HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Unsupervised Image Clustering

Alexander-Sebastian Clauß

Master-Arbeit im Studiengang „Angewandte Informatik“

25. November 2017

Autor Alexander-Sebastian Clauß
Matrikelnummer: 1381164
alexander-sebastian.clauss@hs-hannover.de

Erstprüferin: Prof. Dr. Frauke Sprengel
Abteilung Informatik, Fakultät IV
Hochschule Hannover
frauke.sprengel@hs-hannover.de

Zweitprüfer: Maximilian Zubke, M.Sc.
Abteilung Information und Kommunikation, Fakultät III
Hochschule Hannover
maximilian.zubke@hs-hannover.de

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Master-Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hannover, den 25. November 2017 Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Gegenstand dieser Arbeit ist die Entwicklung eines maschinellen Lernverfahrens zur Kategorisierung von großen Bildmengen. Hintergrund hierfür ist, dass die Fakultät III der Hochschule Hannover (HsH) eine Datenbank mit mehreren Millionen Bildern besitzt, die ohne Information auf Herkunft oder Inhalt gespeichert wurden. Eine Analyse und Gewinnung von Informationen aus solch großen Datenmengen wird *Data Mining* genannt. Die maschinellen Lernverfahren bilden ein Teilgebiet dieser Disziplin und viele der Konzepte gehen bereits auf die 40er Jahre zurück. Doch erst durch den enormen Anstieg der Rechenleistung und günstig gewordenen Speicher ist ein praktischer Einsatz interessant geworden. Ein wesentlicher Faktor dieser Verbreitung ist auch im Fortschritt des *GPU (Graphics Processing Units)* Computing begründet. Hier wird die parallele Architektur der Grafikkarte genutzt, um numerische Probleme um ein vielfaches schneller lösen zu können, als dies mit CPUs möglich wäre. Vor diesem Hintergrund und dem Fakt, dass es eine enorm große Menge an Daten zu verarbeiten gilt, ist das Modell auf eine Ausführung auf Grafikkarten ausgelegt.
Zunächst skizziert der Abschnitt „Aufbau und Ablauf“ die Struktur dieser Arbeit. Im Abschnitt Ziele wird dann formuliert, welchen Anforderungen das Modell genügen muss und welche Ergebnisse erwartet werden.

1.1 Aufbau und Ablauf

Das Grundlagenkapitel beginnt mit einer Einführung in digitale Bilder sowie die Erhebung und Kodierung von „interessanten“ Informationen aus diesen, die sogenannten Features. Es wird definiert, wie Bilder und Features hier mathematisch notiert werden und Operationen, die im Weiteren von Bedeutung sind, auf diesen. Im nächsten Abschnitt wird in das Themengebiet des *Machine Learning* eingeführt. Hier werden die verschiedenen Arten von Lernmethoden definiert und näher auf die Teilbereiche der unüberwachten Lernverfahren eingegangen. Da im Weiteren der Autoencoder und der Bag of Visual Words von Bedeutung sind, sind diese Gegenstand der beiden folgenden Abschnitte. Ein Autoencoder ist ein spezielles neuronales Netzwerk, dass hier zu Kompression von Daten genutzt wird. Das Bag of Visual Words Modell dient dann zur Gruppierung dieser Daten. Einer Erläuterung der Funktionsweise schließt eine Untersuchung

der Parallelisierbarkeit des Algorithmus durch Grafikkarten an. Für die Ausführung dieser Algorithmen auf Grafikkarten wurde Nvidias CUDA Plattform gewählt. Daher folgt eine Einführung in das CUDA Ausführungs- und Programmiermodell. Letztes wird am Beispiel eines CUDA Programms zur Vektor Addition demonstriert.
In der Analyse werden zu Beginn geeignete unüberwachte Lernverfahren, zur Gruppierung von Bildern, aus den Arbeiten anderer betrachtet. Anhand dieser Information wird dann ein geeignetes Modell für den vorliegenden Anwendungsfall gewählt. Im Abschnitt „Features“ werden etablierte Verfahren zur Feature-Detektion und -Extraktion für Bilder betrachtet. Auf dieser Basis wird dann entschieden welcher Detektor bzw. Deskriptor im Weiteren verwendet wird.
In der Konzeption wird zu Beginn ein Überblick über das gesamte Modell gegeben: Es werden die drei Phasen der Verarbeitung skizziert, sowie die Daten welche je Phase erwartet bzw. produziert werden. Anschließend werden die Funktionen und der Ablauf der einzelnen Phasen in je einem Abschnitt detailliert ausgearbeitet. Für den Entwurf eines Autoencoders wurde das Modell aus der Arbeit von Zhao [4] als Basis verwendet. Neben der Architektur des Netzes werden die von Zhao gewählten Hyperparameter näher beleuchtet. Im Abschnitt zum Bag of Visual Words wird dann das Erzeugen eines Modells aus Bild-Features, der Vergleich von Bildern durch das Modell und ein Persistenzmechanismus konzipiert.
Die Implementierung behandelt in drei Abschnitten die konkrete Umsetzung der Extraktion von Features, der Komprimierung dieser durch einen Autoencoder und letztlich das Gruppieren durch einen Bag of Visual Words. Es wird gezeigt, wie die rechenintensiven Operationen durch eine Grafikkarte stattfinden: Der Autoencoder ist in Googles Deep Learning Framework TensorFlow umgesetzt worden und profitiert automatisch von TensorFlows CUDA Schnittstelle. Der Bag of Visual Words ist direkt in CUDA C implementiert. Als Verfahren zum Gruppieren der Features wird ein k-means-Algorithmus verwendet. Insbesondere wird hier die Optimierung des Programms durch die Verwendung von CUDAs *shared memory* veranschaulicht, sowie Einschränkungen aufgezeigt, die dies mit sich bringt.
Das Kapitel „Evaluierung“ stellt zunächst die verwendete Menge an Testdaten, die Bilder des „Caltech101“, vor und illustriert wie ein Experiment durchgeführt wird. Es wird die Qualität der Ergebnisse zwischen dem SIFT-Deskriptor und Zhaos Autoencoderbasierten Deskriptor für Bildfeatures verglichen und die Laufzeiten gegenübergestellt.

Der Quellcode für die in dieser Arbeiten entwickelten Programme, liegt separat auf einem USB-Stick bei. Eine Erläuterung der Applikationen und die Art ihrer Verwendung wurde ebenfalls beigefügt.

1.2 Ziele

Ziel der Arbeit ist es, ein Verfahren zu entwickeln, die Bilder einer große Menge in eine vorgegebene Anzahl von Kategorien einzuteilen, um so semantische Informationen über die Bilder zu gewinnen. Hierbei leiten sich direkt Anforderungen an das Modell ab, doch andere Faktoren, wie der praktische Einsatz der Software und die Vergleichbarkeit der Ergebnisse sollen ebenfalls berücksichtigt werden. Konkret soll diese Arbeit folgende Ergebnisse liefern:

- **Gruppierung der Bilddaten** Eine große Menge Bilder (mehrere Millionen) soll durch das Modell in annehmbarer Zeit gruppiert werden. Bei so vielen Datensätzen sind mehrere Stunden Laufzeit zu erwarten, Tage oder gar Wochen sind aber nicht akzeptabel.
- **Verwendung von CUDA** Die zeitaufwendigsten Berechnungen sollen parallel durchgeführt werden. Da an der HsH und auch insbesondere der Fakultät III CUDA für parallele Berechnungen durch Grafikkarten verwendet wird, soll die Implementierung CUDA als technische Basis verwenden.
- **Bewertung der Ähnlichkeit von Bildern** Die Ähnlichkeit von Bildern soll durch das Modell bewertet werden können. Dabei wird kein „Label“ (eine Bezeichnung / ein Name) für das Bild erzeugt. Ein Algorithmus kann unmöglich von sich aus lernen, dass Menschen die Abbildung einer Katze „Katze“ nennen. Wenn das Modell aber zwei Bilder entgegennimmt, soll bewertet werden, wie ähnlich diese sich sind.
- **Vergleichbarkeit der Ergebnisse** Die Ergebnisse sollen prinzipiell mit denen anderen vergleichbar sein, in dem eine wissenschaftlich anerkannte Testmenge an Bilddaten verwendet wird.

2 Grundlagen

Das Grundlagenkapitel gibt eine Einführung und Übersicht über die verwendeten Begriffe und Konzepte, die im Weiteren dieser Arbeit von Bedeutung sind. Zunächst wird die Darstellung von Bildern behandelt und anschließend die Erhebung von charakteristischen Merkmalen aus Bildern, den Features. Hier genutzte Operationen auf Bildern bzw. Verfahren zur Detektion und Extraktion von Features aus diesen, werden im Anschluss erläutert.
In den folgenden Kapiteln wird ein Modell konzipiert und realisiert, dass eine Gruppierung von Bildern durch einen Autoencoder und Bag of Visual Words ermöglicht. Da diese Algorithmen im Bereich des *Machine Learning* angesiedelt sind, wird zunächst eine Einführung in die Thematik gegeben, um anschließend auf die eingangs erwähnten Modelle im Detail einzugehen. Für den Bag of Visual Words wird hier näher auf die Familie der k-means-Algorithmen eingegangen. Da es sich bei einem Autoencoder um ein spezielles neuronales Netzwerk handelt, beginnt der Abschnitt zu diesem mit einer Einführung in die Grundlagen neuronaler Netze. Auf dieser Basis wird dann die Funktionsweise eines Autoencoders erläutert und zwei Erweiterungen vorgestellt.
Im letzten Teil des Kapitels wird auf die Berechnung numerischer Probleme durch Grafikkarten, das *GPU Computing*, unter Verwendung von CUDA, eingegangen. Ein Beispielprogramm zur Vektoraddition in CUDA schließt das Kapitel ab.

2.1 Bilder und Features

Der erste Abschnitt definiert wie ein Bild mathematisch aufgefasst wird, um eine effiziente Verarbeitung zu ermöglichen. Verfahren der computergestützten Bildverarbeitung erwarten logischerweise ein Bild als Eingabe. Verändern Algorithmen ein Bild, so geben sie die bearbeite Version wieder aus. Bei Analysen hingegen wird ein Bild nicht verändert, es wird auf Muster untersucht und die gefundenen Eigenschaften zurückgegeben. Diese Eigenschaften werden Features genannt. Der Prozess der Featuregewinnung ist in Detektion und Extraktion unterteilt und wird im Anschluss behandelt.

2.1.1 Bilder

Bei einem digitalen Bild handelt es sich um eine Matrix $I(x, y)$, welche Intensitätswerte enthält. Die Anzahl der Spalten m und Zeilen n entspricht dabei den Dimensionen des Bildes in Pixeln. Hier bezeichnen (x, y) die Indizes der Matrix und somit die Pixel des Bildes. Die Darstellung in Matrixform eignet sich sehr gut für Transformationen und Analysen des Bildes. Solche Verfahren betrachten meist jeden Pixel und eine Nachbarschaft dessen. Bei einer Nachbarschaft handelt es sich um die umliegenden Pixel zu einem ausgewählten Pixel. Die Größe der Nachbarschaft ist abhängig von dem Analyseverfahren: Beispielsweise umfasst die 8-er Nachbarschaft eines Pixels nur die direkt angrenzenden Pixel und ist somit eine 3×3 Matrix (mit dem Pixel als Zentrum). Formal werden diese Matrizen wie folgt dargestellt, wobei n und m wieder der Anzahl der Zeilen bzw. Spalten entsprechen:

$$I(x, y) = \begin{bmatrix} i_{0,0} & i_{0,1} & \cdots & i_{0,n-1} \\ i_{1,0} & i_{1,1} & \cdots & i_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ i_{m-1,0} & i_{m-1,1} & \cdots & i_{m-1,n-1} \end{bmatrix}$$

Abhängig vom Typ des Bildes, besitzen die Pixel eine andere Struktur. In der Bilderverarbeitung und im Weiteren dieser Arbeit werden meist folgenden Arten von Bildern verwendet:

- **Monochromatische Bilder** Diese Bilder werden in Graustufen dargestellt, daher besitzt ein Pixel genau einen Intensitätswert.
- **Multispektrale Bilder** Jeder Pixel besitzt einen Vektor an Werten. Der Pixel eines RGB-Bildes ist somit ein Vektor, der drei Komponenten für rot, grün und blau enthält.

Die Intensität eines Pixels bzw. die Intensität seiner Vektoren wird mit in den meisten Programmen mit acht Bit dargestellt und umfasst daher 256 mögliche Werte. Dies ist für die meisten Anwendungen ausreichend und praktisch, da acht Bit genau einem Byte entsprechen und dies eine einfache computergestützte Darstellung und Verarbeitung ermöglicht. Die Werte für die Intensität werden im Folgenden normalisiert als Gleitkommazahlen im Intervall $[0, 1]$ verwendet.

2.1.2 Detektion und Extraktion von Features

Um Bilder zu vergleichen, werden charakteristische Merkmale dieser betrachtet, die sogenannten Features. Ein Feature ist ein allgemeiner Begriff und enthält je nach Verfahren

unterschiedliche Informationen. Dies ist notwendig, da nicht nur die Position oder Intensität eines Pixels, sondern auch generelle Eigenschaften von Interesse sind. Globale Verfahren berücksichtigen bei der Bewertung jeden Pixel des Bildes gleichermaßen, lokale hingegen betrachten nur ein kleine Fenster des Bildes, also einen Pixel und seine Nachbarschaft. Die Suche nach globalen Merkmalen, die ein Bild charakterisieren, kann aber keine Objekte und Details im Bild berücksichtigen. Hierfür eignet sich die Extraktion von lokalen Features. Um Objekte aus unterschiedlichen Perspektiven und in verschiedenen Größen wieder zu erkennen, ist es notwendig, dass die Features affin invariant sind. Abbildung 2.1 zeigt dasselbe Objekt, jedoch rotiert, skaliert, verschoben und unter anderen Beleuchtungsverhältnissen. Ein Algorithmus sollte mit hoher Wahrscheinlichkeit erkennen, dass es sich hier um dasselbe Objekt handelt [5].

Abbildung 2.1: Zwei Bilder mit dem gleichen Objekt aus unterschiedlicher Perspektive und in verschiedenen Lichtverhältnissen[1].

Die Gewinnung der Features ist in die Schritte Detektion und Extraktion aufgeteilt:

- **Detektion** Ein Feature-Detektor in der Bildverarbeitung untersucht das Bild auf „interessante“ Regionen. Die Definition hängt hier maßgeblich vom verwendeten Verfahren ab. Allgemein wird zwischen Detektoren unterschieden die Ecken, Kanten und Regionen betrachten. Einige der Detektoren berücksichtigen hier auch mehr als eine Kategorie: Das Difference of Gaussians Verfahren dient beispielsweise der Erkennung von Ecken und Regionen. Als Ergebnis liefert ein Detektor eine

[1] Quellen: https://www.pexels.com/photo/new-york-statue-of-liberty-usa-monument-64271 (links), http://www.publicdomainpictures.net/view-image.php?image=198474& picture=statue-of-liberty (rechts)

Menge von Pixel und ggf. ihrer Nachbarschaften. Diese Pixel sind die sogenannten *keypoints* und beschreiben, mit ihren Nachbarschaften, die eingangs erwähnten „interessanten“ Stellen.
Um praktisch einsetzbar zu sein, muss ein Detektor ein Feature, dass in verschiedenen Bildern auftaucht, zuverlässig erkennen. Hier sollte aber die angestrebte Allgemeinheit berücksichtigt werden: Ein Feature Detektor für medizinische Bilder, z.B. Röntgenaufnahmen, kann speziellere Annahmen treffen, als einer für eine allgemeine Bildsuche. In Abbildung 2.2 sind die durch einen SIFT-Detektor gefundenen *keypoints* farblich hervorgehoben.

- **Extraktion** Die Feature Extraktion erzeugt aus den vom Detektor gefundenen *keypoints* und seinen lokalen Informationen den Deskriptor. Ein Feature-Deskriptor ist eine kompakte Darstellung eines Features. Die *keypoints* und deren Nachbarschaften werden in Zahlen kodiert. Die meisten Deskriptoren kodieren die Informationen bereits normalisiert im Intervall zwischen 0 und 1. Diese Sequenz wird als ein Vektor notiert, sodass auch von Feature-Vektor gesprochen wird. Einige Verfahren reichern den Deskriptor mit zusätzlichen Informationen an, z.B. über die Stärke eines Gradienten bei der Kantendetektion.

Abbildung 2.2: Gefundene *keypoints* der beiden vorigen Bilder der Freiheitsstatuen farblich hervorgehoben (SIFT Detektor)

2.1.3 Bildoperationen

Zur Detektion und Extraktion von Features werden eine Reihe mathematischer Operationen und Verfahren genutzt, wie sie in der Bildverarbeitung üblich sind. Hiervon werden diejenigen vorgestellt, die im Weiteren in der Arbeit verwendet werden.

Histogramm

Ein Histogramm ist eine diskrete Funktion, welche die Häufigkeitsverteilung in einer Menge abbildet. Hierfür wird jeder Wert der Menge einer Klasse zugeordnet. Die Größe der Intervalle einer Klasse leiten sich aus der Größe des gesamten Wertebereichs ab. So kann beispielsweise die Verteilung der Pixel eines Bildes auf die Intensitäten als Histogramm betrachtet werden. Bei einem monochromatischen Bild liegen 256 Intensitätswerte vor, die je eine Klasse repräsentieren. Beim Bilden des Histogramms wird jeder Pixels betrachtet und der Zähler der Klasse um eins inkrementiert, in deren Wertebereich der Intensitätswert des Pixels fällt. Ein Histogramm ist normalisiert, wenn es die relative Verteilung in der Menge darstellt. In Abbildung 2.3 sind im Wesentlichen zwei Bereiche zu erkennen: ein sehr heller Hintergrund und eine dunkle Katze, die den Großteil des Bildes ausmacht. Dies spiegelt sich auch im Histogramm wieder: Es ist eine große Mengen an Punkten im dunklen Bereich vorhanden (Intensität < 128) und eine kleine, extreme Häufung im hellen Bereich.

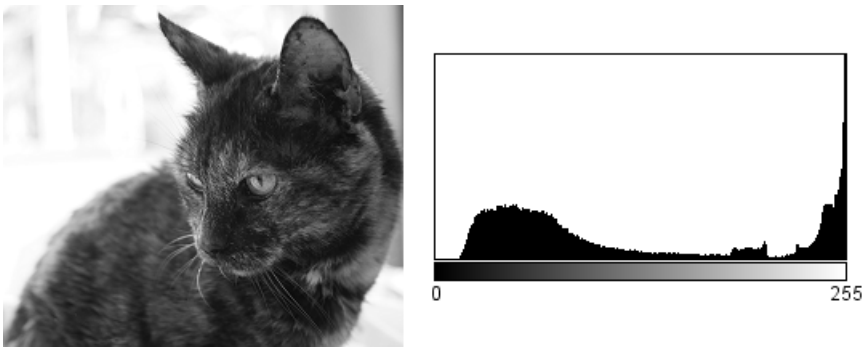

Abbildung 2.3: Graustufenbild einer Katze[2] und das Histogramm der Intensitätswerte.

Filter

Ein Filter (auch Filterkern, Faltungsmatrix) ist eine, meist quadratische, Matrix von Koeffizienten. Eine Filteroperation auf einem Bild soll bestimmte Bestandteile, z.B. Rauschen, reduzieren oder andere wie Kanten bzw. Ecken hervorheben. Als Ergebnis

[2]Quelle: https://www.pexels.com/photo/grayscale-photography-of-cat-164089/

dieses Prozesses resultiert also wieder ein Bild. Die Anwendung des Filters wird Faltung genannt und durch das $*$ Zeichen ausgedrückt. Hier handelt es sich aber nicht um eine klassische Matrixmultiplikation. Bei der Faltung wird zur Berechnung der neuen Intensität eines Pixels seine Nachbarschaft berücksichtigt. Bei Verwendung einer 3×3 Matrix zur Filterung bedeutet dies, dass der Pixel selbst und alle direkt angrenzenden, mit den entsprechenden Koeffizienten des Kerns multipliziert und anschließend aufsummiert werden.

Gauß-Filter Ein Gauß-Filter wird in der Bildbearbeitung verwendet um ein Bild zu verzerren und somit Rauschen aber auch Details zu unterdrücken. Diesem Filter liegt, wie der Name sagt, die Gauß-Funktion zu Grunde:

$$g_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{x^2}{2\sigma^2}}, \quad g_\sigma(x,y) = \frac{1}{2\pi\sigma^2} \mathrm{e}^{-\frac{x^2+y^2}{2\sigma^2}}$$

Wird ein Gauß-Filter verwendet, so bedeutet dies, dass die Einträge der Matrix aus der Gauß-Funktion konstruiert sind. Für einen 3×3 Kernel mit einer Standardabweichung von 1.0 ergibt sich folgende Matrix[3] (Annäherung durch 1000 Iterationen):

$$\begin{bmatrix} 0.077847 & 0.123317 & 0.077847 \\ 0.123317 & 0.195346 & 0.123317 \\ 0.077847 & 0.123317 & 0.077847 \end{bmatrix}$$

Difference of Gaussians Durch den Difference of Gaussian (DoG) Operator können insbesondere Kanten hervorgehoben werden. Wie der Name andeutet wird hier die Differenz zwischen zwei Bildern gebildet, auf die jeweils ein Gauß-Filter angewandt wurde. Hier variieren die Standardabweichungen der Gauß-Funktionen um zwei unterschiedlich stark verzerrte Versionen des Originals I zu erzeugen. Hierbei gilt, dass die Standardabweichung $\sigma_2 > \sigma_1$ ist:

$$DoG(x,y)_{\sigma_1,\sigma_2} = I * g_{\sigma_2}(x,y) - I * g_{\sigma_1}(x,y)$$

Gradienten Ein Gradient ist ein Vektor, der die Richtung und Größe einer Änderung enthält. Bei Bildern wird so die Richtung und Veränderungen der Intensität gemessen. Starke Veränderungen der Intensität sind besonders von Bedeutung: Hier handelt es sich potentiell um Ecken oder Kanten, die für weitere Analysen des Bildes verwendet werden können. Da zunächst nur ein Pixel vorliegt, muss senkrecht zum Gradienten untersucht werden, ob ähnliche Veränderungen der Intensität vorliegen. In Abbildung 2.4 ist links wieder das Bild der Freiheitsstatue dargestellt, rechts die Kanten, die senkrecht zu den Gradientenvektoren verlaufen. Hier ist deutlich zu sehen, dass Kanten hervorgehoben und Details ausgeblendet werden.

[3] Berechnet durch das Programm: http://dev.theomader.com/gaussian-kernel-calculator/

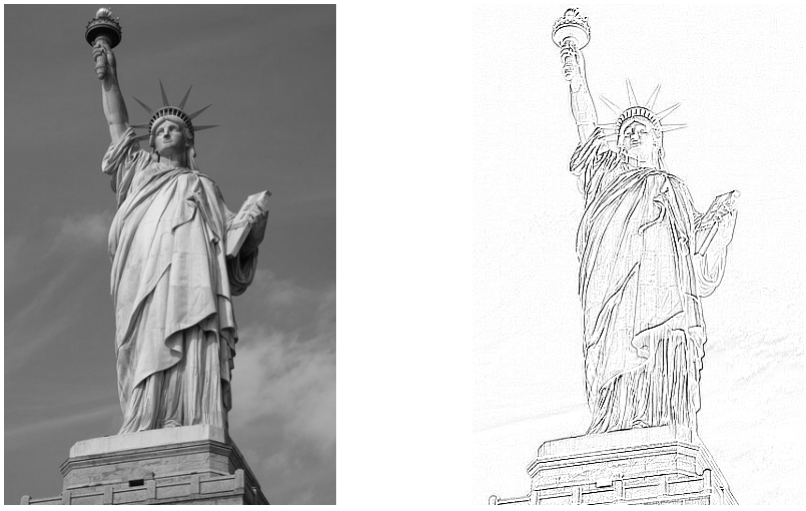

Abbildung 2.4: Links: Graustufenbild. Rechts: Kanten senkrecht zu den ermittelten Gradientenvektoren eines Bildes in horizontale und vertikale Richtung.

2.1.4 Histogram of Oriented Gradients

Der Histogram of Orientied Gradients (HOG) Deskriptor beschreibt die Features als Histogramm der Richtung der Gradienten eines Bildes. Da sich Gradienten eignen, um Kanten in Bildern zu erkennen, wird so die Form der Objekte eines Bildes erkannt. Dalal und Triggs [6] entwickelten und nutzten diesen Verfahren bereits 2005, mit großem Erfolg, um Menschen in Bilder zu erkennen. Um ein HOG zu generieren, wird das Bild in gleich große Zellen eingeteilt, die eine Nachbarschaft von Pixeln umfassen. Über die Pixel jeder Zelle werden nun die lokalen Gradienten berechnet. Diese fließen proportional zu ihrer Stärke in den kumulierten Gradienten der Zelle ein. Dabei hat jede Zelle eine feste, vorgegebene Anzahl an Klassen für die Orientierungen der Gradienten. Dalal und Triggs nutzten in ihrer Originalarbeit als Fenster eine 64×128 Nachbarschaft von Pixeln, eine Zellgröße von 8×8 Pixeln und 2×2 Zellen pro Block. In Abbildung 2.5 ist links das Fenster dargestellt und rechts die entsprechenden Gradienten der Zellen.

2.1.5 SIFT

SIFT ist ein Feature-Detektor und Deskriptor der 1999 von Lowe [1] entwickelt wurde. Neben der Größe bzw. den Maßstab, berücksichtigt SIFT teilweise auch die Rotation, Beleuchtung, Perspektive und Illumination. Der Algorithmus kombiniert mehrere mathematische Operationen und Verfahren um den Deskriptor eines Features als Ergebnis zu erhalten. Die Erzeugung des Deskriptors wird nach Lowe in vier wesentliche Schritte unterteilt:

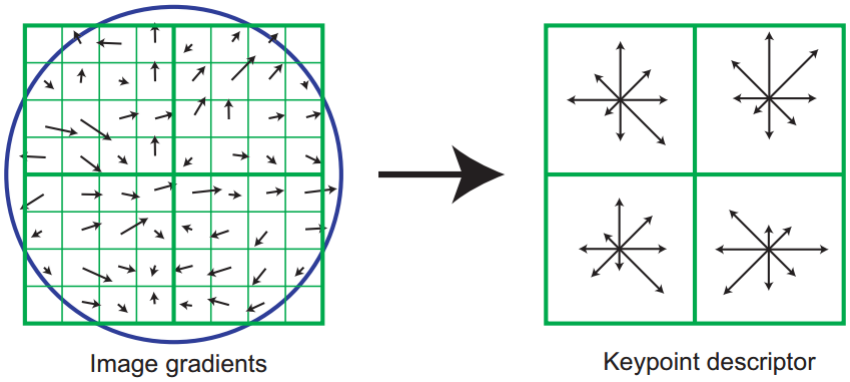

Abbildung 2.5: Schematische Darstellung des Histogram of Oriented Gradients [1]

1. **Scale space** Im ersten Schritt wird der Maßstab behandelt. SIFT konstruiert hierfür einen *scale space*. Hier werden aus dem Originalbild durch den *Difference of Gaussians* immer verzerrtere Versionen erzeugt. Die Größe der neuen Bilder wird halbiert und die Prozedur wiederholt. Alle verzerrten Bilder einer Größe bilden eine sogenannte Oktave. SIFT verwendet vier Oktaven und fünf Bilder pro Oktave um den *scale space* zu erzeugen. Um Ecken und Kanten in einem Bild zu ermitteln, die sich als Kandidaten für *keypoints* eignen, wird nun zwischen allen aufeinanderfolgenden Bildern einer Oktave die Differenz gebildet. Das Prinzip ist für zwei Oktaven in Abbildung 2.6 dargestellt. Das Ergebnis ist eine Approximation des *Laplacian of Gaussians*[4], jedoch ist dieses Verfahren weit weniger rechenintensiv.

2. **Keypoint Ermittlung** Nicht alle Kandidaten werden zu *keypoints*. Aus den DoG-Bildern werden nun die Extrempunkte bestimmt. Hierfür wird immer eine Nachbarschaft des DoG-Bildes und des Vorigen und Nachfolgenden im *scale space* betrachtet. Da die Extremwerte nicht immer exakt auf einem Pixel liegen, muss die genaue Position noch berechnet werden. SIFT verwendet hierfür eine Taylor Entwicklung im angenäherten *keypoint*.

3. **Bestimmung der Orientierung** Bei dem Aufbau der Feature-Vektoren pro *keypoint* wird die lokale Orientierung berechnet. Auf diese Weise sind die SIFT Deskriptoren invariant gegenüber Rotationen. Der SIFT Algorithmus berechnet ein HOG. Hierfür werden zufällig Punkte aus der Nachbarschaft des *keypoints* ausgewählt. Das Maximum des Histogramms wird dann als dominante Orientierung verwendet.

4. **Deskriptor** Für jeden durch den Detektor gefundenen *keypoint* wird nun ein Feature-Vektor gebildet. Der Feature-Vektor enthält Informationen über die Nach-

[4]Der Laplacian of Gaussians (LoG) oder auch Marr-Hildreth-Operator dient ebenfalls der Kantendetektion in Bildern.

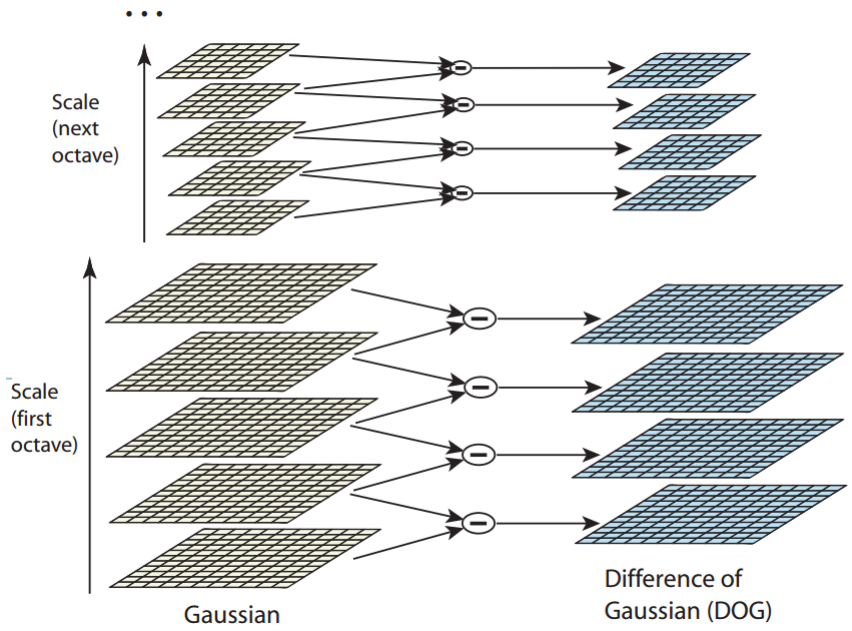

Abbildung 2.6: Schematische Darstellung der Subtraktion von Bildern im *scale space* beim Difference of Gaussians Verfahren [1].

barschaft in Form der Gradienten. Das Fenster für die Auswahl der Nachbarschaft wird auf dem *keypoint* zentriert und in vier Teilfenster unterteilt. Die Gradienten in allen Teilfenster werden in acht Richtungen gemessen, sodass der resultierende Deskriptor 128 Komponenten enthält.

2.2 Machine Learning

Machine Learning ist ein Teilgebiet der künstlichen Intelligenz und wird genutzt um Systeme zu entwerfen, die nicht explizit programmiert werden. Stattdessen lernen diese Systeme: Lernen bedeutet in diesem Kontext, dass ein System durch eine Eingabe seine Struktur verändert, um so die erwartete Leistung zu steigern. Dabei ist *Machine Learning* ein interdisziplinäres Feld: Es sind sowohl Informatik, Statistik als auch biologische und kognitive Wissenschaften involviert. Bisher haben sich viele Anwendungsfälle für *Machine Learning* Verfahren ergeben, die bis in den Alltag reichen. Einige Beispiele sind:

- **Optical Character Recognition (OCR)** Unter OCR wird das Übersetzen eines (hand-)geschriebenen Textes in ein digitales Dokument bezeichnet. Beispielsweise kann so das Einfügen von Daten in CRM / ERP System automatisiert werden.

- **Spam Filterung** Das automatische Erkennen von unerwünschten E-Mails, die Werbung enthalten oder Betrugsversuche sind, ist inzwischen bei jedem Mail-Anbieter Teil des Angebots.
- **Spracherkennung** Auch eine Spracherkennung ist bereits auf den meisten digitalen Assistenten verfügbar und wird sogar zur Steuerung der häuslichen Elektronik verwendet.
- **Anomalie Erkennung** Digitale Geldtransaktionen werden heute von Algorithmen überwacht, die Abweichungen im Zahlungsverhalten beobachten. Wird eine ungewöhnlich hohe Summe überwiesen oder abgehoben, kann so informiert und auch interveniert werden.

All diese Verfahren nutzen eine große Menge an Trainingsdaten, um so ein Modell zu generieren, welches eine Klassifizierung weiterer Daten ermöglicht. Beispielsweise müssen bei einem System zur Spam-Filterung sowohl „normale“ als auch Spam E-Mails verwendet werden, eine Spracherkennung benötigt digitale Aufnahmen von Wörtern und Sätzen zum Lernen, etc. Nach dem Aufbau des Modells findet dann durch das System die Klassifizierung von Test- bzw. realen Daten statt. Es wird beispielsweise bei OCR entschieden, welches digitale Pendant zum vorliegenden Zeichen gehört oder bei der Spam-Filterung wie hoch die Wahrscheinlichkeit ist, dass es sich bei einer Mail um Spam handelt. Diese Trainings- und Testphase sind typisch für maschinelle Lernmethoden. Allgemein eignet sich solch ein Ansatz:

- Um Beziehungen und Muster in den Daten zu entdecken, die nicht offensichtlich sind. Genau mit dieser Fragestellung beschäftigt sich die Disziplin des *Data Mining*. Hierfür werden u.a. maschinelle Lernalgorithmen genutzt.
- Wenn kein klassischer Algorithmus für die Problemstellung entworfen werden kann oder das Programm zu komplex ist, als dass es von Menschen kodiert und gewartet werden könnte.
- Um neue Informationen einzubeziehen. Das Modell basiert auf den Daten, daher kann das System sich theoretisch durch neue Daten verändern und so der Situation anpassen.

2.2.1 Lernverfahren

Je nach Fragestellung haben sich unterschiedliche Methoden entwickelt, um den Lernprozess in einem System abzubilden. Im Wesentlichen werden drei Arten des maschinellen Lernens unterschieden:

- **Überwachtes Lernen (supervised learning)** Ein überwachtes Lernverfahren soll eine Funktion f lernen, die Eingaben (x) ihren Ausgaben (y) zuordnet, sodass gilt: $y = f(x)$. Diese Funktion wird anhand von Trainingsdaten gelernt, die demzufolge aus Paaren von Eingaben und ihren dazugehörigen Ausgaben bestehen.
- **Unüberwachtes Lernen (unsupervised learning)** Ziel unüberwachter Lernalgorithmen ist es, großen Mengen von nicht kategorisierten Daten zu gruppieren oder zu komprimieren. Dadurch können Beziehungen in den Daten entdeckt bzw. kompaktere Darstellungen erreicht werden.
- **Verstärkendes Lernen (reinforcement learning)** Beim verstärkenden Lernen hat ein Agent die Aufgabe ein vorgegebenes Ziel zu erreichen, indem er mit seiner Umwelt agiert. Die Umwelt ist dabei eine Menge von Zuständen zwischen denen der Agent durch eine Aktion wechselt. Dabei hat jede Aktion eine Belohnung oder Bestrafung zufolge. Der Agent optimiert dann sein Verhalten, um die erhaltenen Belohnungen zu maximieren.

Beim überwachten Lernen ist es notwendig, dass sowohl Ursachen (Eingaben) als auch Effekte (Ausgaben) gemessen wurden. Hier sollen dann durch ein trainiertes Modell eine Vorhersage der Ausgabe abhängig von der Eingabe erfolgen. Beim unüberwachten Lernen hingegen sind die Eingaben latente Variablen. Das heißt, sie sind nicht direkt gemessen worden, sondern durch mathematische Verfahren von Observationen abgeleitet. Dadurch ist ein exploratives Vorgehen möglich. Es können Beziehungen in den Daten entdeckt und Gruppierungen bzw. Klassifizierungen durchgeführt werden.

2.2.2 Clustering und Dimensionality Reduction

Die unüberwachten Lernverfahren lassen sich in zwei Kategorien einteilen: Zum einen handelt es sich um Methoden zum Clustering, also dem Gruppieren von Daten, und zum anderen handelt es sich um sogenannte assoziative Verfahren, um eine Kompression der Daten zu erreichen. Beide werden im folgenden vorgestellt.

Clustering-Verfahren quantisieren die Daten in Gruppen. Eine Gruppe steht hier für ein semantisches Merkmal und vertritt eine Menge von konkreten Daten. Unter Clustering-Verfahren fallen Algorithmen wie k-means, hierarchical clustering oder etwa das Gaussian Mixture Model.
Ein Clustering-Algorithmus hat in dem Kontext dieser Arbeit das Ziel, eine großen Menge Feature-Deskriptoren auf die Wesentlichen zu reduzieren. Durch diese Quantisierung in n Gruppen müssen, bei einer Bewertung eines neuen Deskriptors, nur n Deskriptoren betrachtet werden, statt jedes Deskriptors der Ursprungsmenge.

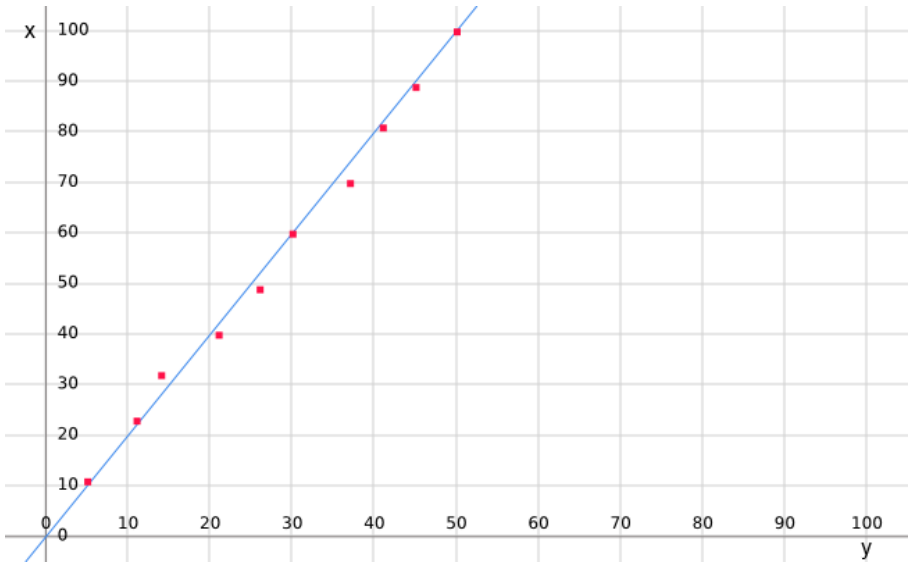

Abbildung 2.7: Beispielhafte Verteilung von Messwerten.

Verfahren zu Reduzierung der Dimensionalität nehmen an, dass es eine unterliegende Struktur gibt, welche entdeckt werden kann. Ein Vektor aus einem n-dimensionalen Raum wird auf einen m-dimensionalen Raum mit $m < n$ abgebildet. Um die Grundidee zu vermitteln, soll ein einfaches Beispiel dienen: Die roten Punkte in Abbildung 2.7 stellen Messwerte dar. Durch die eingezeichnete Linie wird deutlich, dass die Funktion $f(x) = 2y$ die Verteilung sehr gut annähert. Würden diese Werte beispielsweise über ein Netzwerk übertragen, so reicht es x zu senden. Der Empfänger kann dann y anhand der Funktion bestimmen. Dadurch könnte 50% der Daten bei der Übertragung eingespart werden und nur wenig Information geht verloren. In der Praxis bestehen zwischen den Daten aber selten lineare Zusammenhänge, was das Finden einer geeigneten Abbildung erschwert.

2.3 Autoencoder

Durch den Aufschwung des maschinellen Lernens in den letzten Jahren sind neuronale Netze stark in den Fokus der Industrie und Wissenschaft gerückt. Solche künstlichen neuronalen Netze werden genutzt, um aus Beispielen Muster zu lernen und dieses Wissen zu transferieren.
Ein spezielles neuronales Netzwerk zum unbeaufsichtigten Lernen ist der Autoencoder. Diese Art von Netzwerk lernt selbstständig eine komprimierte Darstellung der Eingabe. Als erstes wird im folgenden Abschnitt eine Übersicht über das Themegebiet der

neuronalen Netze gegeben, um darauf aufbauend den Aufbau und die Funktionsweise eines Autoencoders zu erläutern. Im Anschluss werden zwei Erweiterungen des Autoencoders vorgestellt: Der Stacked Autoencoder und der Denoising Autoencoder. Ersterer wird verwendet, um effektiv tiefe Netzwerke zu konstruieren, letzterer ermöglicht eine korrekte Rekonstruktion aus verzerrten Daten.

2.3.1 Neuronale Netze

Neuronale Netze sind dem Aufbau und der Funktionsweise der Neuronen des menschlichen Gehirns nachempfunden. Es wird eine Menge von künstlichen Neuronen genutzt, um eine Lösung für ein Problem zu gestalten. Erste theoretische Überlegungen tauchten bereits in den 40er Jahren auf. Durch die wachsende Rechenleistung und neue Forschungsgebiete wie *Machine Learning* finden neuronale Netze seit Mitte der 80er Jahre vermehrt praktische Anwendung und akademische Zuwendung [7]. Dadurch, dass neuronale Netze von Natur aus hoch parallel arbeiten können, eignen sie sich vor allem für parallele Architekturen und die Verarbeitung großer Datenmengen.

Modell

Ein neuronales Netz besteht aus einer Menge Neuronen, die in Schichten im Netzwerk angeordnet sind. Jedes Neuron besitzt einen Aktivierungszustand und einen Schwellwert. Von diesen hängt ab, ob ein Signal weitergeleitet wird. Neuronen benachbarter Schichten sind, meist vollständig, durch gewichtete Kanten miteinander verbunden. Diese Beziehung wird in einer Gewichtsmatrix ausgedrückt. Eine Kante zwischen zwei Neuronen die nicht verbunden sind, hat dann den Wert 0 als Gewicht, um die Abwesenheit auszudrücken. Das Netz verarbeitet ein Signal, welches hier als Vektor $x \in [0,1]^n$ aus $\mathbb{R}^n$ dargestellt wird. Die erste Schicht des Netzwerks, der *Input Layer*, leitet das Signal nur an die nächste Schicht weiter und besitzt daher n Neuronen. Die letzte Schicht, der *Ouput Layer*, dient zur Ausgabe des Ergebnisvektors $z \in [0,1]^m$, wobei m die Anzahl der Elemente des Vektors angibt. Zwischen diesen beiden Schichten können sich beliebig viele *Hidden Layer* befinden. Die *Hidden Layer* bilden somit den Kern des Netzes, deren Kantengewichtungen, Kanten oder Neuronen während eines Lernprozess angepasst werden können.
In Abbildung 2.8 ist ein Netz mit drei Schichten dargestellt. Der *Input Layer* nimmt einen Vektor mit vier Komponenten als Eingabe entgegen. Die Werte des Vektors werden an jedes der fünf Neuron im *Hidden Layer* weitergeleitet, was durch die Kanten symbolisiert wird. Jede Kante besitzt dabei ein Gewicht, dass aus Gründen der Übersicht nicht aufgeführt ist. Schließlich werden die Ausgaben des *Hidden Layers* an das einzige Neuron des *Output Layers* geschickt und von diesem als Ergebnis z ausgegeben.

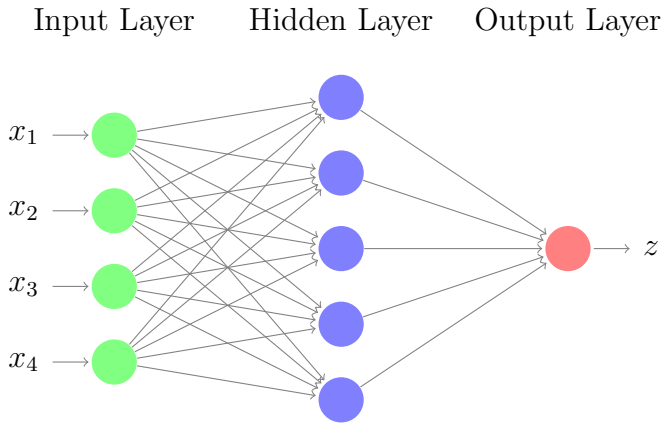

Abbildung 2.8: Beispiel eines dreischichtigen neuronalen Netzes.

Die Verarbeitung eines Signals in einem Neuron ist in Abbildung 2.9 schematisch dargestellt und lässt sich in drei Schritte untergliedern:

- **Propagierungsfunktion** Aus der Eingabe aller verbundenen Neuronen wird die Netzeingabe net_{in} berechnet. Meist wird hier die gewichtete Summe zwischen Eingabe und Gewicht verwendet:

  $$net_{in} = \sum_{i=1} w_i x_i + b$$

  Die Variable b ist hier der sogenannter Bias-Vektor, der dazu verwendet werden kann Einfluss auf die Aktivierung zu nehmen. Dieser wird als zusätzliches Neuron pro Schicht modelliert.

- **Aktivierungsfunktion** Es wird der neue Aktivierungszustand des Neurons aus dem alten Zustand und der Netzeingabe net_{in} durch die Aktivierungsfunktion a berechnet. Häufig wird hier die ReLU (Rectified Linear Unit) a_{relu} bzw. die Sigmoidfunktion a_s verwendet:

  $$a_{relu}(x) = \max(0, x) \qquad a_s(x) = \frac{1}{1 + e^{-x}}$$

- **Ausgabefunktion** Wird ein Neuron aktiviert, so wird das resultierende Signal durch die Ausgabefunktion net_{out} berechnet und an alle Neuronen in der folgenden Schicht weitergeleitet. Oft wird hier die Identitätsfunktion verwendet.

Training und Lernprozess

Wurde ein neuronales Netz konstruiert, folgt darauf die Trainingsphase. Durch das Training ist es möglich, dass ein Netzwerk Neuronen oder Verbindungen hinzufügt bzw.

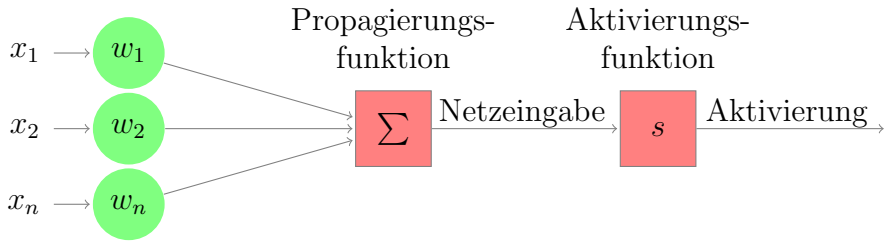

Abbildung 2.9: Verarbeitung eines Signales in einem Neuron (ohne Ausgabefunktion).

entfernt, den Schwellwert für die Aktivierung von Neuronen verändert oder die Gewichte zwischen Neuronen anpasst. Pro Trainingselement wird die Fehlerquote F zwischen dem angestrebten Ergebnis z' und der Ausgabe des Netzes z berechnet:

$$F(z, z') = \sum_{i=0}^{m} (z'_i - z_i)^2$$

Durch eine Lernregel werden abhängig von der Fehlerquote beispielsweise die Gewichte der Kanten angepasst. Für Netze ohne *Hidden Layer* eignet sich hier die Hebbsche oder Delta Lernregel. Für Netze mit mindestens einem *Hidden Layer*, hat sich das *Backpropagation* Verfahren etabliert. *Backpropagation* minimiert den Gradientenabstieg auf der Fehleroberfläche die F aufspannt. Der Algorithmus geht in drei Schritten vor:

1. **Forward Pass** Die Gewichte des Netzwerks werden initialisiert und eine Eingabe durch das Netz propagiert. Als Resultat liegt die Ausgabe vor.
2. **Fehlerberechnung** Die Fehler des *Output Layers* werden, durch einen Vergleich mit den erwarteten Werten, berechnet. Falls die Fehler unterhalb einer vorgegebenen Grenze liegen, ist das Training beendet. Andernfalls wird fortgefahren.
3. **Backward Pass** In diesem Schritt wird die Fehlerquote rückwärts durch das Netz propagiert. Die Gewichte an den Verbindungen zwischen Neuronen werden in Abhängigkeit ihres Einflusses auf den Fehler Schicht für Schicht aktualisiert.

Hyperparameter

Bisher wurde das Netzwerk und die Parameter, die es optimiert, die Gewichte w und der Bias-Vektor b, betrachtet. Darüber hinaus gibt es eine Reihe von Parametern, die Hyperparameter, die vor Anwendung des Modells festgelegt werden müssen. Die wesentlichen Hyperparameter für neuronale Netze sollen hier vorgestellt und ihr Einfluss auf das Modell erläutert werden. Die aufgeführten Standardwerte wurden bei einer Vielzahl von Modellen beobachtet, gelten aber nicht uneingeschränkt für jeden Anwendungsfall [8].

Initiale Lernrate Bei gradientenbasierten Verfahren wird in jeder Iteration der Fehler zurückpropagiert. Dies wird in den meisten Fällen zu einer zu schnellen Anpassung des Netzes führen und es „vergisst“ schnell, was es bereits gelernt hat. Aus diesem Grund werden die berechneten Fehler mit der Lernrate (*learning rate*), einem kleinen Wert, multipliziert. Praktisch wird für die Lernrate ein Wert kleiner 1 und größer 10^{-6} verwendet.

Anzahl der Trainingsiterationen Eine Iteration entspricht einem *forward* und *backward pass* einer kleinen Teilmenge (ein *batch*) der gesamten Trainingsmenge. Eine zu große Anzahl an Iterationen führt zu einer Überanpassung des Netzes an die Trainingsdaten, daher sollte gestoppt werden, wenn sich andere Metriken nicht mehr verbessern.

Anzahl der Exemplare pro Trainingsiteration (*batch size)* Je größer die *batch size*, desto mehr Trainingsexemplare werden pro Iteration verarbeitet. Durch einen größeren Wert kann die Berechnung beschleunigt werden, allerdings ist auch mehr Speicher erforderlich. In der Praxis liegt der Wert für diesen Parameter meist zwischen 16 und 128 (Zweierpotenzen).

2.3.2 Aufbau und Funktionsweise

Ein Autoencoder (AE) ist ein spezielles neuronales Netzwerk, das eine komprimierte Kodierung der Eingabe lernt. Diese Art von Netzwerk versucht, die Daten zu rekonstruieren und kann daher unbeaufsichtigt lernen: Die rekonstruierten Daten können anhand einer Distanzmetrik mit den Originaldaten verglichen werden. Anschließend kann die Größe des Fehlers berechnet werden und durch *Backpropagation* die Gewichtmatrix aktualisiert werden [9]. Um die Originaldaten als Ergebnis erhalten zu können, muss die Anzahl der Neuronen des *Input Layers* der Anzahl der Neuronen im *Output Layer* entsprechen. Die Anzahl der Neuronen im *Hidden Layer* ist geringer, um die komprimierte Darstellung des Features zu erreichen. Werden mehrere *Hidden Layer* verwendet, so nimmt die Neuronenanzahl von Schicht zu Schicht ab um die Anzahl der Komponenten weiter zu verringern. Dieser Vorgang ist die Enkodierung und liefert die gewünschte komprimierte Abbildung. Die Dekodierung ist umgekehrt aufgebaut, um das Original aus der komprimierten Repräsentation Schicht für Schicht zu rekonstruieren. Wie gut die Dekodierung gelungen ist, lässt sich dann anhand eines Vergleichs der Distanz des Original und der Rekonstruktion bewerten.
Formal wird ein Eingabevektor $x \in [0,1]^n$ auf einen Vektor $y \in [0,1]^p$, mit $p < n$, abgebildet:

$$y = encode_{W,b}(x) = s(Wx + b)$$

W ist die Gewichtsmatrix der Größe $n \times p$ und b der Bias-Vektor. Diese Parameter werden durch den Autoencoder optimiert. Die Darstellung y wird in diesem Kontext auch als *code* bzw. latente Variablen bezeichnet. Die Rekonstruktion erfolgt durch die Dekodierungsfunktion. Der Vektor $z \in [0,1]^n$ berechnet sich durch:

$$z = decode_{W',b'}(y) = s(W'y + b')$$

In Abbildung 2.10 ist ein Autoencoder abgebildet, der als Eingabe einen Vektor $x \in [0,1]^4$ entgegen nimmt. Dieser wird auf den Vektor y mit drei Komponenten abbildet, da der *Hidden Layer* drei Neuronen enthält. Die Rekonstruktion z aus y erfolgt dann durch die Berechnung der Dekodierungsfunktion.

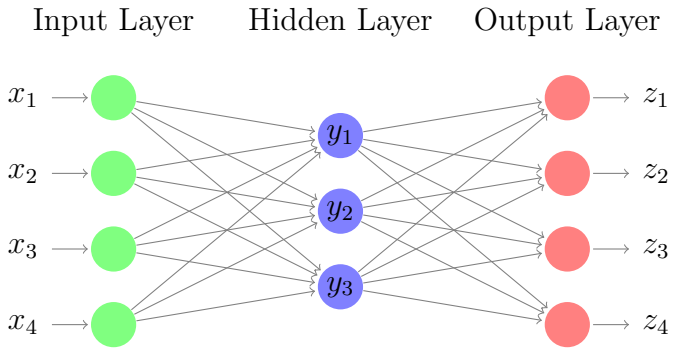

Abbildung 2.10: Beispiel eines Autoencoders mit einem *Hidden Layer*.

2.3.3 Stacked Denoising Autoencoder

Von Hinton and Salakhutdinov wurde 2006 [10] das Konzept des Stacked Autoencoders eingeführt, um einige Probleme mit herkömmlichen Autoencoder zu überwinden. Bei Netzwerken mit mehr als einem *Hidden Layer* erzielt die Methode des steilsten Abstiegs, aufgrund der zunehmenden Verzerrung der Gradienten, bei der Rückpropagierung keine guten Ergebnisse mehr. In vielen Ansätzen wurde auch eine zufällige Initialisierung der Gewichte gewählt. Hier besteht die Gefahr, dass der Algorithmus in einem lokalen Optimum verbleibt. Wenn die anfänglichen Gewichte hingegen bereits nah an einer guten Lösung liegen, sinkt die Wahrscheinlichkeit eines lokalen Optimums. Aus diesem Grund wurde das Pretraining für Autoencoder mit mehr als einer Schicht vorgeschlagen. In diesem Training wird jedes Paar aneinanderliegender Schichten als ein Autoencoder aufgefasst und einzeln trainiert. Das Pretraining besteht aus drei Schritten, die wiederholt werden, bis alle Autoencoder trainiert sind.

1. Es wird der aktuelle Autoencoder trainiert. Zu Beginn besteht dieser aus dem *Input* und folgendem *Hidden Layer*.

2. Nun wird der Decoder des trainierten Autoencoders entfernt und ein neuer Autoencoder erzeugt. Dieser besitzt den *Hidden Layer* des trainierten Autoencoders als *Input Layer*.

3. Das Training wird mit dem neuen Autoencoder fortgeführt.

Ein Denoising Autoencoder [11] dient dazu, eine korrumpierte Eingabe zu korrigieren. Die Korruption ist hier als Rauschen bzw. Verzerrung (*noise*) aufzufassen. In vielen Arten von Features, z.B. Bildern oder Audiomaterial, sind Verzerrungen bereits in den Daten vorhanden, beeinflussen die Semantik des Ganzen aber kaum. Daher soll der Autoencoder dies bereits berücksichtigen, indem er nicht direkt mit der Eingabe x arbeitet. Stattdessen wird x auf die korrumpierte Eingabe $\widetilde{x}$ durch ein stochastisches Verfahren abgebildet. In der Enkodierungsfunktion wird dann $\widetilde{x}$ statt x verwendet:

$$encode_{W,b}(\widetilde{x}) = s(W\widetilde{x} + b)$$

In der Praxis hat sich zur Korruption der Eingabe die *masking corruption* Technik bewährt: Hierbei werden 20% bis 50% der Neuronen des *Input Layers* zufällige ausgewählt und werden „genullt“. Auf diese Weise wird vermieden, dass der Autoencoder nur von bestimmten Teilmengen an Neuronen abhängt [8].

2.4 Bag of Visual Words

Das Bag of Visual Words lehnt sich an das Bag of Words Modell aus dem Bereich Information Retrival an. Daher soll zunächst die Funktionsweise des Bag of Word Modells erläutert werden, um darauf aufbauend den Bag of Visual Words einzuführen.
Der Bag of Words wird zur Klassifizierung von Dokumenten genutzt. Dieses Verfahren zählt das Auftreten jedes Wortes in einem Dokument. Diese Anzahl wird durch die Anzahl aller Wörter im Vokabular dividiert, um die relative Häufigkeit eines Wortes zu ermitteln. Das Vokabular wird in diesem Kontext *Codebook* genannt, die Wörter werden auch als *Codewords* bezeichnet.
Dieses Modell wurde von Faheema und Rakshit [12] adaptiert, um für die digitale Bildverarbeitung von Nutzen zu sein. Der Bag of Visual Words lernt anhand der Features von Trainingsbildern ein visuelles Vokabular, dass zur Klassifizierung von Bildern dient. Die Features können aber nicht direkt statt der Worthäufigkeit verwendet werden: Ein Wort ist ein diskreter Wert, der direkt verglichen werden kann, ein Feature hingegen ist ein Vektor, der aus vielen Gleitkommazahlen besteht. Dies heißt, dass es keinen Sinn ergibt, Features direkt miteinander zu vergleichen: Die Komponenten zweier Vektoren sind unter Umständen ähnlich, aber unter fast keine Umständen gleich. Um also eine Darstellung zu erhalten, die als Basis für einen Vergleich dienen kann, ist es notwendig, die Vektoren zu quantisieren. Die quantisierten Vektoren entsprechen dann den *Codewords* und werden in diesem Kontext auch *Visual Words* genannt.

Zunächst wird die Funktionsweise des Bag of Visual Words im folgenden Abschnitt näher betrachtet. Dem schließt eine Betrachtung des Kernstücks des Algorithms an: Das Clustering der Features. Hier wird ein k-means Algorithmus, Llyods heuristische Variante, verwendet. Es wird eine gängige sequentielle Implementierung angeführt, auf deren Basis dann die Parallelisierbarkeit durch Grafikkarten untersucht wird.
Bei der Einordnung eines Bildes wird ein Histogramm der *Visual Words* generiert, daher wird im Anschluss ein sequentieller Histogramm Algorithmus vorgestellt, der auf Parallelisierbarkeit geprüft wird.

2.4.1 Funktionsweise

Der Bag of Visual Words besteht aus einer Trainings- und Testphase, wie in Abbildung 2.11 dargestellt. Die Extraktion der Features ist beiden Phasen vorgelagert.
Zu Beginn erfolgt die Trainingsphase, das Clustering der extrahierten Features. Das so erzeugte *Codebook* ist das Modell, gegen das anschließend getestet werden kann. Die Idee ist, dass ähnliche Feature-Vektoren nah beieinander im Raum liegen und somit in die gleiche semantische Kategorie gehören. Durch einen Clustering-Algorithmus wie k-means kann die Größe des *Codebooks* bestimmt werden. Wird für k eine große Zahl gewählt, wird ein Vokabular von Exemplaren aufgebaut, ein kleines k hingegen erkennt eher Kategorien. Die Schwerpunkte der Cluster vertreten dann eine Menge von ähnlichen Features und bilden das *Codebook* bzw. Modell.
Der Testprozess erzeugt nun, auf Basis des *Codebooks*, die *Visual Words* von Bildern. Hierfür werden die Features eines Bildes extrahiert und ein Histogrammalgorithmus ermittelt die Verteilung der Visual Words: Für jedes Feature wird das ähnlichste *Visual Word* des *Codebooks* bestimmt und die entsprechende Klasse inkrementiert. Wird dieser Prozess auf zwei Bilder angewendet, so können die resultierenden Histogramme miteinander verglichen werden (z.B. mit dem *MSE (mean squared error)* als Metrik).

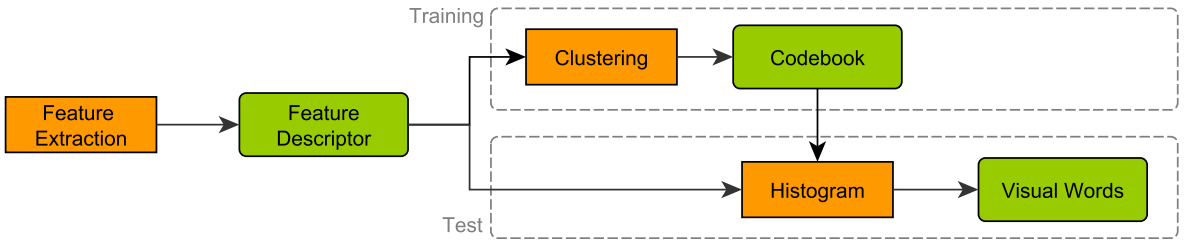

Abbildung 2.11: Training- und Testprozess des Bag of Visual Word Modells.

2.4.2 K-means Clustering

Unter k-means Clustering werden Algorithmen zusammengefasst, die eine Menge Vektoren durch Zuweisung in k vorgegebene Gruppen einteilen, die sogenannten Cluster. Aus den Vektoren werden initial k Stück ausgewählt, die als anfängliche Schwerpunkte der Cluster dienen. Ein k-means-Algorithmus ordnet nun iterativ jeden Vektor dem Cluster zu, der ihm am nächsten ist. Formal werden n Vektoren aus dem Raum $\mathbb{R}^d$ in k Cluster eingeteilt. D ist hierbei eine Distanzmetrik im euklidischen Raum. Oft wird hier der quadrierte Fehler durch k-means minimiert, also beispielsweise $D = \|x_i - c_j\|^2$. Jeder Vektor x_i wird dem Cluster c_j zugeordnet, dessen Distanz zum Vektor am geringsten ist:

$$C(x_i) = arg \min_{j=0,...,k} D(x_i, c_j) \quad i = 0, ..., n$$

Soll ein globales Optimum gefunden werden, so ist k-means NP-schwer. Praktische Implementierungen approximieren daher meist die Schwerpunkte der Cluster, wie beispielsweise der Algorithmus von Llyod. Llyods Algorithmus beginnt mit einer Initialisierung der Schwerpunkte. Schritt zwei und drei werden dann solange wiederholt, bis der Algorithmus konvergiert, also die Vektor-Cluster-Zuordnung sich nicht mehr bzw. nur noch gering ändert oder eine maximale Anzahl an Iterationen erreicht wurde:

1. **Initialisierung** Es werden zunächst k Vektoren durch ein Verfahren (z.B. zufällig oder *farthest neighbour*) als Schwerpunkte der Cluster ausgewählt.
2. **Zuordnung**: Von den verbleibenden Vektoren wird nun mit jedem Clusterschwerpunkt die neue Summe der Varianzen bei Aufnahme des Vektors berechnet. Es wird der Vektor dem Cluster zugeordnet, dessen Varianz sich am geringsten bei Aufnahme des Vektors erhöht.
3. **Vektoren zuweisen**: Die Zentren der Cluster werden neu berechnet, um die neue Zuordnung der Vektoren zu Clustern in Folgeberechnungen miteinzubeziehen.

Zur geometrischen Veranschaulichung sind in Abbildung 2.12 Punkte im zweidimensionalen Raum gegeben. Die Verteilung der Punkte ist in diesem Beispiel idealisiert, um bei einem Clustering mit einem $k = 3$, die Cluster in der rechten Grafik zu erhalten. Dies illustriert auch, dass abhängig von den Daten nicht immer sinvolle Cluster gebildet werden können. Die Wahl der initialen Clusterschwerpunkte kann das Ergebnis ebenfalls stark beeinflussen, daher sollte diese nicht immer zufällig gewählt werden. Beispielsweise wählt die *farthest neighbour* Methode als initial Zentren die Vektoren, welche am weitesten voneinander entfernt sind [13].

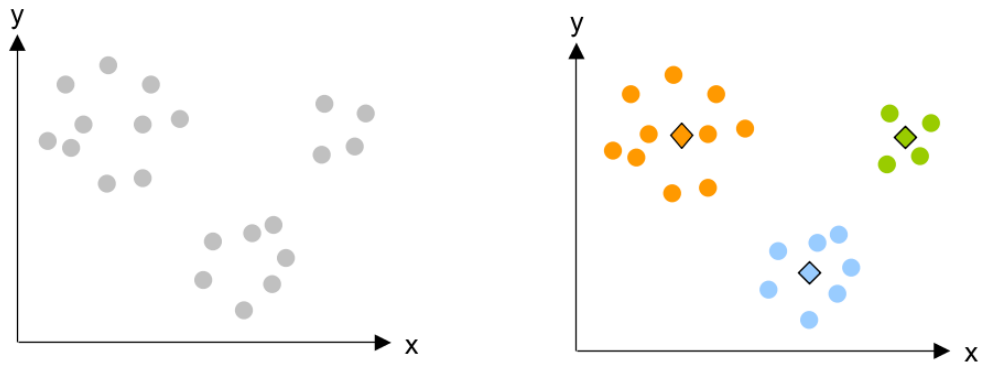

Abbildung 2.12: K-means Clustering im zweidimensionalen Raum mit $k = 3$.

2.4.3 Lloyds Algorithmus

Zunächst soll näher auf die sequentielle Arbeitsweise von Lloyds Algorithmus eingegangen werden, um anschließend eine mögliche Parallelisierung zu diskutieren. In dem Codelisting in Abbildung 4.4 ist der Ablauf des Algorithmus nach Zechner und Granitzer [2] in Pseudocode beschrieben. Als Parameter werden die Vektoren V, eine leere Menge Cluster C und die Anzahl der zu bildenden Cluster k erwartet. In Zeile 2 findet die Auswahl der initialen Schwerpunkte der Cluster c_j statt. Innerhalb der Schleife in Zeile 4 werden zunächst alle k Cluster C_j als leere Menge definiert. Anschließend wird für jeden Vektor v_i aus V der Index j des nächsten Clusterschwerpunktes ermittelt. Der Vektor wird dann in den entsprechenden Cluster C_j aufgenommen. Abschließend wird die Aktualisierung der Schwerpunkte aller Cluster in der Schleife in Zeile 8 und 9 durchgeführt. Hierfür wird pro Cluster die Summe über alle enthaltenen Vektoren gebildet und anschließend normalisiert.
Die Initialisierungsphase muss für die Parallelisierung nicht beachtet werden: Sie nimmt nur wenig Zeit in Anspruch und wird einmalig zu Beginn ausgeführt. Die anderen beiden Schritte des Algorithmus bergen mehr Potential: In Zeile 5 bis 7 wird die Varianz jedes Cluster-Vektor Paares berechnet. Da die Berechnung der Distanz eines Paares unabhängig von der eines anderen ist, kann die Berechnung aller Distanzen parallel erfolgen. Nachdem für einen Durchgang die Veränderung der Mitgliedschaft von Vektoren zu Clustern berechnet wurde, müssen die Cluster-Schwerpunkte aktualisiert werden. Auch die Berechnung der neuen Schwerpunkte der Cluster kann unabhängig voneinander erfolgen: Die Vektoren, aus denen der Mittelwert berechnet wird, sind genau einem Cluster zugeordnet.

```
1  kmeans_lloyd (V, C, k)
2    c_j = rand(v_i) ∈ V, j = 1, ..., k, c_j ≠ c_i ∀i ≠ j
3
4    until convergence
5      C_j = ∅, j = 1, ..., k
6      for each v_i ∈ V
7        j = arg min D(c_j, v_i)
8        C_j = C_j ∪ v_i
9      for each C_j ∈ C
10       c_j = 1/‖C_j‖ Σ_{v_i ∈ C_j} v_i
```

Abbildung 2.13: Ablauf von Llyods Algorithmus in Pseudocode [2].

2.4.4 Histogramme

Ein sequentielles Histogramm kann als Programm in einer Schleife über die Daten ausgedrückt werden: Für jedes Element wird der Index der Klasse des Histogramms berechnet und um eins inkrementiert. Zur Normalisierung des Histogramms ist es anschließend notwendig, jede Klasse des Histogramms durch die Gesamtanzahl der Werte zu dividieren. Da es sich bei der Anzahl der Klassen jedoch um eine kleine Zahl, im Vergleich zur Anzahl der Elemente in den Daten, handelt, ist dieser Aufwand vernachlässigbar. Um das Histogramm der *Visual Words* eines Bildes zu erzeugen, muss für jedes extrahierte Feature das nächste *Visual Word* bestimmt werden. Dies entspricht in Abbildung 4.5 der doppelten Schleife über die Features F und Cluster C in Zeile 2 und 3. Der Index des nächsten Clusters wird dann in Zeile 4 durch $\arg\min D$ berechnet .In der folgenden Zeile wird das Histogramm H an der entsprechenden Stelle i inkrementiert. Abschließend erfolgt in der Schleife in Zeile 6 und 7 die Normalisierung des Histogramms.

```
1  histogram (P, C, H)
2    for each p ∈ P
3      for each c ∈ C
4        i = arg min D(c, p)
5      H_i = H_i + 1
6    for each h ∈ H
7      h = h/‖H‖
```

Abbildung 2.14: Histogramm-Algorithmus in Pseudocode

2.5 CUDA

Der Begriff CUDA ist ein Akronym für *Compute Unified Device Architecture*. Dahinter verbirgt sich eine Architektur von Nvidia für parallele Berechnungen durch Grafikkarten. Durch den Einsatz von Grafikkarten können Berechnungen gerade bei großen Datenmengen stark beschleunigt werden, da diese auf eine hoch-parallele Verarbeitung ausgelegt sind. Die meisten GPU Computing Ansätze, und auch CUDA, verwenden als Ausführungsmodell das Single Program Multiple Data (SPMD) Modell. Im Unterschied zum Multiple Instruction Multiple Data (MIMD) Modell, dass in CPUs verwendet wird, wenden alle Prozessoren das gleiche Programm auf unterschiedliche Daten an. Durch diese Art der Parallelisierung konnten in den letzten Jahren enorme Steigerungen der Gleitkommaoperationen pro Sekunde (flops) und der Speicherbandbreite bei Berechnungen erzielt werden.
Der Abschnitt gibt im Wesentlichen ein Einblick in die im CUDA C Programming Guide [3] detailliert beschriebenen Dokumentation gegeben, sodass ein Leser Teile eines CUDA Programms nachvollziehen kann. Zunächst wird näher auf das Ausführungsmodell und dessen Umsetzung eingegangen. CUDA unterscheidet zwischen verschiedenen Speicherbereichen auf der Grafikkarte. Da die Verwendung der unterschiedlichen Speicherbereiche wesentliche Unterschiede in der Implementierung eines Algorithmus und dessen Laufzeit zur Folge hat, werden diese im Abschnitt 2.5.2 Speicherverwaltung behandelt. Um den praktische Einsatz von CUDA zu illustrieren, wird abschließend ein Programm zur Addition zweier Vektoren auf der Grafikkarte vorgestellt.

2.5.1 Ausführungsmodell

Ein Programm, das auf einer Nvidia Grafikkarte ausgeführt werden soll, muss in der Sprache CUDA C geschrieben sein. Aus der Sicht eines Programmierers handelt es sich hierbei um eine Erweiterung von C um Primitive und Funktionen für Berechnungen auf der Grafikkarte. Zum Übersetzen und Linken des Codes dient der *nvcc* Compiler von Nvidia. Dieser unterscheidet zwischen Code der auf dem *host*, der CPU, und dem *device*, der GPU, ausgeführt wird. Das Kompilieren von *host* Code erfolgt durch den auf dem *host* installierten C Compiler. Der *device* Code wird durch *nvcc* zu *PTX*[5] bzw. *cubin binary code*[6] übersetzt. Nvidia hat das SPMD Modell durch *kernels* umgesetzt. Ein *kernel* ist ein Programm, das parallel auf verschiedene Daten der GPU angewendet wird. Bevor ein *kernel* aufgerufen werden kann, muss der notwendige Speicher auf der Grafikkarte für die Daten und das Ergebnis allokiert werden. Anschließend werden die Daten von *host* zu *device* kopiert. Nach Durchführung der Berechnung kann dann das Ergebnis zurück

[5] PTX steht für Parallel Thread eXecution und ist eine „zwischen“ Assemblersprache. Der generierte Code ist noch nicht ganz optimiert, da PTX als Basis mehrerer GPU Generationen dient.
[6] cubin Dateien sind gerätespezifische Binärdateien

zum *host* kopiert werden. Die Datentransfers weisen eine nicht unbeachtliche Latenz auf. Folglich sollte das Kopieren von Daten nur selten erfolgen.
Die Daten liegen in der Regel als Vektor oder Matrix vor. Der Zugriff auf verschiedene Elemente durch unterschiedliche *kernel* erfolgt dann durch eine Indexberechnung. Diese wird anhand des Beispiels der Vektoraddition näher erläutert.
CUDA erfordert, dass ein Programmierer die Größe bzw. Anzahl von Grids und Blocks festlegt. Ein Block ist eine logische Einheit und entspricht einem Multiprozessor der Grafikkarte. Durch diese Strukturierung können die Blöcke parallel auf der Hardware ausgeführt werden. Das Grid enthält die Blöcke und kann ein, zwei oder dreidimensional sein. Beim Aufruf des *kernels* muss die Anzahl der Threads pro Block und die Anzahl der Blocks in einem Grid angegeben werden. In Abbildung 2.15 ist ein zweidimensionales Grid mit sechs zweidimensionalen Blöcken á 12 Threads schematisch dargestellt. Ein Block kann auf aktuellen Grafikkarten bis zu 1024 Threads beinhalten.

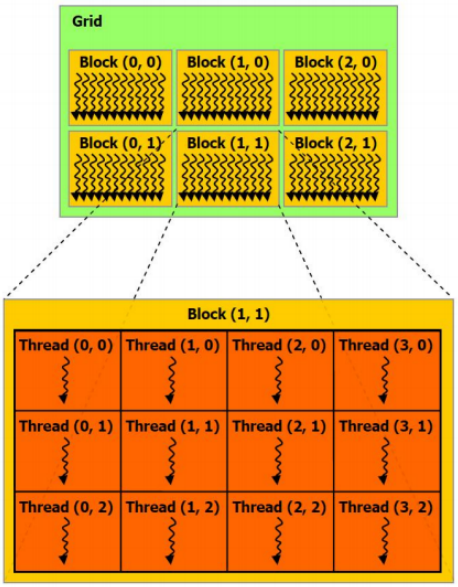

Abbildung 2.15: Organisierung von Threads in Blocks in Grids [3]

2.5.2 Speicherverwaltung

Dadurch, dass die Daten vom *host* zum *device* kopiert werden müssen und vice versa, unterscheidet CUDA zwischen *host memory* und *device memory*. Darüber hinaus ist der *device memory* bereits auf der Grafikkarte in verschiedene Bereiche organisiert, wie in Abbildung 2.16 dargestellt. Jeder Multiprozessor hat Zugriff auf den *global memory* sowie einen eigenen lokalen *shared memory*, *constant* und *texture cache*. Lokal erstellte Variablen werden als *register* bezeichnet. Auf diese ist der Zugriff mitunter am

schnellsten, jedoch sind sie lokal pro Thread. Die Daten liegen als Parameter zunächst im *global memory* vor. Im *constant cache* liegen alle durch das CUDA Schlüsselwort *__constant__* deklarierte Werte. Zugriffe auf den *global memory* sind am langsamsten, da der Speicherbereich zwischen allen Multiprozessoren geteilt wird und Zugriffe mitunter synchronisiert werden müssen. Durch Verwendung des *shared memory* und *texture cache* können Speicherzugriffe beschleunigt werden, jedoch können Daten aus diesen Bereichen nicht zwischen Blöcken geteilt werden.

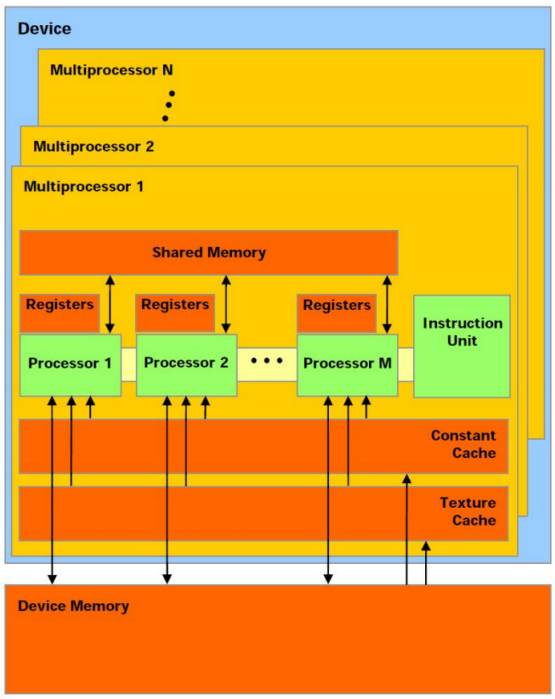

Abbildung 2.16: Organisierung der verschiedenen Speichertypen [3]

Shared Memory

Gerade bei vielen parallelen Lese- / Schreibzugriffen kann hier eine hohe Latenz auftreten, wenn viele Threads auf die gleichen Adressen zugreifen. Aus diesem Grund empfiehlt es sich, die benötigten Daten von dem *global* in den *shared memory* zu kopieren. Der *shared memory* wird von allen Threads in einem Block geteilt. Beim Kopieren der Daten muss ein Index berechnet werden, um die korrekten Daten dem jeweiligen Block zuzuordnen. Soll beispielsweise jeder *kernel* auf dem Element eines Vektors arbeiten, setzt sich der globale Index aus der Position im Block und im Thread zusammen: $blockDim.x * blockIdx.x + threadIdx.x$. Die Variablen $blockDim$ und $blockIdx$ sind, wie $threadIdx$, durch CUDA gesetzt, bzw. durch den Programmierer (im Falle der Blockdimension). Weiterhin ist zu beachten, dass nicht beliebig viel *shared memory* allokiert

werden kann: Je nach Grafikkarten stehen pro Block zwischen 16 und 48 Kilobyte Speicher zur Verfügung. Durch den Einsatz von *shared memory* können gegenüber Algorithmen die exklusiv den *global memory* nutzen, wesentliche Steigerung in der Performance (hier: kürzere Laufzeit) erreicht werden. Allerdings ist der *shared memory* mit 16 bzw. 48 Kilobyte sehr begrenzt und erhöht die Komplexität durch notwendiges Kopieren von Daten und Indexberechnungen.

2.5.3 Vektoraddition

Am Beispiel der Vektoraddition in Abbildung 2.17 soll der Aufbau und Ablauf eines CUDA-Programms illustriert werden. Hierfür wird zunächst die Funktion *vecAdd* angelegt. In Zeile 8 - 10 wird durch *cudaMalloc* der benötigte Speicher für die Vektoren und das Ergebnis allokiert. In den beiden folgenden Zeilen wird der Inhalt der Vektoren a und b vom *host* zum *device* kopiert. In Zeile 15 wird der *kernel add* aufgerufen, der auf der Grafikkarte ausgeführt wird. Einem *kernel* muss durch Angabe in den dreifachen spitzen Klammern, die Dimension des Grids und der Blocks mitgeteilt werden. Im Beispiel werden pro Block 256 Threads verwendet (*numThreads*) und die Anzahl der notwendigen Blocks aus der Vektorgröße und der Threadanzahl berechnet. Da die Verarbeitung asynchron erfolgt, wird durch *cudaDeviceSynchronize* in der nächsten Zeile die Ausführung auf dem *host* pausiert, bis die GPU fertig ist. In Zeile 17 wird das Ergebnis vom *device* zurück zum *host* kopiert und kann ausgegeben oder weiterverarbeitet werden. Der *kernel* wird von jedem Thread ausgeführt. Jeder Thread kümmert sich um die Addition eines Elementes aus a und aus b am selben Index. Damit dieser Index global eindeutig ist, muss neben der *threadId* in x-Richtung die Grid und Blockdimension des Threads berücksichtigt werden. Falls der berechnete Index innerhalb der Vektoren liegt, wird die Summe in c geschrieben.

```
1  void vecAdd (float *a, float *b, float *c, int size) {
2    int numThreads = 256;
3    int numBlocks = (size + numThreads - 1) / numThreads;
4    float *aPtr = 0;
5    float *bPtr = 0;
6    float *cPtr = 0;
7
8    cudaMalloc((void **) aPtr, a, sizeof(float) * size);
9    cudaMalloc((void **) bPtr, b, sizeof(float) * size);
10   cudaMalloc((void **) cPtr, c, sizeof(float) * size);
11   cudaMemcpy(aPtr, a, sizeof(float) * size, cudaMemcpyHostToDevice);
12   cudaMemcpy(bPtr, b, sizeof(float) * size, cudaMemcpyHostToDevice);
13
14   add<<<numBlocks, numThreads>>>(aPtr, bPtr, cPtr, size);
15   cudaDeviceSynchronize();
16
17   cudaMemcpy(c, cPtr, mem, cudaMemcpyDeviceToHost);
18
19   cudaFree(aPtr);
20   cudaFree(bPtr);
21   cudaFree(cPtr);
22 }
23
24 __global__ void add (float *a, float *b, float *c, int size) {
25   int id = blockDim.x * blockIdx.x + threadIdx.x;
26   if (id < size) c[id] = a[id] + b[id];
27 }
```

Abbildung 2.17: Vektoraddition in CUDA C.

3 Analyse

Ziel dieser Arbeit ist es, ein Modell zu entwickeln, das es ermöglicht große Mengen von Bildern zu gruppieren. Wegen der großen Mengen an Daten, die zu verarbeiten sind, sollen *state of the art* Verfahren genutzt werden, die speziell hierauf ausgelegt sind. Der erste der Teil der Analyse befasst sich daher mit der Betrachtung von geeigneten unüberwachten Lernverfahren für diesen Zweck. Hier wird untersucht, welchen Ansatz andere gewählt haben, um das Problem zu lösen. Auf dieser Basis und anhand der Anforderungen und Annahmen wird dann ein unüberwachtes Lernverfahren ausgewählt, dass zu Gruppierung von Bildern dient und in den folgenden Kapiteln weiter ausgearbeitet und realisiert wird.
Im zweiten Teil sollen Möglichkeiten untersucht werden, aus Bildern Features zu gewinnen, welche als Eingabe für das Modell dienen. Um einen Überblick über Features in der Bildverarbeitung zu gewinnen, werden zunächst einige Detektoren und Deskriptoren für unterschiedliche Anwendungsfälle angeführt. Heute ist es kaum vorstellbar, dass ein Feature-Deskriptor jeden möglichen Anwendungsfall abdecken kann. Daher soll abschließend der Fokus dieser Arbeit festgelegt werden: Sollen beispielsweise Gesichter oder Szenen erkannt werden? Sollen Objekte erkannt werden und wenn ja, beliebige Kategorien? Anhand der gewonnenen Erkenntnisse wird dann entschieden, welche Eigenschaften der hier verwendete Feature-Deskriptor aufweisen soll.

3.1 Verwendung der GPU

Zum Aufbauen eines Modells werden mehrere Zehn- bis Hunderttausend Features verarbeitet werden. Viele der Verfahren, die der Erzeugung solcher Modellen zu Grunde liegen, wurden in den vergangenen Jahren durch die Verwendung der GPU statt der CPU beschleunigt. Bei der Betrachtung geeigneter Ansätze wird daher auch berücksichtigt, ob und wie eine Beschleunigung durch parallele Verarbeitung erzielt werden kann. Gerade bei großen Datenmengen und einer enormen Datenparallelität können Probleme durch GPUs um ein vielfaches schneller gelöst werden als durch CPUs. Da Nvidias CUDA an der Hochschule Hannover sowohl gelehrt als auch zu Forschungszwecken genutzt wird und sich CUDA auch international in Forschung und Wirtschaft etabliert hat, soll die Plattform als technische Basis dienen.

3.2 Geeignete unüberwachte Lernverfahren

Für die Gruppierung des Bildmaterials eignen sich maschinelle Lernverfahren besonders, da sie zum einen auf eine große Menge an Trainingsdaten angewiesen sind, um ein nützliches Modell zu generieren und zum anderen eine parallele Verarbeitung begünstigen. Der Einsatz unüberwachter Lernverfahren ist hier aus folgenden Gründen plausibel:

- Das Gruppieren bzw. Kategorisieren von Daten ist ein Teilgebiet der unüberwachten Lernverfahren. Hier wird von Clustering gesprochen.
- Die Feature-Vektoren umfassen oft viele Komponenten, welche zur Kodierung der Eigenschaften erforderlich sind. Unüberwachte Lernalgorithmen ermöglichen ein exploratives Vorgehen: Die Feature-Vektoren werden auf ihre wesentlichen Komponenten analysiert und so eine kompaktere Darstellung erzeugt.
- Es sollen Strukturen in den Daten entdeckt werden, die nicht a priori bekannt sind. Ein überwachter Ansatz erfordert zum Training *gelabelte* Daten, um Vorhersagen zu treffen. Da die gesuchten Strukturen aber gerade nicht bekannt sind, scheidet ein überwachter Ansatz aus.

Das Gruppieren von Vektoren, hier den Bild-Features, kann also durch einen *Clustering* Algorithmus umgesetzt werden. Außerdem scheint es sinnvoll die Features vor dem Clustering aufzubereiten: Durch eine kompaktere Darstellung der Vektoren kann der notwendige Speicher reduziert und die Berechnung beschleunigt werden. Solche Methoden zur Kompression sind sogenannte assoziative Verfahren: Hier werden Strukturen in den Daten gesucht, die nicht offensichtlich sind. Durch die Verwendung dieser unterliegenden Strukturen kann auf einen Teil der ursprünglichen Information verzichtet werden. Im nächsten Abschnitt wird betrachtet, wie sich andere Autoren dieser Fragestellung angenähert haben und welche Vorarbeit bereits geleistet wurde.

3.2.1 Verwandte Arbeiten

Methoden, um großen Mengen Daten zu gruppieren, sind aktuell gefragter denn je: Die zunehmende Vernetzung und Nutzung von Technik im privaten sowie in der Wirtschaft, führt zu großen Datenströmen. Zentrales Element in den Arbeiten andere ist meist ein k-means Algorithmus. Dieser wird hinsichtlich verschiedener Ziele, wie Ausführungszeit oder Präzision, von den Autoren optimiert. Einige dieser Arbeiten werden im folgenden vorgestellt.
Bevor ein Clustering zum Einsatz kommt, ist es wünschenswert, dass die Daten in kompakter Form vorliegen. Oft sind in den Daten viele Informationen vorhanden, die nicht substantiell sind, also als „Rauschen “ betrachtet werden. Außerdem ist es von Vorteil die Daten in einem Raum mit wenig Dimensionen zu beschreiben, um den „Fluch der

Dimensionalität“[1] zu vermeiden. Klassische Verfahren wie die Hauptkomponentenanalyse gelangen bei komplexen, nicht lineare Daten an ihre Grenzen. Aus diesem Grund schließt sich der Vorstellung verschiedener Clustering-Verfahren eine Betrachtung von modernen Ansätzen zur Verringerung der Dimensionalität an.

Yedla et al. [14] haben ihre verbesserte Version des k-means-Algorithmus einfach „Improved K-Means“ getauft. Fokus dieser Arbeit ist eine Methode, die bessere initiale Cluster wählt, als ein Zufallsverfahren. Hierdurch ist es möglich, die Vektoren den Clustern in geringerer Zeit zuzuweisen und die Genauigkeit dabei zu erhöhen (gegenüber dem originale k-means-Algorithmus).

Der sogenannte „Bag of keypoints“ wurde 2004 von Csurka et al. [15] vorgeschlagen. Die Idee lehnt sich an den Bag of Words aus dem Bereich Information Retrival an. Statt Wörtern und Dokumenten sind hier jedoch Bilder und ihre Features Gegenstand. Die Deskriptoren der Features werden quantisiert um so Cluster zu bilden, die jeweils ein semantisches Merkmal bilden sollen. Hierdurch ist es den Autoren in Experimenten gelungen, Bilder mit hoher Genauigkeit korrekt zu kategorisieren.

Der von Seldan et al. [16] entwickelten „Joint Image Segmentation“ liegt ebenfalls die vorige Analogie (Word/Dokument, Bild/Features) zugrunde. Hier werden die Texturen als Features aufgefasst. Ähnliche Bilder sollten viele Texturen gemeinsam haben, daher ist es Ziel dieses Algorithmus, eine Verbindung zwischen ähnlichen Texturen verschiedener Bilder herzustellen. In Experimenten wurde das Verfahren auf natürliche Bilder (Meer, Landschaften) und verschiedene Malstile (van Gogh, Rembrandt, ...) angewendet. Die Autoren demonstrieren, dass in allen Experimenten sinnvolle Cluster gefunden wurden.

Eine Möglichkeit Datenstrukturen in einem Raum mit geringerer Dimensionalität unüberwacht zu lernen, ist der Einsatz von speziellen neuronalen Netzen, den sogenannten Autoencodern. Das Konzept des Autoencoders reicht zwar bis in die 80er Jahre zurück, eine Methode zum effektiven Training tiefer Netze ist erst 2006 von Hinton [10] entwickelt worden. Zhao [4] hat 2016 einen Autoencoder entworfen, der einen Deskriptor aus Gradienten eines Bildes erzeugt. In einem Experiment mit den Mikolajczyk-Bilddaten[2] wurden die Ergebnisse mit denen von Lowes SIFT-Deskriptor verglichen. Der Deskriptor schneidet ebenso gut ab wie SIFT, ist aber ca. 3.5 mal kleiner: Der Feature-Vektor enthält 36 statt 128 Komponenten.

[1] Mit zunehmenden Dimensionen unterscheiden sich Distanzen zwischen den Vektoren kaum noch voneinander. Der Einsatz einer Distanzmetrik ist dann sinnlos.

[2] Hier handelt es sich um Bilder, zu denen transformierte Versionen vorliegen (verzerrt, rotiert, skaliert, etc.). So kann getestet werden, ob ein Deskriptor, unter verschiedenen Bedingungen, die gleichen Features erkennt.

Hinton et al. haben 2017 [17] ihr Konzept über das „dynamische Routen zwischen Kapseln“ veröffentlicht. In diesem Ansatz wird die Position von Objekten, bzw. Teilen dieser, zueinander berücksichtigt. Bisherige Verfahren können dies nicht leisten: Sie erkennen ob ein Objekt in zwei verschiedenen Bilder vorliegt, ist das Objekt in Bild eins gegenüber dem in Bild zwei um 45° gedreht, ist diese Information nicht verfügbar. Eine Kapsel (*capusle*) vertritt in einem neuronalen Netzwerk eine Gruppe von Neuronen, die einen Teil eines Objektes repräsentieren. Diese Gruppen werden trainiert und aktivieren, abhängig von ihrer Eingabe, weitere Kapseln. Auf diese Weise werden Objekte aus verschiedenen Konfigurationen von Kapseln modelliert.

3.2.2 Auswahl des Modells

Um die Features zu gruppieren, soll ein k-means Clustering-Verfahren verwendet werden. Anhand der durch k-means gewonnenen Cluster kann eine Histogrammdarstellung für Features erzeugt werden. Die Kombination dieser Verfahren wird Bag of Visual Words genannt und lehnt sich an das Bag of Words-Modell an. K-means ist einer der einfachsten Vertreter der Clustering-Algorithmen, doch die Adaptierung zur Ausführung auf Grafikkarten bringt eine hohe technische Komplexität mit (Speicherverwaltung auf *host* und *device*, Wahl der Blockdimension, Threadanzahl etc.). Bevor also ein anspruchsvollerer Clustering-Algorithmus in Betracht gezogen wird, soll zunächst die Realisierung des k-means-Algorithmus gelingen. Das parallele Verarbeiten von Histogrammen ist ein Lehrbuchbeispiel für den Einsatz von Grafikkarten, da es durch parallele Reduzierung, ein Muster für einige Probleme, erreicht werden kann.
Für die Kompression der Features soll hier ein Autoencoder genutzt werden. Ein mehrstufiger Autoencoder kann mit jeder Schicht einen kompakteren Deskriptor erzeugen und kodiert die gelernten Informationen in den Gewichten. Der Autoencoder bringt darüber hinaus den Vorteil mit sich, das seine Architektur bereits auf eine parallele Verarbeitung ausgelegt ist und nicht „extra“ berücksichtigt werden muss[3].

3.3 Feature Detektion und Deskription

Für die weitere Verarbeitung der Features ist es erstrebenswert, dass ihre Darstellung möglichst kompakt ist. Deskriptoren werden als Vektoren von Zahlen kodiert, die abhängig vom Verfahren Informationen über einen Pixel und seine Nachbarschaft oder auch ein ganzes Bild enthalten. Je größer die Anzahl der Einträge eines Vektors, desto größer wird der Speicherbedarf und Rechenaufwand. Die erste Stufe des vorgestellten Modells sieht daher die Komprimierung der Feature-Vektoren durch einen Autoencoder

[3]Natürlich ist es auch möglich einen rein sequentiellen Autoencoder zu entwickeln, doch da im Wesentlichen nur Matrixoperationen stattfinden, ist eine Ausführung durch Grafikkarten naheliegend.

vor. Auf diese Weise kann ein initial recht umfangreicher Feature-Vektor aufgebaut werden: Jede Stufe des Autoencoders lernt dann eine kompaktere Darstellung des Feature-Vektors bis zu einer gewünschten Untergrenze.
Da für die vorliegenden Bilddaten der HsH keine speziellen Annahmen getroffen werden können, ist nicht bekannt was für eine Art von Deskriptor gute Ergebnisse liefern kann. In der Literatur findet sich eine große Anzahl an Verfahren zur Detektion und Extraktion von Features für etliche Zwecke. Um einen Überblick zu geben, sollen einige Vertreter angeführt werden, um den Leser einzuführen.

3.3.1 Detektoren

Feature-Detektoren für Bilder sind in die Kategorien *single-scale*,*multi-scale* und *affine invariant* eingeteilt. Detektoren berücksichtigen im Allgemeinen Transformationen wie Rotationen oder Verschiebungen sowie Variationen in der Beleuchtung. Die *multi-scale* Detektoren berücksichtigen zusätzlich Änderungen im Maßstab. Liegen also zwei Bilder vor, die das gleiche Objekt in unterschiedlicher Größe zeigen, werden die gleichen *keypoints* gefunden. Da nicht die Annahme getroffen werden kann, dass die Objekte in den Daten der HsH im gleichen Maßstab vorliegen, liegt hier der Fokus auf *multi-scale* Detektoren.
Die populärsten *multi-scale* Detektoren sind der *Laplacian of Gaussians (Log)* und *Difference of Gaussians*. Ersterer berechnet hierfür die zweite Ableitung der Komponenten und ist daher sehr empfindlich gegenüber Rauschen. In der Praxis wird das Bild bei Verwendung des *LoG* aus diesem Grund zunächst geglättet (das Rauschen reduziert). Da die Glättung ebenfalls eine Faltung des Bildes bedeutet und die Faltung assoziativ ist, wird meist erst der *LoG* mit dem Filter zu Glättung gefaltet: Die Matrizen des *LoG* und Glättungsfilters sind um ein vielfaches kleiner als das Bild und eine Faltung des ganzen Bildes muss so nur einmal statt zweimal erfolgen. Der *DoG* ist eine von Lowe entwickelte Alternative zum *LoG*. Dieser Algorithmus ist zwar nicht genauso präzise, erreicht aber in kürzerer Zeit eine Annäherung die „gut genug“ ist. Da die Bilder der verschiedenen Oktaven im *scale space* (siehe Grundlagen) voneinander subtrahiert werden, ist hier keine Faltung notwendig. Der *DoG* fungiert so als Bandfilter: Es werden Details in den hohen Frequenzen und Flächen in den tiefen Frequenzen gefiltert. Dadurch resultiert auch ein Verlust von Kontrast im Bild.

3.3.2 Deskriptoren

Hier werden einige ausgewählte Deskriptoren vorgestellt, die auf unterschiedliche Anwendungsfälle ausgelegt sind. Der *Spatial Envelope* beurteilt beispielsweise die „Art“ einer Szene, die Local Binary Patterns werden vorwiegend zur Gesichtserkennung verwendet.

Local Binary Patterns Die Local Binary Patterns (LBP) kodieren eine Nachbarschaft eines Pixels, also einen lokalen Teil eines Bildes, indem der Pixel mit seinem Nachbarn verglichen wird. Klassisch wird hier einer 3×3 Matrix verwendet, sodass sich acht Werte und somit 256 mögliche Kodierungen ergeben. Praktisch erzielt der Einsatz von LBP vor allem im Bereich der Gesichtserkennung und Erkennung von Nummernschildern gute Ergebnisse. Durch die kleine 3×3 Matrix werden gerade feine Details berücksichtigt, allerdings können dadurch keine makroskopischen Zusammenhänge berücksichtigt werden. Hierfür können auch größere Nachbarschaften gewählt werden, allerdings gehen dann die Details verloren.

Spatial Envelope In diesem Ansatz wird davon ausgegangen, dass Menschen eine Szene auch einordnen können, wenn diese in geringer Auflösung vorliegt. Der *Spatial Envelope* beschreibt daher das Bild durch globale Features. Torralba und Olivia [18] haben mit dem *Spatial Envelope* ein Verfahren entwickelt, um die z.B. die Natürlichkeit oder Offenheit einen Szene zu beurteilen. Eine hohe Natürlichkeit weist zum Beispiel auf das Bild einer Landschaft hin: Hier kommen in der Regel kaum gerade vertikale und horizontale Linien vor, im Gegensatz zu Bildern, die von Menschen angefertigt wurden.

SIFT Der 1999 von Lowe entwickelte SIFT-Deskriptor, ist der einer der häufigst genutzten für die Objekterkennung in Bildern. Der Deskriptor besitzt zwar keine affine Invarianz, in praktischen Anwendungen werden jedoch auch mit skalierten, rotierten und verschobenen Objekten gute Ergebnisse erzielt. Der mathematische Hintergrund von SIFT wurde wurde bereits im Grundlagen behandelt. Mikolajczyk und Schmid [19] haben 2005 SIFT mit anderen Deskriptoren (u.a. shape context, komplexe Filter, gradient location and orientation histogram, moment invariants, ...) verglichen und kamen zu dem Ergebnis, dass SIFT sehr gut hinsichtlich der Präzison abschneidet. Die Konstruktion des Deskriptors ist allerdings aufwändig und die Beschreibung erfordert einen Feature-Vektor mit 128 Komponenten.

3.3.3 Features in dieser Arbeit

In dieser Arbeit wird das Gruppieren der Bilder durch eine Objekterkennung realisiert. Hiermit eignen sich vor allem *multi-scale* Detektoren. Bei der Auswahl des Deskriptors scheint SIFT, aufgrund der praktischen Erfolge, naheliegend. Die Frage ist, ob bei der Verwendung von SIFT eine weitere Komprimierung noch sinnhaft ist. Die Neuronenanzahl der Schichten eines Stacked Autoencoders nehmen strikt ab, um die Komprimierung zu erreichen. Dies führt dazu, dass zwischen *Input* und dem folgenden *Hidden Layer* maximal $128 \times 127 = 16256$ Verbindungen existieren können. Zhao [4] hat aus diesem Grund einen anderen Ansatz gewählt: Die *keypoints* werden zunächst, wie bei SIFT, durch den

DoG-Operator ermittelt. Anschließend wird ein großer Deskriptor aus den Gradienten der Nachbarschaften um die *keypoints* erzeugt, der dann als Eingabe für den Autoencoder dient. Dieser komprimiert den Vektor auf 36 Komponenten. Gegenüber SIFT ist diese Darstellung ca. $3,5$ mal kleiner, was einen nicht unerheblichen Teil, gerade wegen der großen Menge an Features, ausmacht.
Aus diesem Grund sollen zwei Varianten in der Konzeption verfolgt werden: Zum einen die Verwendung von SIFT-Features und zum anderen das Erzeugen des Deskriptors nach Zhao durch einen Autoencoder. Durch die Kompaktheit des Letzteren ist zu erwarten, dass der Clustering-Vorgang bei diesem um ein vielfaches schneller abgeschlossen ist. Dabei war die Qualität der Ergebnisse von Bildervergleichen in Zhaos Test sehr ähnlich. Ob dies auch für den Bag of Visual Words der Fall ist, soll ein Experiment zeigen.

4 Konzept

Die Konzeption beschäftigt sich zunächst mit einem Überblick des ganzen Prozesses, von der Feature-Extraktion über die Komprimierung bis hin zur Gruppierung. Anschließend folgt eine nähere Betrachtung dieser drei wesentlichen Bestandteile und wie sie ineinandergreifen.
Im Abschnitt „Feature Extraktion“ werden die Feature-Deskriptoren vorgestellt, die hier verwendet werden: Zum einen der SIFT-Deskriptor, da dieser, neben guten praktischen Resultaten, bereits einigermaßen kompakt ist und auch direkt für die Komprimierung verwendet werden kann. Zum anderen wird ein Feature-Vektor konstruiert, der wesentlichen mehr Komponenten umfasst und sich somit für den Einsatz der Komprimierung eignet.
Im folgenden Abschnitt „Autoencoder“ wird auf der Basis der Arbeit von Zhao [4] ein Stacked Denoising Autoencoder eingeführt, der aus einem Feature-Vektor mit 3042 Komponenten, eine Darstellung des Features in einem Raum mit 36 Dimensionen lernt.
Im letzten Abschnitt wird das Bag of Visual Words Modell näher betrachtet: Es werden auf Basis der Analyse parallele Varianten des Clustering- und Histogramm-Algorithmus entworfen, die sich zur Ausführung auf Grafikkarten eignen, die CUDA unterstützen. Abschließend wird behandelt, wie der Bag of Visual Words verwendet werden kann, um ein Modell zu generieren bzw. die Ähnlichkeit zweier Bilder zu bewerten.

4.1 Modell

In der Analyse wurden die wesentlichen Bestandteile identifiziert, welche hier zur Gruppierung von Bildern dienen sollen. Um dies zu erreichen wird ein dreistufiges Modell vorgeschlagen, dass in Abbildung 4.1 skizziert ist. Jede „Zeile“ entspricht hier einer Phase der Verarbeitung. Die Ein- bzw. Ausgaben sind mit durch die Farbe grün gekennzeichnet, die Schritte zur Datenverarbeitung durch Orange. Diese Phasen werden im Folgenden detailliert dargestellt, hier soll jedoch zuerst ein Überblick über den Ablauf gegeben werden. Die drei Phasen zum Erzeugen eines Modells laufen wie folgt ab:

- **Extraktion** Zuerst muss das Verfahren bestimmt werden, mit denen die Features gewonnen werden sollen. Je nach Anwendungsfall sollte dieses Verfahren aus-

tauschbar sein. Zu Beginn werden aus den Bilddaten *Images* durch einen Feature-Extraktor die Feature-Vektoren *Features A* erhoben. Bei den Bilddaten handelt es sich um eine Liste von Matritzen, welche die Intensitätswerte von Bildpixeln kodiert. *Features A* ist ein Vektor von Features, die abhängig vom Verfahren verschiedene Eigenschaften beschreiben.

- **Komprimierung** Um die Komponenten in einem Feature-Vektor auf die Wesentlichen zu reduzieren, erfolgt eine Komprimierung durch einen Autoencoder. Die Phase des Autoencoders ist hierbei optional und daher gestrichelt dargestellt. Sollten die Features nicht durch den Autoencoder komprimiert werden, so ist der Vektor *Features B* gleich dem Vektor *Features A*. Andernfalls enthält *Features B* die komprimierten Features.
- **Gruppierung** In der dritten Phase nimmt der Bag of Visual Words die Features aus der vorigen Phase entgegen und generiert hieraus das *Codebook*. Da die *Visual Words* des *Codebooks* den erzeugten Clustern entsprechen, ist das Codebook selbst eine Liste von Clustern. Die Cluster wiederum sind Stellvertreter einer Menge von Features, daher sind sie ebenfalls ein Vektor mit der gleichen Anzahl an Komponenten wie die Features in *Features B*.

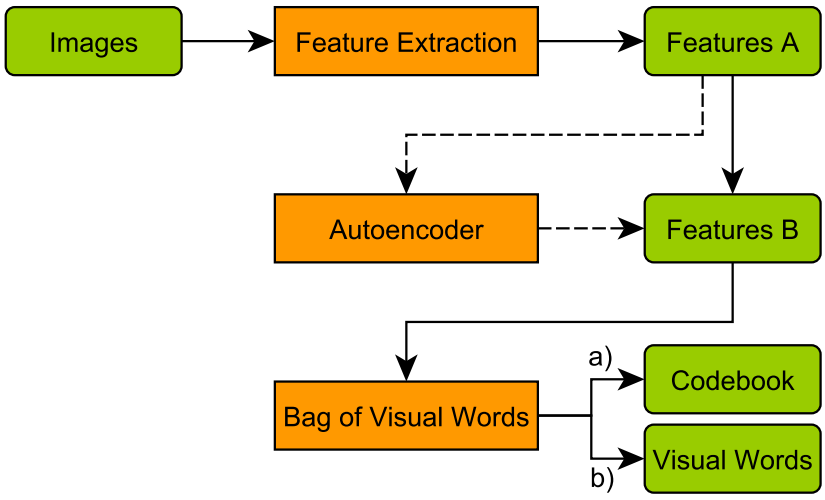

Abbildung 4.1: Aufbau des Modells

Das Erzeugen der *Visual Words* aus einem Bild läuft in den ersten beiden Schritten analog ab, nur das die Menge *Images* aus genau einem Bild, dem zu verarbeitenden, besteht. Die Features werden erst extrahiert und dann optional komprimiert. Für diese Schritte müssen auch exakt die gleichen Parameter verwendet werden, d.h.:

- Der gleiche Feature-Detektor bzw. -Deskriptor muss gewählt werden.

- Eine Komprimierung muss stattfinden, wenn dies im Training der Fall war. Andernfalls darf sie nicht erfolgen.
- Das gleiche Netzwerk zur Erzeugung der komprimierten Features muss gewählt werden.

In der dritten Phase verwendet der Bag of Visual Words das vorher generierte *Codebook* und die Features aus der vorigen Phase, um die *Visual Words* zu ermitteln.

4.2 Feature Extraktion

Da es zahlreiche Verfahren zur Detektion und Extraktion von Features aus Bildern in Literatur und Praxis gibt, können diese unmöglich alle gleichermaßen Beachtung finden. In der Analyse wurde daher beschlossen, zunächst ein Clustering der Bilder anhand einer Objekterkennung zu realisieren. Um den Prozess des Clusterings zu beschleunigen, soll der Feature-Vektor kompakt sein. Eine solche Kompression kann durch einen Autoencoder erreicht werden. Die Effektivität hängt hierbei stark von den verwendeten Daten ab. Die Vektoren sollten initial möglichst viele Komponenten umfassen, um ein tiefes Netz konstruieren zu können. Ein solcher Deskriptor muss aber eigens für den Autoencoder trainiert werden: Die meisten etablierten Deskriptoren ermöglichen eben nicht die Konstruktion eines tiefen Netzes. Die Verwendung eines alternativen Deskriptors ist daher aus zwei Gründen plausibel:

- Da die Schritte im Prozess aufeinander aufbauen, hängt die Qualität der Ergebnisse des Bag of Visual Word Verfahrens auch vom Autoencoder ab. Um den Bag of Visual Words an sich testen zu können, soll ein bereits kompakter, praktisch bewährter Deskriptor als Alternative dienen.
- Es ist denkbar, dass der Schritt der Komprimierung nicht in allen Fällen notwendig bzw. sinnvoll ist, da der Deskriptor bereits kompakt genug ist.

Der von Zhao [4] entwickelte Autoencoder nimmt daher einen eigens kreierten Feature-Vektor als Eingabe entgegen. Die *keypoints* werden hier durch den Difference of Gaussians ermittelt. Um jeden der *keypoints* wird eine Nachbarschaft der Größe 41×41 betrachtet. Von diesen Ausschnitten werden die Gradienten in horizontale und vertikale Richtung bestimmt. Bei einem Einsatz eines Gaußfilters mit einer Filterkerngröße von 3×3, ergeben sich somit pro Richtung 1521 Werte, die den Ausschnitt beschreiben. Der resultierende Feature-Vektor besitzt somit, für beide Richtungen, insgesamt 3042 Komponenten.
Um das Clustering-Verfahren auch ohne einen Autoencoder anwenden zu können, bzw. sicherzustellen, dass andere Feature-Deskriptoren für Bilder direkt verwendet werden können, soll in einem Test auch SIFT Gegenstand sein. So kann nicht nur die Qualität

der Ergebnisse der beiden Deskriptoren verglichen werden, sondern auch Auswirkungen auf die Laufzeit des Bag of Visual Words.

4.3 Autoencoder

In diesem Ansatz wird ein Stacked Denoising Autoencoder zur Komprimierung eines Feature-Vektors entworfen, wie er in der Arbeit von Zhao [4] vorgeschlagen wurde. Hierfür wird zunächst im Abschnitt „Modell“ ein objektorientiertes Modell entworfen, dass den Autoencoder, wie er in der Analyse beschrieben wurde, abbildet. Durch dieses Klassenstruktur lassen sich nun beliebige Autoencoder definieren. Im Teil „Parameter“ wird dann auf den konkreten Autoencoder eingegangen, den Zhao entworfen hat.

4.3.1 Entwurf des Modells

Im Klassendiagramm in Abbildung 4.2 sind die wesentlichen Bestandteile eines Autoencoders erfasst: Der Großteil der Parameter sind private Variablen, die bei Erzeugung, also im Konstruktor, übergeben werden müssen. Die Anzahl der Schichten ergibt sich durch die Anzahl der Elemente in *layers*. Jede Ganzzahl steht hier für die Anzahl der Neuronen in einer Schicht. Die Gewichte zwischen zwei Schichten sind in *weights* gespeichert. Ein Gewicht w_{ij} von Neuron i zu Neuron j zwischen zwei Schichten befindet sich an der Position *weights[i][j]*. Da zwischen Schichten unterschiedliche Aktivierungsfunktionen verwendet werden können, wird eine Liste mit einer Aktivierungsfunktion pro Schichtpaar erwartet.
Durch *encode* wird der Autoencoder genutzt, um die Testdaten zu komprimieren. Die Anzahl der Koordinaten eines Features muss hier gleich der Anzahl der Neuronen in der ersten Schicht des Autoencoders sein. Die Methode *decode* kehrt den Prozess der Enkodierung wieder um. Hier werden entsprechend Features erwartet, die genauso viele Koordinaten wie die letzte Schicht des Netzwerks aufweisen.

4.3.2 Parameter des Netzes

Nachdem nun eine Struktur geschaffen wurde, in der sich beliebige Autoencoder definieren lassen, soll das von Zhao vorgeschlagene Netzwerk näher betrachtet werden. Die Leistung dieses Autoencoders wurden unter verschiedenen Kriterien mit denen der Hauptkomponentenanalyse (PCA) und SIFT-PCA verglichen. Dabei erkannte der Autoencoder in fast allen die gleichen Features, jedoch durch einen 36 statt 128-elementigen Feature-Vektor. Der Encoder des vorgeschlagenen Modells besteht aus fünf Schichten, deren Neuronenanzahl sukzessive reduziert wird, bis schließlich die kleinste Schicht mit

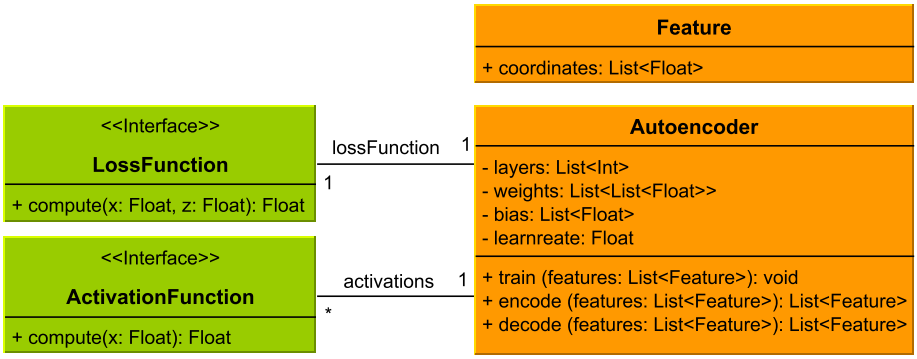

Abbildung 4.2: Klassendiagramm des Autoencoders

36 Neuronen erreicht wird. Abbildung 4.3 zeigt die Schichten des Encoders sowie Decoders. Der Decoder ist umgekehrt aufgebaut und auch die Gewichte der Kanten zwischen zwei Neuronen entsprechen ihren Pendants im Encoder.

Die Hyperparameter für das verwendete Modell sind vollständig dokumentiert. Dabei wird zwischen dem *Pretraining* und dem *Finetuning* unterschieden: Zunächst findet das *Pretraining* statt, also das trainieren der einzelnen Enkoder-Paare. Jedes Paar wird für 500 Iterationen trainiert und die Lernrate auf 0.05 gesetzt. Beim *Finetuning* wird dann das Backpropagation-Verfahren angewendet. Das ganze Netz wird für 700 Iterationen mit einer Lernrate von 0.03 trainiert. Zhao verwendete hier für die Lernrate einen Wert von 2%, sodass $l = 0.02$. Als Aktivierungsfunktion wird in beiden Phasen die logistische (sigmoid) Funktion verwendet und ein *batch* enthält 100 Elemente. Das Netz sieht so in einem *foward* und *backward pass* je 100 Trainingsexemplare. Da dies im Wesentlichen die Laufzeit durch eine Verwendung von mehr Speicher bewirkt, kann diese Zahl bei ungenügenden Ressourcen auch reduziert werden. Sowohl beim *Pretraining* als auch beim *Finetuning* werden 30% der Eingabe korrumpiert.

4.4 Bag of Visual Words

In der Analyse wurde bereits eine sequentielle Varianten von Lloyds-Algorithmus und ein sequentieller Histogramm-Algorithmus vorgestellt und aufgezeigt, an welchen Stellen eine Parallelisierung der Berechnung durch Grafikkarten erfolgen kann. Im Folgenden wird aus diesen Informationen je Algorithmus eine parallele Version abgeleitet, welche sich für die Realisierung als CUDA Programm eignen. Im Abschnitt Modell wird dann der Aufbau des Modells und Ablauf der Funktionsaufrufe skizziert. Zur Interaktion stehen einem Anwender im Wesentlichen eine Funktion zu Generierung eines Modells und zur Berechnung der *Visual Words* eines Bildes zur Verfügung.

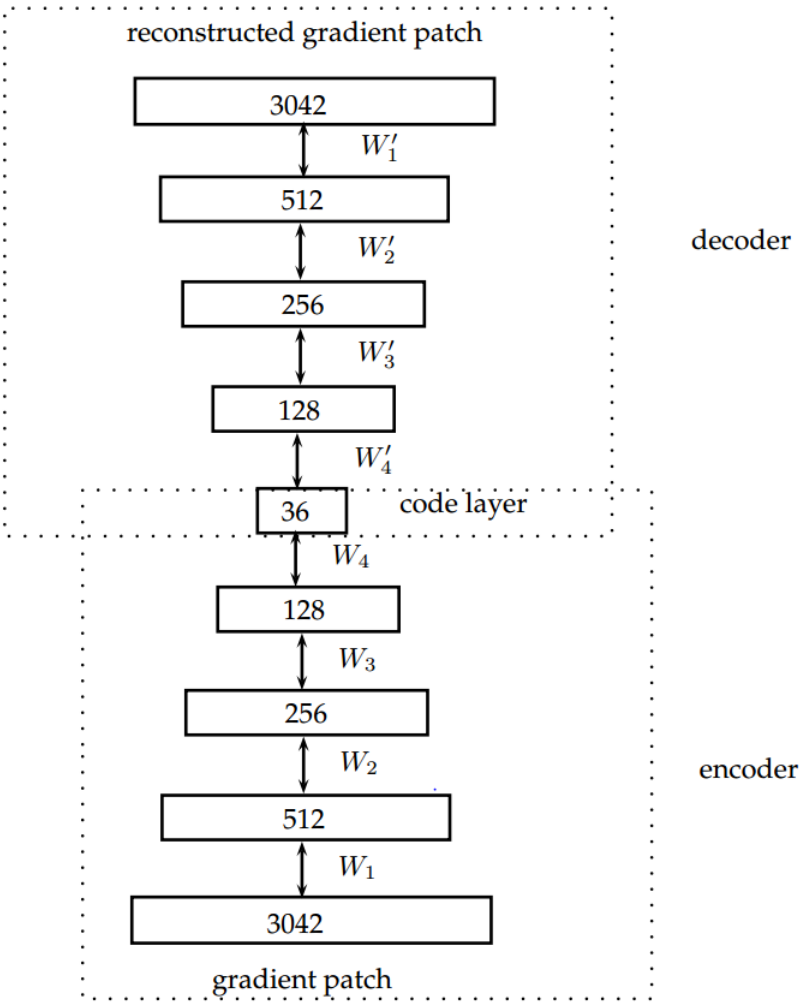

Abbildung 4.3: Schichten des verwendeten Autoencoders [4]

4.4.1 Paralellisierung von Llyods Algorithmus

Da im der Grundlagenkapitel bereits anhand einer sequentiellen Variante von Lloyds Algorithmus, die zu parallelisierenden Berechnungen identifiziert wurden, wird hier der Ablauf einer parallelen Version vorgestellt. Das Listing aus Abbildung 4.4 der Arbeit von Zechner und Granitzer [2] illustriert den Ablauf solch eines Algorithmus in Pseudocode. Die Funktion *lloyd_gpu* erwartet als Parameter die zu quantisierenden Vektoren V und eine initial leere Liste der Cluster C. Der Thread in einem Block mit der *threadId* 0 fungiert hier als Master für die anderen Threads. Die Initialisierung der Cluster mit zufälligen Vektoren aus V wird ebenfalls von diesem übernommen. Die Zuweisung von Vektoren zu Clustern kann parallelisiert werden, in dem pro Feature-Vektor ein Thread

verwendet wird: Jeder Thread berechnet für seine Feature-Vektoren die Distanz zu allen Clusterschwerpunkten. Der Index des Clusters, der am Nächsten ist, wird in einer Liste *l (labels)* unter dem Index des Vektors gespeichert. Dieser Prozess ist in Pseudocode in Zeile 6 und 7 ausgedrückt. Bevor die Cluster aktualisiert werden, müssen die Threads synchronisiert werden: Andernfalls ist nicht garantiert, dass die Berechnung jedes Threads abgeschlossen ist. In Zeile 11 übernimmt der Master-Thread die Neuberechnung der Clusterschwerpunkte. In Zeile 13 werden die Vektoren in den jeweiligen Clusterschwerpunkt mitaufgenommen (aufsummiert). In Zeile 14 wird *m members*, die Anzahl der Vektoren pro Cluster, für jeden neuen Vektor inkrementiert. In Zeile 15 und 16 erfolgt dann die Berechnung des Durchschnitts aller Clusterschwerpunkte.

```
1  lloyd_gpu(V, C, k)
2    if threadId == 0
3      c_j = rand(v_i) ∈ V, j = 1, ..., k, c_j ≠ c_i ∀i ≠ j
4    synchthreads
5
6    until convergence
7      for each v_i ∈ V_threadId
8        l_i = arg min D(c_j, v_i)
9      synchthreads
10
11     if threadId == 0
12       for each v_i ∈ V
13         c_{l_i} = c_{l_i} + v_i
14         m_{l_i} = m_{l_i} + 1
15       for each C_j ∈ C
16         c_j = 1/‖C_j‖ Σ_{v_i ∈ C_j} v_i
```

Abbildung 4.4: Ablauf der parallelen Variante von Lloyds Algorithmus [2].

4.4.2 Parallele Reduzierung von Histogrammen

Die Berechnung eines Histogramms kann parallelisiert werden, da die Operation assoziativ und kommutativ ist: Es spielt keine Rolle, in welcher Reihenfolge die Daten abgearbeitet werden bzw. in welcher Reihenfolge die Klassen inkrementiert werden. Wenn das zu beschreibende Histogramm im *global memory* vorliegt, wird die Berechnungsgeschwindigkeit stark reduziert, da viele Threads auf die gleichen Speicheradressen des Histogramms schreibend zugreifen. Damit es nicht zu Lese- / Schreibanomalien kommt, muss das Inkrementieren einer Klasse atomar sein, d.h. zwischen Lese- und Schreibzugriff darf kein anderer Thread auf die Adresse zugreifen. Dies wird in CUDA durch die Operation *atomicAdd* realisiert. Damit die Anzahl an Threads, die auf dieselbe Adresse schreiben eingeschränkt wird, arbeitet jeder Block auf einem lokalen Histogramm im *shared memory*. Wenn alle Blöcke ihre lokalen Histogramme berechnet haben, müssen

diese noch in das Histogramm im *global memory* kumuliert werden.
In dem Codelisting in Abbildung 4.5 ist solch ein Histogrammalgorithmus in CUDA C aufgeführt. Als Parameter werden hier die Features *features*, die Cluster *clusters*, das zu beschreibende Histogramm *histo*, die Anzahl an Komponenten in einem Feature *featureSize*, die Anzahl der Features *count* sowie die Anzahl der Cluster k erwartet. Zunächst wird in Zeile 3 und 4 der nötige *shared memory* allokiert. Das Schlüsselwort *extern* gibt an, dass die Größe des *shared memory* bei Aufruf des *kernels* mitgegeben wird (in den dreifachen, spitzen Klammern). Die Klassen des privaten Histogramms werden alle mit 0 initialisiert und die Threads synchronisiert. In Zeile 11 wird der Index des ersten Features berechnet, dessen Cluster dieser Thread berechnen soll. Die Anzahl der Features die der Block insgesamt berechnet ergibt sich dann aus der Block- und Griddimension. Die *while*-Schleife in Zeile 14 wird dann für all diese Features durchlaufen. Hier wird der Index des nächsten Clusters berechnet und die entsprechende Klasse des privaten Histogramms inkrementiert. Da nicht jeder Block unbedingt gleich viele Features bearbeitet, also wenn die Anzahl der Features nicht ein Vielfaches der Blockdimension ist, müssen die Threads nach der Schleife wieder synchronisiert werden. Abschließend werden die Werte des privaten Histogramms eines Blocks in das globale kopiert.

4.4.3 Aufbau des Bag of Visual Words Algorithmus

Der Aufbau des Bag of Visual Words Modell ist als Klassendiagramm in Abbildung 4.6 dargestellt. Cluster und Features werden hier als Punkte in einem n-dimensionalen Raum aufgefasst, deren Position durch eine Liste von n *float*-Werten definiert ist. Diese Gemeinsamkeit wird durch Point abstrahiert. Ein Objekt der Klasse Cluster enthält zusätzlich eine Liste *members*, welche die Features enthält, die dem Cluster zugeordnet sind. Die BagOfVisualWords-Klasse selbst umfasst nur *host code* und steuert den Prozessablauf. Die Kmeans-, Histogramm- und Shared-Klasse enthalten die CUDA *kernels* und führen die Berechnungen auf der GPU aus.
Die folgenden Abschnitte „Generierung des Modells“, „Speichern und Laden eines Modells“ und „Berechnung der Visual Words“ illustrieren die Kernfunktionalitäten der BagOfVisualWords-Klasse sowie den jeweiligen Prozessablauf anhand des Klassendiagramms.

Generierung des Modells

Die Generierung eines Modells kann durch die Funktion *createModel* gestartet werden. Die Funktion ist überladen und erwartet als Parameter entweder eine Liste der Features oder den Pfad zu einer Datei, in der die Features gespeichert sind. Der Ablauf der Methodenaufrufe ist dann wie folgt:

```
1 __global__
2 void histogram_kernel (float *features, float *clusters, unsigned int *
    histo, int featureSize, int count, int k) {
3   extern __shared__ int *sharedMemory[];
4   unsigned int histo_private = (unsigned int*) sharedMemory;
5
6   if (threadIdx.x < k) {
7     histo_private[threadIdx.x] = 0;
8   }
9   __syncthreads();
10
11  int i = threadIdx.x + blockDim.x * gridDim.x;
12  int stride = blockDim.x * gridDim.x;
13
14  while (i < count) {
15    float *feature = &features[i * featureSize];
16    int bin = findNearestCluster(feature, clusters, featureSize, k));
17    atomicAdd(&(histo_private[bin]), 1);
18    i += stride;
19  }
20  __syncthreads();
21
22  if (threadIdx.x < k) {
23    atomicAdd(&(histo[threadIdx.x]), histo_private[threadIdx.x]);
24  }
25 }
```

Abbildung 4.5: Histogramm-Algorithmus in CUDA C.

- Es werden bei Aufruf mit einem String die Features eingelesen und anschließend *createModel* mit dieser Feature-Liste aufgerufen. Der k-means-Algorithmus wird mit den Features und k aufgerufen. Die Daten werden zum *device* kopiert und der Prozess gestartet.
- Kmeans nutzt die ausgelagerte Funktion *findNearestCluster* um den nächsten Cluster eines jeden Features zu bestimmen. Intern wird hier durch *computeEuclidean-Distance* die Entfernung eines Cluster-Feature-Paares bestimmt.
- Anschließend prüft *computeDelta* ob sich die Zuordnung von Features zu Clustern nicht mehr wesentlich geändert hat, d.h. die relative Veränderung unter dem Schwellwert liegt.
- Ist der Schwellwert oder eine maximale Anzahl an Iterationen erreicht, ist der Prozess abgeschlossen und die *device* Cluster werden in die Cluster des aufrufenden BagOfVisualWords-Objektes kopiert.

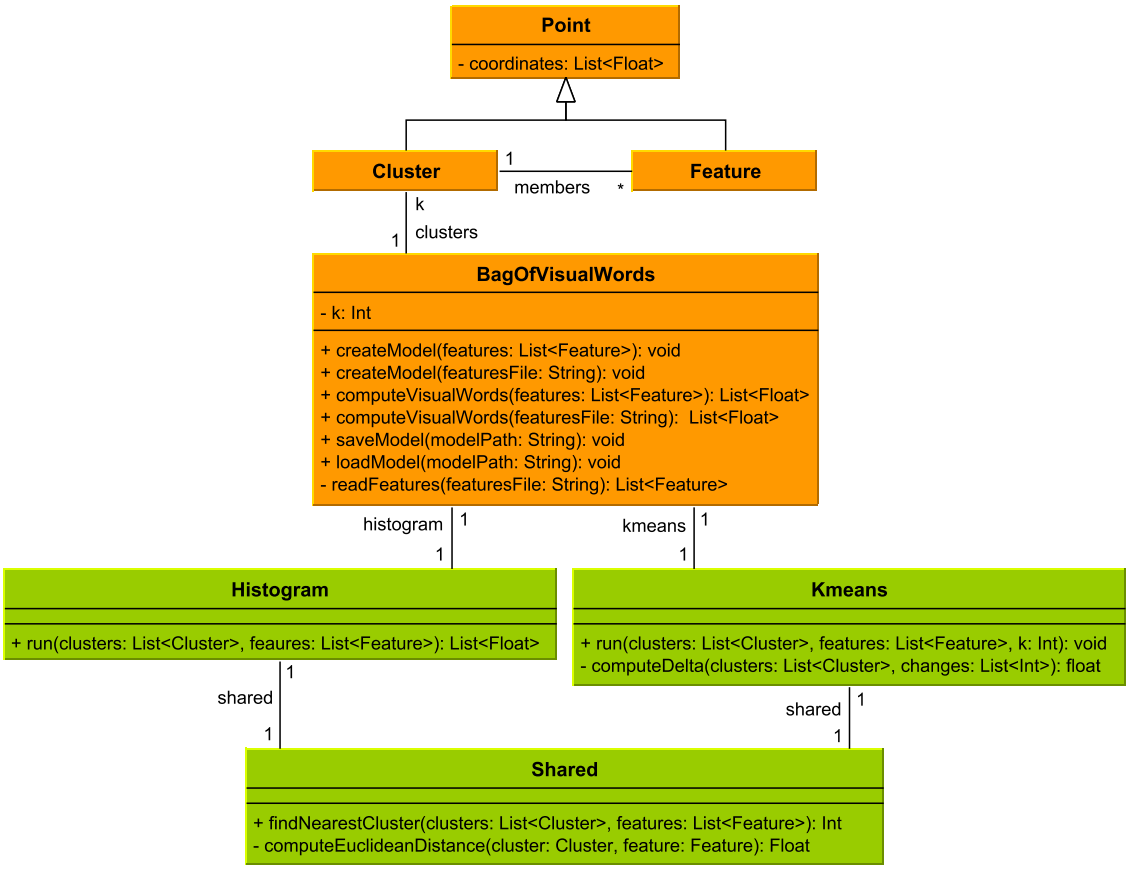

Abbildung 4.6: Klassendiagramm des Bag of Visual Words

Speichern und Laden eines Modells

Um Modelle über den Speicher hinaus verwenden zu können, soll die Funktion angeboten werden, diese persistent zu speichern und wieder einlesen zu können. Die Methode *saveModel(modelPath: String)* der BagOfVisualWords-Klasse speichert die Anzahl der Cluster k, die Liste der Cluster *clusters* sowie die Zuordnung der Features *members* unter dem Pfad *modelPath*. Durch das Pendant *readModel(modelPath: String)* kann ein so gespeichertes Modell, also k, *clusters* und *members*, wieder eingelesen werden.
So ergibt sich zum Beispiel für ein $k = 5$ und 100 Features mit je 128 Komponenten, eine Datei mit 106 Zeilen: Die erste Zeile enthält k. Darauf folgen k viele, also hier fünf, Zeilen mit den Zentren der Cluster und anschließend 100 Zeilen mit den Features. Die Zeilen mit den Clustern enthalten hier 128 Werte, durch Leerzeichen separiert. Die Features enthalten einen Wert am Ende der Zeile mehr: Diese Ganzzahl gibt den Index des Clusters an, zu der das Feature gehört.

Berechnung der Visual Words

Die Erzeugung der *Visual Words* wird durch die BagOfVisualWords-Klasse angestoßen. Wie bei der Modellgenerierung wird intern ein CUDA-Programm, die Histogram-Klasse, verwendet. Für den Gebrauch muss die *computeVisualWords* Methode aufgerufen werden. Der Ablauf ist dann wie folgt:

- Wie beim kmeans Algorithmus kann *computeVisualWords* der BagOfVisualWords-Klasse mit einer Liste der Features oder dem Pfad zu einer Feature-Datei aufgerufen werden. Letztere ließt die Features ein und ruft dann wiederum *computeVisualWords* mit der Feature-Liste auf. Es wird geprüft, ob ein Modell vorhanden ist, also eine Liste von Clustern vorliegt. Ist dies nicht der Fall, wird abgebrochen.
- Die *run* Methode der Histogram-Klasse wird nun mit den *clustern* der aufrufenden BagOfVisualWords-Instanz und den Features aus dem vorigen Schritt aufgerufen. Um das *Visual Word* für ein Feature zu bestimmen, wird *findNearestCluster* aus der Shared-Klasse genutzt. Da diese Funktion den Index des Clusters zurückgibt, kann dieser direkt für die zu inkrementierende Position im Histogramm verwendet werden.
- Das Histogramm wird zum *host* kopiert und an den Aufrufer zurückgegeben. Es kann nun gespeichert oder für weitere Analyse verwendet werden.

5 Implementierung

Die Implementierung ist in drei Teile gegliedert. Zu Beginn wird gezeigt, wie mit Python und der *opencv*-Bibliothek Gradienten von Bildern erzeugt wurden. Diese stellen die Basis für den zweiten Schritt, der Komprimierung durch einen Autoencoder dar. Die Verarbeitung kann an dieser Stelle ohne technischen Bruch fortfahren, da der Autoencoder in TensorFlow und somit ebenfalls in Python umgesetzt wurde. Die so erzeugten komprimierten Features werden zunächst zwischengespeichert, da sie nicht direkt weiterverarbeitet werden können: Der Bag of Visual Words wurde direkt in CUDA C umgesetzt und muss daher die Features wieder einlesen. Ursprünglich war die Implementierung des gesamten Projektes in CUDA geplant. Der Aufwand für die Implementierung des gesamten Modells, insbesondere den Autoencoder, bzw. neuronale Netze im Allgemeinen, geht leider über den Rahmen dieser Arbeit hinaus.

5.1 Extraktion der Features

Zu Ermittlung der *keypoints* durch den SIFT-Detektor wird auf die *opencv*[1] Implementierung von SIFT zurückgegriffen. Zur Verwendung des SIFT Algorithmus ist es erforderlich das *opencv* Projekt zusammen mit dem *opencv-contrib*[2] Projekt selbst zu kompilieren. Bei SIFT handelt es sich um einen patentierten Algorithmus, daher ist er seit Version 3.0 nicht mehr standardmäßig im *opencv* Projekt enthalten.

Da der Autoencoder in Python geschrieben ist, erfolgt die Gewinnung der Patches in der gleichen Sprache, um so eine einfache weitere Verarbeitung zu ermöglichen. Der Prozess lässt sich in drei Schritte untergliedern. Um alle Features eines Bildes zu erhalten, wird die Funktion *extractFeatures* mit dem Pfad zu einer Bilddatei ausgerufen. Es wird das Bild eingelesen, konvertiert und durch den *opencv* SIFT-Detektor die *keypoints* ermittelt. Die Berechnung der Gradienten der Nachbarschaften um diese *keypoints* erfolgt dann durch die Funktion *computeDescriptors*:

```
1 def computeDescriptors(image, keypoints):
2   descriptors = []
3
```

[1] https://github.com/opencv/opencv
[2] https://github.com/opencv/opencv-contrib

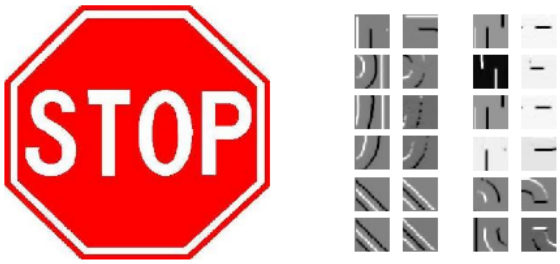

Abbildung 5.1: Stop-Schild und Gradienten um einige der gefundenen *keypoints*[3].

```
4   for keypoint in keypoints:
5     patch = getPatch(image, keypoint)
6     gradients = computeGradients(patch)
7     descriptors.append(gradients)
8   return descriptors
9
10 def getPatch (image, keypoint):
11   x, y = keypoint.pt[0], keypoint.pt[1]
12   return image[y-20:y+21, x-20:x+21]
```

Für jeden *keypoint* werden nun wiederum *Patches* berechnet: Hierbei handelt es sich um die Nachbarschaften der Größe 41×41. Die Gradienten in vertikale und horizontale Richtung eines solchen *Patches* werden durch *computeGradients* bestimmt. In Zeile 2 und 3 in nachfolgendem Codelisting findet die Faltung des Patches mit dem Gauß-Filter statt, den *opencv* anbietet.

```
1 def computeGradients(image):
2   grad_x = cv2.GaussianBlur(image, (3, 1), 0)
3   grad_y = cv2.GaussianBlur(image, (1, 3), 0)
4   return [grad_x, grad_y]
```

In Abbildung 5.1 sind auf der rechten Seite sind, in zwei Reihen, einige der gefundenen Gradienten des Bildes auf der linken Seite dargestellt. Dadurch, dass das Bild des Stop-Schildes viele deutliche Kanten aufweist, sind die Gradienten leicht zuzuordnen.

Sollte im Späteren eine Umsetzung des Autoencoders in CUDA C erfolgen, so lässt sich die Gewinnung der Patches leicht portieren. Die Verwendung von SIFT erfolgt in C analog (in entsprechender Syntax).

[3]Quelle: Caltech101 (stop_sign/image_0005.jpg)

5.2 Autoencoder

In diesem Abschnitt wird behandelt wie, der Autoencoder in Python und TensorFlow umgesetzt wurde. Hierfür wird zunächst eine Übersicht über die Projektstruktur gegeben und anschließend wird an Programmcode demonstriert, wie der von Zhao entworfene Autoencoder abgebildet werden kann.

5.2.1 Projektstruktur

Das Python-Projekt für des Autoencoders besteht aus fünf Dateien. In autoencoder.py ist eine gleichnamige Klasse zur objektorientierten Verwendung enthalten. Die Funktionen zur Feature-Extraktion sind in der Datei feature_extractor.py und werden im folgenden Abschnitt näher behandelt. Die Datei util.py stellt eine Sammlung allgemein verwendeter Funktionen bereit, z.B. zur Messung von Zeit oder Konvertierung von Datenstrukturen. Wie beim Bag of Visual Words liegt hier eine main-Datei bei, welche die Benutzung des Autoencoders durch die Kommandozeile erlaubt.

```
src
├── autoencoder.py
├── feature_extractor.py
├── main.py
├── test.cpp
└── util.py
```

5.2.2 TensorFlow

TensorFlow ist ein Deep-Learning Framework, dass in Python geschrieben ist. Darüber hinaus gibt aus aber auch Schnittstellen zu anderen Sprachen wie beispielsweise C oder Java. Aus dem Namen leitet sich die bereits die Idee ab, die TensorFlow zugrunde liegt: Ein Tensor ist ein (multidimensionales) Array aus Daten. Diese Tensoren sind mit mathematischen Operationen, den Knoten, miteinander verbunden, sodass die Daten durch die Tensoren und Knoten „fließen“ und dabei transformiert werden. Daher eignet sich TensorFlow hervorragend für die Darstellung neuronaler Netze: Die Kanten des Netzes entsprechen den Tensoren, die Neuronen in den Schichten führen mathematische Operationen auf den eingehenden Daten aus und leiten diese weiter an den nächsten Tensor. Die in TensorFlow definierten Modelle lassen sich sowohl auf mehreren CPUs sowie Nvidia GPUs ausführen. Für die Ausführung auf Nvidia Hardware ist es notwendig, dass auf dem System mindestens CUDA 7.5 und die cuDNN Bibliothek 4.0 installiert ist. TensorFlow folgt dem sogenannten „lazy“ Programmierparadigma. Dies bedeutet, dass zunächst aus den Definitionen ein Modell aufgebaut wird. Dieses Modell kann durch

TensorFlow automatisiert geprüft und visualisiert werden. Im nächsten Schritt werden alle nötigen Variablen initialisiert. Erst durch das Erzeugen und Aufrufen einer „Session“ wird das Modell trainiert bzw. auf Testdaten ausgeführt.

```
1 import tensorflow as tf
2
3 a = tf.placeholder(tf.int16)
4 b = tf.placeholder(tf.int16)
5
6 addOp = tf.add(a, b)
7
8 init = tf.initialize_all_variables()
9
10 with tf.Session() as sess:
11     sess.run(init)
12     print "Addition: %i" % sess.run(addOp, feed_dict={a: 2, b: 3})
13
14 sess.close()
```

5.2.3 Modell in TensorFlow

Diese Umsetzung des Autoencoders ist in TensorFlow unter Verwendung der Bibliothek *libsdae-autoencoder-tensorflow*[4] von Rajarshee Mitra erfolgt. Der in der Konzeption vorgestellte Stacked Autoencoder von Zhao kann so wie folgt definiert werden:

```
1 from deepautoencoder import StackedAutoEncoder
2 import cv2
3 import numpy
4
5 model = StackedAutoEncoder(
6   dims=[3042, 1024, 512, 128, 36],
7   activations=['sigm', 'sigm', 'sigm', 'sigm', 'sigm'],
8   epoch=[700, 700, 700, 700, 700],
9   loss='rmse',
10   noise='mask-0.3',
11   lr=0.03,
12   batch_size=100
13 )
14
15 def train(features):
16   model.fit(features)
17
18 def encode(features):
19   model.transform(data=features, mode='encode')
20
21 def decode(features):
```

[4] https://github.com/rajarsheem/libsdae-autoencoder-tensorflow

```
22    model.transform(data=features, mode='decode')
```

In den Zeilen 5 bis 12 findet die Definition des Autoencoders statt. Der *StackedAutoEncoder* entstammt dem eingangs erwähnten Projekt von Rajarshee Mitra. Hier ist es nicht möglich, die Parameter des *Pretrainings* und des *Finetunings* separat zu definieren. Als Ausgangspunkt werden hier daher die Werte des *Finetunings* für beide Prozesse verwendet. In *dims* wird die Menge der Schichten des Encoder-Teils durch eine Liste von Ganzzahlen dargestellt: Eine Zahl steht für die Anzahl der Neuronen pro Schicht. Hier wird davon ausgegangen, dass benachbarte Schichten voll verbunden sind. Der Decoder-Teil ist umgekehrt aufgebaut, daher leitet sich dieser aus der Encoder-Definition ab und muss nicht notiert werden. Folglich werden in *activations* die Aktivierungsfunktion auch nur einmal notiert. Die Abkürzung *sigm* steht hier für die sigmoid Funktion. Die Schichten können über verschiedene Aktivierungsfunktionen miteinander verbunden sein, hier wird aber für alle die sigmoid Funktion benutzt. Wichtig ist hier, dass die Aktivierungsfunktion nicht linear ist, damit komplexe Zusammenhänge gelernt werden können. Die Liste *epoch* enthält die Anzahl der Trainingsiteration für die Autoencoder, die aus den benachbarten Schichten konstruiert und einzelnen trainiert werden. In Zeile 9 wird unter *loss* die Metrik definiert, mit welcher der Fehler der Rekonstruktion gemessen wird. *rmse* steht für *root-mean-square error* und ist somit die Wurzel aus dem gemittelten quadratischen Fehler. Da der konzipierte Autoencoder mit einer verzerrten Eingabe trainiert werden soll, wird unter *noise* die Art der Verzerrung angegeben: 30% des Vektors werden maskiert (zufällig ausgewählte Komponenten). Die Lernrate wird hier mit *lr* bezeichnet und wurde entsprechend des konzipierten Autoencoders auf 3% gesetzt.
In Zeile 15, 18 und 21 werden die in der Konzeption vorgestellten Methoden *train*, *encode* und *decode* definiert. Diese ermöglichen eine Verwendung des konkreten Autoencoders. Der Parameter *features* muss bei *train* und *encode* eine Liste sein, die als Elemente Listen mit 3042 Komponenten enthält. Als Eingabe werden hier *numpy* Arrays verwendet, sodass sich die Form *shape(n, 3042)* bei n Features ergibt. Dementsprechend erwartet *decode* ein Array der Form *shape(n, 36)*.

Zusammenhang mit TensorFlow Das Modell verbirgt jegliche Interaktion mit TensorFlow. Intern wurden die Operationen jedoch in TensorFlow umgesetzt. Pro Schicht wird die *run* Funktion aufgerufen. Hier wird zunächst eine Session *sess*, ein Platzhalter für die Eingabe x und die Ausgabe $x\_$, sowie die Funktionen *encode* und *decode* deklariert:

```
1 sess = tf.Session()
2 x = tf.placeholder(dtype=tf.float32, shape=[None, inp_dim], name='x')
3 x_ = tf.placeholder(dtype=tf.float32, shape=[None, inp_dim], name='x_')
4
5 encode = {
6   'weights': tf.Variable(tf.truncated_normal([inp_dim, hid_dim], dtype=tf.
      float32)),
```

```
7   'biases': tf.Variable(tf.truncated_normal([hid_dim], dtype=tf.float32))
8 }
9 decode = {
10  'biases': tf.Variable(tf.truncated_normal([inp_dim], dtype=tf.float32)),
11  'weights': tf.transpose(encode['weights'])
12 }
```

Die Dimensionen der Eingabe und Schichten (*inp_dim* und *hid_dim*) sind Parameter des Algorithmus. Durch *truncted_normal* werden zufällige, normalverteilte Zahlen erzeugt. Anschließend wird dann definiert, wie die Daten kodiert bzw. dekodiert werden, der Fehler berechnet wird und welches Verfahren zum Training angewendet wird (gekürzter Code, im Original werden mehrere Metriken zu Fehlerberechnung angeboten):

```
1 encoded = activate(tf.matmul(x, encode['weights']) + encode['biases'])
2 decoded = tf.matmul(encoded, decode['weights']) + decode['biases']
3 loss = tf.sqrt(tf.reduce_mean(tf.square(tf.subtract(x_, decoded))))
4 train_op = tf.train.GradientDescentOptimizer(lr).minimize(loss)
```

Die Funktion *activate* entspricht hier der angegebenen Aktivierungsfunktion. Da bei Zhao die sigmoid Funktion verwendet wird, entspricht dies dem Aufruf von *tf.nn.sigmoid*. Die Variable *lr* gibt die Lernrate an und ist Paramter der *run*-Methode. Zum Schluss findet die Anwendung des Modells für die definierte Anzahl an Epochen statt:

```
1 sess.run(tf.global_variables_initializer())
2 for i in range(epoch):
3   batch_x, batch_x_ = utils.get_batch(data_x, data_x_, batch_size)
4   sess.run(train_op, feed_dict={x: batch_x, x_: batch_x_})
```

Verwendung durch die Kommandozeile Die Datei *main.py* importiert dann den Feature-Extraktor sowie den parametrierten Autoencoder. In Zeile 6 und 7 werden durch die *readImages* Methode die Bilddaten eingelesen. Hierfür müssen die Pfade zu den Trainigs- bzw. Test-Dateien auf der Kommandozeile angegeben werden: *python3 main.py /train/path test/path*. In Zeile 10 wird dann die *train*-Methode des Autoencoders genutzt, um die Gewichte anhand der Trainingsdaten zu initialisieren. In Zeile 11 erfolgt die Komprimierung der Testdaten und abschließend in der nächsten Zeile die Speicherung.

```
1 import sys
2 import util
3 import featureExtractor
4 import autoencoder
5
6 train = util.readImages(sys.argv[1])
7 test = util.readImages(sys.argv[2])
8 target = sys.argv[3];
9
10 autoencoder.train(featureExtractor.extractAll(train))
11 result = autoencoder.encode(featureExtractor.extractAll(test))
12 util.writeFeatures(target, result)
```

5.3 Bag of Visual Words

Zunächst wird eine Übersicht über die Projektstruktur der Bag of Visual Words Implementierung gegeben. Neben der konkreten Klassenstruktur wird auf Abweichungen zum Konzept eingegangen. Dem schließt die Betrachtung des k-means Clustering der Features an, sowie Unterschiede und Limitierungen der *global* bzw. *shared memory* Varianten.

5.3.1 Projektstruktur

Der Bag of Visual Words ist als Klasse in C++ umgesetzt worden und ist auch die öffentliche API des Programms. Der k-means- und Histogramm-Algorithmus sind CUDA C-Programme und tragen somit die Endung .cu. Neben den CUDA Programmen sind hier aber auch Varianten in C zur Ausführung auf CPUs enthalten. In der util.cpp Datei sind Funktionen zur Messung von Ausführungszeiten, Lesen / Schreiben von Dateien und und Formatierung von Zeichenketten enthalten. Zur direkten Ausführung im Projekt ist eine main.cpp Datei enthalten. Hier werden Argumente der Kommandozeile geparst, um einen entsprechenden Bag of Visual Words zu generieren bzw. auszuführen. Inklusive header-Dateien ergibt sich somit folgender Aufbau des src-Ordners:

```
src
├── BagOfVisualWords.h
├── BagOfVisualWords.cpp
├── histogram.h
├── histogram.cu
├── kmeans.h
├── kmeans.cpp
├── main.cpp
├── shared.h
├── shared.cu
└── util.cpp
makefile
```

5.3.2 Abweichungen zum Konzept

Da CUDA C eine sehr hardwarenahe Sprache ist, obliegt die effiziente Verwendung des Speichers dem Programmierer. Insbesondere auf dem *device* sollte mit dem Speicher genügsam umgegangen werden. Daher weicht die Implementierung von dem vorgestellten Konzept bei den verwendeten Datentypen ab: Statt Objekte für Cluster oder Features zu erzeugen, werden diese Typen einfach als *pointer* von *float*-Werten dargestellt. Dies ist für die Features problemlos möglich, doch die Information über die Mitgliedschaft

seiner Features kann nicht länger in einem Cluster enthalten sein. Diese Informationen wird stattdessen global vom BagOfVisualWords in der privaten Variablen *membership* gehalten und als *pointer* zu Integer-Werten dargestellt. Diese Liste enthält für jedes Feature den Index des Clusters, dem es zugeordnet ist. Der Index eines Features an der Stelle i in der *features* Liste, ist dann an der Stelle $membership_i$ gespeichert.
Dadurch dass *pointer* verwendet werden, erfordern die meisten Funktionsaufrufe nun zusätzliche Parameter:

- Die Anzahl der Features *count* muss angegeben werden. In C ist es nicht auf einfachem Weg möglich zu bestimmen, auf wie viele Elemente ein *pointer* zeigt.
- Die Anzahl an Komponenten *featureSize* in einem Feature bzw. Cluster muss angegeben werden. Auch hier handelt es sich wieder um einen *pointer*, daher ist es nicht möglich, die Anzahl der Elemente zu ermitteln.

Um Vergleiche mit sequentiellen Varianten zu ermöglichen, kann durch den Aufruf *setMode(mode: Int)* auf einem BagOfVisualWords-Objekt festgelegt werden, ob die GPU (0) oder CPU (1) verwendet werden soll.

5.3.3 Paralleler k-means-Algorithmus

Als Referenzimplementierung für die Umsetzung in CUDA C diente hier das Projekt von Serban Giuroiu [5]. Der *kernel* wurde adaptiert und leicht verändert, um gemeinsam genutzte Methoden auszulagern. Die Funktion *kmeans* in der Datei kmeans.cuda kann genutzt werden, um Clustering auf der GPU auszuführen. Der Algorithmus erwartet dabei als Parameter:

- **float **features** Eine Liste von Feature-Vektoren, die zu gruppieren sind.
- **float **clusters** Eine Liste, welche mit den Clustern befüllt wird.
- **int *membership** Diese Liste wird die Cluster-Vektor-Zuordnung enthalten. An der Stelle $membership_i$ wird der Index des Clusters aus *clusters* eingetragen, dem das Feature $features_i$ zugeordnet wird.
- **int featureSize** Die Anzahl der Komponenten in einem Feature-Vektor, also die Anzahl der *float*-Werte.
- **int count** Die Anzahl der Features.
- **int k** Die Anzahl der zu bildenden Cluster k.
- **int iterations** Die maximale Anzahl an Iterationen, die durchlaufen wird, falls bisher keine Konvergenz erreicht wurde.

[5] https://github.com/serban/kmeans

- **float conv** Ein Schwellwert, der in jeder Iteration mit der relativen Veränderung der Mitgliedschaft verglichen wird. Ist er größer, wird das Clustering beendet.

Die Datenstruktur der Features und Cluster ist dabei spaltenorientiert, d.h. dass *features* und *clusters* jeweils auf *featureSize* viele Elemente zeigen. Dahinter liegen dann im Fall der Features *count* viele *pointer* zu den eigentlichen Koordinaten, im Fall der Cluster k viele. An der Stelle *features[0][0]* befindet sich dann die erste Koordinate des ersten Elements, an der Stelle *features[0][1]* die erste Koordinate des zweiten Elements, etc.
Vor dem Aufruf des *kernels* wird für die Features und Cluster der notwendige Speicher allokiert und die Daten zum *device* kopiert. Die Dimensionen der Features sowie der Cluster werden durch *Float*-Werte dargestellt. Ein SIFT Feature-Vektor belegt so $128 \times 4 = 512$ Byte. Der Deskriptor, der durch den Autoencoder erzeugt wurde, belegt $36 \times 4 = 144$ Byte. Da die Features ursprünglich als zweidimensionales Array vorliegen (*pointer-pointer* zu *float*-Werten), werden diese in ein eindimensionales Arrays konvertiert werden, damit die Daten von *host* zu *device* kopiert werden können.
Der Clustering-Vorgang wird nun in einer Schleife durchgeführt. Dabei werden die in der Konzeption vorgestellten Funktionen *computeEuclideanDistance*, *findNearestCluster* und *computeDelta* innerhalb der Schleife nacheinander aufgerufen. Abweichend zum Konzept, erfordern auch diese Funktionen *count* und *featuresSize* als zusätzliche Parameter, um korrekt über die *pointer* iterieren zu können. Wenn das Konvergenzkriterium *conv* oder die maximale Anzahl an Iterationen *iterations* erreicht wurde, ist der Vorgang beendet und die das Ergebnis wird vom *device* zum *host* in *clusters* respektive *membership* kopiert.

5.3.4 Shared memory

Zur Beschleunigung der Berechnung bei der Suche des nächsten Clusters zu einem gegebenen Punkt, soll CUDAs *shared memory* genutzt werden. Hierfür werden die Cluster pro Block vom *global* in den *shared memory* kopiert. Daraus ergibt sich eine Anpassung an mehreren Stellen im Programm. Um beide Implementierungen zu unterstützen, fragt eine Direktive für den Präprozessor den Wert der Variable *SHARED_MEM* ab. Falls der Wert der Variable *true* ist, wird Code nur für die *shared memory* Implementierung eingebunden, andernfalls für die *global memory* Variante:

```
1 #if SHARED_MEM
2 // shared memory Logik
3 #else
4 // global memory Logik
```

Die Größe des extra zu allokierenden Speichers pro Block muss in *blockSharedDataSize* berücksichtigt werden und ergibt sich aus der Anzahl der Cluster und der Anzahl der Elemente eines Features:

```
1 const unsigned int membershipDataSize = numThreads * sizeof(unsigned char)
    ;
2 const unsigned int clusterDataSize = k * featureSize * sizeof(float);
3
4 #if SHARED_MEM
5     const unsigned int blockSharedDataSize = membershipDataSize +
    clusterDataSize;
6 #else
7   const unsigned int blockSharedDataSize = membershipDataSize;
```

In der Funktion *findNearestCluster* wird der Parameter *clusters* in *deviceClusters* umbenannt. Wenn die *global memory* Variante genutzt wird, sind diese gleich den Clustern *clusters*. Andernfalls werden die *deviceCluster* in die *clusters* im *shared memory* kopiert. Da die Größe von *clusterDataSize* hier von k und *size* abhängt, also der Anzahl der Cluster und der Anzahl der Komponenten eines Features, ist diese Implementierung nur einsetzbar, wenn k und *size* in Hinsicht auf den verfügbaren *shared memory* nicht zu groß gewählt werden. Die Größe von *membershipDataSize* ist nur abhängig von der Anzahl der Threads. Werden beispielsweise 256 Threads pro Block gewählt, benötigt dies konstant, unabhängig von k und *size*, 1024 Byte *shared memory*. Für die hier verwendeten Deskriptoren kann somit eine Obergrenze für k berechnet werden. Gängige Modelle Nvidias CUDA kompatibler Grafikkarten sind mit 16 oder 48 Kilobyte *shared memory* ausgestattet. Von 256 Threads pro Block ausgehend ergibt dies ein maximales k von $(memory - 1024)/512$. Für SIFT sind dies bei 48 Kilobyte maximal 91, für 16 Kilobyte 29 Cluster. Der von Zhao entworfene Deskriptor ermöglicht bei 48 Kilobyte Speicher 326 Clustern, für 16 Kilobyte bis zu 104. Sind dennoch mehr Cluster notwendig, muss auf die *global memory* Implementierung zurückgegriffen werden. Hier werden, je nach Anzahl der Cluster, erheblich höhere Berechnungszeiten erwartet, jedoch gibt es kein Limit für die Gesamtanzahl an Clustern.

6 Evaluierung

Die Qualität der Ergebnisse des in der Implementierung realisierten Modells wird in diesem Kapitel durch ein Testexperiment überprüft. Weiterhin werden Ausführungszeiten für beispielsweise verschieden große k beim Bag of Visual Words betrachtet, um eine Einsicht in die Performance der Algorithmen zu gewinnen. Im Kapitel Experimentaufbau wird zunächst das Experiement sowie verwendete Metriken und Ergebnistypen behandelt. Nachfolgend wird erläutert, wie die großen Menge an benötigten Trainings- und Testdaten wiederholbar und automatisiert aufgebaut wird. Dieser Abschnitt illustriert auch, wie eine Durchführung des Experiments vonstatten geht. Im letzten Teil werden dann konkrete Testgruppen aus den Caltech101 [20] Bilddaten erzeugt. Diese Menge an Bilddaten hat große Verbreitung im Bereich der Objekterkennung gefunden. Auf diese Weise ist ein Vergleich mit Arbeiten Anderer prinzipiell möglich.

6.1 Testdaten und Testgenerierung

Der Abschnitt Testdaten stellt die hier verwendete Menge von Bildern vor, die Caltech101, welche extra für den Test von Algorithmen bezüglich der Objekterkennung in Bildern entwickelt wurde.
Im folgenden Abschnitt zur Testgenerierung wird ein Verfahren zur zufälligen Auswahl von Trainings- und Testbildern unter Berücksichtigung verschiedener Restriktionen, wie z.B. dem Verhältnis der Anzahl von Trainings- zu Testbildern vorgestellt.

6.1.1 Testdaten

Als Testmenge wurden die Bilder der Caltech101-Menge verwendet. Bei Caltech101 handelt es sich um eine weit verbreitete Menge von Bilddaten, die vorwiegend zum Test von Algorithmen bezüglich der Objekterkennung in Bildern dient. Insgesamt liegen, wie der Name sagt, 101 Kategorien vor, die jeweils zwischen 40 und 800 Bildern enthalten. Auf der offiziellen Webseite [1] und im Artikel der Autoren wird empfohlen, die eigene Arbeit mit derer anderen vergleichbar zu halten, indem:

[1] http://www.vision.caltech.edu/Image_Datasets/Caltech101/

- Eine feste Anzahl an Trainings- und Testbildern verwendet wird.
- Experimente mit einer zufälligen Auswahl an Bildern wiederholt werden.
- Ähnlich viele Bilder, wie in den Arbeiten anderer, verwendet werden (1, 3, 5, 10, 15, 20 oder 30 Trainingsbilder; 20 oder 30 Testbilder).

Die Caltech101 Daten liegen nach Download kategorisiert im JPG-Format vor. Die Struktur wurde so beibehalten und ist noch für die Testgenerierung relevant. Direkt unter dem Caltech101-Ordner ist pro Kategorie ein Ordner vorhanden, der die jeweiligen Bilder immer im gleichen Namensschema enthält:

```
Caltech101
├── accordion
│   ├── image_0001.jpg
│   ├── image_0002.jpg
│   └── ...
├── airplanes
│   └── ...
└── ...
```

Abbildung 6.1 zeigt vier Bilder aus der Kategorie „Erdbeere“. Neben Bildern von realen Rosengewächsen sind auch Zeichnungen und Objekte enthalten, die Form und Farbe der Erdbeere nachempfunden sind. Auch sind die Objekte auf einem Bild zum Teil in unterschiedlicher Menge vorhanden. Durch diese Variation ist ein Algorithmus so gefordert, tatsächlich eine Abstraktion zu lernen.

Abbildung 6.1: Verschiedene Bilder aus der Kategorie „Erdbeere“ der Caltech101 Bilddaten.

6.1.2 Testgenerierung

Die Generierung von Testdaten bietet sich aus mehreren Gründen an. Zum einen sind enorm viele Trainingsdaten notwendig, um ein leistungsfähiges Modell zu generieren, zum anderen sollen im Test ca. 2000 Bildpaare verwendet werden. Ein manueller Aufbau

der Testfälle wäre ein enormer Aufwand, fehleranfällig und nicht sehr flexibel. Da ein praktisch taugliches Modell erst durch die Variation einiger Parameter gefunden werden kann, ist es wünschenswert, Testdaten mit verschiedenen Eigenschaften generieren zu können:

- Die Anzahl der Kategorien sollte bestimmbar sein. Dies entspricht einer Kategorie der Caltech101-Daten. Somit sind hier theoretisch bis zu 101 Kategorien im Test denkbar.
- Das Verhältnis bzw. die Anzahl an Trainings- und Testbildern muss definierbar sein.

Letztendlich sollen die Histogramme der *Visual Words* zweier Bilder miteinander verglichen und so die Ähnlichkeit gemessen werden. Ein Programm automatisiert daher die Generierung solcher Paare: Es werden zufällige Bildpaare aus ausgewählten Kategorien (*airplanes*, *anchor*, ...) selektiert. Diese Paare, sowie die Information, ob die Bilder in der selben oder einer verschiedenen Kategorie liegen, stellen einen Testkandidaten dar. Zwei Bilder liegen dabei in der selben Klasse, wenn sie im selben Ordner im Dateisystem, also hier im Caltech101-Ordner, enthalten sind. Das Ergebnis wird dann als Datei *test _time.txt* gespeichert. Die Pfade der Bilder werden hierbei relativ zum Caltech101-Ordner gespeichert, die Information über die Kategorie wird als „+“ bzw. „-“ kodiert. Eine Datei für die Kategorien *airplanes* und *anchor* könnte dann so beginnen:

```
1 airplanes/image_0023.jpg airplanes/image_0009.jpg +
2 airplanes/image_0002.jpg anchor/image_0015.jpg -
3 anchor/image_0013.jpg airplanes/image_0002.jpg -
4 anchor/image_0001.jpg anchor/image_0005.jpg +
5 airplanes/image_0006.jpg anchor/image_0006.jpg -
6 ...
```

Neben den zu verwendenden Kategorien muss bei Erzeugung die Anzahl der Testkandidaten, das Verhältnis von positiven zu negativen Kategorien sowie die Anzahl der Trainings- und Testbilder angegeben werden.
Die Bilder, welche durch das Programm für das Training ausgewählt wurden, werden separat als *train_time.txt* gespeichert. Pro Zeile ist hier der relative Pfad des Bildes innerhalb des Caltech101-Ordners enthalten.

6.2 Experimentaufbau

Das Experiment soll sowohl die Feature-Kompression durch einen Autoencoder testen als auch die Kategorisierung bzw. den Vergleich der Bilder durch den Bag of Visual Words. Aus diesem Grund ist das Experiment zweigeteilt:

(a) In dieser Variante findet ein reiner Test des Bag of Visual Words statt. Hierfür werden durch SIFT die Feature-Deskriptoren von Trainingsbildern extrahiert und direkt als Eingabe an den Bag of Visual Words gegeben. Anschließend folgt die Verarbeitung der Testbilder.

(b) Hier wird nach Extraktion der *keypoints* durch den SIFT-Detektor der Feature-Deskriptor durch den Autoencoder erzeugt. Die so erhaltenen Features werden dann wie in (a) durch den Bag of Visual Words gruppiert und anschließend die *Visual Words* der Testbilder erzeugt.

Wichtig ist, dass pro Durchführung des Experiments in beiden Varianten die gleichen Trainings- und Testbilder verwendet werden, damit die Ergebnisse beider Deskriptoren miteinander vergleichbar sind.
Nach Erzeugung des Modells mit den Trainingsbildern, werden nun die Features der Testkandidaten extrahiert und pro Bild dies *Visual Words* berechnet. Die Ähnlichkeit *sim (similarity)* der resultierenden Histogramme h_1 und h_2 wird dann als $1 - MSE$ *(mean squared error)* gemessen:

$$sim(h_1, h_2) = 1 - MSE(h_1, h_2)$$
$$MSE(h_1, h_2) = \frac{1}{n} \sum_{i=0}^{n} (h_{1_i} - h_{2_i})^2$$

Beträgt die Ähnlichkeit von zwei Histogrammen 1 werden sie als identisch angesehen. Werte nahe 0 drücken aus, dass die Histogramme sich sehr voneinander unterscheiden. Damit nun unterschieden werden kann, ob zwei Bilder in derselben Klasse sind, muss die Ähnlichkeit mit einem Schwellwert verglichen werden. So kann beispielsweise definiert werden, dass es sich bei einer Ähnlichkeit größer als 0.8 um dieselbe Klasse handelt. Somit hat der Schwellwert unmittelbar Auswirkungen auf die Ergebnisse und sollte selbst Bestandteil des Experiments sein. Das Resultat eines solchen Vergleichs ist dann einer der beiden folgenden Kategorien zuzuordnen:

- **True Positives (TP)** Bei *True Positives* handelt es sich um zwei Bildern die entweder in der gleichen oder einer verschiedenen Klasse liegen und die Vorhersage des Modells diesbezüglich korrekt ist.
- **False Positives (FP)** In diesem Fall ist die Klassifizierung durch das Modell nicht korrekt: Bei gleicher Klasse wurde eine geringe Ähnlichkeit erkannt, bei verschiedenen eine Hohe.

Damit ein Modell zuverlässige Ergebnisse liefert, muss es größtenteils *True Positives* erkennen, bzw. der Anteil der *True Positives* sollte im Verhältnis zu den *False Positives* bei weitem überwiegen. Für eine visuelle Darstellung dieses Verhältnisses eignet sich die

Receiver Operating Characteristic (ROC): Diese stellt die *TP* auf der Ordinate und die *FP* auf der Abzisse dar.

6.3 Experimentdurchführung

Das Clustering wurde in drei Experimenten getestet. Pro Experiment wurden Eigenschaften einer Bildmenge variiert, um zu beobachten, wie sich diese Veränderungen auf die Qualität der Ergebnisse auswirkt. Die Testmengen sind jeweils wie folgt gewählt:

- **Test 1** besteht aus zwei Kategorien: „Bonsai“ und „Leopard“. Es wurden 50 Bilder verwendet, aus beiden Kategorien je 25. Von diesen Bildern sind 30% für die Trainings- und die anderen 70% für die Testphase bestimmt. Im Training liegen ca. 10.000 Features vor, im Test knapp 27.000.
- **Test 2** enthält ebenfalls, wie Test 1, die Kategorien „Bonsai“ und „Leopard“. Hier werden aber insgesamt 100 Bilder verwendet. Aus beiden Kategorien wurden wieder gleich viele Bilder gewählt und die Aufteilung in Trainings- bzw. Testdaten bleibt auch gleich. Hier wurden für das Training ca. 21.000 Features extrahiert und für den Test 55.000.
- **Test 3** enthält drei Kategorien von Bildern, um zu testen, ob eine komplexere Kategorisierung prinzipiell möglich ist. Als Kategorien wurden hier „Flugzeug“, „Motorrad“ und „Armbanduhr“ gewählt. Insgesamt werden 80 Bilder verwendet, jede Kategorie steuert ein Drittel der Bilder bei. Die Aufteilung in Trainings- und Testdaten wird auch hier beibehalten. In der Trainingsphase lagen ungefähr 18.000 Features vor, in der Testphase ca. 52.000.

Für den Bag of Visual Words wurde eine maximale Anzahl von 500 Iterationen festgelegt, falls das Konvergenzkriterium nicht erreicht wurde. Dabei gilt, dass bei einer relative Veränderung der Vektor-Cluster-Zuordnung kleiner 0.02% das Konvergenzkriterum erfüllt ist. Als Schwellwert für eine positive Klassifizierung wurde hier 0.8 gewählt. Als Grafikkarte für die Experimente wurde eine Nvidia Quadro M4000 (GM204GL) verwendet. Bei dieser Grafikkarte stehen 48kB *shared memory* zur Verfügung, sodass bei 256 Threads 91 Cluster für SIFT und ca. 300 Cluster für die Autoencoder-Features möglich sind. Die Programme wurden unter CUDA 7.5 ausgeführt.

6.3.1 Qualität der Ergebnisse

Für die Experimente ist in den Abbildungen 6.2 bis 6.7 das Ergebnis der Klassifizierung der SIFT- und Autoencoder-Features dargestellt. Hierbei handelt es sich um

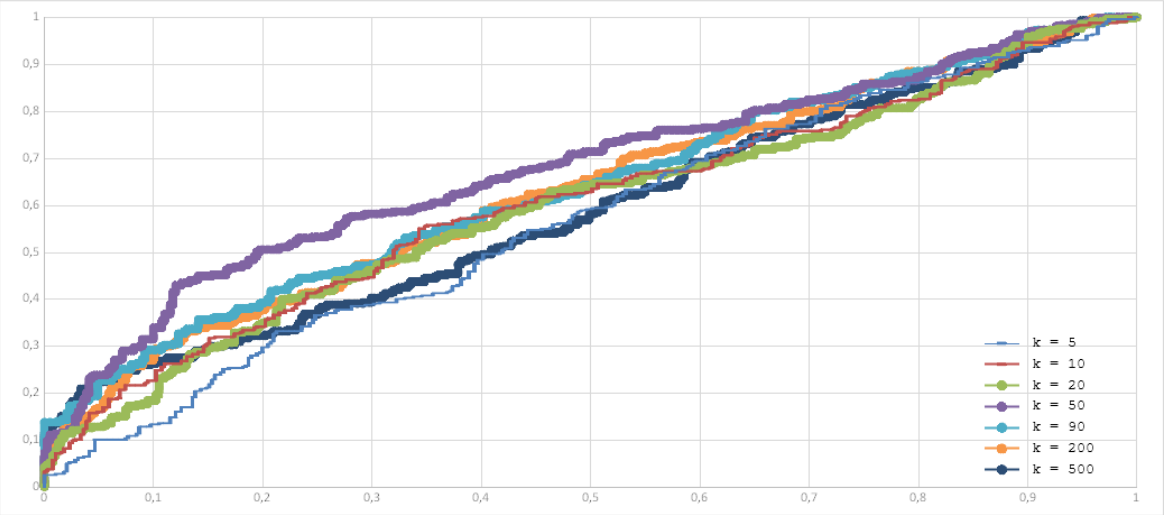

Abbildung 6.2: ROC-Kurven für die SIFT-Features und verschiedene k aus Experiment 1.

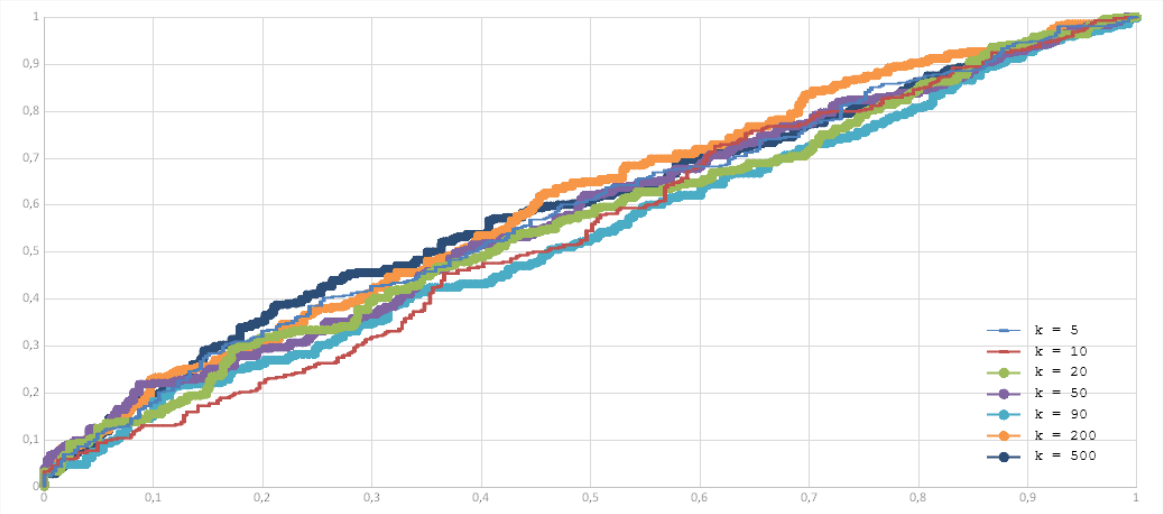

Abbildung 6.3: ROC-Kurven für die Autoencoder-Features und verschiedene k aus Experiment 1.

die *global memory* Varianten. Die Werte der *shared memory* Versionen sind die Gleichen, soweit diese erfasst werden konnten (bei SIFT sind maximal 90 Cluster und bei den Autoencoder-Features maximal 326 Cluster möglich). In den Grafiken wird das Verhältnis der richtig erkannten Kategorien (*True Positives*) zu falsch erkannten Kategorien (*False Negatives*). Hieraus ist ersichtlich, dass die Klassifizierung der SIFT- und Autoencoder-Features mäßig erfolgreich war: Ein Kurve nahe der 45° Linie sagt aus, dass das Verhältnis zwischen *TP* und *FP* ausgeglichen ist. In diesem Fall beträgt die Fläche unter der Kurve (*Area Under Curve, AUC*) 50%. Der *AUC* für die verschiedenen

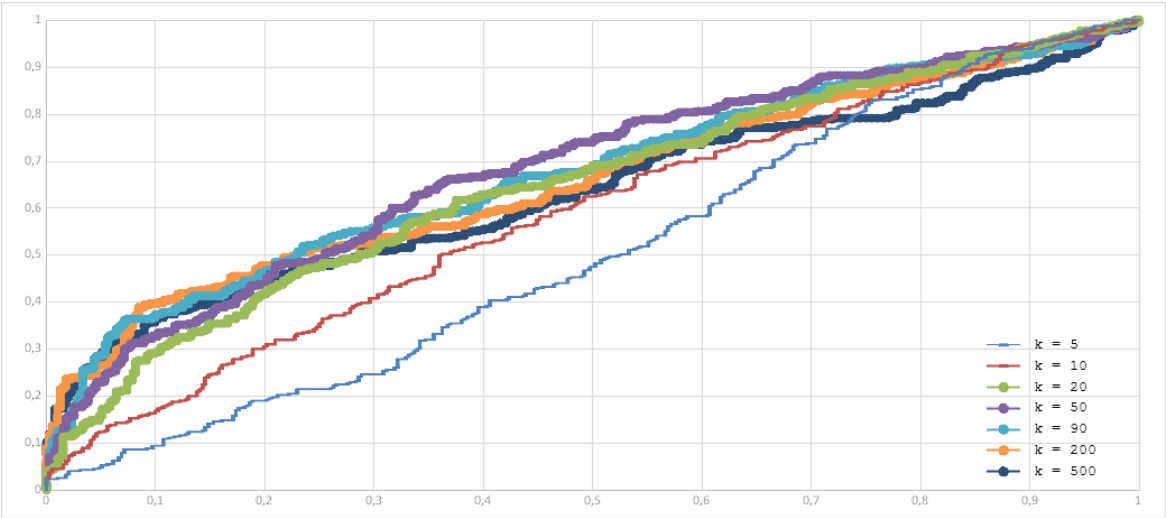

Abbildung 6.4: ROC-Kurven für die SIFT-Features und verschiedene k aus Experiment 2.

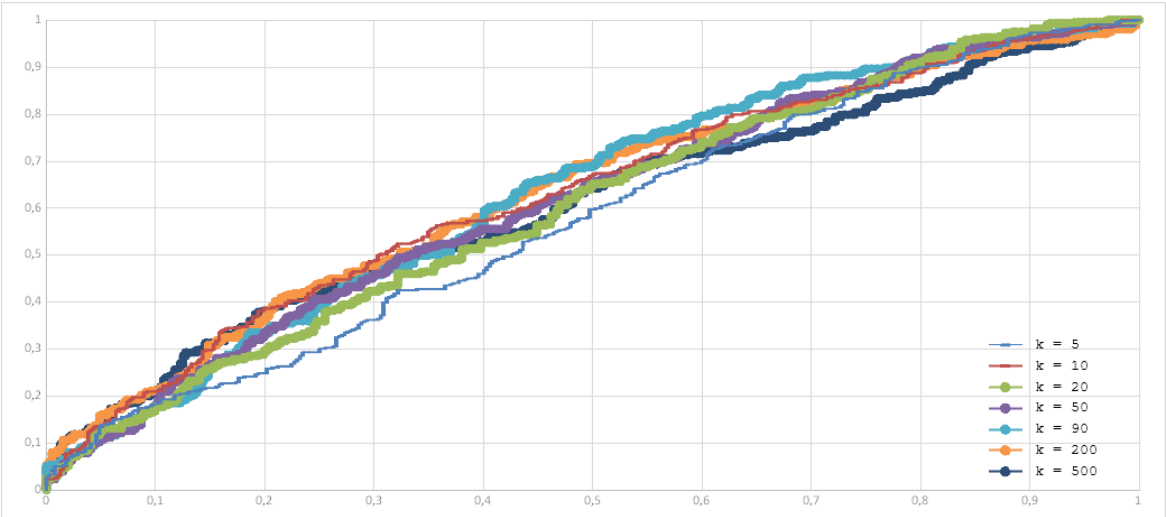

Abbildung 6.5: ROC-Kurven für die Autoencoder-Features und verschiedene k aus Experiment 2.

Konfigurationen ist in Tabelle 6.1 dargestellt. Die Fläche unter der ROC-Kurve beträgt in Experiment 1 für SIFT, je nach k, zwischen 56.73% und 67.7%. Die Features des Autoencoders schneiden im gleichen Test meist schlechter ab. Mit einem Bestwert von 60.49% bei 200 Clustern, ist eine Klassifizierung nicht erfolgreich gewesen.
Ein ähnliches Bild zeichnet sich bei den anderen beiden Testgruppen ab: Meist weisen die SIFT-Features eine höhere AUC als die Autoencoder-Features auf, die Bestwerte für ein Experiment erzielen stets die SIFT-Features (68.31% in Experiment 2 und 69.65% in Experiment 3).

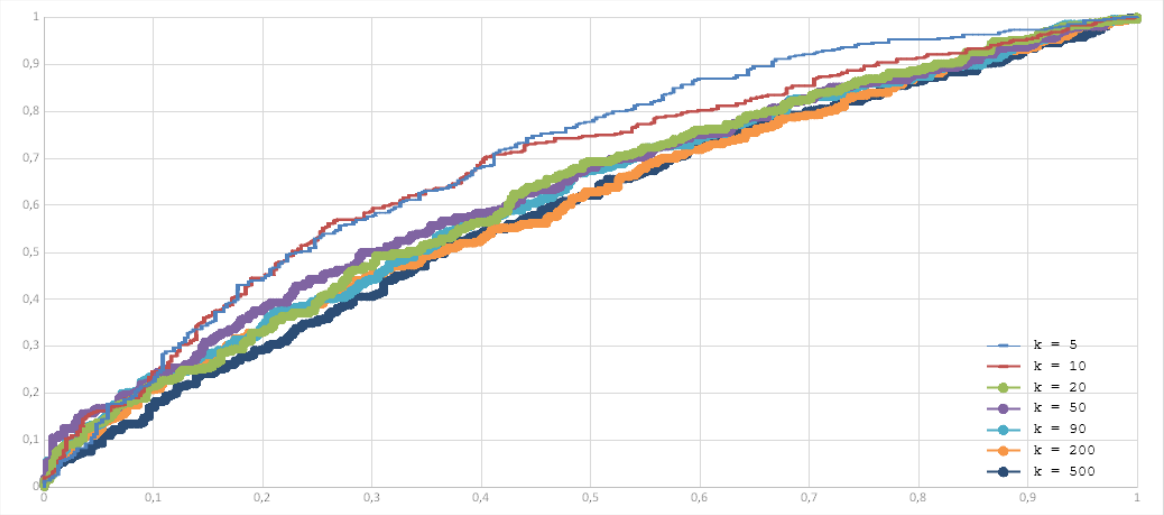

Abbildung 6.6: ROC-Kurven für die SIFT-Features und verschiedene k aus Experiment 3.

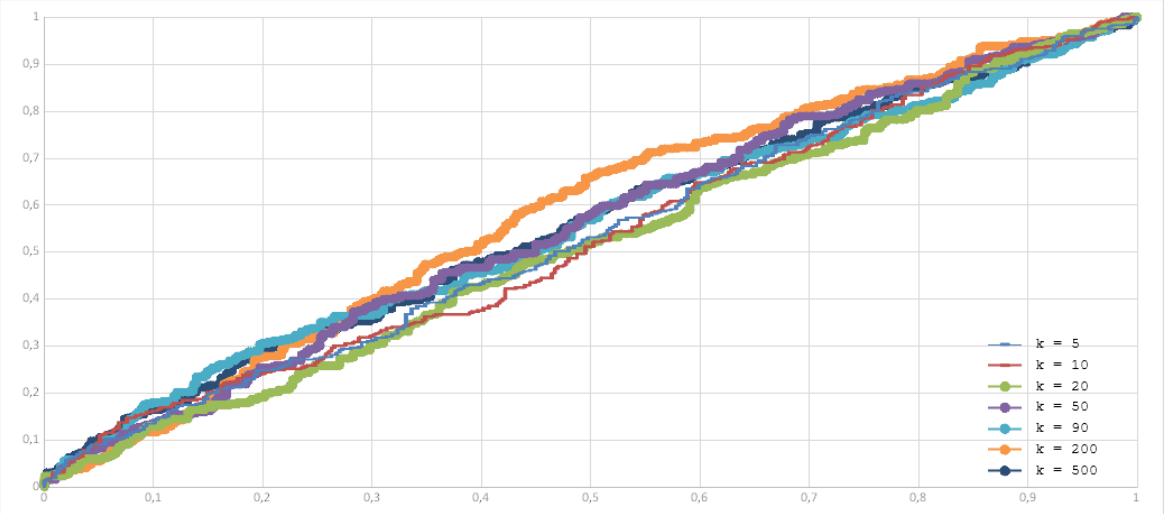

Abbildung 6.7: ROC-Kurven für die Autoencoder-Features und verschiedene k aus Experiment 3.

Auffallend ist auch, dass in Experiment 2 gegenüber Experiment 1, sowohl die Klassifizierung auf Basis der SIFT- als auch der Autoencoder-Features, erfolgreicher war. Der Aufbau beider Experimente ist recht ähnlich, nur das in Experiment zwei die doppelte Menge an Features für das Training verwendet wurde. Die Bestwerte steigen nur ein wenig (+0.61 für SIFT, +2.75 für den Autoencoder), doch die meisten Clusterkonfigurationen weisen bessere Ergebnisse auf.
Überraschend ist, dass in Experiment 3 die Klassifizierung für kleine k, bei Verwendung der SIFT-Features, am erfolgreichsten ist ($k = 5$ ist die beste Konfiguration mit 69.65%

	Experiment 1		Experiment 2		Experiment 3	
k	SIFT	AE	SIFT	AE	SIFT	AE
5	56.73	58.50	50.49	57.68	**69.65**	53.12
10	60.43	54.56	58.56	62.99	67.81	52.74
20	59.62	56.37	64.91	60.34	62.29	51.37
50	**67.70**	57.78	**68.31**	61.52	63.18	55.62
90	63.59	53.62	67.47	63.19	61.77	55.11
200	62.80	**60.49**	66.04	**63.24**	60.02	**58.17**
500	58.90	59.56	63.39	60.70	58.84	55.67

Tabelle 6.1: *Area Under Curve (AUC)* der ROC-Kurven in Prozent für alle drei Experimente.

AUC). Alle anderen Experimente lassen erkennen, dass eine Verwendung von 50 bis 200 Clustern die besten Ergebnisse liefert.
Allgemein beträgt der *AUC* in allen Experimenten zwischen 50% und 70% und die Kurven verlaufen meist nah beieinander. Nur in Abbildung 6.5 ist zu erkennen, das weniger als 20 Cluster deutlich schlechter gegenüber Modellen mit mehr Clustern abschneiden.

6.3.2 Laufzeiten

Für alle drei Experimente wurden sowohl die Laufzeiten der *global* als auch der *shared memory* Implementierung gemessen. Auf diese Weise können nicht nur die Laufzeiten der verschiedenen Features (SIFT, Autoencoder) gegenübergestellt werden, sondern auch überprüft werden, ob die Verwendung von *shared memory* den Algorithmus beschleunigt. Da der durch den Autoencoder erzeugt Deskriptor ca. 3.5 mal kleiner als der SIFT-Deskriptor ist, kann ein Die Werte sind über zehn Testläufe gemittelt.
In Tabelle 6.2 sind die Laufzeiten der *global memory* Variante für beide Feature-Typen dargestellt. Es ist deutlich zu erkennen, dass das Clustering der Autoencoder-Features in allen Fällen, großteils sogar sehr deutlich, schneller abgeschlossen ist, als das Clustering der SIFT-Features. In Experiment 1 und 2 ist der Prozess im Schnitt 4 mal schneller abgeschlossen, in Experiment 3 im Schnitt nur ca. 2.8 mal schneller.
Tabelle 6.3 zeigt die Laufzeiten der *shared memory* Implementierung. Auch hier ist das Clustering der Autoencoder-Features in Experiment 1 und 2 um einen ähnlichen Faktor schneller: Das Gruppieren des SIFT-Features ist in Experiment 1 ca. 4.6 mal langsamer, in Experiment 2 4.5 mal. Für verschiedene k schwankt die Reduzierung der Berechnungsdauer enorm: So ist in Experiment 2 das Clustering der Autoencoder-Features für $k = 20$ gerade mal 1.8 mal schneller, für $k = 50$ hingegen fast 7 mal schneller. Auch hier ist in Experiment 3 die Berechnung nicht so stark beschleunigt worden, wie in Experiment 1 und 2. Mit einem Faktor von 3.4 ist dies aber immer noch näher an dem erwarteten Wert von 3.5, als Experiment 3 der *global memory* Variante.

	Experiment 1		Experiment 2		Experiment 3	
k	SIFT	AE	SIFT	AE	SIFT	AE
5	1.29	0.51	3.62	0.78	1.51	0.57
10	2.57	0.95	12.44	3.00	3.45	1.05
20	7.96	1.70	10.06	5.19	7.54	3.08
50	14.41	2.93	34.13	4.95	13.05	5.04
90	15.24	4.22	52.64	9.78	28.12	9.46
200	22.30	4.86	67.24	14.13	69.52	12.33
500	37.81	6.52	100.92	19.87	72.08	17.44

Tabelle 6.2: Laufzeiten der *global memory* Implementierung des Bag of Visual Words in Sekunden.

	Experiment 1		Experiment 2		Experiment 3	
k	SIFT	AE	SIFT	AE	SIFT	AE
5	1.29	0.45	3.02	0.76	1.51	0.57
10	2.58	0.79	10.27	2.73	3.41	1.04
20	7.91	1.37	8.28	4.68	7.45	3.04
50	14.34	2.28	30.66	4.44	12.95	5.02
90	15.15	3.23	46.93	8.54	26.02	9.39
200	-	3.70	-	12.47	-	12.26

Tabelle 6.3: Laufzeiten der *shared memory* Implementierung des Bag of Visual Words in Sekunden.

7 Fazit und Ausblick

In dieser Arbeit wurde demonstriert, wie ein Gruppieren von Bilddaten mittels unüberwachter Lernverfahren realisiert werden kann. Um den großen Datenmengen gerecht zu werden und eine annehmbare Ausführungszeit zu erzielen, wurde das Modell auf eine parallele Verarbeitung durch Nvidia Grafikkarten ausgelegt. Für die Gruppierung der Features wurden zwei Ansätze getestet: Zum einen ein Clustering auf Basis von SIFT-Features, zum anderen auf Basis von Features, die durch einen Autoencoder komprimiert wurden. In drei Experimenten wurde getestet, wie erfolgreich eine Klassifizierung von Bildern durch diese Features durchgeführt werden kann. Mit knapp 70% in allen drei Experimenten, liefert SIFT hier deutlich bessere Ergebnisse als die Autoencoder-Features (zwischen 60% und 63%). Erfolgreich ist die Klassifizierung letztendlich nicht: Knapp jedes dritte Bild würde falsch kategorisiert werden.
Das aktuelle Modell besitzt viele Parameter und bietet somit eine Vielzahl an weiteren Möglichkeiten, die getestet werden können. Beim Autoencoder kann die Anzahl an der Iterationen im Training oder die Lernrate variiert werden, um den Einfluss auf die Ergebnisse zu beobachten. Beim Bag of Visual Words kann ebenfalls die Anzahl an Trainingsiterationen sowie der Wert für das Konvergenzkriterum angepasst werden. Außerdem kann studiert werden, wie sich die Ergebnisse der Klassifizierung verändern, wenn als Schwellwert für eine positive Klassifizierung andere Werte als 0.8 verwendet werden.
Der aktuelle Stand der Implementierung dient als *Proof of Concept*. Um eine praktische Verwendung zu ermöglichen, sollte der „technische Bruch“ beseitigt werden: Aktuell findet die Kompression der Daten durch TensorFlow statt, das Clustering hingegen ist in CUDA C geschrieben. Eine reine TensorFlow oder CUDA Implementierung ist nicht nur leichter zu überschauen und zu warten, sondern kann auch unnötige Datentransfers eliminieren. Hier ist zu erwarten, dass eine Umsetzung in TensorFlow weniger komplex ist als in CUDA. CUDA hingegeben bietet mehr Möglichkeiten zur Optimierung, um den Algorithmus zu beschleunigen. Momentan ist die *shared memory* Implementierung abhängig von der Anzahl der Cluster k. Um dies zu umgehen, kann ein Mechanismus zum Laden der Daten in Gruppen (*chunks*) implementiert werden: Ist nicht genug *shared memory* vorhanden, werden die Daten in *chunks* eingeteilt und nacheinander verarbeitet.

Literaturverzeichnis

[1] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, Nov. 2004.

[2] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via cuda," in *First International Conference on Intensive Applications and Services*, pp. 7–15, 2009.

[3] NVIDIA, *NVIDIA CUDA C Programming Guide*. NVIDIA, 2012.

[4] C. Zhao, *An Autoencoder-Based Image Descriptor for Image Matching and Retrieval*. PhD dissertation, Wright State University, 2016.

[5] A. I. Awad and M. Hassaballah, *Image Feature Detectors and Descriptors: Foundations and Applications*. Springer Publishing Company, Incorporated, 1st ed., 2016.

[6] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pp. 886–893, IEEE Computer Society, 2005.

[7] D. Kriesel, *A Brief Introduction to Neural Networks*. 2007.

[8] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *Computing Research Repository*, vol. abs/1206.5533, 2012.

[9] A. Zell, *Simulation neuronaler Netze*. Oldenbourg, 1997.

[10] G. E. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," pp. 504–507, 2006.

[11] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *The Journal of Machine Learning Research*, vol. 11, pp. 3371–3408, Dec. 2010.

[12] A. Faheema and S. Rakshit, "Feature selection using bag-of-visual-words representation," Advance Computing Conference (IACC), IEEE 2nd International, 2010.

[13] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011.

[14] M. Yedla, S. R. Pathakota, and T. M. Srinivasa, "Enhanced k-means clustering algorithm with improved initial center," pp. 121–125, 2010.

[15] G. Csurka, C. Bray, C. Dance, and L. Fan, "Visual categorization with bags of keypoints," *Workshop on Statistical Learning in Computer Vision, ECCV*, pp. 1–22, 2004.

[16] S. Starik, Y. Seldin, and M. Werman, "Unsupervised clustering of images using their joint segmentation," 2003.

[17] S. Sabour, N. Frosst, and G. Hinton, "Dynamic routing between capsules," 2017.

[18] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *International Journal of Computer Vision*, vol. 42, pp. 145–175, May 2001.

[19] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, pp. 1615–1630, Oct. 2005.

[20] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories," Conference on Computer Vision and Pattern Recognition, Workshop on Generative-Model Based Vision, 2004.