tabelle 6.1 zu entnehmen. Es ist deutlich zu erkennen das sowohl AUC als auch Accuracy bereits konvergiert sind. Precision und Recall hingegen erlauben noch Spielraum für weitere Trainingsiterationen. Ihr mittlerer Fehler im Mittelwerte zeigt, dass noch keine Konvergenz erreicht ist. Das Training für *SimpSeq-HexConv-70* und *SimpSeq-Conv-70* wurden daher testweise auf 400.000 Iterationen erhöht, um eine mögliche Konvergenz zu überprüfen.

| Modell | AUC | Precision | Recall | Accuracy |
|---|---|---|---|---|
| SimpSeq-HexConv-70 | 0.85 ±0.001 | 0.90 ±0.001 | 0.96 ±0.002 | 0.90 ±0.000 |
| SimpSeq-Conv-70 | 0.85 ±0.001 | 0.90 ±0.002 | 0.96 ±0.002 | 0.90 ±0.000 |

**Tabelle 6.2:** Gemittelte Messwerte mit Fehler aus den letzten 20 Evaluationsschritten nach 400.000 Trainingsiterationen.

Die Experimente unterstützen die Vermutung, dass beide Verfahren ähnliche Resultate erbringen, da eine Konvergenz aller Güten im selben Zeitraum stattfindet, wie in Tabelle 6.2 zu erkennen ist. Ein Vorteil des bekannten Verfahrens ist allerdings die große Unterstützung vieler Frameworks, wodurch eine Umsetzung leicht fällt. Hexagonale Convolution benötigt dafür weniger Parameter zum Training, da ihre Filter aus echten Hexagone bestehen. Solche können auch in der klassischen Convolution imitiert werden, indem die Filtermatrix mit Null-Werten aufgefüllt wird. Eine Arbeit von Stefan Rötner [30] nutzte dieses Verfahren bei der Bildanalyse von FACT-Aufnahmen.
Im Fall von SimpSeq mussten gut 12098 Parameter im rechteckigen Verfahren und nur 8710 auf HexConv trainiert werden. Dieser Unterschied kann, bei guter Optimierung, Trainingszeit und Speicheraufwand reduzieren.

## 6.1 Performance

Beide Verfahren wurden eine identische Anzahl Epochen lang trainiert, allerdings benötigte HexConv in Durchschnitt 30% länger. Es stellt sich die Frage, wie gut das implementierte HexConv Verfahren skaliert.
In Abbildung 6.4 ist die relative Veränderung der Anzahl bearbeiteter Daten pro Sekunde zu sehen. Getestet wurden unterschiedliche SimpSeq Modelle mit vergrö-

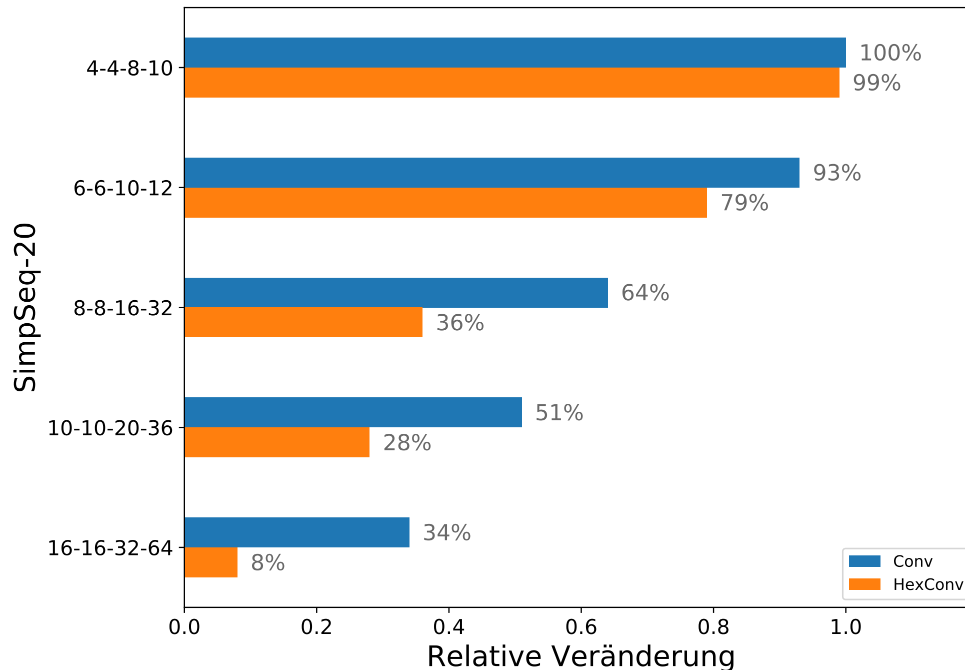



**Abbildung 6.4:** Relative Anzahl bearbeiteter Events pro Sekunde bezogen auf zu trainierenden Parametern. Ausgehend vom schnellsten Modell (Conv-4-4-8-10) sind SimpSeq Modelle mit unterschiedlichen Parameterräumen der relativen Veränderung zur diesem aufgetragen. Die prozentuale Verlangsamung zur Basline ist jedem Balken angefügt. Testversuche sind mit TensorFlow Release 1.9, CUDA 9.2 und cuDNN 7.1.4 auf einer GTX 970 mit lokalen Datenzugriff ausführt worden.

ßerten Filtern. Die Anzahl der getesteten Filtern sind der y-Achse zu entnehmen. Die prozentuale Geschwindigkeit ist den Balken angefügt. Diese zeigt einen direkten Vergleich zu der schnellsten Variante. Beispielshalber bearbeitete HexConv 92% weniger Events pro Sekunde in der Filterkonfiguration $16-16-32-64$ als die schnellste Variante. Dagegen konnte das rechteckige Verfahren noch 34% der ursprünglichen Anzahl bearbeiten, also verbuchte nur ein Defizit von 66%.
Anhand der Graphik ist deutlich zu erkenne, dass die derzeit implementierte HexConv nicht gut mit einer hohen Anzahl von zu trainierenden Parametern skaliert. In der Basiskonfiguration konnte das Verfahren auf der getesteten GTX 970 noch mithalten, mit jeder Steigerung von zu trainierenden Parametern verschlechterte sich dies jedoch.
Wie bereits in Kapitel 4 angedeutet, kann dies der CUDA-Implementierung zugeschrieben werden. Die derzeit implementierte Variante der GPU-Operatoren greifen nur auf den globalen Speicher zu, welcher einen deutlich langsameren Zugriff ermöglicht als beispielsweise *shared* oder *texture memory*[1]. Die Umstellung auf diese

[1] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

Speicherzugriffe war aufgrund der zeitlichen Begrenzung dieser Arbeit nicht mehr realisierbar. Da ein großes Verständnis von CUDA-internen Methoden verlangt wird, ist diese Umstellung auch nicht trivial. Die Nutzung von *shared memory* verlangt ein tiefes Verständnis der Speicherverwaltung, welche die Reihenfolge des Speicherzugriffs mit berücksichtigt.

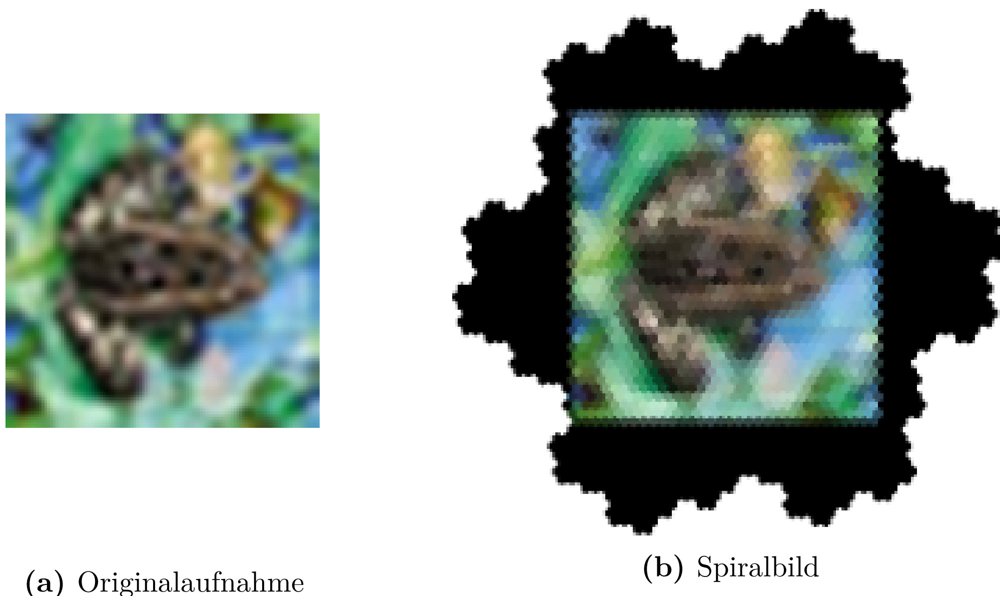
# 7 Zusammenfassung

In dieser Arbeit wurde eine hexagonale Convolution (HexConv) erarbeitet und in TensorFlow integriert[1]. Weiterhin wurden Experimente mit FACT-Daten durchgeführt und ein direkter Vergleich zwischen bekannter rechteckiger Convolution und HexConv aufgestellt.
Zunächst wurden verschiedenen Adressierungsarten für hexagonale Bilder erläutert. Als Adressierung wurde die Spiraladressierung aus dem HIP System der mehrdimensionalen mit orthogonalen Achsen vorgezogen. Zwischen Mehrdimensionaler Adressierung und Spiraldarstellung versprach die Letztere aufgrund des Spiral-Rechensystems eine effizientere Speicherung der Daten. Ein Nachteil der Spiraladressierung, das exponentielle Anwachsen der Level, wurde mit Hilfe einer Ringindexierung umgegangen. Diese vergrößert sich mit ihrer Ringanzahl nur noch quadratisch und eignet sich besser aufgrund ihrer Struktur für hexagonale Convolution.
Weiterhin wurden die notwendigen Schritte zum Übergang zur HexConv näher beschrieben. Dazu gehören die allgemeine Arithmetik zu Berechnung der resultierenden Formen nach Anwendung von Filtern. Anstelle von Höhe und Breite wurde eine Ringzahl eingeführt, die an Stelle beider Dimensionen nun zur Berechnung der Output-Form genutzt wird. Die dadurch notwendigen Anpassungen für Padding, Schrittweite und Dilation wurden ebenfalls erläutert.
Es wurden markante Merkmale des FACT-Datensatzt extrahiert, die anschließend in zwei Datensätze mit unterschiedlicher Verteilung gesampelt wurden. Dabei wurde ein Random-Undersampling genutzt, um die mehrheitliche Klasse zu reduzieren. Mit den extrahierten Datensets wurde ein Modell (SimpSeq) trainiert. Dieses besteht aus mehreren Convolutional-, Normalisierung- und Aktivierungsebenen.
Es wurden sowohl rechteckige als auch hexagonale Convolution auf beiden Datensätze mit dem vorgestellten Modell trainiert. Dabei konnte kein auffälliger Unterschied sowohl zwischen den beiden Datensätzen, als auch zwischen den beiden Verfahren festgestellt werden.
Aufgrund fehlender Optimierung im CUDA-Code trainierten Modelle mit HexConv deutlich langsamer als mit dem rechteckigen Verfahren, das in TensorFlow enthalten war. Die Auswertung zeigte deutliche Schwächen in der Skalierung von Modellgröße und Parameteranzahl. Besonders breite Filter erwiesen sich als Problemquellen für langsame Ausführungszeiten. Lösungsansätze durch Umstellung der Speicherverwaltung wurden genannt, aber aufgrund der Komplexität nicht umgesetzt.
Das entwickelte HexConv Verfahren erbringt gleichwertige Ergebnisse mit einer reduzierten Anzahl an zu trainierenden Parameter. Dies kann allerdings dem zugrunde liegenden Modell zugeschrieben werden, welches aufgrund seiner Topologie keine Unterschiede aufweisen lässt. Da die Skalierung des Parameterraums zu erhöhten Laufzeiten geführt hat, sind keine größeren Modelle hier trainiert worden. Die Auswirkung größere Parameterräume wurde daher nicht weiter untersucht.

[1] `https://bitbucket.org/maymic/tensorflow`

## 7.1 Ausblick

Ein wichtiger Aspekt ist weiterhin die Optimierung des Verfahrens. Zum einem muss der CUDA-Code zur GPU-Unterstützung verbessert werden. Dazu müssen effizientere Routinen genutzt werden, um die Convolution zu berechnen. Wie bereits in Kapitel 4 angedeutet kann der Einsatz von Shared-Memory dies ermöglichen. Aufgrund der zeitlichen Begrenzung dieser Arbeit konnte dies allerdings nicht mehr im vollen Maße auf allen Bereichen der hexagonalen Convolution übertragen werden.
Weiterhin müssen Operatoren entwickelt werden, die neben der Convolution selbst auch Pooling umsetzen können. In dieser Arbeit wurde das HexPooling in Abschnitt 3.5 beschrieben, allerdings wird diese derzeit nur für CPU-Operatoren umgesetzt. Es werden noch GPU-Operatoren benötigt, die sowohl das Pooling und dessen Gradienten implementieren.
Die Resultate zeigten, dass die vorgestellte HexConv auf dem FACT-Datensatz sehr ähnliche Ergebnisse erzielen. Daher sollten weitere Modelle getestet werden. Eine Umsetzung bestehender Modelle wie ResNet [17, 40, 16] erscheint eine sinnvolle Erweiterung zu sein. Ein Test auf den ImageNet Datensatz ist dann auch von Interesse, da dies der bekanntest Testdatensatz im Bereich von Bildanalysen ist. Dies verlangt allerdings ein sinnvolles Sampling der rechteckigen Bildstrukturen in hexagonale. Daher sollten geeignete Maßnahmen zum Sampling rechteckiger Bilder näher untersucht werden. Der im HIP System [27] zur Verfügung gestellt Python-Code scheint hier eine gute Basis zu geben. Abbildung 7.1 stellt die beiden Formate dar. Ein konvertiertes CIFAR Bild kann so in die Spiraldarstellung umgewandelt werden.



**(a)** Originalaufnahme

**(b)** Spiralbild

**Abbildung 7.1:** Übertragung eines Bildes im rechteckigen Gitter (7.1a) auf das HIP System (7.1b).

Die nächsten Schritte würden nun eine Eingrenzung des Spiralbildes auf die Ringindexierung benötigen. So eingegrenzte Bilder können dann direkt in TensorFlow mit HexConv Erweiterung eingeführt und weiter analysiert werden.

# A HexConv

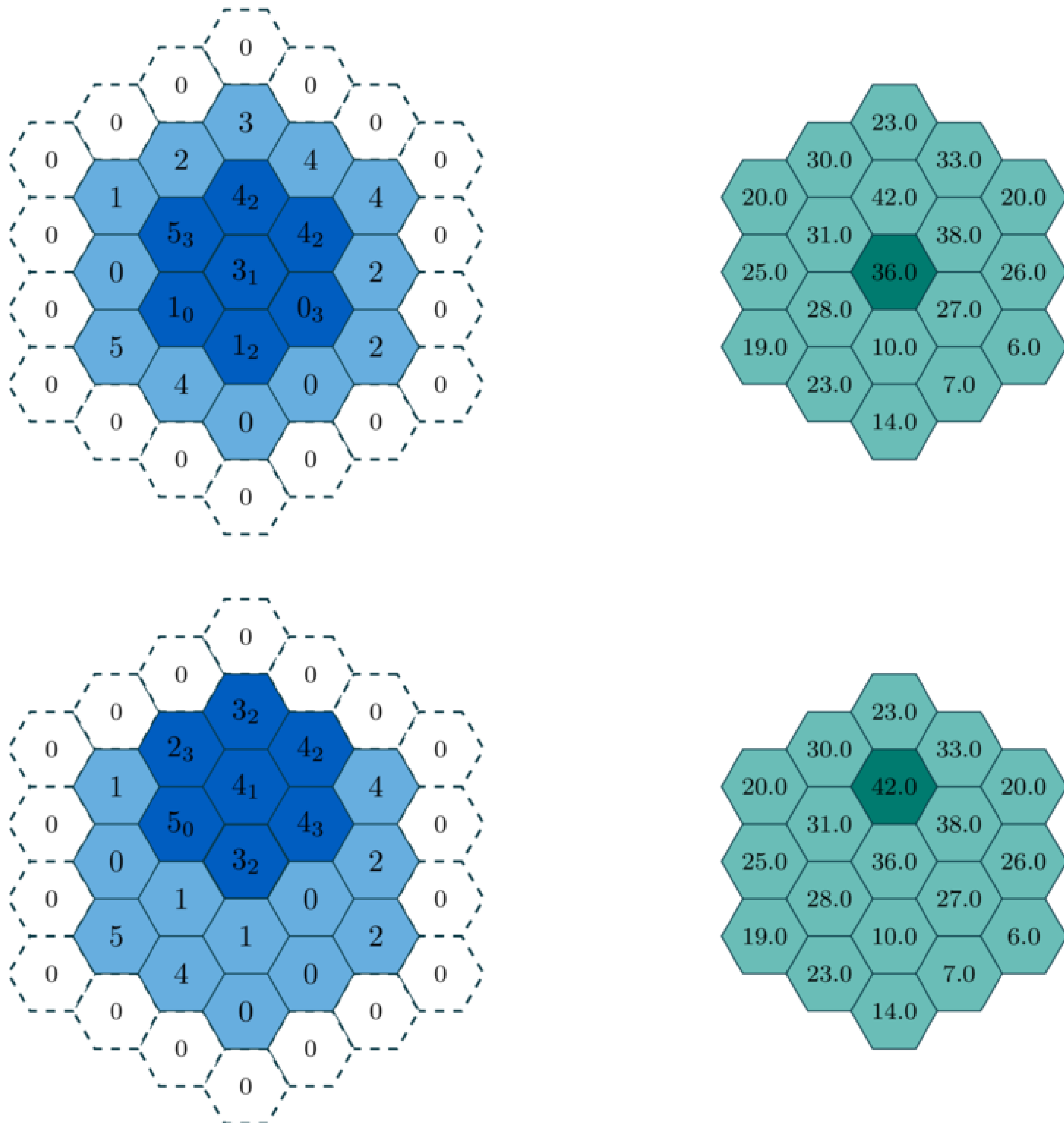



**Abbildung A.1:** Weitere Beispiele der HexConv mit $R_I = 2$, $R_P = 1$, $R_S = 1$ und $R_W = 1$. Linke Seite ist die Eingabe (Blau), rechts die Ausgabe (Cyan) mit Hervorhebung der Aktiven Berechnung. Gewichte sind den Eingabewerten als Subskript angefügt.

# Abbildungsverzeichnis

1.1 Auszug einer Aufzeichnung aus dem Monitoring-Tool `smartfact`[1]. Die Legenden wurden zur Lesbarkeit vergrößert. Gezeigt sind die einzelnen Pixel einer FACT-Aufnahme. Die Farbwerte entsprechen den Spannungswerten. . . . . 3

2.1 Nachbarschaft auf rechteckigen Gittern . . . . . 7
2.2 Horizontale, vertikale und diagonale Geraden im hexagonalen Gitter . 8
2.3 Zweidimensionale Adressierung über 90° versetze Achsen. . . . . . 9
2.4 Beispielhafte Platzierung der um 60° verzerrten Achsen in einer zweidimensionalen Adressierung. . . . . 10
2.5 Nummerierung eines Level 1 Cluster im HIP System. Die Nummerierung startet im Zentrum (0) und setzt sich in einer spiralen Bewegung fort (1-6). . . . . 10
2.6 Rekursive Adressierung eines Level 2 Clusters . . . . . 11
2.7 Abstände von Hexagone im Gitter . . . . . 13
2.8 Spiraladdition in Vektordarstellung. Dargestellt wird die Addition der Element ***25*** und ***36***. Das Ergebnis ***33*** entstammt der Vektoraddition beider Adressen. . . . . 15
2.9 Ringindexierung mit vier Ringen und zugehörige Spiraladressen (Spiralaresse/Ringindex). In blau hervorgehoben sind die Spiraladressen aus $\mathbb{S}^4$. . . . . 18

3.1 Vereinfachte Darstellung künstlicher neuronaler Netzwerke. Gezeigt sind Input-Layer (blau), Hidden-Layer (grün) und Output-Layer (rot). Kanten zwischen zwei Knoten stellen die gewichtete Übertragungsfunktion dar. . . . . 20
3.2 Visualisierung der Auswirkung des Paddings auf die Ausgaben. Abbildungen (a) - (f) zeigen Teilschritte der hexagonalen Convolution ohne Padding, und (d) - (f) mit Padding. . . . . 24
3.3 Überspringen von Ringen und Elemente durch höhere Schrittgrößen $R_S = 2$ mit $R_I = 3$, $R_P = 0$ und $R_W = 1$. Dargestellt sind die ersten sieben Schritte der iterativen Convolution. . . . . 25
3.4 Visualisierung unterschiedlicher Ausdehnungsfaktoren für HexConv . 26
3.5 MAX-HexPooling mit $R_I = 3$, $R_P = 0$, $R_S = 2$ und $R_W = 1$ und Beispielwerten. Auf der linke Seite ist die Eingabe (Blau) und rechts die Ausgabe (Cyan) zu erkennen. Gewichte sind den Eingabewerten als Subskript angefügt. . . . . 28

4.1 Architektur des TensorFlow-Frameworks. Illustriert sind die einzelnen Schichten der Architektur. . . . . 29
4.2 Mögliche Spezifiktation eines HexConv Operators. . . . . 30

4.3 Implementation eines binären Operator mit Klassentemplates für Datentyp und Ausführungsdevice . . . . . . . . . . . . . . . . . . . . . 31

4.4 Registierung der HexConv2DOp-Klasse mit Namen HexConv2D und CPU-Unterstützung . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

4.5 Verknüpfung der *blocks* und *threads* in CUDA. Ein verteiltes Programm wird in parallelisierte Blöcke unterteilt, welche jeweils eine festgelegte Anzahl von Threads ausführen können. Bild dem CUDA *programming guide* entnommen: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/grid-of-thread-blocks.png`. 33

5.1 FACT-Aufnahmen auf Spiraler- und Ring-Adressierung mit einem Overlay der Gesamtgröße des jeweiligen Verfahrens. . . . . . . . . . . 35

5.2 `streams`-XML zur Extraktion der Lichtintensität. Anwendung erfolgte mit einer `streams` Erweiterung mit Sparks Unterstützung, die von der Projektgruppe 594 der TU Dortmund entwickelt wurde. Ebenfalls wird `fact-tools` zur Extraktion der FACT-Daten genutzt. . . . . . . 37

5.3 Inhalte eines exportierten FACT-Exports im TFRecord-Format. Gezeigt wird die exportierten Elemente und deren Datentyp. . . . . . . 38

5.4 Visualisierung einer FACT-Aufnahme durch rechteckige und hexagonale Pixel. Die hexagonalen Pixel wurden hierbei durch eine Vielzahl Rechteckiger approximiert. Diese werden auch Hyperpixel genannt. . 39

5.5 Die zwei Grundbausteine des SimpSeq Modells. Der obere Baustein enthält eine einzelne Downsampling-Schicht (VALID-CONV) und der Untere wird um eine zusätzliche erweitert. . . . . . . . . . . . . . . 40

5.6 Gesamte Netzwerkstruktur des SimpSeq Modells. . . . . . . . . . . . 41

6.1 Graphiken zu AUC (6.1a), Recall (6.1b), Precision (6.1c) und Accuracy (6.1d) aus Training und Evaluierung des SimpSeq-Modells mit HexConv. Aufgetragen ist die erreichte Güte auf der y-Achse in Abständen von 100 Iterationen (x-Achse) für die Trainingsphase. Güten der Evaluationsphase werden in Abständen von zehn Minuten, die das Netzwerk trainiert, aufgezeichnet. Die Datensätze *fact-70* (rot) und *fact-50* (grün) sind farblich hervorgehoben. Die Trainingskurve gibt den allgemeinen Trend des gesamten Trainings wieder, während für Evaluierung Momentaufnahmen aus der jeweiligen Iteration mit dem trainierten Modell angezeigt werden. Die Legende setzt sich aus genutzten Datensatz, Training- oder Evaluationsphase und Art der Convolution zusammen; *dataset*-type-*conv*. . . . . . . . . . . . . . . . 43

6.2 Graphiken zu AUC (6.2a), Recall (6.2b), Precision (6.2c) und Accuracy (6.2d) aus Training und Evaluierung des SimpSeq-Modells mit klassischer Convolution. . . . . . . . . . . . . . . . . . . . . . . . . 44

6.3 Überlagerung der Güten AUC (6.3a), Recall (6.3b), Precision (6.3c) und Accuracy (6.3d) beider Convolutionen Verfahren auf 150.000 Iterationen. HexConv (grün) und Conv (rot) werden farblich voneinander getrennt. Die Legende unterteilt sich in den genutzten Datensatz, das Evaluationsdaten genutzt werden und die Art der Convolution; *dataset*-type-*conv*. . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.4 Relative Anzahl bearbeiteter Events pro Sekunde bezogen auf zu trainierenden Parametern. Ausgehend vom schnellsten Modell (Conv-4-4-8-10) sind SimpSeq Modelle mit unterschiedlichen Parameterräumen der relativen Veränderung zur diesem aufgetragen. Die prozentuale Verlangsamung zur Basline ist jedem Balken angefügt. Testversuche sind mit TensorFlow Release 1.9, CUDA 9.2 und cuDNN 7.1.4 auf einer GTX 970 mit lokalen Datenzugriff ausführt worden. . . . . . . . 47

7.1 Übertragung eines Bildes im rechteckigen Gitter (7.1a) auf das HIP System (7.1b). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

A.1 Weitere Beispiele der HexConv mit $R_I = 2$, $R_P = 1$, $R_S = 1$ und $R_W = 1$. Linke Seite ist die Eingabe (Blau), rechts die Ausgabe (Cyan) mit Hervorhebung der Aktiven Berechnung. Gewichte sind den Eingabewerten als Subskript angefügt. . . . . . . . . . . . . . . . . . . 53

# Literatur

[1] Martín Abadi u. a. „Tensorflow: a system for large-scale machine learning“. In: *OSDI*. Bd. 16. 2016, S. 265–283.

[2] Martín Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[3] H Anderhub u. a. „Design and operation of FACT–the first G-APD Cherenkov telescope“. In: *Journal of Instrumentation* 8.06 (2013), P06008.

[4] H Anderhub u. a. „FACT—The first Cherenkov telescope using a G-APD camera for TeV gamma-ray astronomy“. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 639.1 (2011), S. 58–61.

[5] Fayas Asharindavida, Nisar Hundewale und Sultan Aljahdali. „Study on hexagonal grid in image processing“. In: *Saudi Arabia: Stultan Aljahdali Taif University* (2012).

[6] RK Bock u. a. „Methods for multidimensional event classification: a case study using images from a Cherenkov gamma-ray telescope“. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 516.2 (2004), S. 511–528.

[7] Christian Bockermann und Hendrik Blom. „The streams framework“. In: *TU Dortmund University, Tech. Rep* 5 (2012), S. 12.

[8] Christian Bockermann u. a. „Online analysis of high-volume data streams in astroparticle physics“. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2015, S. 100–115.

[9] François Chollet u. a. *Keras*. `https://keras.io`. 2015.

[10] Sonya Coleman, Bryan Gardiner und Bryan Scotney. „Adaptive tri-direction edge detection operators based on the spiral architecture“. In: *Image Processing (ICIP), 2010 17th IEEE International Conference on*. IEEE. 2010, S. 1961–1964.

[11] Franklin C Crow. „Summed-area tables for texture mapping“. In: *ACM SIGGRAPH computer graphics* 18.3 (1984), S. 207–212.

[12] Jia Deng u. a. „Imagenet: A large-scale hierarchical image database“. In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE. 2009, S. 248–255.

[13] Vincent Dumoulin und Francesco Visin. „A guide to convolution arithmetic for deep learning“. In: *ArXiv e-prints* (2016). eprint: `1603.07285`. URL: `https://github.com/vdumoulin/conv_arithmetic`.

[14] Apache Software Foundation. *Keras*. `https://hadoop.apache.org/`. 2011.

[15] Bryan Gardiner, Sonya A Coleman und Bryan W Scotney. „Fast Multiscale Operator Development for Hexagonal Images“. In: *DAGM-Symposium*. Springer. 2009, S. 282–291.

[16] Kaiming He u. a. „Deep residual learning for image recognition“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, S. 770–778.

[17] Kaiming He u. a. „Identity mappings in deep residual networks“. In: *European conference on computer vision*. Springer. 2016, S. 630–645.

[18] Innchyn Her. „Geometric transformations on the hexagonal grid“. In: *IEEE Transactions on Image Processing* 4.9 (1995), S. 1213–1222.

[19] Innchyn Her und Chi-Tseng Yuan. „Resampling on a pseudohexagonal grid“. In: *CVGIP: Graphical Models and Image Processing* 56.4 (1994), S. 336–347.

[20] Emiel Hoogeboom u. a. „HexaConv“. In: *arXiv preprint arXiv:1803.02108* (2018).

[21] Sergey Ioffe und Christian Szegedy. „Batch normalization: Accelerating deep network training by reducing internal covariate shift“. In: *arXiv preprint arXiv:1502.03167* (2015).

[22] Yangqing Jia u. a. „Caffe: Convolutional Architecture for Fast Feature Embedding“. In: *arXiv preprint arXiv:1408.5093* (2014).

[23] Alex Krizhevsky und Geoffrey Hinton. „Learning multiple layers of features from tiny images“. In: (2009).

[24] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Advances in neural information processing systems*. 2012, S. 1097–1105.

[25] Yann LeCun, Corinna Cortes und Christopher JC Burges. „MNIST handwritten digit database“. In: *AT&T Labs [Online]* (2010). URL: `http://yann.lecun.com/exdb/mnist`.

[26] Andrew L Maas, Awni Y Hannun und Andrew Y Ng. „Rectifier nonlinearities improve neural network acoustic models“. In: *Proc. icml*. Bd. 30. 1. 2013, S. 3.

[27] Lee Middleton und Jayanthi Sivaswamy. *Hexagonal image processing: A practical approach*. Springer Science & Business Media, 2006.

[28] Adam Paszke u. a. „Automatic differentiation in PyTorch“. In: (2017).

[29] Victor Podlozhnyuk. „Image convolution with CUDA“. In: *NVIDIA Corporation white paper, June* 2097.3 (2007).

[30] Stefan Rötner. „Deep Learning on Raw Telescope Data“. Masterarbeit. TU Dortmund University, 2017.

[31] Bryan Scotney, Sonya Coleman und Bryan Gardiner. „Biologically motivated feature extraction using the spiral architecture“. In: *Image Processing (ICIP), 2011 18th IEEE International Conference on*. IEEE. 2011, S. 221–224.

[32] Phillip Sheridan. „Spiral Architecture for machine vision“. Diss. 1996.

[33] David Silver u. a. „Mastering the game of Go with deep neural networks and tree search“. In: *nature* 529.7587 (2016), S. 484.

[34] Karen Simonyan und Andrew Zisserman. „Very deep convolutional networks for large-scale image recognition“. In: *arXiv preprint arXiv:1409.1556* (2014).

[35] Christian Szegedy u. a. „Going deeper with convolutions“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, S. 1–9.

[36] Vinod Kumar Vavilapalli u. a. „Apache hadoop yarn: Yet another resource negotiator“. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, S. 5.

[37] Gang Wu und Edward Y Chang. „Class-boundary alignment for imbalanced dataset learning“. In: *ICML 2003 workshop on learning from imbalanced data sets II, Washington, DC*. 2003, S. 49–56.

[38] Qiang Wu, Xiangjian He und Tom Hintz. „Virtual Spiral Architecture.“ In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'04*. Bd. 1. Jan. 2004, S. 399–405.

[39] Fisher Yu und Vladlen Koltun. „Multi-scale context aggregation by dilated convolutions“. In: *arXiv preprint arXiv:1511.07122* (2015).

[40] Sergey Zagoruyko und Nikos Komodakis. „Wide residual networks“. In: *arXiv preprint arXiv:1605.07146* (2016).

[41] Matei Zaharia u. a. „Spark: Cluster computing with working sets.“ In: *HotCloud* 10.10-10 (2010), S. 95.