

Westfälische Wilhelms-Universität Münster

Institut für Informatik

Praktikum *Computer Vision: Camera Trap Challenge* – Wintersemester 2018/19

Dozenten: Jun.-Prof. Dr. Benjamin Risse, Andreas Nienkötter

Auswertung von Kamerafallenbildern mithilfe von Principal Components Analysis, Spatial Pyramid Matching und Support Vector Machines

Thomas Poschadel
Rudolf-Harbig-Weg 36, 48149 Münster
M.Sc. Informatik
Matrikelnummer: 429 698
poschadel@wwu.de

Joschka Strüber
Rudolf-Harbig-Weg 36, 48149 Münster
M.Sc. Informatik
Matrikelnummer: 418 702
j.st@wwu.de

Sufian Zaabalawi
M.Sc. Informatik
Matrikelnummer: 465 224
m_zaab01@uni-muenster.de

Münster, 29. März 2019

Inhaltsverzeichnis

1 Einleitung	2
2 Aufteilung auf Sequenzen	3
2.1 Sortierung mithilfe der Exif-Daten	3
2.2 Camera Trap Sequencer	4
3 Lokalisierung mit Principal Components Analysis (PCA)	5
3.1 Hintergrundapproximation mit PCA	5
3.2 Sliding-Window Lokalisierung mit PCA	7
3.2.1 Objektdetektion mit PCA	7
3.3 Ergebnisse	9
4 Klassifizierung mit Histograms of Gradients und Support Vector Machines	9
4.1 Einführung zu Histograms of Oriented Gradients	9
4.2 Technische Umsetzung der Klassifizierung	12
5 Klassifizierung mit Spatial Pyramid Matching	13
5.1 Grundidee Spatial Pyramid Matching	13
5.2 Locality-constrained Linear Coding	15
5.2.1 Grundidee	15
5.2.2 Optimierungsproblem und analytische Lösung	15
5.2.3 Pooling und Normalisierung	17
5.2.4 Klassifizierung mit Support Vector Machines	17
5.3 Implementierung	18
5.3.1 Berechnung der Features	18
5.3.2 LLC Encoding	19
5.3.3 Scikit-learn SPM-Transformer und LinearSVC	19
6 Evaluierung	19
6.1 Daten	20
6.2 Experimentelle Optimierung und Auswertung der SVM mit HOG	20
6.2.1 Arten der Testdaten	20
6.2.2 Parameterbestimmung HOG-Deskriptor und Support Vector Machines	21
6.2.3 Ergebnisse	23
6.3 Auswertung Spatial Pyramid Matching mit LLC	23
6.3.1 Laufzeit	23
6.3.2 Klassifizierung auf DDD	24
6.3.3 Klassifizierung auf DDD+	25
6.3.4 Zusammenfassung der Ergebnisse	26

7 Fazit	27
Literaturverzeichnis	29
Anleitung	31
Eigenständigkeitserklärung	35

1 Einleitung

Kamerafalle bieten eine immer wichtiger werdende Möglichkeiten Populationen zu erforschen und zu überwachen. Forschungsinteressen sind beispielsweise die Veränderung der Biodiversität, der Einfluss des Klimawandels und anderer Einflüsse auf die Lebensräume und Migrationsmuster [Yu et al. 13].

Durch die immer größer werdende Akzeptanz von Kamerafallen, steigende Qualität und sinkende Preise kommt es zu einer exponentiell wachsenden Datenmenge. Diesen Daten manuell Herr zu werden stellt eine Herausforderung dar, denn jedes Bild muss einzeln von einem Experten auf Tiere untersucht werden. Aufgrund der Verwendung von unveröffentlichten Daten in aktuellen Forschungsprojekten ist Crowd-Sourcing oft unmöglich. Zudem zeichnen sich die anfallenden Bilder durch einen hohen Anteil von False-Positives (Bilder ohne Tiere) und eine große Vielfalt von Arten in verschiedensten Posen, Entferungen und bei wechselnden Witterungsbedingungen aus, was die automatische Analyse erschwert.

Im Kontext dieser Arbeit beziehen wir uns auf einen Datensatz aus dem niederländischen Nationalpark De Hoge Veluwe. Die Datenbank umfasst 40 GB Bilder von neun einheimischen Tierspezies, die mithilfe von Reconyx-Kamerafallen gesammelt wurden. Dabei wurden sowohl Farbbilder am Tag als auch Infrarotbilder in schwarz-weiß in der Nacht aufgenommen.

Um diese komplexe Aufgabe zu automatisieren, stellen wir eine Softwarepipeline vor, mit deren Hilfe es möglich ist alle nacheinander anfallenden Herausforderungen zu lösen. Der erste Schritt besteht in der Ordnung der Daten. Dafür haben wir mit dem *Camera Trap Sequencer* eine Software mit grafischer Benutzeroberfläche implementiert, die es dem Benutzer erlaubt nach Tierarten vorsortierte Datenbanken oder einzelne Ordner von Bildern auf zusammenhängende Sequenzen aufzuteilen.

Der nächste Schritt ist die Lokalisierung von Tierarten in Bildern. Das ermöglicht zum einen das Aussortieren von Bildern, die in Wirklichkeit keine Tiere zeigen, und zum anderen die Identifikation von Bildausschnitten, die für die spätere Klassifizierung relevant sind. Hierfür verwenden wir eine Pipeline mit verschiedenen Vor- und Nachbereitungsschritten, die mithilfe von *Principal Components Analysis* ein Hintergrundbild auf einer Bildsequenz berechnet und somit die Segmentierung von relevanten Bildausschnitten erlaubt. Um auch die Lokalisierung von Tieren auf Einzelbildern zu ermöglichen, wurde zusätzlich ein PCA-unterstütztes *Sliding-Window*-Verfahren implementiert.

Den Abschluss jeder Auswertung bildet das Klassifizieren von Spezies in zuvor bestimmten *Regions of Interest*. Hierzu stellen wir zwei verschiedene Techniken vor:

Die erste ist die Klassifizierung mit Hilfe einer *Support Vector Machine* mit *Radial-Basis-Function*-Kernel auf dem *Histogram-of-oriented-Gradients*-Feature, einem Strukturfeature. Das zweite Verfahren ist *Spatial Pyramid Matching* (SPM) mit *Locality-constrained Linear Coding* [LSP06]. Hierbei wird das Eingabebild in immer feinere Teilbilder unterteilt, auf denen dann SIFT- oder

LBP-Features berechnet werden. Diese Features werden mit LLC kodiert, wobei die räumliche Aufteilung erhalten bleibt. Abschließend werden die so bestimmten Codes zum Trainieren einer Support Vector Machine mit linearem Kernel benutzt, da sich empirisch erwiesen hat, dass sie gut linear separierbar sind [Yang et al. 09].

Zum Abschluss unserer Ausarbeitung evaluieren wir unsere Verfahren. Betrachtet werden sowohl die Laufzeit der Algorithmen als auch ihre Güte auf den uns zur Verfügung stehenden Daten. Da der ursprüngliche Datensatz zu groß ist, betrachten wir dabei lediglich zwei repräsentative Teilmengen: In der DDD befinden sich Tagbilder von Dachsen und Damhirschen. Die geringe Datenmenge erlaubte schnelle Ergebnisse, ohne grundlegende Probleme, wie beispielsweise unterschiedlich unbalancierte Klassenhäufigkeiten, aus dem Blick zu verlieren. Für die DDD+ haben wir über 2000 Tag- und Nachtbilder von insgesamt sechs verschiedenen Tierarten zusammengestellt. Ziel ist hierbei die Bestimmung der Güte der Verfahrens auf einem komplexen Datensatz.

2 Aufteilung auf Sequenzen

2.1 Sortierung mithilfe der Exif-Daten

Die handelsüblichen Kamerafallen haben einen Sensor, der bei Bewegung auslöst und zunächst zehn Bilder im Abstand von jeweils einer Sekunde schießt. Sollte es am Ende einer Zehnersequenz weiterhin Bewegung im Sichtfeld der Kamera geben, löst der Sensor erneut aus und es werden weitere zehn Bilder aufgenommen. Das sorgt dafür, dass der Datensatz aus zusammenhängenden Sequenzen von jeweils zehn oder mehr Bildern besteht. Oft befinden sich gerade auf den letzten Aufnahmen einer Sequenz keine Tiere mehr, da sich diese aus dem Bild bewegt haben.

Unglücklicherweise sind die Daten auf der Datenbank lediglich nach Tierart sowie dort jeweils nach Tag und Nacht, leeren Bildern und Fehlklassifizierungen sortiert. Für das korrekte Funktionieren unserer Segmentierungstechnik PCA ist es aber nötig, dass die Daten in zusammenhängenden Sequenzen vorliegen. Aus diesem Grund benutzen wir die Exif-Metadaten, um die Daten aufzuteilen. Diese Exif-Daten umfassen Informationen über das Bild, wie beispielsweise die Abmessungen, das Aufnahmedatum, die Belichtungszeit, den ISO-Wert und das Kameramodell.

Unser Algorithmus sammelt zunächst die Metadaten aller aufzuteilenden Bilder in einer Liste. Diese Liste wird primär nach Seriennummer der Kamera und sekundär nach Aufnahmezeitpunkt sortiert. Sollte es zwischen zwei benachbarten Bildern in der Liste zu einem Wechsel der Seriennummer kommen, so wissen wir, dass eine neue Sequenz beginnt. Ebenso gilt das für zwei Bilder, die von derselben Kamera aufgenommen wurden, deren Aufnahmezeitpunkte sich jedoch um mehr als ein paar Sekunden unterscheiden. Diese beiden Situationen markieren den Wechsel einer Sequenz anhand der wir unterscheiden können, welche Bilder zusammenhängen.

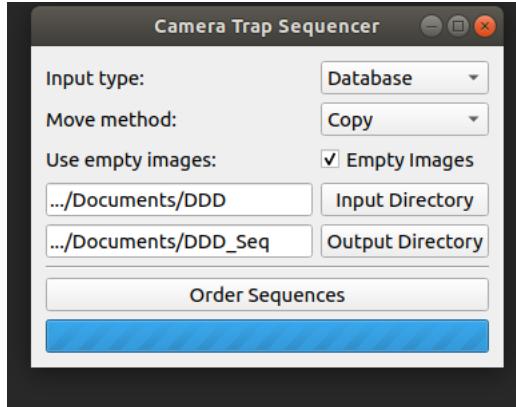


Abbildung 1: Der Camera Trap Sequencer im Linux-Design.

Leider gehört die Seriennummer nicht zu den ursprünglichen Exif-Metadaten. Stattdessen wird sie in die sogenannten *Maker Notes* geschrieben, einem freien Datenfeld, das die Kamerahersteller für ihre Zwecke benutzen können. Aktuell gibt es keine Python-Bibliothek, die das Maker-Note-Feld auslesen kann. Aus diesem Grund waren wir gezwungen die Perl-Bibliothek „ExifTool“ [Harvey 03] zu benutzen und auf sie mithilfe einer Python-Schnittstelle zuzugreifen. Die Verwendung der Metadaten hat - anders als das Auslesen und Abgleichen der Pixel aus der linken oberen Ecke des Bildes - den Vorteil unabhängig vom Kamerahersteller und -modell zu sein und ist sehr effizient.

2.2 Camera Trap Sequencer

Der Camera Trap Sequencer ermöglicht es dem Anwender seine Datenbank mit Kamerafallenbildern unkompliziert und schnell auf Sequenzen aufzuteilen. Als Eingabe kann sowohl eine komplette Datenbank als auch ein einzelner Ordner mit Bildern benutzt werden. Man kann sich zwischen dem Verschieben und Kopieren der Bilder entscheiden. Das Verschieben von Bildern hat den Vorteil, dass keine Daten dupliziert werden. Die ursprüngliche Ordnerstruktur bleibt momentan jedoch noch erhalten.

Des Weiteren hat man im Fall einer Bilddatenbank die Möglichkeit sich dafür zu entscheiden die Informationen über leere Bilder zu speichern. Alle vorsortierten Datenbanken haben Verzeichnisse mit dem Namen „empty“, in denen sich Bilder auf Sequenzen befinden, auf denen keine Tiere zu sehen sind. Unsere Segmentierungstechnik PCA ist darauf angewiesen möglichst auf möglichst langen Bildsequenzen angewendet zu werden. Deshalb ist es nützlich False-Positives mitzuverwenden, insbesondere weil leere Bilder einen großen Beitrag zur Berechnung eines Hintergrundbilds leisten können. Da wir als Label für die Klassifizierung aber die Namen der Tierordner benutzen, müssen diese vor der Klassifizierung aussortiert werden. Deshalb speichern wir, falls die empty-Option gesetzt ist, eine Textdatei mit den Dateipfaden aller leeren Bilder, die dann im Anschluss an PCA aussortiert werden.

Der Camera Trap Sequencer selbst ist mit PyQt5 umgesetzt. Er zeichnet sich deshalb durch Benutzerfreundlichkeit und ein natives Design auf jeder Plattform aus.

3 Lokalisierung mit Principal Components Analysis (PCA)

Die automatische Lokalisierung von Objekten in digitalen Bildern ist ein wesentlicher Bestandteil vieler Anwendungen. Für das Lokalisierungsproblem in dieser Arbeit bietet sich die Verwendung der Methoden *Hintergrund-Subtraktion* und *Sliding-Window* mit *Principal Components Analysis* (PCA) an.

3.1 Hintergrundapproximation mit PCA

Um Bewegungen in Bildsequenzen erkennen zu können, wird in der Praxis sehr häufig das Verfahren der *Hintergrund-Subtraktion* angewendet. Dabei handelt es sich um eine klassische Technik aus dem Bereich der Bilderkennung. Das Hintergrundbild kann mithilfe von PCA approximiert werden. Anschließend wird das Vordergrundbild über die Differenz zum Hintergrundbild extrahiert. PCA, oder auch Hauptkomponentenanalyse, ist ein statistisches Verfahren um große Mengen von Datensätzen zu vereinfachen und zu strukturieren, indem die Datenpunkte im p -dimensionalen Raum \mathbb{R}^p in einen q -dimensionalen Unterraum \mathbb{R}^q mit ($q < p$) projiziert werden. Diese Transformation muss dabei so gewählt werden, dass möglichst wenige Informationen verloren gehen [Verbeke et al. 2007]. Grundsätzlich benutzt PCA die *Niedrigrangapproximation*. Damit kann eine Matrix durch eine andere Matrix im allgemeinen Rang angenähert werden. Sei eine Matrix A mit $\text{Rang}(A) = r$ und $r > k$:

$$\min_{\text{rang}(A)=k} \|A - B\|_2 \quad (1)$$

Dabei soll die Differenz zwischen A und B minimiert werden [Markovsky et al. 2008]. Mithilfe der *Singulärwertzerlegung* (SVD) können die Singulärwerte einer Matrix abgelesen werden. Die SVD von Matrix A ist dann:

$$A = U\Sigma V^T \quad (2)$$

Somit kann ein Hintergrundbild aus einer Sequenz von Bildern wie folgt approximiert werden (Abbildung 2):

- ▷ Berechne Singulärwertzerlegung aller Bildern von Sequenz X :

$$\text{SVD}(X) = C = U\Sigma V^T \quad (3)$$

- ▷ Leite die Matrix Σ_k von Σ her, indem die Werte $n - k$ entlang der Diagonalen durch 0 ersetzt werden.

- ▷ Dies ergibt die Niedrigrangapproximation von Matrix X :

$$\text{SVD}(X)_k = C_k = U \Sigma_k V^T \quad \text{mit} \quad \Sigma_k = \text{diag}(\sigma_1, \dots, \sigma_k, 0, \dots, 0) \quad (4)$$

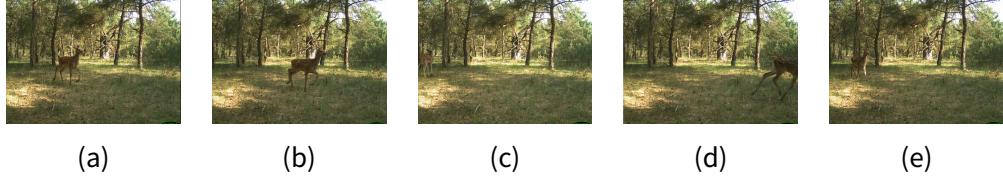


Abbildung 2: (a)-(e) Bilder aus einer Sequenz und (f) das approximierte Hintergrundbild.

Anschließend kann das Vordergrundbild durch die klassische *Hintergrundsubtraktion* extrahiert werden (Abbildung 3).

The diagram illustrates the foreground extraction process. It shows three images arranged horizontally: the original image M , the approximated background L , and the resulting foreground S obtained by subtracting L from M . The subtraction is represented by a minus sign between M and L , followed by an equals sign before S .

Abbildung 3: Das Vordergrundbild S ergibt sich durch die Subtraktion des approximierten Hintergrundbildes L vom Eingabebild M .

Zur Nachbearbeitung des Vordergrundes gehört eine Vielzahl von Operationen z.B. *morphologische Operationen*, *Thresholding* und *Filterung*. Damit können kleinere Bildstrukturen und Rauschen entfernt, vergrößert, geschlossen oder aufgefüllt werden. Jedoch können diese Operationen zu einer Veränderung der Größe der Vordergrundelemente führen, was bei der Lokalisierung des Elements aber kein Störfaktor ist. Durch Kombination der Operationen in einer bestimmten Reihenfolge kann eine Größenveränderung verhindert und dennoch die Vorteile der Operationen genutzt werden. Durch *Opening* werden zunächst kleine Strukturen bzw. Rauschen, welches zum Hintergrund gehört, entfernt. Danach werden kleine Löcher innerhalb der

Vordergrundelemente durch *Closing* geschlossen. In (Abbildung 4) ist die vollständige Pipeline zur Nachbearbeitung des Vordergrundes zu sehen [Giraldo et al. 2018].

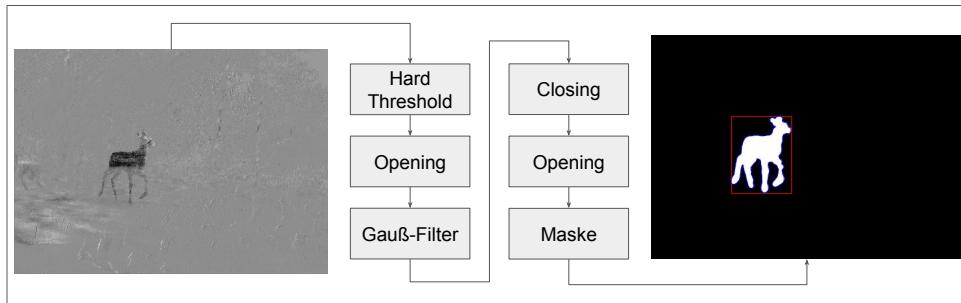


Abbildung 4: Die Pipeline der Nachbearbeitung des Vordergrundbildes. Durch *Opening* und *Closing* werden kleine Bildstrukturen bzw. Rauschen entfernt und kleine Löcher geschlossen. Die Gauß-Filterung dient in diesem Fall dazu, die Silhouette des Vordergrundelements leicht zu vergrößern.

Bei Tierbildern kommt es häufig vor, dass sich auf einem einzigen Bild mehrere Tiere befinden. Für ein gutes Lokalisierungsverfahren ist es in diesem Kontext deshalb unerlässlich, dass es mehrere relevante Regionen im gleichen Bild erkennen kann. Im Falle von PCA ist das kein Problem, weil jede Zusammenhangskomponente im Vordergrundbild, die groß genug ist, als eine solche Region gespeichert wird.

3.2 Sliding-Window Lokalisierung mit PCA

Sliding-Window ist eine Brute-Force-Suche über das Bild mit fester Fenstergröße, um Objekte zu finden. Für jedes dieser Fenster wird ein Bildklassifikator angewendet, um zu bestimmen, ob das Fenster ein bekanntes Objekt enthält. In diesem Fall wird eine auf PCA basierende Technik als Objekt-Klassifikator angewendet [Belhumeur et al. 97].

3.2.1 Objektdetektion mit PCA

Jedes Bild ist ein Punkt in einem hochdimensionalen Raum. Durch das PCA-Verfahren lassen sich die Datenpunkte in einen niederdimensionalen Unterraum abbilden. PCA sucht die ersten k -Hauptkomponenten, welche die Daten mit einer maximalen Varianz beschreiben. Damit wird eine niederdimensionale Darstellung gefunden, bei der die Klassifizierung leichter wird.

Algorithmus

- ▷ Phase I: Initialisierung

- ▷ Berechne das Mittelwertbild der Trainingsbilder

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i \quad (5)$$

- ▷ Berechne die zentrierten Daten durch Subtraktion der Trainingsbilder vom Mittelwertbild

$$C = X - \mu \quad (6)$$

- ▷ Berechne die Eigenwerte und Eigenvektoren für die Kovarianzmatrix CC^T

$$\text{SVD}(C) = \mathbf{U}\Sigma V^T \quad (7)$$

- ▷ Projiziere die Trainingsbilder in den r -Unterraum

$$\mathbf{Y} = \mathbf{U}_r^T C \quad (8)$$

- ▷ Phase II: Klassifikation

Gegeben ist ein unbekanntes Bild M

- ▷ Projiziere das Bild M in den r -Unterraum

$$\mathbf{W} = \mathbf{U}_r^T (M - \mu) \quad (9)$$

- ▷ Finde den nächsten Nachbarn zwischen den projizierten Trainingsbildern \mathbf{Y} und dem projizierten Bild \mathbf{W} .

Die Sliding Windows laufen das Bild mit unterschiedlichen Fenstergrößen ab. Dabei werden einzelne Schnittbilder mit PCA klassifiziert, indem der nächste Nachbar der projizierten Schnittbilder gefunden und zugeordnet wird. Der bunte Rahmen in (Abbildung 5) weist darauf hin, dass ein Objekt mit einer bestimmten Wahrscheinlichkeit innerhalb des jeweiligen Fenster existiert. Anschließend können diese Rahmen miteinander verschmolzen werden, um das Tier zu lokalisieren.

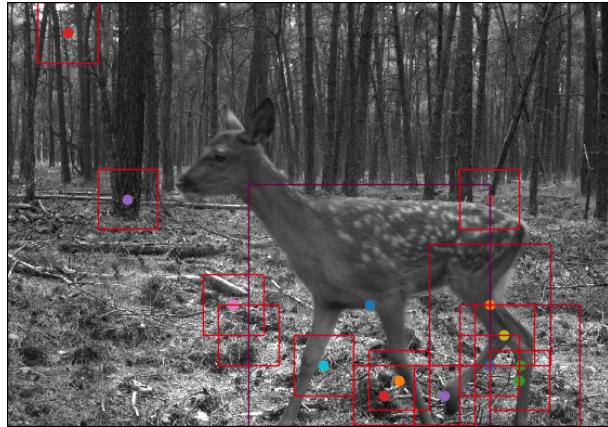


Abbildung 5: Lokalisierung mit Sliding-windows und PCA.

3.3 Ergebnisse

Da keine *Ground-Truth*-Bilder für die Evaluierung vorhanden sind, erfolgt die Evaluierung der Lokalisierungsverfahren nur über subjektive Beobachtungen. Die Ergebnisse der *Hintergrund-Subtraktion* und *Hintergrundapproximation* erzielten eine Präzision von über 82 % auf den Damhirsch- und Dachsdatensätzen. Die Einschränkung für dieses Verfahren ist, dass die Sequenzen aus mindestens drei Bildern bestehen müssen, weil sonst nicht über Bewegung erkannt werden kann, ob ein Pixel zum Vorder- oder Hintergrund gehört. Darüber hinaus müssen die Bilder einer Sequenz auch möglichst unterschiedlich voneinander sein.

Die Lokalisierungsergebnisse mit dem *Sliding-Windows*-Verfahren und PCA-Klassifikator erzielten eine Präzision von etwa 60-65 % auf Damhirsch- und Dachsdatensätzen. Weil der PCA-Klassifikator eine hohe Sensibilität für Variationen zwischen den Trainingsbildern und dem zu klassifizierenden Bild hat, könnte die Anwendung mit Sliding-windows zu False-Positive Klassifikation führen. Deshalb wurde auf die gleiche Art und Weise das *Sliding-Windows*-Verfahren mit einem HOG-Klassifikator (*Histograms of Oriented Gradients*) erweitert. In Kapitel 4 wird das HOG-Verfahren im Detail betrachtet. Mit diesem Verfahren erzielten wir bei der Lokalisierung eine Präzision von 80 % auf dem Damhirsch- und Dachsdatensatz.

4 Klassifizierung mit Histograms of Gradients und Support Vector Machines

4.1 Einführung zu Histograms of Oriented Gradients

Die Bezeichnung *Histograms of Oriented Gradients* (HOG) bezeichnet einen Feature-Deskriptor und wurde bekannt durch die Arbeit von Navneet Dalal und Bill Triggs [DT05]. Diese benutzten den HOG-Deskriptor ursprünglich nur, um Fußgänger zu detektieren. Später wurde er auch

für andere Objekte eingesetzt. Die Berechnung des HOG-Deskriptors lässt sich in drei Schritte unterteilen: Gradientenberechnung, Gruppierung der Orientierungen und Histogrammerzeugung. Häufig werden auch Vorverarbeitungsschritte, wie zum Beispiel ein Histogrammausgleich durchgeführt.

Zur Gradientenberechnung wird der Sobel-Operator verwendet. Eine visuelle Darstellung des Ergebnisses ist in Abbildung 6 gezeigt. Zu sehen ist die Magnitude der Ableitung über die vertikale (links) und horizontale (rechts) Bildachse. Dabei entspricht in dieser Darstellung der Grauwert der positiv definierten Magnitude des Gradienten. Je heller, desto größer ist der Gradient. Mit den folgenden Formeln lassen sich aus den beiden Gradientenbildern die positiv definierte Gesamtmagnitude $|G|$ und die Orientierung Θ berechnen:

$$|G| = \sqrt{G_x^2 + G_y^2} \quad (10)$$

$$\Theta = \arctan(G_x, G_y) \quad (11)$$

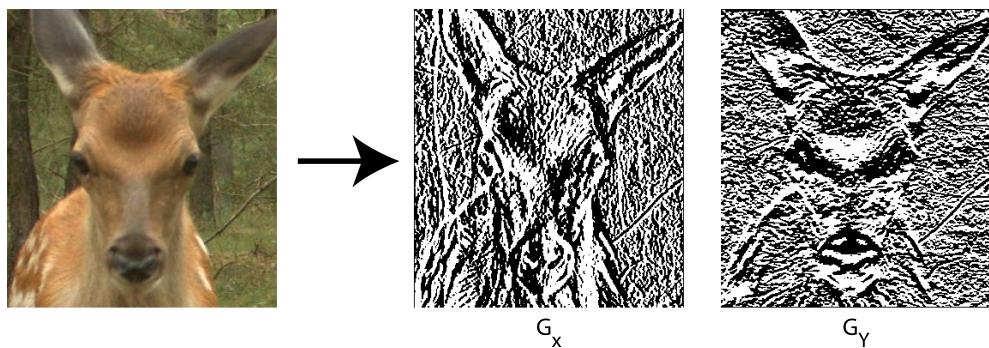


Abbildung 6: Exemplarische Ergebnisbilder des Sobel-Operators für die vertikale und horizontale Bildachse. Die Grauwerte entsprechen der Magnitude des Gradienten.

Für jedes Pixel eines Bildes werden die Magnitude und Orientierung berechnet. Anschließend werden sie entsprechend ihrer Orientierung in Gruppen sortiert. Häufig wird das Vorzeichen der Orientierung ignoriert und sie werden in neun Gruppen zusammengefasst. Durch das Ignorieren der Richtung wird also nur ein halber Kreis betrachtet ($0 - 180^\circ$). Bei neun Gruppen bedeutet dies, dass jede Gruppe 20° abdeckt. Der Wert dieser Gruppe ist dann die Summe aller Magnituden der Pixel, die dieser Gruppe zugewiesen wurden. Es ist jedoch auch möglich andere Einteilungen zu verwenden.

Formal ist ein solches Histogramm aus neun Werten bereits ein HOG. Allerdings ist der Informationsgehalt dieser Repräsentation sehr gering. Um den Informationsgehalt zu erhöhen wird das Bild stattdessen systematisch in gleich große Bereiche unterteilt und für jeden Bereich ein Histogramm gebildet. Diese Histogramme werden anschließend konkateniert und als ein HOG-Deskriptor verwendet. Durch die Unterteilung werden Informationen der einzelnen Bildbereich bewahrt und somit kann ein Bild wesentlich besser beschrieben werden.

Die systematische Unterteilung des Bildes wird wie folgt durchgeführt. Das Bild wird in gleich große Zellen unterteilt, sodass jedes Pixel in genau einer Zelle enthalten ist (siehe Abbildung 7, rotes Raster). Im Anschluss werden jeweils vier dieser Zellen zu einem Block zusammengefasst (siehe Abbildung 7, grünes Rechteck) und für jeden dieser Blöcke wird nach dem *Sliding-Window*-Prinzip ein Histogramm erstellt. Anschließend wird jedes dieser Histogramme normalisiert, um eine belichtungsunabhängige Repräsentation zu erhalten. Dies bedeutet, dass jede Zelle - außer den Randzellen - viermal in einem Histogramm repräsentiert wird. Diese Methode wird angewendet, da so durch die Normalisierung alle lokalen Kontexte betrachtet werden und weniger Informationen durch die Normalisierung verloren gehen. Das resultiert beispielsweise in einer erhöhten Invarianz zu Schatten im Bild.

Exemplarisch ist in Abbildung 8 eine visuell aufbereitete Repräsentation eines HOG-Deskriptors gezeigt. In dieser Darstellung ist das gezeigte Reh noch zu erkennen, da der Deskriptor deutlich die Kanten des Tieres hervorhebt. Diese veranschaulicht, dass die essentiellen Informationen erhalten bleiben, obwohl die Datenmenge auf einen Bruchteil reduziert wurde. Um die Vergleichbarkeit der HOG-Deskriptoren zu gewährleisten, muss garantiert werden, dass die Deskriptoren dieselbe Dimension besitzen. Das wird erreicht, indem die betrachteten Bilder oder Bildausschnitte auf eine einheitliche Größe skaliert werden und anschließend das selbe Raster aus Zellen verwendet wird. [DT05][HOG1]

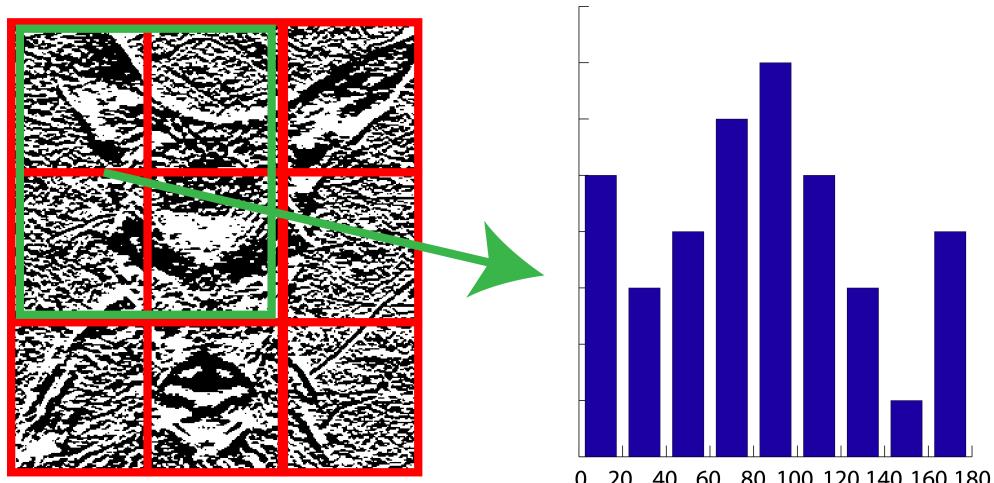


Abbildung 7: Veranschaulichung der Erstellung eines HOG-Deskriptors. Das rote Raster visualisiert die Unterteilung in Zellen. Jeweils vier dieser Zellen werden zusammen als ein Block betrachtet, sodass hier vier Blöcke möglich wären. Für jeden Block wird anschließend ein normalisiertes Histogramm erstellt. Hier exemplarisch nur für den grünen Block gezeigt.

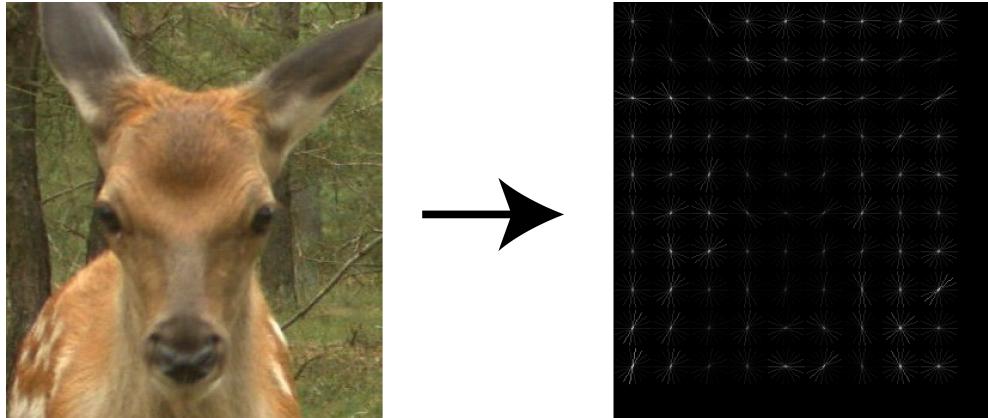


Abbildung 8: Exemplarischer Bildausschnitt (links) und die visuell aufbereitete Repräsentation als HOG-Deskriptor (rechts). In der HOG-Repräsentation ist es immer noch möglich das Reh zu erkennen.

4.2 Technische Umsetzung der Klassifizierung

Umgesetzt wurde der HOG-basierte Klassifizierer in Python mit OpenCV (Version: opencv-contrib-python 3.4.2.17). Der Klassifizierer setzt voraus, dass Bilder und die interessanten Regionen (ROIs) bereits bestimmt wurden. Details zur ROI-Bestimmung werden in Kapitel 3 gegeben. Eine Übersicht der Arbeitsweise wird in Abbildung 9 gezeigt. Dabei gibt es zwei Phasen: 1. Training und 2. Klassifizierung. Die Trainingsphase wird durch den oberen Verlauf in der Abbildung skizziert. Zuerst werden aus einem gelabelten Bilddatensatz deren relevante ROIs selektiert. Für jedes ROI dieser Bilder wird, wie in Abschnitt 4.1 erklärt, ein HOG-Deskriptor berechnet. Dabei wurde darauf geachtet, dass jedes ROI das selbe Seitenverhältnis besitzt. Zur Beschleunigung der Berechnung wurden die ROIs nur als Graustufenbilder verarbeitet und auf eine Größe von 64×128 Pixel² skaliert. Eine Zelle wurde auf 16×16 Pixel festgelegt, sodass eine Blockgröße von 32×32 Pixel² verwendet wurde. Diese Parametersatz hat sich empirisch als der mit den besten Ergebnissen herausgestellt (siehe Kapitel 6.2.2). Außerdem wurden die Orientierungen in neun Gruppen sortiert, sodass eine richtungsunabhängige Orientierung betrachtet wurde. Sonstige Parameter des HOG-Deskriptors wurden auf den entsprechenden Standardwerten der OpenCV-Implementierung belassen.

Diese Deskriptoren sollen mit einer Support Vector Machine (SVM) trainiert und im Anschluss klassifiziert werden. Dazu wurde die von OpenCV zur Verfügung gestellte Implementierung einer SVM verwendet. Als Kernel der SVM wurde eine radiale Basisfunktion gewählt. Experimentell wurden die Parameter $C = 12,5$ und $\gamma = 0,50625$ als optimal bestimmt. Details zur Parameterwahl werden im Kapitel 6.2.2 beschrieben.

Mithilfe der so trainierten SVM kann in der 2. Phase die Klassifizierung der Bilder durchgeführt werden. Dazu werden analog für Bildausschnitte der Testdaten Feature-Deskriptoren berechnet und diese von der SVM klassifiziert. Die Qualität dieser Klassifizierung wird im Unterkapitel 6.2 beschrieben.

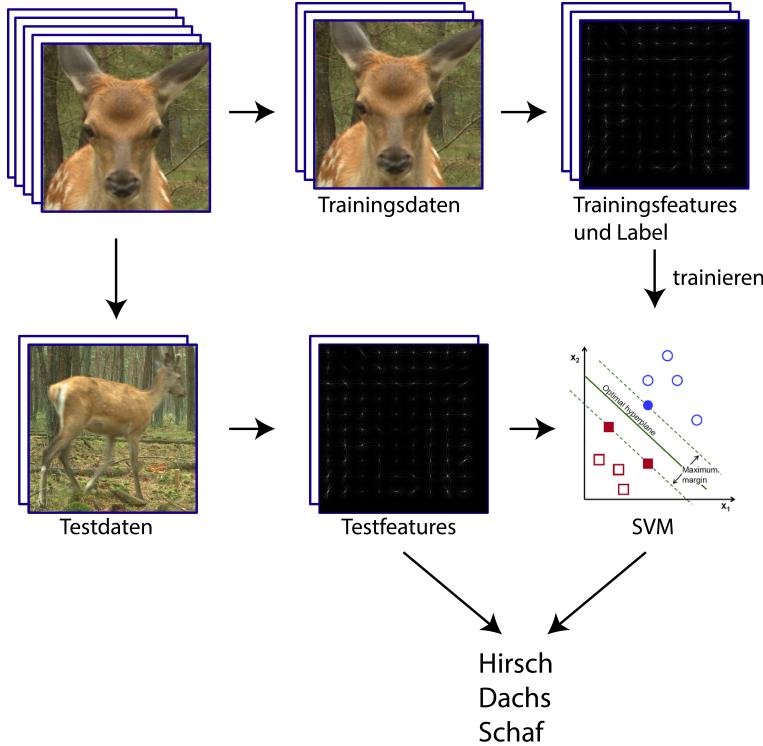


Abbildung 9: Schaubild zur Klassifizierung mit einer SVM. Der obere Pfad repräsentiert die Trainings- und der untere die Klassifizierungsphase. (Das SVM Bild (rechts unten) wurde aus der OpenCV-Dokumentation übernommen [SVM1])

5 Klassifizierung mit Spatial Pyramid Matching

5.1 Grundidee Spatial Pyramid Matching

Spatial Pyramid Matching ist eine Weiterentwicklung des Bag-of-Visual-Words-Ansatzes von Lazebnik et al. [LSP06]. Er wurde 2006 eingeführt und ursprünglich zur Klassifizierung von Szenerien benutzt. Ziel war es zu erkennen, ob ein Bild beispielsweise eine Stadt, einen Wald oder einen Strand zeigt. Beim Spatial Pyramid Matching werden nicht einfach alle Features ungeordnet betrachtet. Stattdessen wird das Bild in immer feinere Teilbilder unterteilt, deren Features jeweils in einem Spatial Bin gesammelt werden. Der L_0 -Bin wird in jeder Dimension halbiert und bildet somit vier L_1 -Bins. Diese wiederum werden wieder geviertelt, was insgesamt 16 L_2 -Bins bedeutet. Durch das Beibehalten der räumlichen Anordnung aller Features lässt sich das endgültige Klassifikationsergebnis verbessern.

Die Features aus jedem der insgesamt 21 Bins werden anschließend über die Lösung eines Optimierungsproblems den Repräsentanten eines Codebooks B zugeordnet. Bei dem Codebook handelt es sich um eine Menge von Features, die den Featureraum über der Datenmenge gut widerspiegeln. Dazu werden üblicherweise Features über zufälligen Bildern oder Bildausschnitten

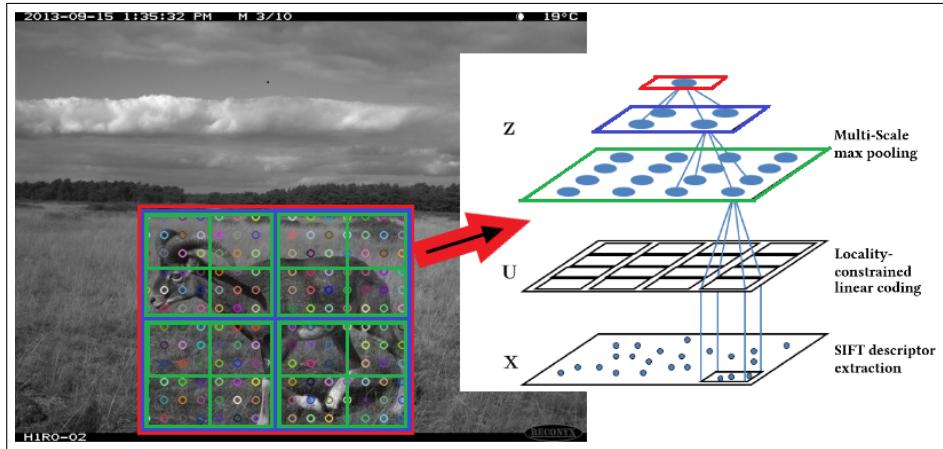


Abbildung 10: Architektur des Algorithmus mit Veranschaulichung der Spatial Pyramid auf SIFT-Features und Pooling. Im Anschluss werden der L_0 - (rot), die vier L_1 - (blau) und 16 L_2 -Codes (grün) konkateniert - angelehnt an [Yang et al. 09].

der Datenbank berechnet und anschließend mithilfe von Clustering Repräsentanten bestimmt. Bei einigen Verfahren wird das Codebook auch online weiter optimiert, worauf wir jedoch verzichten [Wang et al. 10]. Wir haben in unseren Evaluierungen Codebooks mit 256, 512, 1024 und 2048 Features getestet. Je größer das Codebook ist, desto besser können auch komplizierte Datenbanken mit vielen und vielfältigen Klassen abgebildet werden. Auf der anderen Seite steigt die Komplexität der Berechnungen bei einem größeren Codebook deutlich an.

Nachdem für jeden Spatial Bin alle Features mit dem Codebook kodiert wurden, werden diese pro Bin gepoolt, um einen einzige Zuordnung für diesen zu bilden. Die Länge entspricht dabei der Anzahl der Features im Codebook. Abschließend werden die Kodierungen aller Bins konkateniert. Für ein Codebook mit 1024 Features ergibt sich so ein SPM-Code der Länge $21 \cdot 1024 = 21504$. Diese SPM-Codes können dann mit Klassifizierern wie beispielsweise Support Vector Machines (SVMs) eingesetzt werden. Durch die Länge der Codes könnte man erwarten, dass die Laufzeit der Klassifizierung sehr hoch ist. Wie wir im nächsten Abschnitt sehen werden, sind diese gut linear separierbar, weshalb eine SVM mit dem effizienten linearen Kernel verwendet werden kann. Ein Überblick über das Verfahren befindet sich in Abbildung 10.

Unser Algorithmus orientiert sich insgesamt lose an dem Verfahren von [Yu et al. 13]. Dort wurden zunächst SIFT- und cLBP-Features dicht auf Bildern berechnet. Mit dem *Feature Sign Solver* haben sie dann ein dünnbesetztes Kodierungsproblem gelöst [Lee et al. 07]. Die Codes wurden mit Max-Pooling gepoolt, mit der euklidischen Norm normalisiert und anschließend wurden die Bins konkateniert. Die unabhängig voneinander kodierten SIFT und cLBP-Features wurden abschließend mithilfe von AdaBoost und einer linearen SVM klassifiziert.

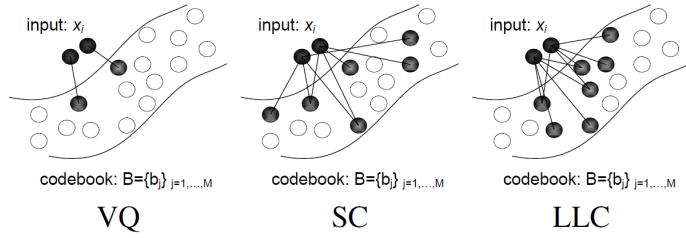


Abbildung 11: Vergleich der drei Kodierungsstrategien Vector Quantization, Sparse Coding und Locality-constrained Linear Coding [Wang et al. 10].

5.2 Locality-constrained Linear Coding

5.2.1 Grundidee

Wir verwenden zur Kodierung unserer Features das *Locality-constrained Linear Coding* (LLC) [Wang et al. 10]. Bei diesem werden jedem Feature f der Spatial Bins mehrere Features des Codebooks zugeordnet, die insgesamt keinen zu großen euklidischen Abstand von f haben dürfen. Bei der ursprünglichen Variante des Spatial Pyramid Matchings wurde lediglich Vektorquantisierung benutzt [LSP06]. Das heißt, dass jedes f dem nächsten Nachbarn im Codebook zugewiesen wird. Dabei kann es zu schlechten Zuordnungen kommen, wenn es beispielsweise kein Feature im Codebook gibt, das zu f ähnlich ist (Fehlertyp 1) oder wenn zwei recht ähnliche Features f und g unterschiedlichen Features in B zugeordnet werden (Fehlertyp 2). Fehler dieser Art nennt man *Quantisierungsfehler*.

Eine deutliche Verbesserung stellte das Verfahren von [Yang et al. 09] dar. Beim *SPM based on sparse coding* (ScSPM) wird jedes Feature f anteilig gleich mehreren Features in B zugeordnet. Über einen Regularisierungsterm wird dabei sichergestellt, dass die Kodierungen insgesamt dünnbesetzt sind. Dieser Algorithmus bietet eine deutliche Verbesserung bei Fehlertyp (1), denn möglicherweise lässt sich f durch Kombination mehrerer Features in B besser darstellen. Da lediglich die Dünnsbesetzung (Sparsity) der Kodierungen gefordert ist, können weiterhin zwei ähnliche Features durch vollkommen unterschiedliche Kombinationen von Features aus B repräsentiert werden.

Wang et al. haben festgestellt, dass für eine optimale Zuordnung zum Codebook Sparsity allein nicht ausreicht [Wang et al. 10]. Deshalb stellen sie in ihrem Optimierungsproblem über einen Regularisierungsterm zusätzlich sicher, dass die Zuordnung lokal stattfindet. Die Lokalität stellt gleichzeitig auch die Sparsity sicher, während das Gegenteil nicht immer der Fall ist. Ein Vergleich dieser Strategien befindet sich in Abbildung 11.

5.2.2 Optimierungsproblem und analytische Lösung

Die Beschreibung von LLC folgt der Veröffentlichung von [Wang et al. 10]. Es seien eine Menge von D -dimensionalen Features $X = [x_1, \dots, x_N] \in \mathbb{R}^{(D \times N)}$ und ein Codebook $B = [b_1, \dots, b_M] \in$

$\mathbb{R}^{(D \times M)}$ mit M Einträgen gegeben. Bei X handelt es sich um Features aus einem Bild und bei B um eine Menge von Deskriptoren, die den Merkmalsraum über allen Bildern gut widerspiegelt.

Gesucht ist eine lokale Zuordnung $C = [c_1, \dots, c_N] \in \mathbb{R}^{(M \times N)}$ von Features aus X auf Visual Words aus B . Hierfür verwenden wir das folgende Optimierungsproblem:

$$\min_C \sum_{i=1}^N \|x_i - B \cdot c_i\|^2 + \alpha \cdot \|d_i \odot c_i\|^2, \text{ s.t. } 1^T c_i = 1 \forall i \quad (12)$$

Betrachten wir den Regularisierungsterm $\alpha \cdot \|d_i \odot c_i\|^2$ genauer. \odot ist die elementweise Multiplikation zweier Vektoren. $d_i = \exp\left(\frac{\text{dist}(x_i, B)}{\sigma}\right) \in \mathbb{R}^M$ ist ein Vektor, der die Distanz des Features x_i zu allen Visual Words aus B angibt. Dabei ist $\text{dist}(x_i, B) = [l_2(x_i, b_1), \dots, l_2(x_i, b_m)]^T \in \mathbb{R}^M$ der Vektor der euklidischen Distanzen von x_i zu jedem Visual Word aus B . In der Praxis wird d_i zusätzlich normalisiert, indem das Maximum aller $l_2(x_i, b_j)$ bestimmt wird. Dieses wird von jeder Zeile von $\text{dist}(x_i, B)$ abgezogen, wodurch sich Werte im Intervall $(-\infty, 0]$ ergeben. Die Anwendung der Exponentialfunktion sorgt dann für normalisierte Distanzwerte in $(0, 1]$. α und σ sind Hyperparameter, die bestimmen wie lokal die Zuordnungen sein müssen.

Es ist zu beachten, dass die Kodierungen c_i nicht zwangsläufig im Sinne der l_0 -Norm dünnbesetzt sind. Vielmehr ergeben sich nicht viele relevante Werte, sodass alle zu kleinen Koeffizienten mithilfe eines Thresholds auf 0 gesetzt werden können, um echte Sparsity sicherzustellen.

Anders als Sparse Coding besitzt LLC eine analytische Lösung und muss nicht mit dem Feature Sign Solver gelöst werden. Als erstes wird die Kovarianzmatrix eines Features x_i über dem Codebook B berechnet, indem wir x_i zeilenweise von B abziehen und die resultierende Matrix mit dem Transponierten seiner selbst multiplizieren:

$$C_i = (B - 1 \cdot x_i^T) \cdot (B - 1 \cdot x_i^T)^T \quad (13)$$

Die Kovarianzmatrix C_i benutzen wir, um ein lineares Gleichungssystem über dieser Matrix und dem Regularisierungsterm nach \tilde{c}_i zu lösen und anschließend zu normalisieren:

$$(C_i + \alpha \text{diag}(d_i)) \cdot \tilde{c}_i = (1, \dots, 1)^T \quad (14)$$

$$c_i = \tilde{c}_i / 1^T \tilde{c}_i \quad (15)$$

[Wang et al. 10] geben die analytische Lösung von LLC als Vorteil gegenüber Verfahren an, die den Feature Sign Algorithmus benutzen, da dieser im besten Fall eine Laufzeit in $\mathcal{O}(M \cdot K)$ hat, wobei K die Anzahl der Elemente ungleich Null ist. Da C_i nicht dünnbesetzt ist, hat die Lösung des linearen Gleichungssystems im Allgemeinen jedoch eine Komplexität von $\mathcal{O}(M^3)$. Möglicherweise lässt sich das Gleichungssystem mit einem speziellen Solver schneller lösen, da

die Kovarianzmatrix symmetrisch und positiv semidefinit ist. In der Praxis hat sich dieser Schritt als Flaschenhals des Verfahrens herausgestellt, wie wir in Abschnitt 6 feststellen werden.

5.2.3 Pooling und Normalisierung

Das Resultat von LLC auf eine Menge Features $X \in \mathbb{R}^{(D \times N)}$ ist eine Menge von Kodierungen $C \in \mathbb{R}^{N \times M}$. C wird auf einen einzelnen trainierten Deskriptor abgebildet, indem wir sie spaltenweise poolen:

- ▷ Sum-Pooling: $c_{out} = \sum_{i=1}^N c_{in,i}$
- ▷ Max-Pooling: $c_{out} = \max c_{in,1}, \dots, c_{in,N}$

Das Pooling stellt eine Möglichkeit dar, die Informationen aller Kodierungen aus C in einem Deskriptor zu bündeln und die aussagekräftigsten Informationen zu erhalten. Aufgrund der Assoziativität der Pooling-Operationen ist es nicht nötig alle 21 Bins einzeln zu kodieren. Das erspart viel Aufwand, da jedes Feature aus einem L_2 -Bin auch in einem L_1 - und dem L_0 -Bin auftaucht. Stattdessen reicht es alle L_2 -Bins einzeln zu kodieren, zu poolen und danach sukzessive das Pooling auf die vier jeweils zusammengehörigen Bins anzuwenden. Dieses *Multi-scale Pooling* wurde auch in Abbildung 10 veranschaulicht.

Zum Abschluss des LLC-Verfahrens werden alle 21 Deskriptoren der einzelnen Bins zu einem einzigen langen LLC-Code konkateniert. Um die Vergleichbarkeit zwischen diesen zu gewährleisten, normalisieren wir sie mit einer der folgenden Methoden:

- ▷ Sum-Normalisierung: $c_{out} = c_{in} / \sum_j c_{in}(j)$
- ▷ l^2 - oder euklidische Normalisierung: $c_{out} = c_{in} / \|c_{in}\|_2$

5.2.4 Klassifizierung mit Support Vector Machines

Die LLC-Codes lassen sich mit einer Support Vector Machine mit linearem Kernel klassifizieren. Diese separiert die Klassen mit einer linearen Hyperebene. Ohne auf die Details einzugehen verwenden wir als Loss-Funktion den differenzierbaren *Squared Hinge Loss* und im Falle von mehr als zwei Klassen werden mit der *One-against-all*-Strategie L verschiedene Klassifikatoren trainiert.

Der von uns verwendete lineare Kernel der SVM hat den Vorteil, dass wir den Klassifikator in $\mathcal{O}(N)$ Zeit trainieren können, während das Testen eines Codes sogar in $\mathcal{O}(1)$ Zeit möglich ist [Yang et al. 09]. Die Laufzeit wird lediglich von der Dimensionalität der Daten, also der Größe des Codebooks bestimmt, nicht aber von der Gesamtanzahl der Features. Demgegenüber benötigen nichtlineare Mercer-Kernels, wie beispielsweise der *Chi-square Kernel* oder der im vorigen Kapitel verwendete RBF-Kernel, Laufzeiten von $\mathcal{O}(N^2)$ bis $\mathcal{O}(N^3)$ fürs Training beziehungsweise $\mathcal{O}(N)$ fürs Testen. Diese Erkenntnis deckt sich auch mit unseren Evaluierungen, in der das Training der

SVM für die Laufzeit des Verfahrens nicht relevant ist. Diese wird stattdessen von der Laufzeit des LLC dominiert.

5.3 Implementierung

Die Implementierung des Verfahrens erfolgte in der Programmiersprache Python, wobei verschiedene Bibliotheken zum Einsatz gekommen sind. Zu diesem Zeitpunkt seien uns die Pfade der Bilder und die Regions of Interest der zu klassifizierenden Tiere als Bounding Boxes gegeben. Um die Laufzeit des Verfahrens zu verringern, werden alle Teilbilder auf eine maximale Länge oder Breite von 300 Pixeln verkleinert und in Graustufenbilder umgerechnet. Das Seitenverhältnis bleibt konstant.

5.3.1 Berechnung der Features

Wir verwenden zur Klassifizierung der Bilder zwei verschiedene Arten von Features. Das *Scale-invariant-Feature-Transform*-Feature (SIFT) ist ein Strukturfeature, bei dem über eine *Scale-Space*-Pyramide von *Difference-of-Gaussian*-Bildern Histogramme von Gradienten berechnet werden [Lowe 99]. Wir benutzen die Implementierung von OpenCV, die sich in dem Modul opencv-contrib der patentierten Verfahren befindet und gegebenenfalls nicht von sich aus mit OpenCV installiert wird [**ocv**]. SIFT findet vielfältig Verwendung und zeichnet sich durch hohe Aussagekraft sowie die Invarianz gegenüber Skalierungen und geringen Perspektiv- und Belichtungswechseln aus.

Das zweite Feature ist das Texturfeature *Local Binary Binary* (LBP) [OPH 94]. Wir verwenden die uniforme Variante mit Radius zwei und 16 benachbarten Punkten von „Scikit-image“ [SKI]. Hierbei wird ein Graustufenpixel mit 16 symmetrisch verteilten Pixeln im Abstand von zwei Pixeln verglichen. Falls der Nachbar größer ist als der zentrale Pixel, merken wir uns für diese Stelle eine 0, ist er kleiner, eine 1. Das ergibt bei 16 Nachbarn eine Kodierung von zwei Bytes. In der uniformen Variante prüfen wir zusätzlich, ob der Deskriptor uniform ist, also ob es höchstens zwei Transformationen der Form $0 \rightarrow 1$ oder $1 \rightarrow 0$ gibt. Bei der anschließenden Berechnung des Histogramms über einem Block von Pixeln erstellen wir für jedes uniforme Muster einen Bin im Histogramm und einen einzelnen Bin für alle nicht-uniformen Muster. Die Verwendung der uniformen Variante sorgt dafür, dass die Histogramme lediglich die Länge 18 statt 512 haben und sorgen zusätzlich für Graustufen- und Rotationsinvarianz.

Die Klasse FeatureExtraction ermöglicht das Berechnen von Featuremengen und Features in Spatial Pyramids für übergebene Featureextraktoren, wie beispielsweise denen für SIFT oder LBP. Die Features werden in einem dichten Gitter der Schrittweite 16 über Blöcken von 16 mal 16 Pixeln berechnet. Für die Klassifizierung wäre eine geringere Schrittweite (beispielsweise vier Pixel) vermutlich besser, da bei nicht überlappenden Blöcken wichtige Kanten und Eckpunkte verloren gehen können. Dieses Verfahren wurde auch von [Yu et al. 13] verwendet. Wir verzichten

bewusst darauf, um die Anzahl der Features und damit die Komplexität der Berechnungen zu verringern.

5.3.2 LLC Encoding

Den Kern unseres Verfahrens bildet die Klasse `LlcSpatialPyramidEncoder`, der über die `encode`-Methode Features mit LLC kodieren kann. Zuerst wird mithilfe von Scikit-learns `MiniBatchKMeans` ein Codebook über einer Menge von Features trainiert, die auf der Datenbank der Bilder zufällig bestimmt wurden [SKL]. Anschließend wurde der Algorithmus aus Abschnitt 5.3.3 mit „Numpy“ umgesetzt. Leider hat sich die Laufzeit der Kodierung als problematisch herausgestellt (siehe Kapitel 6), weshalb der ursprünglich sequentielle Algorithmus in klassischem Python mehrmals optimiert wurde. Die Hilfsfunktionen der Klasse wurden in das Modul `llc_optimization` ausgelagert und mit „Numba“ annotiert [Numba]. Numba bietet die Möglichkeit Pythonfunktionen, die lediglich die Standardbibliothek und Numpy verwenden, mit sogenannten *Decorators* zu markieren. Bei der Ausführung des Programms werden diese Funktionen mithilfe des LLVM-Compilers in Echtzeit kompiliert, was in einer deutlich schnelleren Laufzeit im Vergleich zu interpretiertem Code resultiert.

5.3.3 Scikit-learn SPM-Transformer und LinearSVC

Alle obigen Funktionen wurden in der Klasse `SpmTransformer` zusammengefasst. Diese implementiert als Unterklasse von Scikit-learns `BaseEstimator` die Methoden `fit` und `transform`, wodurch sich das LLC-Verfahren problemlos in Scikit-learns *Pipelines* integrieren lässt [SKL]. Das ermöglicht einem spielend leicht die Kombination mit Klassifizierern, Ensemblemethoden und Modelselektionsverfahren der Bibliothek.

Abschließend wurde in dem Modul `spatial_pyramid_matching` das komplette Verfahren von der Segmentierung, über Locality-constrained Linear Coding bis zur Klassifizierung mit Scikit-learns `LinearSVC` auf zwei verschiedenen Datenbanken umgesetzt. `LinearSVC` setzt glücklicherweise bereits alle unsere Anforderungen bezüglich des Kernels, der Loss-Funktion und der Strategie für nicht-binäre Klassifikation um. Um der unbalancierten Datenlage Genüge zu leisten, setzen wir den Parameter `class_weight="balanced"`. Das sorgt dafür, dass der Strafterm `c` reziprok zum Anteil der Label jeder Klasse modifiziert wird, wodurch das Verfahren besser auf Datensätzen funktioniert bei denen ein Label häufiger vorkommt als andere.

6 Evaluierung

In diesem Kapitel widmen wir uns der Evaluierung unserer Ergebnisse. Dabei betrachten wir die Laufzeit des SPM-Verfahrens und die Güte aller Klassifikationsverfahren. Leider liegen für die Segmentierungen keine Ground-Truth-Daten vor, bei denen jedes Pixel als innerhalb oder außer-

halb eines Tieres markiert ist. Solche Daten wären aber nötig, um die Lokalisierungstechniken und insbesondere PCA verlässlich testen zu können. Stattdessen haben wir die Technik lediglich subjektiv durch Anwendung auf Bildern getestet. Dort waren die Ergebnisse überzeugend. Des Weiteren fließen die Ergebnisse der PCA-Segmentierung indirekt auch in die Güte der Klassifikationen von SPM ein, weil als gelabelte Trainings- und Testdaten die vorsegmentierten Bilder von PCA verwendet wurden. Wären die Ergebnisse von PCA schlecht, hätte sich das auch deutlich in der Klassifikation zeigen müssen.

6.1 Daten

Als Testdaten liegt uns eine umfangreiche Datenbank von Bildern aus dem niederländischen Nationalpark De Hoge Veluwe vor. Diese umfasst 40 GB Bilder von neun verschiedenen Säugetierarten. Die Bildsequenzen wurden mit Reconyx-Kameras erstellt und umfassen zehn bis 120 Bilder. Jedes Bild der Größe 2048×1536 Pixel ist entweder eine farbige Tagaufnahme oder ein Graustufenbild, das nachts mit Infrarotbeleuchtung aufgenommen wurde.

Für unsere Tests haben wir zwei verschiedene Datenbanken angelegt. In der Dachs-Damhirsch-Datenbank (DDD) befinden sich alle Tagbilder der Spezies *Meles Meles* (Dachs) und *Dama Dama* (Damhirsch). Für diese liegen 37 beziehungsweise 262 Bilder vor.

Um unser Verfahren auf komplexeren Daten zu testen, haben wir zusätzlich die Dachs-Damhirsch-Datenbank+ (DDD+) zusammengestellt. Diese umfasst insgesamt 2336 Tag- und Nachtbilder von Rotfuchs, Damhirsch, Wildschwein, Feldhase, Dachs und Wildschaf. Für jede Klasse liegen zwischen 240 und 490 Bilder vor, wobei die Anteile von Tag- und Nachtbildern zwischen den Klassen abweichen. Die Bilder in der DDD+ wurden zufällig aus der Gesamtdatenbank ausgewählt, ohne auf die Komplexität (z.B. Witterung oder Pose der Tiere) zu achten. Die einzigen Voraussetzungen waren die Auswahl von zusammenhängenden Sequenzen und dass sich auf den Bildern nicht Individuen verschiedener Spezies befinden. Das wäre problematisch, weil wir die Ordnerstruktur zum Auslesen der Label benutzen.

6.2 Experimentelle Optimierung und Auswertung der SVM mit HOG

In diesem Kapitel wird die experimentelle Bestimmung der besten Parameter für den HOG-Deskriptor und die SVM behandelt. Für diesen Teil wurden die DDD mit Tagesbildern von Dachs und Damhirsch verwendet.

6.2.1 Arten der Testdaten

Der HOG-Klassifizierer wurde auf einer Datenbank mit bereits bestimmten ROIs verwendet. In ersten Experimenten und zur optimalen Parameterbestimmung wurden Daten mit manuell selektierten ROIs verwendet. Dies wurde getan, um einen möglichen Fehler durch die automa-

tische ROI-Selektion mit PCA auszuschließen. Später wurde die SVM jedoch mit automatisch selektierten ROIs trainiert und zur Klassifizierung benutzt.

Da der Datensatz der Dachsbilder sehr gering war, wurde versucht die Trainingsdatenmenge durch vertikale Spiegelung der Bilddaten künstlich zu erhöhen. Auf zehn Testläufen wurde eine geringe Verbesserung der Ergebnisse festgestellt. Auch wenn diese Erhöhung nicht signifikant war und in seltenen Fällen eine Reduktion der Präzision verursachte, wurde die künstliche Testdatenerhöhung für alle Tests im folgenden Teil angewendet.

Ein weiteres Problem der geringen Menge der Daten für den Dachs war, dass bei prozentualer Selektion der Trainingsdaten der Großteil aus Damhirschbildern bestand. Wurde die SVM mit dieser unausgewogenen Verteilung der Tierklassen trainiert, konnte die SVM fast nur noch Damhirsche erkennen. Die Erkennungsrate für Dachse lag dabei bei unter 50 %. Dieses Problem wurde dadurch gelöst, dass darauf geachtet wurde gleich große Mengen an Trainingsdaten zu verwenden. Durch diese Maßnahme wurde die Präzision mit geeigneten Parametern auf durchschnittlich über 90 % erhöhen. In der Zukunft scheint auch eine Verwendung des Parameters `class_weight="balanced"` sinnvoll, um bessere Ergebnisse auf ungleichmäßig verteilten Datenmengen zu erreichen.

6.2.2 Parameterbestimmung HOG-Deskriptor und Support Vector Machines

Zwei der wichtigsten Parameter des HOG-Deskriptors sind die Größe des betrachteten Bildausschnittes und die Einteilung in Zellen, denn diese Größen bestimmen zum einem die benötigte Rechenzeit und zum anderem wieviele Details des Bildausschnittes regional aufgelöst werden. Außerdem ergibt sich aus dieser Kombination die Dimension des Ergebnisvektors. Dieser hat wiederum Einfluss auf den Rechenaufwand der SVM. Ziel der ersten Parameterfindung war es, die Präzision der Klassifizierung zu maximieren. Dabei wurde besonders darauf geachtet, dass die Präzision für beide trainierten Klassen maximal wurde. Dazu wurden die folgenden Bildausschnittgrößen betrachtet: 64×128 , 128×256 , 64×64 , 128×128 , 128×64 und 256×128 Pixel². Jede dieser Größen wurde mit einer Zellgröße von 16×16 und 32×32 Pixeln getestet. Für die Blockgröße wurde jeweils die Kombination aus 4 Zellen verwendet. Die Ergebnisse sind in Tabelle 1 gezeigt. Ebenfalls wurde versucht die Parameter der SVM zu optimieren. Dazu wurde die `trainAuto` Methode von OpenCV verwendet. Die besten Parameter für die Kostenfunktion waren $C = 12,5$ und $\gamma = 0,50625$. Diese Werte liegen sehr nah an den Standardwerten ($C = 12$ und $\gamma = 0,5$) für die SVM, sodass keine nennenswerten Änderungen an den Werten aus Tabelle 1 bestimmt werden konnten.

In den gezeigten Tabellen wird deutlich, dass die Präzision für die beiden Zellgrößen in einem ähnlichen Genauigkeitsbereich liegt. Jedoch scheint eine größere Zellgröße etwas besser geeignet, um Dachse zu erkennen, und etwas schlechter geeignet, um Damhirsche zu erkennen. Die besten Ergebnisse wurden für die Kombination aus einer Bildausschnittgröße von 64×128 Pixel² und einer Zellgröße von 16×16 Pixel² erhalten. Dieser Wert entspricht den üblichen Standardwerten für die Fußgängerdetektion in OpenCV. Für die Tierdetektion ist dieses Ergebnis

Tabelle 1: Präzision der Klassifizierung abhängig von der Bildausschnittgröße und der Zellgröße. Es werden aus 50 zufällig generierten Trainings- und Testdatenreihenfolgen die jeweils erreichte durchschnittliche Präzision \bar{p} , die minimale sowie maximale erhaltene Präzision (p_{min}, p_{max}) und die Standardabweichung σ angegeben.

Ausschnitt [Pixel ²]	Zellgröße 16 × 16
64 × 128	Dachs: $\bar{p} = 0.913, p_{min} = 0.750, p_{max} = 1.000, \sigma = 0.062$ Damhirsch: $\bar{p} = 0.914, p_{min} = 0.831, p_{max} = 0.987, \sigma = 0.034$
128 × 256	Dachs: $\bar{p} = 0.488, p_{min} = 0.167, p_{max} = 0.833, \sigma = 0.161$ Damhirsch: $\bar{p} = 0.996, p_{min} = 0.949, p_{max} = 1.000, \sigma = 0.011$
64 × 64	Dachs: $\bar{p} = 0.908, p_{min} = 0.833, p_{max} = 1.000, \sigma = 0.045$ Damhirsch: $\bar{p} = 0.859, p_{min} = 0.797, p_{max} = 0.916, \sigma = 0.035$
128 × 128	Dachs: $\bar{p} = 0.818, p_{min} = 0.500, p_{max} = 1.000, \sigma = 0.111$ Damhirsch: $\bar{p} = 0.951, p_{min} = 0.882, p_{max} = 0.992, \sigma = 0.023$
128 × 64	Dachs: $\bar{p} = 0.933, p_{min} = 0.833, p_{max} = 1.000, \sigma = 0.050$ Damhirsch: $\bar{p} = 0.881, p_{min} = 0.709, p_{max} = 0.979, \sigma = 0.069$
256 × 128	Dachs: $\bar{p} = 0.675, p_{min} = 0.250, p_{max} = 1.000, \sigma = 0.308$ Damhirsch: $\bar{p} = 0.781, p_{min} = 0.367, p_{max} = 1.000, \sigma = 0.249$

Ausschnitt [Pixel ²]	Zellgröße 32 × 32
64 × 128	Dachs: $\bar{p} = 0.875, p_{min} = 0.750, p_{max} = 1.000, \sigma = 0.067$ Damhirsch: $\bar{p} = 0.854, p_{min} = 0.755, p_{max} = 0.907, \sigma = 0.044$
128 × 256	Dachs: $\bar{p} = 0.883, p_{min} = 0.750, p_{max} = 1.000, \sigma = 0.076$ Damhirsch: $\bar{p} = 0.894, p_{min} = 0.810, p_{max} = 0.928, \sigma = 0.037$
64 × 64	Dachs: $\bar{p} = 0.975, p_{min} = 0.917, p_{max} = 1.000, \sigma = 0.038$ Damhirsch: $\bar{p} = 0.859, p_{min} = 0.708, p_{max} = 0.574, \sigma = 0.072$
128 × 128	Dachs: $\bar{p} = 0.908, p_{min} = 0.667, p_{max} = 1.000, \sigma = 0.102$ Damhirsch: $\bar{p} = 0.840, p_{min} = 0.705, p_{max} = 0.903, \sigma = 0.062$
128 × 64	Dachs: $\bar{p} = 0.967, p_{min} = 0.917, p_{max} = 1.000, \sigma = 0.041$ Damhirsch: $\bar{p} = 0.834, p_{min} = 0.776, p_{max} = 0.932, \sigma = 0.045$
256 × 128	Dachs: $\bar{p} = 0.892, p_{min} = 0.833, p_{max} = 1.000, \sigma = 0.075$ Damhirsch: $\bar{p} = 0.867, p_{min} = 0.797, p_{max} = 1.000, \sigma = 0.954$

Tabelle 2: Präzision des Klassifizierers der sowohl mit den aus PCA stammenden ROIs trainiert und getestet wurde. Es werden aus 50 zufällig generierten Trainings- und Testdatenreihenfolgen die jeweils erreichte durchschnittliche Präzision \bar{p} , die minimale sowie maximale erhaltene Präzision (p_{min}, p_{max}) und die Standardabweichung σ angegeben.

Tiergattung	Präzision
Dachs	$\bar{p} = 0.859, p_{min} = 0.761, p_{max} = 0.957, \sigma = 0.052$
Damhirsch	$\bar{p} = 0.833, p_{min} = 0.763, p_{max} = 0.930, \sigma = 0.032$

überraschend, da die Seitenverhältnisse der betrachteten Tiere eher 1:1 bzw. 1:2 entsprechen. Eine Begründung für diese Abweichung kann an dieser Stelle nicht gegeben werden.

Im folgenden werden Experimente nur noch mit den hier gefundenen optimalen Parametern durchgeführt.

6.2.3 Ergebnisse

Die in Kapitel 6.2.2 bestimmten optimalen Parameter wurden mit denen durch PCA ermittelten ROIs und der künstlichen Erhöhung der Testdaten getestet. Die in Tabelle 2 gezeigten Ergebnisse stammen aus 50 Tests mit den selben Daten, aber einer zufälligen Unterteilung der Daten in Trainings- und Testdaten. Es ist zu erkennen, dass die Präzision etwas geringer ist, als mit den manuell ausgewerteten ROIs. Dies war zu erwarten, weil die automatische Bestimmung der ROIs mit PCA kleine Fehler enthält. Der Vorteil dieser Variante ist dennoch die automatische ROI-Selektion und damit eine deutliche Aufwandsreduktion für den Benutzer.

Experimentell wurde der HOG-Klassifizierer auch auf eine Datenbank mit Bildern von Damhirsch, Dachs, Wildschein und Schaf angewendet. Mit Erhöhung der Datenklassen sank die Güte der Ergebnisse auf den jeweiligen Klassen deutlich. Die Präzision lag üblicherweise unter 70 %, für Dachse sogar unter 50 %. Deshalb halten wir diese Technik ungeeignet für größere und kompliziertere Klassifizierungsprobleme und haben sie nicht weiter auf der DDD+ evaluiert.

6.3 Auswertung Spatial Pyramid Matching mit LLC

6.3.1 Laufzeit

In der Praxis hat sich die Laufzeit von SPM mit LLC als der limitierende Faktor herausgestellt. Unsere Bildausschnitte, die von PCA bestimmt wurden, werden vor der Berechnung auf eine Größe von 300 Pixeln Seitenlänge oder Höhe verkleinert, um die Anzahl der zu berechnenden Features zu verringern. Trotzdem liegen die Laufzeiten pro Bild momentan in einem Bereich, der eine praktische Anwendung schwierig macht. Alle folgenden Tests wurden auf einem Desktop-PC aus dem Jahr 2014 mit einer i5-4590 CPU mit vier Kernen à 3.3GHz und 8 GB RAM durchgeführt.

Variante	n = 256	n = 512	n = 1024	n = 2048
sequentiell	0,44±0,25s	1,55±0,94s	8,32±3,99s	39,92±20,62s
multiprocessing	1,71±0,19s	2,57±0,79s	9,69±4,87s	38,37±19,46s
Numba	0,32±0,29s	1,13±0,66s	5,96±2,70s	24,39±14,27s

Tabelle 3: Gegenüberstellung durchschnittlichen Laufzeiten mit Standardabweichung der drei implementierten Varianten des Algorithmus für verschiedene Größen des Codebooks.

Wir haben festgestellt, dass der überwiegende Teil der Berechnungszeit für die Berechnung der LLC-Codes verwendet wird. Das Training der SVM mit linearem Kernel und die Berechnung der Features sowie das Training des Codebooks sind wie erwartet schnell. Die genaue Laufzeit ist abhängig von der Größe des Codebook, wobei die Lösung des linearen Gleichungssystems etwa 80 bis 95 % der Berechnung ausmacht.

Um die Enkodierung zu beschleunigen wurden mehrere Varianten des Algorithmus implementiert. Die erste entspricht einer klassischen sequentiellen Umsetzung mit Numpy ohne zusätzliche Optimierung. In der zweiten Variante haben wir versucht die unabhängigen Kodierungen der 16 L_2 -Bins zu parallelisieren. Dafür wurde mit der Funktion `pool.map` aus dem Modul `multiprocessing` je ein Thread pro Bin gestartet. Leider hat sich in unseren Tests keine signifikante Verbesserung oder sogar eine Verschlechterung der Laufzeit eingestellt.

Zuletzt haben wir die sequentielle Variante wiederhergestellt und die Low-Level-Funktionen mithilfe von Numba automatisch optimieren lassen [Numba]. Numba ist eine Bibliothek, die Pythoncode mit einem JIT-Compiler auf Basis des LLVM-Compilers optimiert. Dieser Schritt sorgte für eine deutliche Verbesserung der Laufzeit, die aber leider immer noch nicht zufriedenstellend ist.

Eine Gegenüberstellung der Ergebnisse befindet sich in Tabelle 3. Die lange Laufzeit für Codebooks der Größe 2048 hatte zur Folge, dass diese Größe nicht zur Klassifizierung benutzt haben. Der nächste Schritt wird es sein die Enkodierung mit *Tensorflow* zu programmieren und die Berechnungen der Codes aller Features eines Bilds parallel auf der GPU durchzuführen.

6.3.2 Klassifizierung auf DDD

Dank der Implementierung des `SpmTransformers` als Scikit-learn-BaseEstimator waren wir in der Lage die Klassifizierung mit Spatial Pyramid Matching und einer linearen SVM problemlos in eine Pipeline einzubetten. Zur Bestimmung der optimalen Parameter haben wir die randomisierte Cross Validation `RandomizedSearchCV` mit fünf Folds benutzt. Hierbei wurden 80 % der Daten fürs Training und 20 % zum Testen benutzt. Insgesamt wurden für SIFT- und LBP-Features jeweils zwanzig zufällige Parametersätze getestet.

Die Parameterverteilung sah wie folgt aus:

- ▷ Codebookgröße: [256, 512, 1025]
- ▷ Alpha: `reciprocal(100, 1000)`

- ▷ Sigma: reciprocal(50, 500)
- ▷ Pooling: ['max', 'sum']
- ▷ Normalisierung: ['sum', 'eucl']
- ▷ C: reciprocal(0.1, 5)

Eine reziproke Verteilung bedeutet, dass die Parameter in dem Bereich zufällig bestimmt werden, wobei kleinere Werte mit einer höheren Wahrscheinlichkeit gewählt werden. Der Logarithmus der Verteilung ist ungefähr gleichverteilt. Bei Listen werden alle Werte gleichverteilt ausgewählt.

Als bester Parametersatz für SIFT hat sich eine Codebookgröße von 1024, Alpha = 357, Sigma = 330, Sum-Pooling, euklidische Normalisierung und C = 0,55 herausgestellt. Als Scoring wurde F1-Scoring benutzt, um es zu bestrafen alle Tiere als Damhirsche zu klassifizieren. Die F1-Scores auf den Validation Sets waren sehr vielversprechend:

Fold k =	1	2	3	4	5
F1-Score	0,9901	0,97029	0,98	1,0	0,9583

Auf dem Testdatensatz ergab sich ein gewichteter F1-Score von 0,95493. Alle acht Dachsbilder wurden korrekt klassifiziert. Bei den 63 Damhirschbildern gab es drei Misklassifikationen.

Auf die Daten für das uniforme LBP-Feature gehen wir nicht im Detail ein. Die Ergebnisse sind etwas schlechter, für die besten Parametersätze aber durchaus überraschend gut, wenn man bedenkt, dass das Feature lediglich 17 Dimensionen hat. Die Varianz zwischen den Ergebnissen war deutlich größer und überraschenderweise nahm die Genauigkeit bei Codebooks der Größe 1024 deutlich ab. Des Weiteren generalisieren die Ergebnisse bei LBP weniger gut. Als Unterstützung für SIFT scheint es aber eine gute Wahl zu sein.

6.3.3 Klassifizierung auf DDD+

Die Datenbank DDD+ zeichnet sich durch deutlich kompliziertere Klassen aus. Um ein Gefühl für die Daten und die Parameter zu erhalten, haben wir zunächst einen Test mit SIFT auf wenigen Trainingsdaten mit RandomSearchCV und den gleichen Einstellungen wie im vorherigen Abschnitt durchgeführt. Aus dem kompletten Pool an Daten wurden lediglich 100 Trainings- und 100 Testbilder ausgewählt. Das sorgt für 12 bis 25 Trainingsbilder pro Klasse, durchschnittlich sind es etwa 17.

Bei zehn Durchläufen mit zufälligen Parametersätzen haben wir einen Mean Test Score von 0,43 auf den Trainingsdaten erhalten. Überraschenderweise lag der gewichtete F1-Score auf den Testdaten bei sogar 0,52 also höher als auf den Validation Sets. Diese Ergebnisse sind nicht überragend, aber für die geringe Anzahl an Trainingsdaten und die Komplexität der Aufgabe durchaus überzeugend. Sie decken sich in etwa mit denen von [Wang et al. 10] auf ähnlich komplexen Daten.

	Dachs	Rotfuchs	Feldhase	Wildschwein	Wildschaf	Damhirsch
Dachs	19	0	3	2	0	2
Rotfuchs	2	45	1	3	1	4
Feldhase	2	4	19	0	0	2
Wildschwein	4	6	1	41	7	5
Wildschaf	0	5	2	0	59	6
Damhirsch	1	1	2	2	4	65

Tabelle 4: Confusion Matrix der sechs Tierarten nach Anwendung von LLC-kodierten SIFT-Features und einer linearen SVM auf der DDD+.

Anschließend haben wir die empirisch bestimmten optimalen Parameter zur Evaluierung des Verfahrens auf der kompletten DDD+ benutzt. Aufgrund der Datenmenge war Cross Validation hier nicht möglich. Wieder werden die Daten im Verhältnis 80:20 auf Trainings- und Testdaten aufgeteilt. Wir wenden LLC auf SIFT-Features mit einem Codebook der Größe 1024, Alpha = 500, Sigma = 100, Max-Pooling und euklidischer Normalisierung an. Die Klassifikation der Trainingsdaten selbst ergibt als Bias einen gewichteten F1-Score von 0,9853. Das beweist eindeutig die lineare Separierbarkeit der LLC-Codes. Leider fällt das Ergebnis auf den Testdaten mit 0,7678 etwas schlechter, aber nach wie vor gut aus. Möglicherweise lässt sich das Overfitting mit noch stärkerer Regularisierung durch Erhöhung von Alpha verringern. Eine Confusion Matrix für die DDD+ befindet sich in Abbildung 4. Überraschenderweise waren die Tests bei denen nur SIFT benutzt wurde besser als bei konkatinierten Codes für SIFT und LBP. Dort betrug der durchschnittliche Score 0,7399 für den besten Klassifikator. Wie bei den Tests von LBP auf der DDD angedeutet könnten sich mit kleineren Codebooks für dieses Feature sogar bessere Ergebnisse erzielen lassen.

6.3.4 Zusammenfassung der Ergebnisse

Zusammenfassend lässt sich sagen, dass LLC mit SPM ordentliche Ergebnisse liefert, auch wenn nur wenige Daten vorliegen. Auf der DDD war unsere Klassifikation der menschlichen ebenbürtig, die in der Regel mit 96 % angegeben wird. Auch auf der komplexeren DDD+ haben wir zufriedenstellende Ergebnisse erhalten, die durch bessere Parameter eventuell noch gesteigert werden können. Die in der Literatur angegebenen Werte von Alpha = 500 und Sigma = 100 haben auch für uns gute Ergebnisse geliefert [Wang et al. 10]. Anders als dort angegeben, haben wir keine Unterschiede zwischen den Pooling- und Normalisierungsverfahren feststellen können. Bei den SIFT-Features hat insbesondere die Vergrößerung des Codebooks zu signifikanten Steigerungen in der Genauigkeit geführt. So ließen sich die Scores von 0,656 (256 Visual Words), über 0,7168 (512 Visual Words), bis auf 0,7678 für 1024 Visual Words steigern. Nach einer Reimplementierung des Verfahrens mit Tensorflow ist es denkbar, dass sich mit einer Vergrößerung des Codebooks auf zum Beispiel 2048 Repräsentanten noch bessere Ergebnisse einstellen.

7 Fazit

In dieser Ausarbeitung haben wir zahlreiche Methoden gesehen, mit deren Hilfe sich alle Probleme, die bei der Bearbeitung eines Projekts mit Kamerafallenbildern anfallen, lösen lassen. Der Camera Trap Sequencer sortiert einzelne Ordner oder ganze Datenbanken in zusammenhängende Sequenzen. Er ist über ein unkompliziertes grafisches Tool benutzbar.

Die Sequenzen können anschließend mithilfe des PCA-Verfahrens über die Berechnung eines Hintergrundbilds pro Sequenz segmentiert werden. Dadurch lassen sich relevante Bildausschnitte bestimmen und Tiere lokalisieren. Dank der ausgefeilten Pipeline aus Vor- und Nachbereitungsschritten sind überzeugende Segmentierungsergebnisse möglich. Gerade die Identifizierung von mehreren ROIs im gleichen Bild ist bei Bildern von Tiergruppen oder -herden ein unschätzbarer Vorteil. Sollten Bilder einmal nicht in Sequenzen, sondern als einzelne Dateien vorliegen, können wir das Sliding-Window-Verfahren anwenden, welches Lokalisierung und Klassifizierung miteinander verbindet.

In der ersten Iteration des Projekts haben wir einen SVM-Klassifikator implementiert, der einen RBF-Kernel auf HOG-Features benutzt. Das erste Ziel einer Präzision von über 80 % auf der DDD wurde mit durchschnittlich 83 % auf dem besten Parametersatz erreicht. Zudem zeichnet sich der Algorithmus durch seine schnelle Laufzeit aus. Leider sinkt die Güte der Ergebnisse bei komplizierteren Klassifizierungsproblemen deutlich ab, sodass ein Einsatz auf der DDD+ nicht sinnvoll war.

Für die Klassifizierung eines komplizierten Datensatzes mit vielen Klassen und Tag- und Nachtbildern, haben wir Spatial Pyramid Matching mit LLC implementiert. Dabei werden auf immer feineren Teilbildern SIFT- oder LBP-Deskriptoren berechnet und auf einem Codebook lokal encodiert. Die LLC-Codes lassen sich anschließend mit einer linearen SVM klassifizieren. Für dieses Verfahren werden auf der kleinen DDD Ergebnisse erreicht, die denen von Menschen ebenbürtig sind. Auch auf der komplizierteren DDD+ lag der gewichtete F1-Score fast bei 80 %. Durch die Implementierung als `BaseEstimator` lässt sich SPM leicht in Scikit-learn Pipelines integrieren, was die Verwendung einer Vielzahl von Ensemblemethoden und Modellselektionsverfahren erlaubt.

Das größte Problem unserer Verfahren ist momentan die schlechte Laufzeit des Spatial Pyramid Matchings. Während das Training der SVM sehr effizient möglich ist - trotz Codes, die mehrere zehntausend Elemente lang sind - verbraucht ebenjene Kodierung sehr viel Zeit. Das nächste Ziel ist deshalb eine Reimplementierung des LLC-Verfahrens mit Tensorflow, um eine höhere Parallelität zu erreichen. Durch die Verwendung von größeren Codebooks und der besseren Möglichkeit die optimalen Hyperparameter zu suchen, sind dadurch bessere Klassifikationsergebnisse denkbar.

Auch komplexere Ensemblemethoden, wie beispielsweise *AdaBoost* oder *VotingClassifier*, sind Ansätze, die unser Ergebnis verbessern könnten [Yu et al. 13]. Des Weiteren wäre es sinnvoll die

Sliding-Window-Methoden so zu erweitern, dass man einen eigenen vortrainierten Klassifikator benutzen kann, der dann für die Lokalisierung und Klassifizierung benutzt wird.

Literatur

- [Belhumeur et al. 97] Peter N Belhumeur, João P Hespanha u. a. „Eigenfaces vs. fisherfaces: Recognition using class specific linear projection“. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 7 (1997), S. 711–720.
- [DT05] N. Dalal und B. Triggs. „Histograms of oriented gradients for human detection“. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Bd. 1. Juni 2005, S. 886–893. DOI: 10.1109/CVPR.2005.177.
- [Giraldo et al. 2018] Jhony-Heriberto Giraldo-Zuluaga, Alexander Gómez u. a. „Camera-Trap Images Segmentation using Multi-Layer Robust Principal Component Analysis“. In: *CoRR* abs/1701.08180 (2017). arXiv: 1701.08180. URL: <http://arxiv.org/abs/1701.08180>.
- [Harvey 03] *ExifTool* by Phil Harvey. <https://www.sno.phy.queensu.ca/~phil/exiftool/>. Accessed: 04.03.2019.
- [HOG1] *Histogram of Oriented Gradients*. <https://www.learnopencv.com/histogram-of-oriented-gradients/>. Accessed: 27.03.2019.
- [Lee et al. 07] Honglak Lee, Alexis Battle u. a. „Efficient sparse coding algorithms“. In: *NIPS Proceedings 2007*. NIPS, 2007.
- [Lowe 99] David Lowe. „Object recognition from local scale-invariant features“. In: *Proceedings of the International Conference on Computer Vision*. IEEE, 1999.
- [LSP06] Svetlana Lazebnik, Cordelia Schmid u. a. „Beyond bags of features: spatial pyramid matching for recognizing natural scene categories“. In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE, 2006.
- [Markovsky et al. 2008] Ivan Markovsky. „Structured low-rank approximation and its applications“. In: *Automatica* 44.4 (2008), S. 891–909. ISSN: 0005-1098. DOI: <https://doi.org/10.1016/j.automatica.2007.09.011>. URL: <http://www.sciencedirect.com/science/article/pii/S0005109807003950>.
- [Numba] *Numba*. <http://numba.pydata.org/>. Accessed: 04.03.2019.

- [OPH 94] Timo Ojala, Mati Pietikäinen u. a. „Performance evaluation of texture measures with classification based on Kullback discrimination of distributions“. In: *Proceedings of the 12th IAPR International Conference on Pattern Recognition*. Elsevier, 1994.
- [SKI] *Scikit-learn. scikit-image.org*. Accessed: 28.03.2019.
- [SKL] *Scikit-learn. scikit-learn.org*. Accessed: 04.03.2019.
- [SVM1] *OpenCV SVM Tutorial*. https://docs.opencv.org/3.4.2/d4/db1/tutorial_py_svm_basics.html. Accessed: 28.03.2019.
- [Verbeke et al. 2007] Nicolas Verbeke und Nicole Vincent. „A PCA-Based Technique to Detect Moving Objects“. In: *SCIA*. 2007.
- [Wang et al. 10] Jinjun Wang, Jianchao Yang u. a. „Locality-constrained Linear Coding for Image Classification“. In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE, 2010.
- [Yang et al. 09] Jianchao Yang, Kai Yu u. a. „Linear Spatial Pyramid Matching Using Sparse Coding for Image Classification“. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2009.
- [Yu et al. 13] Xiaoyuan Yu, Jiangping Wang u. a. „Automated identification of animal species in camera trap images“. In: *EURASIP Journal on Image and Video Processing* (2013).

Anleitung

Im Folgenden wird eine kurze Anleitung zur Verwendung der in dieser Arbeit implementierten Algorithmen gegeben. Aufgrund des Umfangs kann nicht auf alle spezifischen Parameter und Optionen eingegangen werden. Daher ist diese Anleitung eher als ein *Quick Start Guide* zu betrachten. Weitere Details können der Codedokumentation entnommen werden.

Systemvoraussetzungen

Zur Verwendung des Codes wird eine Python 3.6-Umgebung vorausgesetzt. Die Installation von Python 3.6 wird auf der offiziellen Homepage beschrieben <https://www.python.org/>. Des Weiteren wird zur Sequenzierung das ExifTool von Phil Harvey in Version 11.33 verwendet. Die Installation für verschiedene Betriebssysteme wird auf der entsprechenden Homepage beschrieben <http://www.sno.phy.queensu.ca/~phil/exiftool/>. Zusätzlich müssen die folgenden Python-Pakete installiert sein: Matplotlib (Version 3.0.2), Numpy (Version 1.15.4), Scipy (1.2.1), OpenCV mit opencv-contrib-python (Version 3.4.2.17), python-dateutil (Version 2.7.5), Scikit-image (Version 0.14.2), Scikit-learn (Version 0.20.2), PyQt5 (Version 5.12) und Numba (Version 0.43.0).

Sequenzierung

Die Sequenzierung der Bilddatenbank kann auf zwei Art und Weisen durchgeführt werden. Zum einen über die DataProvider-Klasse (siehe Abschnitt Verwendung DataProvider Klasse) oder über die einfachere und benutzerfreundliche Variante mit GUI. Dazu muss das Python Skript `camera_trap_sequencer.py` ausgeführt werden, das im Verzeichnis `src/sequences` zu finden ist. Die GUI ist in Abbildung 1 gezeigt.

Schritt-für-Schritt-Anleitung:

Vorbedingung: Es existiert ein Ordner, der alle zu bearbeitenden Tierbilder nach Tierart sortiert in Ordnern besitzt (<data folder>)

1. Starte `camera_trap_sequencer.py`
2. Selektiere Input type: Directory und Move method: Copy
3. Wähle das Input Directory als <data folder>
4. Wähle einen **leeren** Ordner als Output Directory
5. Klicke auf Order Sequences

Je nach Anzahl der Bilder dauert dieser Prozess einen Moment, da unter Umständen eine große Datenmenge bearbeitet werden muss.

Lokalisierung

Nach der Sequenzierung der Bilderdatenbank können die Sequenzen mit *Hintergrund-Subtraktion* und *Hintergrundapproximation* lokalisiert bzw. segmentiert werden. Dafür kann die Methode `segment` in der Klasse `segment.py` verwendet werden. Die Parameter sind der Pfad des Überordners, der alle Unterordner der Sequenzen enthält, die „Label“ der jeweiligen Tiere und der Ausgabepfad der segmentierten Bilder.

Für die Lokalisierung mit *Sliding-Windows* und dem PCA-Klassifikator kann man die Methode `TrainingsPhase` in der Klasse `pca_knn.py` anpassen, um die Pfade von den Schnittbildern festzulegen. Die Schnittbilder sollen möglichst quadratisch sein und müssen das Tier enthalten. Defaultmäßig werden das trainierte `pca.sav`- und `knn.sav`-Model geladen. Anschließend kann das gesuchte Tier in den Bildern mithilfe der Methode `localisation` gefunden und ausgegeben werden. Analog kann die Lokalisierung mit *Sliding-Windows* und HOG-Klassifikator in der Klasse `hog_svm.py` verwendet werden.

Verwendung DataProvider Klasse

Die DataProvider-Klasse ist eine Klasse zu vereinfachten Handhabung der Trainings- und Testdaten, die für dieses Projekt benötigt werden. Sie bietet die Möglichkeit die Sequenzierung und Segmentierung der Datenbank Bilder auszulösen, aber vor allem eine gute Methode, um die Daten in Trainingsdaten und Testdaten aufzuteilen. Es ist ebenfalls möglich die Daten zufällig anzutragen. Die DataProvider Klasse ist in `<src>/datautils/data_provider.py` zu finden.

Beispiel Verwendung:

```
1 # Erzeugen eines DataProvider-Objektes
2 provider = DataProvider(
3     "<Path>/data", # Pfad zu den Bilderdaten in separaten Ordnern für jedes Tier. Optional
4     # falls die Sequenzierung bereits durchgeführt wurde
5     "<Path>/sequ", # Pfad an die Sequenzen gespeichert sind, bzw. gespeichert werden sollen.
6     # Optional falls die Sequenzierung bereits durchgeführt wurde
7     "<Path>/npy", # Pfad zu den Numpy-Arrays, die Dateipfad, ROI und Label enthalten
8     True, # Ungenutztes Artefakt, wurde verwendet um die ROIs anzuzeigen
9     {"dayvision", "day"}, # Unterordner die zur Sequenzierung betrachtet werden sollen
10    # Optional falls die Sequenzierung bereits durchgeführt wurde
11    0.66, # Prozentsatz der Daten die zum Training verwendet werden
12    True, # Angabe ob alle Tierklassen mit gleich vielen Trainingsdaten trainiert werden
13    True, # Ob Training- und Testdaten zufällig sortiert werden sollen Ermöglicht leichte
14    # Variation der Datenmenge
15    0) # Setzen des Seeds für den Zufallsgenerator zum mischen der Daten 0 entspricht einer
16    # zufälligen Sortierung. Alle anderen positiven Werte geben eine konstante Sortierung
17    # vor. Dient der Reproduzierbarkeit
```

12

```

13 provider.generate_sequences() # Sequnziert die Daten. nur aufrufen wenn dies noch nicht
   ↳ durchgeführt wurde
14 provider.segment_sequences() # Segmentiert die Daten mit PCA. Nur aufrufen wenn dies noch
   ↳ nicht durchgeführt wurde
15
16 # Abrufen der Trainings- und Testdaten
17 trainingData = provider.get_training_data()
18 testData = provider.get_test_data()

```

Klassifizierung mit HOG

Zur Klassifizierung mit HOG muss die Segmentierung erfolgreich durchlaufen sein. Insbesondere werden die mit PCA gespeicherten Numpy-Arrays benötigt, welche die Dateipfade, ROIs und Label enthalten. Diese können wie bei der Lokalisierung beschrieben erzeugt werden oder durch Verwendung des DataProvider-Objekts:

Beispiel Verwendung unter der Annahme, das provider ein gültiges DataProvider-Objekt ist:

```

1 # Erzeugen einer Klassifikator-Objektes
2 classifier = HogClassifier()
3 # Trainieren der SVM
4 classifier.train(provider.get_training_data(), False, True)
5 # Testen der Testdaten
6 classifier.test(provider.get_test_data())

```

Es ist auch möglich multiple Iterationen der Trainings- und Testdurchläufe zu starten. Dazu muss lediglich die Klassenmethode `test_multiple_times(<number of runs>)` aufgerufen werden. Spezifische Testparameter müssen in der Methode selbst angepasst werden. Dies betrifft vor allem das dort erstellte DataProvider-Objekt.

Klassifizierung mit SPM

Zur Klassifizierung werden in der Python-Datei *spatial_pyramid_matching.py* vier praktische Pipelines zur Verfügung gestellt:

```

call_DDD_sift_pipeline()

call_DDD_lbp_pipeline()

call_DDD_plus_sift_pipeline():

call_DDD_plus_sift_lbp_pipeline():

```

Die ersten beiden Pipelines sind für die Dachs- und Damhirsch-Datenbank konzipiert und wenden, wie der Pipelinename andeutet, entweder den auf SIFT basierenden oder den auf Local Binary Patterns basierenden Deskriptor an. Die beiden unteren Pipelines arbeiten auf der erweiterten DDD+-Datenbank.

Zur Verwendung der jeweiligen Pipeline muss das in der Pipeline verwendete DataProvider-Objekt an die lokale Datenstruktur angepasst werden. Es wird vorausgesetzt, dass die Daten bereits auf Sequenzen aufgeteilt wurden (siehe Lokalisierung). Zur Anpassung der Datei an das eigene System genügt es die Pfade für die Ordner mit den Sequenzen und Segmentierungsdateien am Anfang der Datei manuell anzupassen:

```
1 # directories for sequences and segmentation files for DDD and DDD+
2 DIR_DDD_SEQUENCES = "/home/joschi/Documents/DDD_seqs"
3 DIR_DDD_SEGMENTS = "/home/joschi/Documents/DDD_segs"
4 DIR_DDD_PLUS_SEQUENCES = "/home/joschi/Documents/DDD+_seqs"
5 DIR_DDD_PLUS_SEGMENTS = "/home/joschi/Documents/DDD+_segs"
```

Falls PCA noch nicht ausgeführt wurde, muss für die Aufrufe `$# provider.segment_sequences()` einmalig die Kommentierung entfernt werden. Es wird aus Zeitgründen empfohlen diese Zeilen wieder zu kommentieren, wenn PCA mit der unveränderten Datenbank bereits durchgeführt wurde, die Pipeline aber erneut aufgerufen wird. So kann viel Rechenzeit gespart werden. Für die erstmalige Ausführung ist bereits alles korrekt gesetzt.

Eigenständigkeitserklärung

Hiermit versichern wir, dass die vorliegende Ausarbeitung *Auswertung von Kamerafallenbildern mit Hilfe von Principal Components Analysis, Spatial Pyramid Matching und Support Vector Machines* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

(Ort, Datum)

(Unterschrift)