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“

15. Mai 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
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 15. Mai 2017 Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1. Beschreibung der Kapitel, des Ablaufs

1.1 Motivation

1. Entwicklung einer Bildsuche / eines Verfahrens zur Klassifizierung von Bilder

1.2 Ziele der Arbeit

1. Vergleich etablierter Verfahren (BoVW / AE)
2. Nutzung von Grafikkarten für die Berechnung

2 Grundlagen

Das Grundlagenkapitel gibt eine Einführung und Übersicht über die verwendeten Begriffe und Konzepte, auf die im Weiteren der Arbeit zurückgegriffen wird. Zunächst wird die Darstellung von Bildern behandelt und anschließend die Erhebung von charakteristischen Merkmalen aus Bildern, den Features. Zur Detektion und Extraktion von Features aus Bildern haben sich zahlreiche verschiedene Verfahren etabliert. Von diesen wird der SIFT Feature Detektor und Deskriptor näher betrachtet, da er im Weiteren als Basis für die Feature Gewinnung dient. Es werden anschließend zwei Modelle vorgestellt, die eine Klassifikation von Bildern auf Basis von Features ermöglichen. Das Bag of Visual Words Modell wurde aus dem Bereich Information Retrival adaptiert. Es wird anhand der Features von Trainingsbildern eins visuelles Vokabular gelernt, dass zur Klassifizierung von Bildern dient. Alternativ zu diesem Ansatz wird der Autoencoder vorgestellt. Ein Autoencoder ist ein spezielles neuronales Netzwerk, dass selbständig eine komprimierte Darstellung der Eingabe, in diesem Fall der Features, lernt. Im letzten Teil wird auf die Berechnung allgemein mathematischer Probleme auf Grafikkarten, das GPGPU Programming eingegangen. Durch den Einsatz von Grafikkarten können Berechnungen gerade bei großen Datenmengen stark beschleunigt werden, da diese massiv parallel auf den Daten arbeiten. Mit Nvidias cuda wird eine Sprache eingeführt, mit der sich Modelle wie der Bag of Visual Words und Autoencoder auf Nvidia Grafikkarten ausführen lassen.

2.1 Bilder und Features

Zunächst wird definiert wie ein Bild mathematisch aufgefasst wird um eine effiziente Verarbeitung zu ermöglichen. Jedes Verfahren erwartet 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 $I(x, y)$. Die Anzahl der Spalten m und Zeilen n entspricht dabei den Dimensionen des Bildes in Pixeln. Hier bezeichnen (x, y) diskrete Koordinaten der Matrix und somit die einzelnen Pixel des Bildes. Die Darstellung in Matrixform eignet sich sehr gut für Transformationen und Analysen des Bildes. Solche Verfahren betrachten oft jeden Pixel und seine Nachbarschaft. Die direkte Nachbarschaft eines Pixels ist eine 3×3 Matrix (mit dem Pixel als Zentrum), die alle unmittelbar anliegenden Pixel beinhaltet.

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

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 nur in Graustufe dargestellt, daher besitzt ein Pixel genau einen Intensitätswert.
- **Multispektrale Bilder** Jeder Pixel besitzt einen Vektor an Werten. Im Falle eines Farbbildes enthält der Vektor drei Intensitätswerte für rot, grün und blau.

Die Intensität eines Pixels bzw. die Intensität seiner Vektoren wird mit acht Bit dargestellt und umfasst daher 256 mögliche Werte. Diese Werte werden im Folgenden normalisiert im Intervall $[0, 1]$ angegeben.

2.1.2 Feature Detektion und Extraktion

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

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

sind. Abbildung 2.1 zeigt dasselbe Objekt, jedoch rotiert, skaliert und verschoben. Ein Algorithmus sollte mit hoher Wahrscheinlichkeit erkennen, dass es sich hier um dasselbe Objekt handelt.

Die Gewinnung der Features ist in zwei Schritte aufgeteilt:

- **Detektion** Zuerst ermittelt ein Detektor Muster im Bild. Die untersuchten Muster sind abhängig vom Detektor Pixel, Linien oder Regionen einer Nachbarschaft. Hierfür wird jeder Pixel und seine Nachbarschaft betrachtet und entschieden ob es sich um einen *key point* handelt. Ein Detektor gibt als Ergebnis alle Pixel zurück, bei denen es sich um key points handelt. Manche Verfahren geben zusätzlich zu den *key points* auch Eigenschaften wie die Orientierung oder den Maßstab aus. 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 kann spezieller Annahmen treffen, als einer für eine allgemeine Bildsuche.
- **Extraktion** Die Feature Extraktion erzeugt aus den vom Detektor gefundenen *key points* den Deskriptor. Ein Feature Deskriptor ist eine kompakte Darstellung eines Features. Die *key points* und deren Eigenschaften werden in Zahlen kodiert. Oft wird hier nicht nur ein Feature erzeugt, sondern ein Feature-Vektor, der auch Deskriptoren der Nachbarschaft des *key points* enthält. Von einem Deskriptor ist es wünschenswert, dass er affin invariant ist. Somit kann das Feature auch erkannt werden, wenn das Bild rotiert, verschoben oder skaliert wurde.

Abbildung 2.2: Gefundene *key points* in einem Bild farblich hervorgehoben (SIFT Detektor)

Im Folgenden werden zwei Verfahren zur Feature Detektion und Extraktion vorgestellt, die sich in der Praxis bewährt haben und teilweise affine Invarianz aufweisen. Das Histogram of Oriented Gradients ist ein weit verbreiteter Feature Deskriptor, der sich im Bereich der Objektdetektierung bewährt hat. SIFT ist ein von Lowe entwickeltes Verfahren zur Feature Extraktion und Detektion. Bei der Detektion von Features werden Rotationen, Skalierungen und Translationen von gefundenen Features erkannt und diese durch den Deskriptor als ein Histogramm der Orientierungen dargestellt.[HAA16]

2.1.3 Histogram of Oriented Gradients

Ein Histogramm ist eine diskrete Funktion, welche die Häufigkeitsverteilung 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 und der Anzahl der Klasse 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 Klassen 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 die Anzahl der Werte einer Klasse durch die Anzahl der Gesamtwerte dividiert wurde. In Abbildung 2.3 sind im Wesentlichen zwei Bereiche zu erkennen: ein

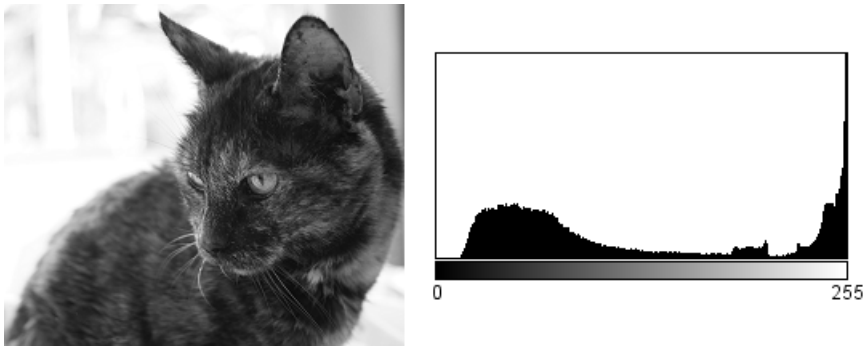

Abbildung 2.3: Graustufenbild und Verteilung der Intensitätswerte

sehr hellerer 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.

Das Histogramm of Oriented Gradients beschreibt die Änderung der Intensität von einer Menge benachbarter Pixel in einer Richtung an. 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 TODO: Abbildung ist links das Fenster dargestellt und rechts die entsprechenden Gradienten der Zellen.

TODO: Abbildung

2.1.4 Scale-invariant feature transform

SIFT ist ein Feature Detektor und Deskriptor der 1999 von Lowe entwickelt wurde. Neben der Invarianz der Skalierung der Features, berücksichtigt SIFT auch die Rotation, Beleuchtung und Perspektive. *keypoints*. Der Algorithmus lässt sich in vier wesentliche Schritte unterteilt:

1. **Scale space** Im ersten Schritt wird die Invarianz der Skalierung behandelt. SIFT konstruiert hierfür einen *scale space*. Hier werden aus dem Originalbild immer verzerrte Versionen durch den Gaussian Operator erzeugt. Die Größe der neuen Bilder

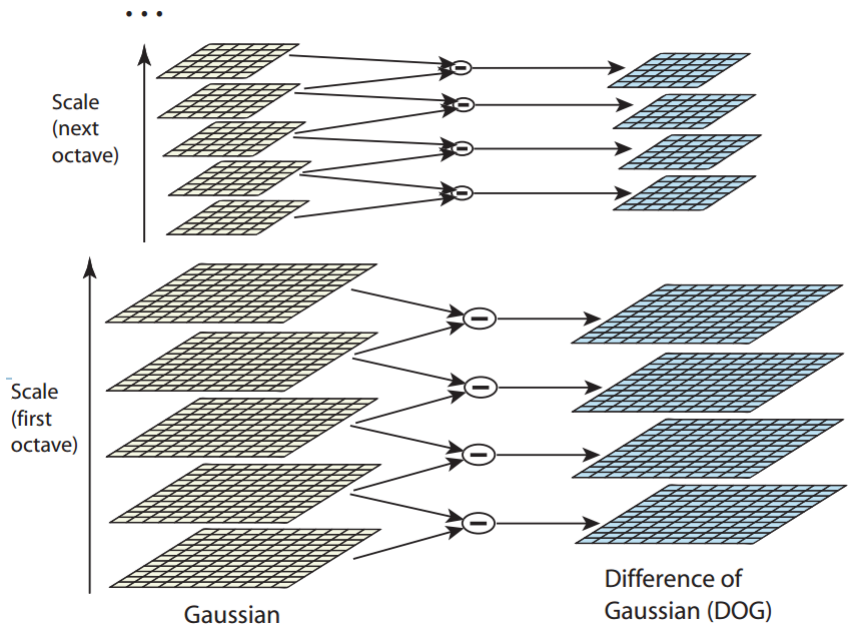

Abbildung 2.4: Difference of Gaussians Operator, Abbildung aus [Low04]

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 Kandidaten für *keypoints* sind, wird nun zwischen allen aufeinanderfolgenden Bildern einer Oktave die Differenz gebildet. Das Prinzip ist für zwei Oktaven in Abbildung 2.4 dargestellt. Das Ergebnis ist eine Approximation des Laplacian of Gaussians, 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 Nachfolgendem im *scale space* betrachtet. Da die Extremwerte nicht immer exakt auf einem Pixel liegen, muss die genau Position noch berechnet werden. SIFT verwendet hierfür eine Taylor Expansion vom angenäherten *keypoint* ausgehend.

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

4. **Deskriptor** Für jeden durch den Detektor gefundenen *keypoint* wird nun ein Featurevektor gebildet. Der Featurevektor enthält Informationen über die Nachbarschaft 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 Dimensionen aufweist.

SIFT ist äußerst robust, da Änderungen im Grenzwert von Position und Orientierung den Feature Vektor kaum beeinflussen. Der Deskriptor besitzt zwar keine affine Invarianz, in praktischen Anwendungen werden jedoch auch mit skalierten, rotierten und verschobenen Objekten gute Ergebnisse erzielt. Die Konstruktion des Deskriptors ist allerdings sehr aufwendig und er weist eine hohe Dimensionalität auf. [Low04]

TODO: SIFT-PCA erwähnen

2.2 k-means Clustering

Unter k-means Clustering werden Algorithmen zusammengefasst, die eine Menge Vektoren durch Zuweisung in k vorgegeben 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, dessen Varianz sich bei Aufnahme des Vektors am Wenigsten erhöht. Aus diesem Grund setzt das k-means Clustering einen euklidischen Raum voraus. 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. Zunächst beginnt Llyods Algorithmus 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 ändert, oder eine maximale Anzahl an Iterationen erreicht wurde:

1. **Initialisierung** Es werden zunächst k zufällige Vektoren als Schwerpunkte der Cluster ausgewählt.
2. **Zuordnung**: Von den verbleibende Vektor wird nun mit jedem Cluster 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 den Cluster werden neu berechnet, um den neu zugeordneten Vektor in Folgeberechnungen miteinzubeziehen.

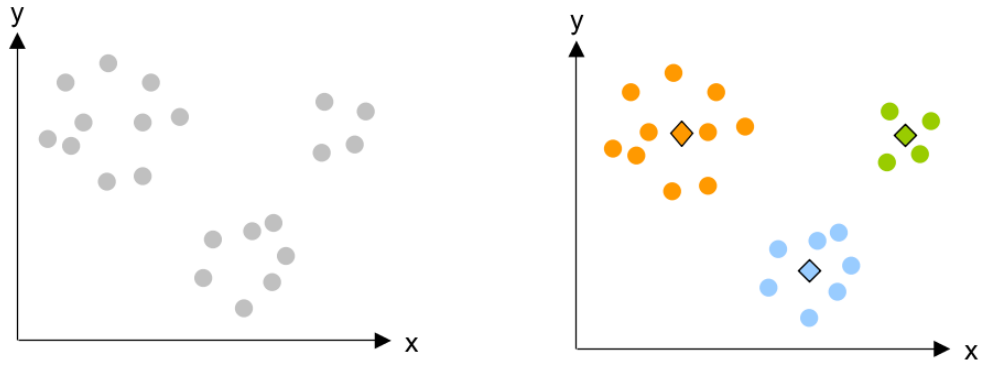

Abbildung 2.5: K-means Clustering im zweidimensionalen Raum mit $k = 3$.

Zur geometrischen Veranschaulichung sind in Abbildung 2.5 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 abähngig von den Daten nicht immer sinvolle Cluster gebildet werden können. [RU11]

2.3 Bag of Visual Words

Das Bag of Words Modell stammt aus dem Bereich Information Retrival und wird zur Klassifizierung von Dokumenten genutzt. Es wird das Auftreten jedes Wortes in eine Dokument gezählt und durch die Anzahl aller Wörter im Vokabular dividiert, um so einen normalisierten Wert zu erhalten, welcher die relative Häufigkeit eines Wortes angibt. Das Vokabular wird *Codebook* genannt, die Wörter sind die *Codewords*. Dieses Modell wurde von der Computer Vision adaptiert um Bilder zu klassifizieren. Die Features können aber nicht direkt statt der Worthäufigkeit verwendent werden: Ein Wort ist ein diskreter Wert der direkt verglichen werden kann, ein Feature hingegen ist ein hochdimensionaler Vektor, der Eigenschaften beschreibt. Um konkrete Werte zu erhalten, ist es notwendig die Vektoren zu quantisieren. Die quantisierten Vektoren entsprechen dann den *Codewords* und werden in diesem Kontext auch *Visual Words* genannt. Um *Visual Words* aus einer Menge von Trainingsbildern zu erhalten, werden zunächst Featurevektoren aus den Bildern extrahiert. Die Features werden dann durch ein Clusteringverfahren gruppiert. Die Idee ist, dass ähnliche 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 ein 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*. Auf Basis des *Codebooks* kann der Bag of Words von Bildern erzeugt werden. Hierfür werden wieder die Features eines Bildes extrahiert. Aus diesen Daten wird nun eine Histogrammdarstellung des Bildes erzeugt: Für jedes Feature wird das ähnlichste *Visual Word* des *Codebooks* bestimmt und die entsprechende Klasse inkrementiert. Die Summen aller Klassen werden dann durch die Gesamtanzahl an *Visual Words* dividiert, um die relativen Häufigkeiten zu erhalten.

TODO: Abbildung Prozess

TODO: Quelle

2.4 Autoencoder

Ein Autoencoder ist ein spezielles neuronales Netzwerk. Diese Netze werden für das unbeaufsichtigte Lernen einer komprimierten Darstellung von Daten genutzt. Zunächst wird hierfür ein Überblick über das Themengebiet der neuronale Netze im Allgemeinen gegeben und anschließend die Funktionsweise eines Autoencoders erläutert. Darauf aufbauend werden zwei Erweiterungen des Autoencoders vorgestellt, um Rauschen in den Daten und tiefe Netzwerke berücksichtigen.

2.4.1 Neuronale Netze

Neuronale Netze (ANN, KNN, NN) 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 Deep Learning und Machine Learning finden neuronale Netze seit Mitte der 80er Jahre vermehrt praktische Anwendung und akademische Zuwendung. Dadurch das neuronale Netze von Natur aus hoch parallel arbeiten, eignen sie sich vor allem für parallele Architekturen und die Verarbeitung großer Datenmengen. Ein neuronales Netz besteht aus einer Menge Neuronen die in Schichten im Netzwerk angeordnet sind. Jedes Neuron besitzt einen Aktivierungszustand und einen Schwellwert, von denen abhängt, ob ein Signal weitergeleitet wird. Neuronen benachbarter Schichten sind vollständig durch gewichtete Kanten miteinander verbunden. Diese Beziehung wird in einer Gewichtsmatrix ausgedrückt. Neuronen zwischen denen dann keine Verbindung vorhanden ist, besitzen das Gewicht den Wert 0. Das Netz verarbeitet ein Signal, welches hier als Vektor $x \in [0,1]^n$ mit Dimension 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 Größe des Vektors angibt. Zwischen

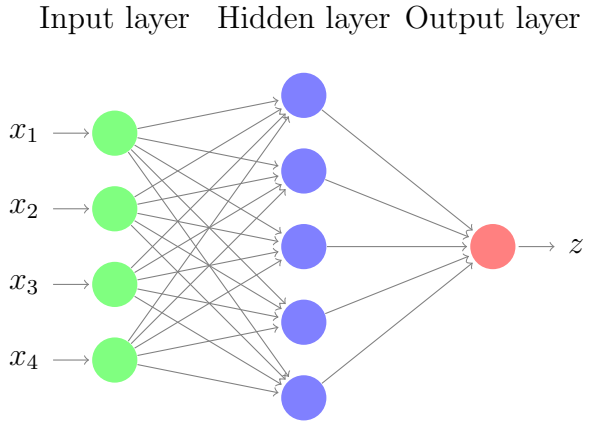

Abbildung 2.6: Beispiel eines dreischichtigen neuronalen Netzes.

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. In Abbildung 2.6 ist ein Netz mit drei Schichten dargestellt. Der *Input layer* nimmt einen vierdimensionalen Vektor 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.

Die Verarbeitung eines Signals in einem Neuron ist in Abbildung 2.7 schematisch dargestellt und lässt sich in drei Schritte untergliedern:

- **Propagierungsfunktion** Aus der Eingabe aller verbundenen Neuronen wird die Netzeingabe berechnet. Meist wird hier die gewichtete Summe zwischen Eingabe und Gewicht verwendet: $\sum_{i=1} w_i x_i + b$.
- **Aktivierungsfunktion** Es wird der neue Aktivierungszustand des Neurons aus dem alten Zustand und der Netzeingabe berechnet. Häufig wird hier die logistische Funktion $s(x) = \frac{1}{1+e^{-x}}$ verwendet.
- **Ausgabefunktion** Wird ein Neuron aktiviert, so wird das resultierende Signal durch die Ausgabefunktion berechnet und an alle Neuronen in der folgenden Schicht weitergeleitet. Oft wird hier die Identitätsfunktion verwendet.

Wurde ein neuronales Netz konstruiert, folgt darauf die Trainingsphase. Durch das Training ist es möglich, dass ein Netzwerk Neuronen oder Verbindungen hinzufügt bzw. entfernt, den Schwellwert für die Aktivierung von Neuronen verändert oder die Gewichte zwischen Neuronen anpasst. Pro Trainingselement wird die Fehlerquote F als die Summe der quadrierten Differenz zwischen dem angestrebten Ergebnis und der Ausgabe

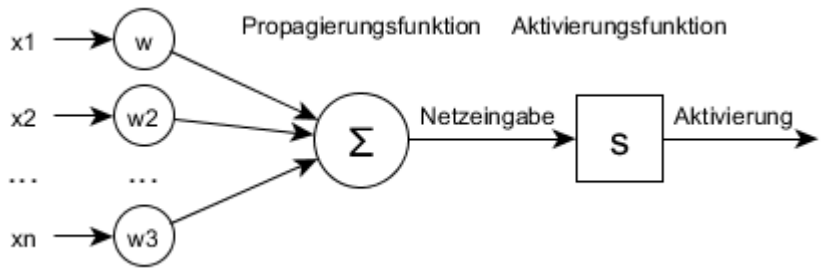

Abbildung 2.7: Verarbeitung eines Signales in einem Neuron (ohne Ausgabefunktion) (Grafik überarbeiten)

des Netzes berechnet. Um diese Veränderungen im Netz bekannt zu machen, wird das *Backpropagation* Verfahren genutzt. *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. **Berechnung** Die Summe der quadrierten Fehler wird berechnet.
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 aktualisiert.

TODO: Gradienten erläutern

2.4.2 Funktionsweise

Ein Autoencoder (AE) ist ein spezielles neuronales Netzwerk, dass eine komprimierte Kodierung der Eingabe lernt. Ein Autoencoder 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 Gewichtesmatrix aktualisiert werden. 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 *Hiddenlayer* ist geringer, um die komprimierte Darstellung des Features zu erreichen. Werden mehrere *Hiddenlayer* verwendet, so nimmt die Neuronenanzahl von Layer zu Layer ab um die Anzahl der Dimensionen 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

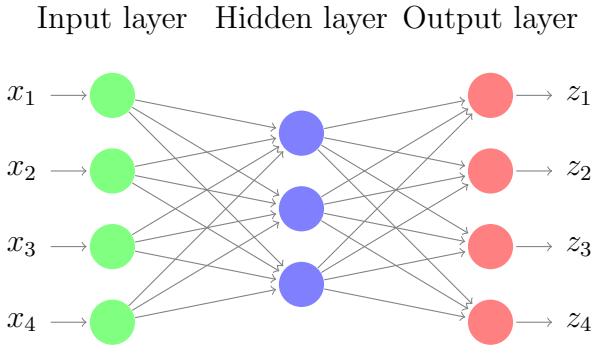

Abbildung 2.8: Beispiel eines simplen Autoencoders

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 $y \in [0,1]^p$ durch $y = encode_{W,b}(x) = s(Wx + b)$ abgebildet. W ist die Gewichtsmatrix $n \times p$ und b der Bias-Vektor. Diese Parameter werden durch den Autoencoder optimiert. Die Rekonstruktion erfolgt durch die Dekodierungsfunktion: $z \in [0,1]^n$ wird dann durch $z = decode_{W',b'}(y) = s(W'y + b')$ berechnet. [Zel97].

In Abbildung 2.8 ist ein Autoencoder abgebildet der als Eingabe einen Vektor $x \in [0,1]^4$ entgegen nimmt. Dieser wird auf einen dreielementigen Vektor y abbildet, da der *Hidden layer* drei Neuronen enthält. Die Rekonstruktion von x aus y erfolgt dann durch die Berechnung der Dekodierungsfunktion.

2.4.3 Stacked Denoising Autoencoder

Von Hinton and Salakhutdinov wurde 2006 das Konzept des Stacked Autoencoders eingeführt, um einige Probleme mit herkömmlichen Autoencoder zu überwinden. Bei Netzwerken mit mehr als einem Hidden Layer erzielt die Gradient Descent Methode bei der Rückpropagierung keine guten Ergebnisse mehr, aufgrund der zunehmenden Verzerrung der Gradienten. 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. Hierfür wird im Training jedes Paar aneinanderliegender Schichten als ein Autoencoder aufgefasst. Das Pretraining besteht aus drei Schritten, die wiederholt werden, bis alle Autoencoder trainiert sind [VLL+10].

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.

TODO: Abbildungen

TODO: Denoising Autoencoder

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. Das Umsetzen von Programmen auf Grafikkarten, die nicht mit der Verarbeitung von grafischen Daten in Zusammenhang stehen, wird GPGPU (General Programming on Graphics Processing Units) genannt. Die meisten GPGPU 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.

2.5.1 Ausführungsmodell

Ein Programm, dass auf einer Nvidia Grafikkarte ausgeführt werden soll, muss in der cuda Sprache geschrieben sein. Aus der Sicht eines Programmierers handelt es sich bei cuda 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 bzw. cubin binary code übersetzt. Nvidia hat das SPMD Modell durch *kernels* umgesetzt. Ein *kernel* ist ein Programm, dass 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 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

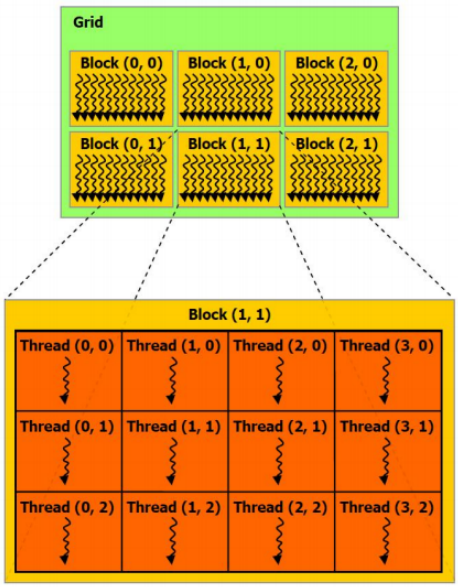

Abbildung 2.9: Organisierung von Threads in Blocks in Grids REF cuda

als Vektor oder Matrix vor. Der Zugriff auf verschiedene Elemente durch unterschiedliche *kernel* erfolgt dann durch eine Indexberechnung. Diese wird anhand des Beispiels der Vektor Addition näher erläutert. Beim Aufruf des Kernels muss die Anzahl der Threads pro Block und die Anzahl der Blocks in einem Grid angegeben werden. 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. In Abbildung 2.9 ist ein eindimensionales Grid mit sechs eindimensionalen Blöcken á 12 Threads schematisch dargestellt. Ein Block kann auf aktuellen Grafikkarten bis zu 1024 Threads beinhalten.

2.5.2 Speicherverwaltung

Dadurch das 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 2.10 dargestellt. Jeder Multiprozessor hat Zugriff auf den *global memory* sowie einen eigenen lokalen *shared memory*, *Constant* und *Texture Cache*. Lokal erstellten 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

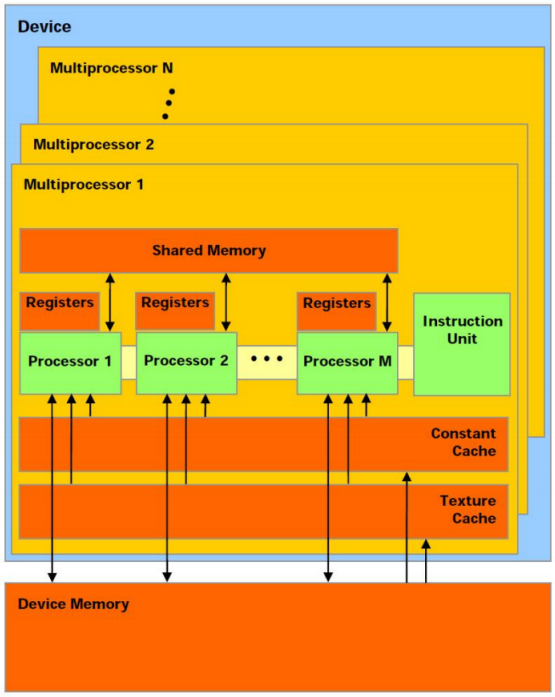

Abbildung 2.10: Organisierung der verschiedenen Speichertypen

synchronisiert werden müssen. Durch Verwendung des *shared memory* und *texture cache* können Speicherzugriffe beschleunigt werden, jedoch können Daten aus diesen Bereichen nicht zwischen Böcken geteilt werden.

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. 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.

texture memory (oder unified)

TODO: Recherche, Text

2.5.3 Vektor Addition

Am Beispiel der Vektor Addition soll der Aufbau und Ablauf eines cuda Programms illustriert werden. Hierfür wird zunächst die Funktion *vecAdd* angelegt. Hier wird in Zeile 8 - 10 durch *cudaMalloc* der benötigte Speicher für die Vektoren und das Ergebnis allokiert. Anschließend werden die Daten von Vektor a und b zum *device* kopiert. 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   vecAddKernel<<<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 }
```

3 Analyse

Ziel der Arbeit ist es, dass ein Anwender die in der HsH Datenbank vorhandenen Bilddaten in eine Menge von k verschiedenen Gruppen einteilen kann, um so semantische Informationen über die Bilder zu gewinnen. Das k kann hierbei vom Anwender vorgegeben werden um verschiedene Gruppen zu erhalten und so unterschiedliche Informationen in Bildern zu entdecken. In diesem Rahmen sollen zwei Methoden miteinander verglichen werden, die eine Gruppierung von Bildern auf Basis von sogenannten Feature Deskriptoren ermöglichen. Die so entstandenen Modelle erlauben es weitere Bilder zu labeln, d.h. die Gruppe zu bestimmen, der sie angehören. Eine wesentliche Eigenschaft die der Deskriptor aufweisen sollte, ist affine Invarianz. Verschiedene Objekte oder Merkmale die auf mehreren Bildern vorhanden sind, weise selten die gleiche Position auf, daher sollten Rotation, Skalierung und Translationen berücksichtigt werden. Gegenwärtig ist aber noch kein Deskriptor vorhanden, der eine Einbeziehung aller möglichen Umstände einbeziehen kann. Eine Reihe von geeigneten und verbreiteten Deskriptoren wird im Kapitel Feature Deskriptoren vorgestellt und diskutiert. Ein Vergleich von Bildern anhand von Features ist nicht unmittelbar möglich. Die Feature-Vektoren weisen sehr hohe Dimensionen auf, was eine effiziente Berechnung nicht möglich macht. Die Folgenden beiden Abschnitte behandeln zum Einen das Bag of Visual Words Modell und zum Anderen den Autoencoder. Beide Ansätze reduzieren auf unterschiedliche Weise die Dimensionen der Daten, um so eine schnellere Berechnung in einem TODO: kleineren Raum zu ermöglichen. Da zum Aufbauen der Modelle Millionen von Features verarbeitet werden müssen, wird bei der Betrachtung der beiden Ansätze geprüft, wie eine Beschleunigung der Berechnung 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.

TODO: cuda besser einführen, kurze cuda Sektion?

3.1 Feature Deskriptoren

Feature Deskriptoren enthalten Informationen von charakteristischen Bereichen in Bildern. Feature Deskriptoren kodieren weitaus mehr als nur die geometrische Position von Pixeln: Es wird beispielsweise die Beleuchtung und teilweise affinen Invarianz berücksichtigt.

In der Literatur finden sich verschiedene Ansätze, die je nach Einsatzgebiet unterschiedliche Stärken besitzen. Nachfolgend werden einige Feature Deskriptoren vorgestellt, die verbreitete Anwendung durch ihre praktischen Erfolge erzielt haben. 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 Dimension eines Vektors, desto größer wird der Speicherbedarf und Rechenaufwand. Neben einer kompakten Darstellung werden daher in der Praxis Verfahren verwendet um die Dimension weiter zu reduzieren.

TODO: Under construction - auch die folgenden Deskriptoren

3.1.1 Local Binary Patterns

Der Local Binary Pattern (LBP) Deskriptor erzeugt aus einem Bild eine Reihe Bitstrings als Deskriptoren. Die Länge der Bitstrings hängt hier von der verendeten Größe der Nachbarschaft eines Pixel ab. Original wurde eine 3×3 Nachbarschaft verwendet, was zu einer Länge von acht führt (nur die Nachbarschaft wird kodiert). Für jeden Pixel wird nun bestimmt ob seine Intensität kleiner oder größer im Vergleich zu einem Schwellwert ist. Abhängig vom Ergebnis wird eine 0 oder 1 kodiert. Im Fall der 3×3 Nachbarschaft ergibt dies 2^8 mögliche verschiedene Bitstrings, sodass der resultierende Featurevektor 256 Dimensionen aufweist. LBP wurden um Invarianz gegenüber Rotationen und die Verarbeitung von Farbbildern erweitert. Durch die Entwicklung dieser Methode wurden vor allem im Bereich der Gesichtserkennung wesentliche Fortschritte gemacht [TMT02].

3.1.2 Spatial Envelope

Einen ganz anderen Ansatz haben Torralba und Olivia verfolgt: Statt Objekte durch lokale Features zu beschreiben, werden globale Eigenschaften betrachtet. Das Bild wird in einem Raum mit wenig Dimensionen abgebildet, dem sogenannten *Spatial Envelope*. Die Autoren nutzen hier wahrnehmbare Dimensionen wie Natürlichkeit und Offenheit um den Raum zu definieren. 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. Bilder die in einer semantischen Kategorie Ähnlichkeiten aufweisen, liegen dann nah beieinander. Dieses Modell hat sich vor allem bewährt um eine Umgebung bzw. Landschaft zu klassifizieren. [OT01]

3.1.3 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 entwickelten und nutzten diesen Verfahren bereits 2005, mit großem Erfolg, um Menschen in Bilder zu erkennen. Obwohl beispielsweise SIFT auch ein lokales Histogramm um einen *keypoint* berechnet, werden beim HOG hingegen die Features des ganzen Bildes (bzw. Bereiches) berechnet, nicht nur von Nachbarschaften.

3.2 Lernverfahren

Ein System, dass ein Lernverfahren implementiert, wird genutzt um Muster in Datenströmen zu entdecken. Hierfür wird das System mit Trainingsdaten angelernt, um diese zu beurteilen und daraus Muster zu gewinnen. Wenn dieser Vorgang abgeschlossen ist, kann der Lernalgorithmus auf die eigentlichen Daten angewendet werden. In der Literatur wird zwischen verschiedene Lernmethoden unterschieden, die geläufigsten sind unüberwachtes und überwachtes Lernen:

- **Überwachtes Lernen (supervised Learning)** Ein Lehrer ist erforderlich, der überprüft, ob die Ausgabe des Netzes bezüglich des Eingabe korrekt ist. Das Netz lernt auf diese Weise Assoziationen zwischen den Ein- und Ausgaben herzustellen.
- **Unüberwachtes Lernen (unsupervised Learning)** Ziel unüberwachter Lernalgorithmen ist aus großen Mengen von nicht kategorisierten Daten versteckte Strukturen zu entdecken. Um dies zu erreichen werden die Daten oft quantisiert oder in eine simplere Darstellung überführt.

Für die Featuregewinnung eignen sich vor allem unüberwachte Lernverfahren, da pro Bild viele hochdimensionale Deskriptoren gefundenen werden. Zum einen muss die Menge der erzeugten Deskriptoren auf die Wesentlichen reduziert werden, zum anderen eignen sich die 128 dimensionalen SIFT Feature Vektoren nur bedingt für einen Vergleich. Eine Möglichkeit besteht also darin durch ein Clustering Verfahren die Deskriptoren in Kategorien zu quantisieren, sodass beim Labelling eines Bildes nur noch ein Bruchteil an Vergleichen durchgeführt werden muss. Ein anderer Ansatz ist das Verringern der Feature Dimensionalität. Verfahren wie beispielsweise die Hauptkomponentenanalyse bilden die Feature-Vektoren auf einen Raum niederer Dimensionen ab und wurden bereits speziell für SIFT adaptiert (SIFT-PCA). Durch den Aufschwung maschineller Lernverfahren wurden jüngst auch neuronale Netze für diesen Zweck adaptiert. Ein Autoencoder ist ein spezielles neuronales Netzwerk, dass sich für das unbeaufsichtigte Lernen einer komprimierten Darstellung von Daten verwenden lässt. Im Folgenden

werden zwei verschiedene unüberwachte Lernverfahren, dass Bag of Visual Words Modell und der Autoencoder, vorgestellt. Ziel ist es beide Verfahren zu Implementieren und die Ergebnisse gegenüberzustellen.

3.3 Ansatz 1: Bag of Visual Words

Im ersten Ansatz soll das Bag of Visual Words Modell genutzt werden. Zu Beginn liegen die Feature-Vektoren vor, die in der vorigen Phase extrahiert wurden. Um das Codebook aufzubauen ist es erforderlich die Visual Words zu generieren. Die Visual Words werden durch ein Clustering der Feature-Vektoren gewonnen, daher handelt es sich hier um ein unüberwachtes Lernverfahren. Als Clustering Algorithmus wird hier Llyods heuristische Variante des k-means Algorithmus verwendet. Zunächst 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.

3.3.1 Lloyds Algorithmus

Im Grundlagenkapitel wurde bereits Lloyds Algorithmus eingeführt, hier soll zunächst näher auf die sequentielle Ausführung eingegangen werden, um anschließend eine mögliche Parallelisierung zu diskutieren. Im Nachfolgendem Codelisting ist der Ablauf des Algorithmus in Pseudocode beschrieben. Als Parameter werden die Punkte P und die Anzahl der zu bildenden Cluster k erwartet. In Zeile 2 findet die Auswahl der initialen Schwerpunkte der Cluster statt. Die Zuordnung von Punkten zu Clustern erfolgt in Zeile 7: $argminD$ wählt den Cluster aus, dessen Varianz am wenigsten bei Aufnahme des Punktes p_i steigt. Abschließend wird die Aktualisierung der Schwerpunkte aller Cluster in Zeile 9 durchgeführt.

```
1 kmeans_lloyd (P,C,k)
2   initialisierung
3   until convergence
4     C_j = 0, j = 1, ..., k
5     for each p_i ∈ P
6       for each c_j ∈ C
7         c_j = argminD(c_j, p_i)
8     for each c_j ∈ C
9       c_j = 1/|c_j| ∑_{n_i ∈ c_j} n_i
```

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 Varianz unabhängig von der eines anderen ist, kann die Berechnung aller Varianzen parallel erfolgen.

TODO: Cluster Aktualisierung

3.3.2 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.

TODO: Histogramm für Cluster erläutern

```
1 histogram (P, C, H)
2   for each p_i ∈ P
3     for each c_j ∈ C
4       k = arminD(c_j, p_i)
5     H_k = H_k + 1
6   for 1..|H|
7     H_i = H_i/|H|
```

3.4 Ansatz 2: 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 diesen Wissen zu transferieren. Ein spezielles neuronales Netzwerk zum unbeaufsichtigten Lernen ist der Autoencoder. Hier soll dieser genutzt werden, um die Dimensionalität des Feature-Vektors zu reduzieren. In der Arbeit von [REF] wurde bereits vorgestellt, wie auf Basis eines Autoencoders ein Bilddeskriptor erzeugt werden kann. Es werden durch den SIFT Detektor die *interest points* eines Bildes ermittelt. Für jeden *intereset Point* werden die lokalen Gradienten in horizontale und vertikale Richtung einer 41×41 großen Nachbarschaft berechnet. Diese werden in einem Vektor der Größe $2 \times 39 \times 39 = 3042$ gespeichert. Der Encoder besteht aus fünf Stufen, um die Gradienten zu komprimieren.

TODO: Autoencoder für unbeaufsichtigtes Lernen, Einführung AE TODO: Auch hier gibt es ein Paper, das mir viel zum Autoencoder liefert, auch bereits hier

4 Konzept

Das Kapitel Konzeption beschäftigt sich zunächst mit dem Prozess der Feature Extraktion. Hierfür wird der SIFT Algorithmus nach Lowe genutzt. Folgend 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 SPMD Architekturen eignen. Anschließend wird der geplante Ablauf zum Generieren eines Bag of Visual Word Modells und Labelling eines Modells durch das Modell erläutert. Abschließend wird auf der Basis der Arbeit von [REF] ein Autoencoder eingeführt, der aus SIFT *keypoints* eine Darstellung des Features in nur 36 Dimensionen lernt.

4.1 Feature Extraktion

Die Extraktion der Features ist die Basis für beide Varianten der Klassifizierung. Das Bag of Visual Words Modell nutzt die von SIFT erzeugten Feature Deskriptor für die weitere Verarbeitung. Der Autoencoder hingegen arbeitet mit Gradienten der *keypoints* die vom SIFT Detektor ermittelt wurden. Aus diesem Grund werden die Feature-Vektoren und *keypoints* beide berechnet und getrennt gespeichert. Der SIFT Deskriptor enthält 128 Dimensionen und ist so für einen Vergleich nur schwer geeignet, da pro Bild ca. 100 bis 1000 Feature Vektoren generiert werden. Es werden daher im folgenden zwei Ansätze vorgestellt, die die Dimensionalität der Features reduzieren und eine durch Grafikkarten gestützte Berechnung von ähnlichen Bildern ermöglichen.

TODO: Was noch hier, oder überhaupt?

4.2 Ansatz 1: Bag of Visual Words

In der Analyse wurde bereits sequentielle Varianten des Lloyd und 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 für SIMD Prozessoren abgeleitet. In den beiden nachfolgenden Abschnitten Generierung des Modells und Labeling eines Bildes wird auf

den Programmaufbau und -ablauf näher eingegangen. Es werden die wesentlichen Funktionen, ihre Parameter und Aufrufe skizziert.

4.2.1 Parellisierung von Llyods Algorithmus

Der Thread in einem Block mit der ID 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 nimmt $\Theta(nk)$ Zeit in Anspruch, wobei n die Anzahl Vektoren und k die Anzahl der Cluster ist. Diese Phase kann parallelisiert werden, in dem pro Feature Vektor ein Thread verwendet wird: Jeder Thread berechnet für seinen Feature Vektor die Distanz zu allen Clusterschwerpunkten und bestimmt den Index des Clusters, der am Nächsten ist. Dieser Prozess ist in Pseudocode in Zeile 6 bis 8 ausgedrückt. Bevor die Cluster aktualisiert werden, müssen die Threads synchronisiert werden: Andernfalls ist nicht garantiert, dass die Berechnung jedes Threads abgeschlossen ist.

```
1  kmeans_gpu
2    if threadId == 0
3      c_j = rand(p_i) ∈ P, j = 1, ..., k, c_j ≠ c_i ∀i ≠ j
4    synchronize threads
5    until convergence
6      for each x_i ∈ P_threadId
7        l_i = argminD(c_j, p_i)
8      synchronize threads
9      if threadId == 0
10       for each p_i ∈ P
11         c_{l_i} = c_{l_i} + p_i
12         m_{l_i} = m_{l_i} + 1
13       for each c_j ∈ C
14         c_j = (1/m_j) c_i
```

4.2.2 Parallele Reduzierung von Histogrammen

TODO: Allgemeines parallel reduction Prinzip in SIMD (Pseudocode?)

4.2.3 Aufbau des Bag of Visual Words Algorithmus

Das Bag of Visual Words Modell soll zwei Anwendungsfälle unterstützen. Zunächst muss aus einer Menge von Bildern ein Modell generiert werden. Da verschiedene Modelle erstellt werden sollen, müssen diese gespeichert und auch wieder eingelesen werden können.

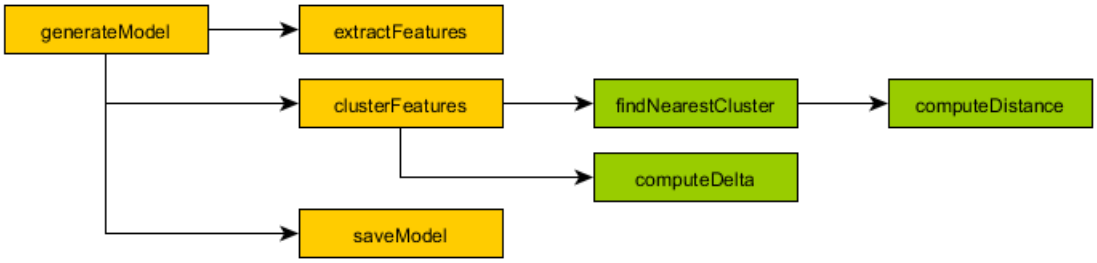

Abbildung 4.1: Funktionen zur Generierung eines Modells

Der Abschnitt Generierung des Modells beschäftigt sich mit einem Entwurf solch eines Systems. Sofern ein Modell generiert wurde, soll es einem Anwender möglich sein ein neues Bild anhand des Modells zu labeln. Dieser Prozess ist in Kapitel Labeling eines Bildes dargestellt.

Generierung des Modells

Die Generierung eines Modells kann durch die Funktion *generateModel* gestartet werden. Als Parameter werden der Pfad für die Bilddaten *imageDir*, der Zielpfad *modelPath* und die Anzahl der Cluster k erwartet. Der Ablauf der folgenden Funktionsaufrufe ist in Abbildung 4.1 dargestellt. Im ersten Schritt wird *extractFeatures* aufgerufen um alle SIFT-Features der Bilder, die in *imageDir* enthalten sind, zu extrahieren. Als nächstes werden durch *clusterFeatures* die Features in k Cluster gruppiert. Die Berechnung der Cluster, der Distanzen von Features zu Clustern und des Konvergenzkriteriums erfolgt durch die GPU. Als Ergebnis werden die k berechneten Schwerpunkte der Cluster und die Mitgliedschaft der Features zurückgegeben. Abschließend speichert *saveModel* die Cluster unter *¡modelPath¿/clusters* und die Mitgliedschaft unter *¡modelpath¿/membership*. TODO: single extractFeature

Labeling eines Bildes

Sofern ein Modell erstellt wurde, können auf dessen Basis Bilder verglichen werden. In Abbildung 4.2 sind schematisch die aufeinanderfolgenden Funktionsaufrufe dargestellt. Die Funktion *getImageLabels* wird mit dem Pfad des Modells und des zu vergleichenden Bildes aufgerufen. Das Modell wird durch *readModel* eingelesen und die Clusterschwerpunkte initialisiert. Die SIFT Features werden, wie bei der Generierung, durch *extractFeatures* ermittelt. Die Funktion *selectLabels* berechnet durch *computeFrequencies* das Histogramm der Visual Words aus den Cluster und Features auf der GPU. Auf dieser Basis werden dann die top X Labels ermittelt und zurückgegeben.

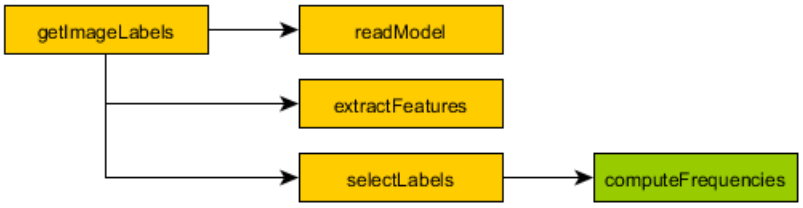

Abbildung 4.2: Funktionen zur Gewinnung von Labels eines Bildes

4.3 Ansatz 2: Autoencoder

In diesem Ansatz werden zur Reduzierung der Dimensionen der Feature-Vektoren wird ein Stacked Denoising Autoencoder verwendet wie er in der Arbeit von TODO [Zha16] vorgeschlagen wurde. Der Autoencoder soll eine komprimierte Darstellung der Gradientenvektoren erzielen, die aus den *intereset points* berechnet werden. Da dieser Vektor 3042 Werte enthält, besitzt der Autoencoder in der Eingabeschicht 3042 Neuronen. Der Encoder des vorgeschlagenen Modells besteht aus fünf Schichten, deren Neuronenanzahl sukzessive reduziert wird, bis schließlich eine Darstellung in 36 Dimensionen erreicht wird. Abbildung 4.3 zeigt die Schichten des Autoencoders. In der Arbeit wurde bereits aufgezeigt, dass der modellierte Autoencoder *state of the art* Ergebnisse erzielt: Die Ergebnisse des Autoencoders wurden unter verschiedenen Kriterien mit den Ergebnissen der SIFT-PCA und TODO Methode verglichen. Dabei erkannte der Autoencoder in fast allen vielen die gleichen Features, jedoch durch einen 36 statt 128 dimensionalen Feature-Vektor. Theoretisch wird hier also das gleiche Ergebnis in einem Drittel der Zeit ermittelt.

TODO: Mit Referenz auf die Arbeit ist es plausibel dieses Modell zu verwenden?

TODO: In der Konzeption bereits TensorFlow erwähnen und somit Beschleunigung durch cuda Bindings?

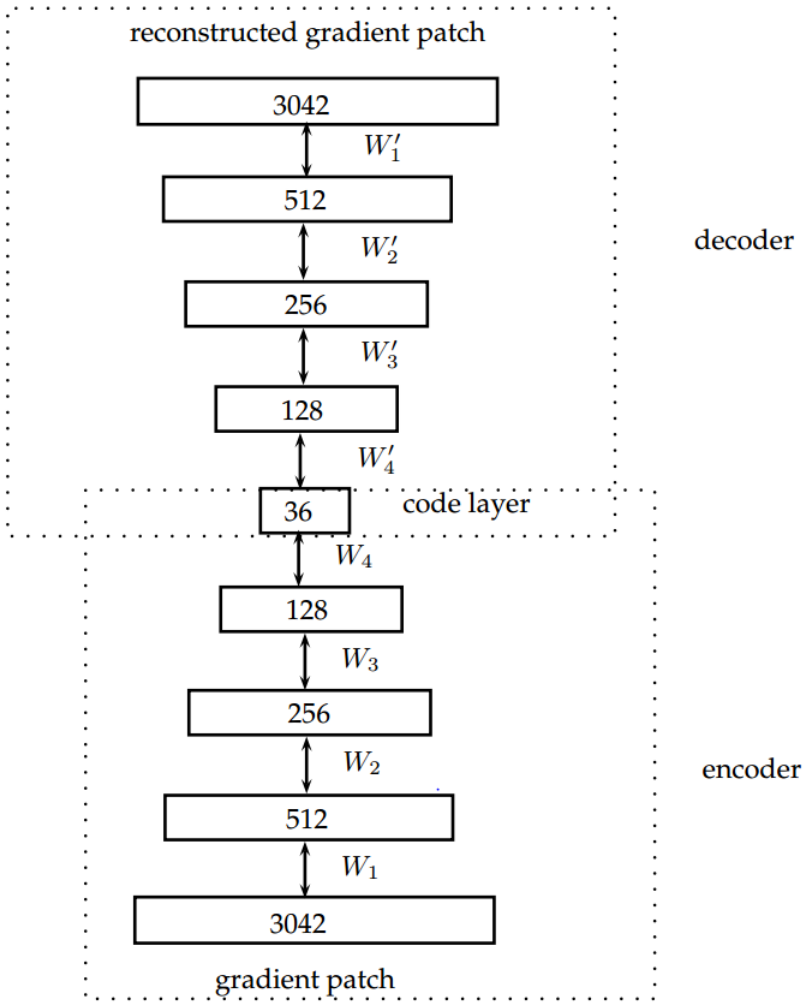

Abbildung 4.3: Schichten des verwendeten Autoencoders, Abbildung aus [Zha16]

5 Implementierung

Die Implementierung ist in drei Teile gegliedert. Zunächst wird die Extraktion und Speicherung der Features näher beschrieben. Auf erfolgt im Anschluss der Aufbau der Modelle. Im zweiten Teil wird die Umsetzung des des Bag of Visual Word Modells in cuda betrachtet. Im Wesentlichen werden hier die cuda Kernels für den k-means und Histogramm Algorithmus angeführt sowie Unterschiede in der Implementierung zwischen *global* und *shared memory* behandelt. Der dritte Abschnitt illustriert eine Autoencoder Implementierung in TensorFlow.

5.1 Feature Extraktion

Da es sich bei der Feature Extraktion für beide Modelle um einen Schritt zur Vorverarbeitung handelt, würde eine Umsetzung genügen. Die beiden Umgebungen der Umsetzung unterschieden sich jedoch sehr voneinander: C und cuda sind sehr hardwarenahe Sprachen, Python ist eine interpretierte Hochsprache und TensorFlow generiert den entsprechenden Code für die Grafikkarte automatisch aus einem Modell. Aus diesem Grund wird eine Funktion zur Extraktion in beiden Sprachen bereitgestellt. Intern wird aber bei beiden 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.

In C wird zur Gewinnung der Feature-Vektoren eines Bildes *extractFeatures* mit dem Pfad des Bildes aufgerufen. Die vollständige Funktion kann nachfolgendem Codelisting entnommen werden. Um die Übersicht zu wahren, wurde hier auf die Importe verzichtet. Neben den *opencv* Standardimporten ist es jedoch zusätzlich notwendig *opencv/nonfree/features2d* einzubinden. Zunächst wird das Bild durch *opencvs imread* Methode eingelesen und liegt im Speicher als Matrix vor. Da SIFT mit monochromatischen Bildern arbeitet, wird vor der Detektion das eingelesen Bild konvertiert. Um hieraus die Deskriptoren zu berechnen, bietet *opencv* eine *SiftFeatureDetector* Klasse an. Via *detect*

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

werden im ersten Schritt die *keypoints* ermittelt (Zeile 8) und in Zeile 9 aus dem Bild und den *keypoints* die Deskriptoren berechnet. Die Deskriptoren werden dann als *opencv* Matrix zurückgegeben.

```
1 using namespace cv;
2
3 Mat extractFeatures (String imagePath) {
4   const Mat image;
5   const Mat source = imread(imagePath, 0);
6   cvtColor(source, image, CV_RGB2GRAY);
7   SiftFeatureDetector detector;
8   vector<KeyPoint> keypoints;
9   Mat descriptors;
10
11  detector.detect(input, keypoints);
12  detector.compute(input, keypoints, descriptors);
13  return descriptors;
14 }
```

In Python funktioniert die Verwendung analog, da auch hier die *opencv* Bibliothek genutzt wurde.

5.2 Ansatz 1: Bag of Visual Words

Das Bag of Visual Words Modell wurde direkt in C und cuda umgesetzt. Sofern die Deskriptoren aus den Trainingsbildern extrahiert wurden, kann durch *generateModel* die Generierung eines Modells gestartet werden. Hier werden die Features, die Anzahl der Cluster und eine Referenz auf eine Liste von Mitgliedschaften erwartet. Neben den Schwerpunkten der Cluster werden die Mitgliedschaften berechnet: Pro Cluster liegt eine Liste vor, die alle *keypoints*, die zu dem Cluster gehören, enthält. Sowohl die die Schwerpunkte der Cluster als auch die Mitgliedschaften werden nach der Verarbeitung auf die Festplatte in verschiedene Dateien geschrieben. Die Schwerpunkte der Cluster werden Zeilenweise in die Datei *model_clusters* geschrieben, sodass sich k Zeilen mit 128 Elementen bei der Verwendung von SIFT ergeben. Die Mitgliedschaften der Features zu Cluster wird in der Datei *model_membership¿* gespeichert und enthält so viele Zeilen wie Features vorliegen. Jede Zeile enthält die x und y Koordinaten des *keypoints* und den Index des zugehörigen Clusters. Der Index bezieht sich hierbei auf die Position des Clusters in der Datei *model_clusters*.

5.2.1 Paralleler k-means Algorithmus

Die Umsetzung des k-means Algorithmus ist direkt in cuda erfolgt. Als Referenzimplementierung diente hier das Projekt von [REF]. Der Algorithmus erwartet die einen Feature-Vektor, die Feature Dimension und die Anzahl der zu bildenden Cluster als Eingabe. Vor dem Kernelaufruf 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, sodass ein SIFT Feature-Vektor bzw. der Schwerpunkt eines Clusters $128 \times 4Byte = 512Byte$ belegt.

- zweidimensionale Daten kopieren, Alternativen
- Übersicht der Kernels (euclidDistance, findNearestCluster, computeDelta)

Global memory

TODO: Wesentliche Teile der global memory Implementierung hier.

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. Die Größe des zu extra zu allokierenden Speichers pro Block muss in *blockSharedDataSize* berücksichtigt werden und ergibt sich aus der Anzahl der Cluster und der Feature Dimension:

```
1 const unsigned int membershipDataSize = numThreads * sizeof(unsigned
      char);
2 const unsigned int clusterDataSize = k * size * sizeof(float);
3 const unsigned int blockSharedDataSize = membershipDataSize +
     clusterDataSize;
```

In der Funktion *findNearestCluster* wird der Parameter *clusters* in *deviceClusters* umbenannt. Vor der Berechnung der Mitgliedschaft wird nun ein lokaler Pointer *clusters* angelegt und alle notwendigen Cluster kopiert.

```
1 float *clusters = (float *)(sharedMemory + blockDim.x);
2 for (int i = threadIdx.x; i < k; i += blockDim.x) {
3   for (int j = 0; j < size; j++) {
4     clusters[k * j + i] = deviceClusters[k * j + i];
5   }
6 }
```

```
7 __syncthreads();
```

Da die Größe von *clusterDataSize* hier von k und *size* abhängt, also der Anzahl der Cluster und der Featuredimension, 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*. Durch die Verwendung von SIFT ist die Featuredimension auf 128 festgelegt, sodass letztendlich eine Obergrenze für k berechnet werden kann. Gängige Modelle cuda kompatibler Grafikkarten sind mit 16 oder 48 Kilobyte *shared memory* ausgestattet. Von 256 Thread pro Block ausgehend ergibt dies ein maximales k von $(memory - 1024)/512$. Dies sind bei 48 Kilobyte maximal 91, für 16 Kilobyte sogar nur 29 Cluster. TODO was tun

5.2.2 Histogramm als parallele Reduktion

TODO: Hier noch als Implementierung behandeln oder eher weglassen?

5.3 Ansatz 2: Autoencoder

Zur Implementierung des Autoencoders wurde TensorFlow verwendet. TensorFlow ist ein DeepLearning Framework und bietet Schnittstellen in diversen Sprachen an. Neben OpenCL wird auch Nvidias cuda unterstützt, sodass TensorFlow Programme automatisch von Grafikkarten profitieren können, ohne das der Entwickler diese explizit berücksichtigen muss. Für diese Umsetzung eines Autoencoders wurde Python und das Projekt *libsdae-autoencoder-tensorflow*[3] von Rajar Sheem genutzt. Ein simpler Autoencoder mit einem HiddenLayer lässt sich wie folgt definieren:

```
1 from deepautoencoder import StackedAutoEncoder
2
3 model = StackedAutoEncoder(dims=[3], activations=['relu'], epoch
    =[1000], loss='rmse', lr=0.007, batch_size=50, print_step=100)
4
5 result = model.fit_transform(x)
```

1. TODO: erklären, tatsächliches Netz
2. Referenzlayout
3. Einlesen der Feature Vektoren etc.

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

4. Speichern / Laden eines Netzes?
5. Training

6 Evaluierung

- Testdaten: MNIST, imagenet, HsH
- Verschiedene Tests auf: Rotation, Skalierung, etc

6.1 MNIST

TODO

6.2 imagenet

TODO

6.3 HsH Daten

TODO

Literaturverzeichnis

[HAA16] M. Hassaballah, Aly Amin Abdelmgeid, and Hammam A. Alshazly. *Image Feature Detectors and Descriptiors*. Springer Verlag, 2016.

[Low04] David G. Lowe. *Distinctive Image Features from Scale-Invariant Keypoints*. International Journal of Computer Vision, 2004.

[OT01] Aude Olivia and Antonio Torralba. *Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope*. International Journal of Computer Vision, 2001.

[RU11] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2011.

[TMT02] Ojala T., Pietikäinen M., and Mäenpää T. *Multiresolution gray-scale and rotation invariant texture classification with Local Binary Patterns*. IEEE Transactions on Pattern Analysis and Machine Intelligence 24(7):971-987, 2002.

[VLL+10] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*. Journal of Machine Learning Research, 2010.

[Zel97] Andreas Zell. *Simulation neuronaler Netze*. Addison Wesley Longman Verlag, 1997.

[Zha16] Chenyang Zhao. *An Autoencoder-Based Image Descriptor for Image Matching and Retrieval*. PhD dissertation, Wright State University, 2016.