

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
Straße Hausnummer, 12345 Münster
M.Sc. Informatik
Matrikelnummer: 123 456
blabla@wwu.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
4 Klassifizierung mit Histograms of Gradients und Support Vector Machines	8
4.1 Einführung zu Histograms of Oriented Gradients	8
4.2 Technische Umsetzung der Klassifizierung	11
5 Klassifizierung mit Spatial Pyramid Matching	12
5.1 Grundidee Spatial Pyramid Matching	12
5.2 Locality-constrained Linear Coding	14
5.2.1 Grundidee	14
5.2.2 Optimierungsproblem und analytische Lösung	14
5.2.3 Pooling und Normalisierung	16
5.2.4 Klassifizierung mit Support Vector Machines	16
5.3 Implementierung	17
5.3.1 Berechnung der Features	17
5.3.2 LLC Encoding	18
5.3.3 Scikit-learn SPM-Transformer und LinearSVC	18
6 Evaluierung	18
6.1 Daten	19
6.2 Laufzeit Spatial Pyramid Matching	19
6.3 Experimentelle Optimierung und Auswertung SVM mit HOG	19
6.3.1 Arten der Testdaten	19
6.3.2 Parameterbestimmung HOG-Deskriptor und Support Vector Machines	20
6.3.3 Ergebnisse	20
6.4 Güte der Klassifizierungstechnik	22
7 Fazit	22
Anleitung	22
Literaturverzeichnis	24

Eigenständigkeitserklärung	25
-----------------------------------	-----------

1 Einleitung

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

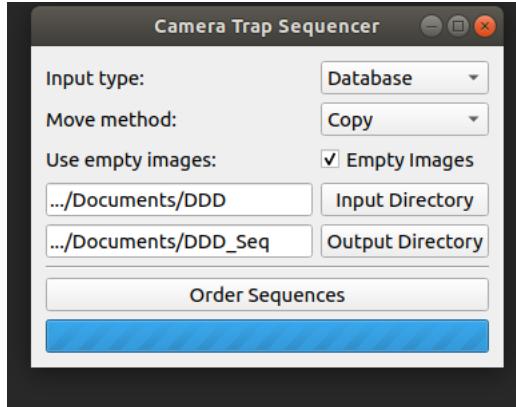


Abbildung 1: Der Camera Trap Sequencer im Linux-Design.

Leider gehört die Seriennummer nicht zu den ursprünglichen Exif-Metadaten. Stattdessen wird sie in die sogenannten *Maker Notes* geschrieben, einem freien Datenfeld, dass 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 mit Hilfe einer Python-Schnittstelle zuzugreifen.

Zugegebenermaßen ist die Verwendung der Metadaten nicht der eleganteste Weg Sequenzen zu bestimmen. Sie hat aber - 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.

Abschließend hat man im Fall einer Bilddatenbank die Möglichkeit sich dafür zu entscheiden mit 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 leeren 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 können.

Der Camera Trap Sequencer selbst ist mit PyQt5 umgesetzt. Er zeichnet sich deshalb durch 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 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* angewandt. Dabei handelt es sich um ein klassisches Verfahren 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 R^p in einen q -dimensionalen Unterraum R^q mit ($q < p$) projiziert werden. Diese Transformation muss dabei so gewählt werden, dass möglichst wenig Information verloren geht. 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. 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 :

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

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

- ▷ Dies ergibt die Niedrigrangapproximation von Matrix X :

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

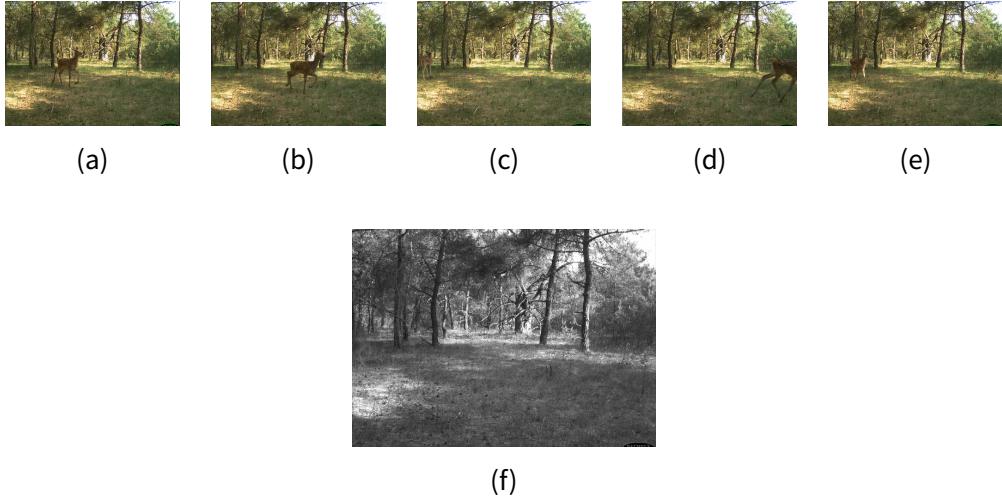


Abbildung 2: (a)-(e) Bilder aus einer Sequenz und (f) das approximierte Hintergrundbild.

Anschließend kann das Vordergrundbild durch die klassische *Hintergrund-Subtraktion* extrahiert werden (Abbildung 3).



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

Zum 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. Können jedoch diese Operationen zu einer Veränderung der Größe der Vordergrundelemente führen, was zur Lokalisierung des Elements aber keinen Störfaktor ergibt. Durch Kombination der Operationen in einer bestimmten Reihenfolge kann 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 eine Kombination dieser Pipeline zur Nachbearbeitung des Vordergrundes benutzt worden.

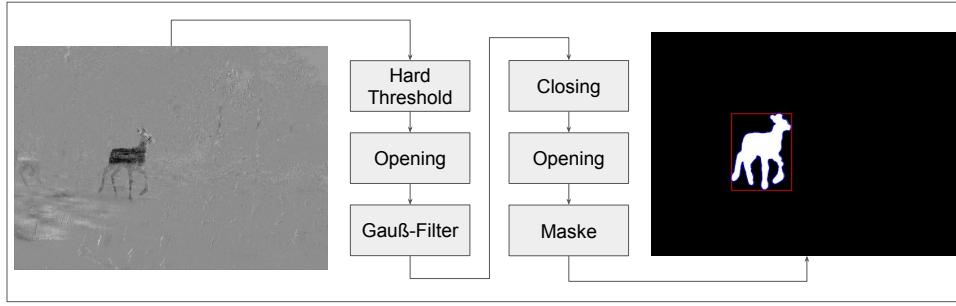


Abbildung 4: Die Pipeline der Nachbearbeitung des Vordergrundbildes. Durch *Opening* und *Closing* werden kleine Bildstrukturen bzw. Rauschen entfernt und kleine Löcher geschlossen werden. Die Gauß-Filterung dient in diesem Fall dazu, die Silhouette des Vordergrundelements grob zu vergrößern.

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 PCA als Objekt-Klassifikator angewandt.

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 es 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

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

- ▷ 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 durch. Demnach werden Schnittbilder einzelne mit PCA klassifiziert. Dabei wird der nächste Nachbar der projizierten Schnittbilder gefunden und zugeordnet (Abbildung 5).

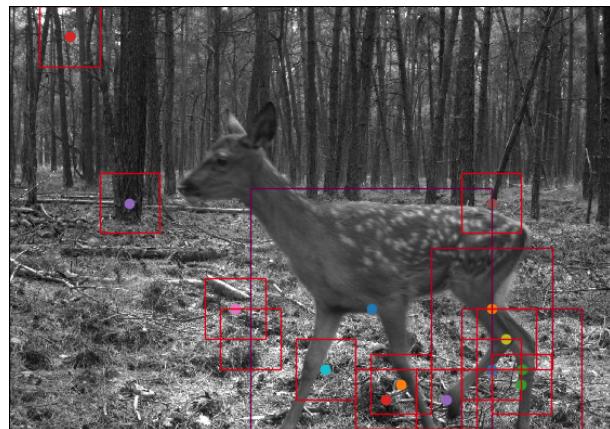


Abbildung 5: Lokalisierung mit Sliding-windows und PCA.

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 setzen den

HOG-Deskriptor ein, 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. Mit den folgenden Formeln lassen sich aus den beiden Gradientenbildern die positiv definierte Gesamtmagnitude $|G|$ und die Orientierung Θ berechnen:

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

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

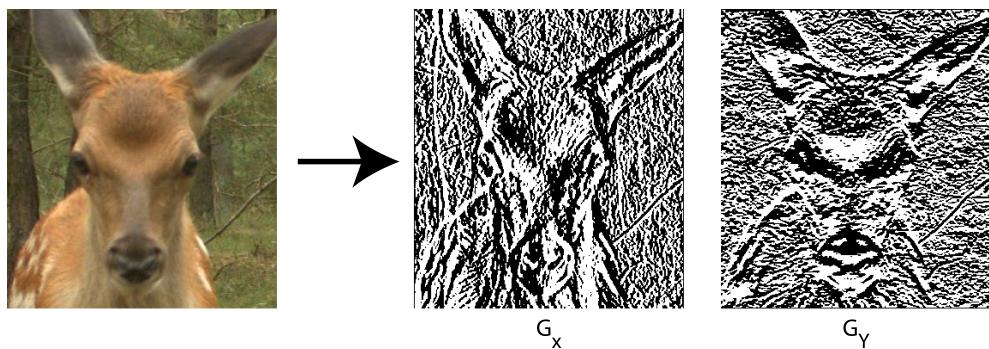


Abbildung 6: Exemplarische Ergebnisbilder des Sobel-Operators für die vertikale und horizontale Bildachse. Die Grauwerte entsprechen der Magnitude des Gradienten.

Für jeden 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 abschließend 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 jeder Pixel in genau einer Zelle enthalten ist (siehe Abbildung 7

rotes Raster). Anschließend 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. Jedes dieser Histogramme wird anschließend 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 Information durch die Normalisierung verloren geht. 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]

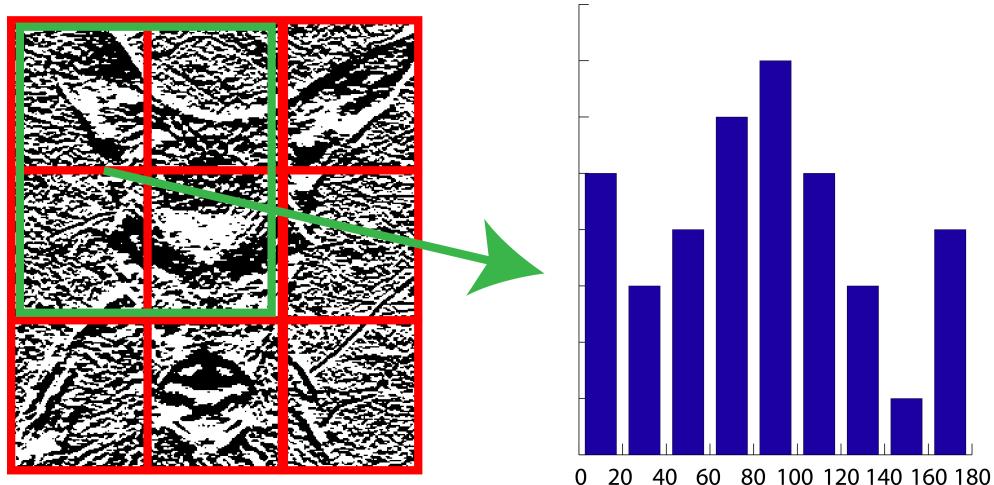


Abbildung 7: Veranschaulichung der Erstellung eines HOG-Deskriptors. Das rote Raster visualisiert die Unterteilung in Zellen. Jeweils vier dieser Zellen werden zusammen als ein Block betrachtet, sodass hier vier Blöcke möglich wären. Für jeden Block wird anschließend ein normalisiertes Histogramm erstellt.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

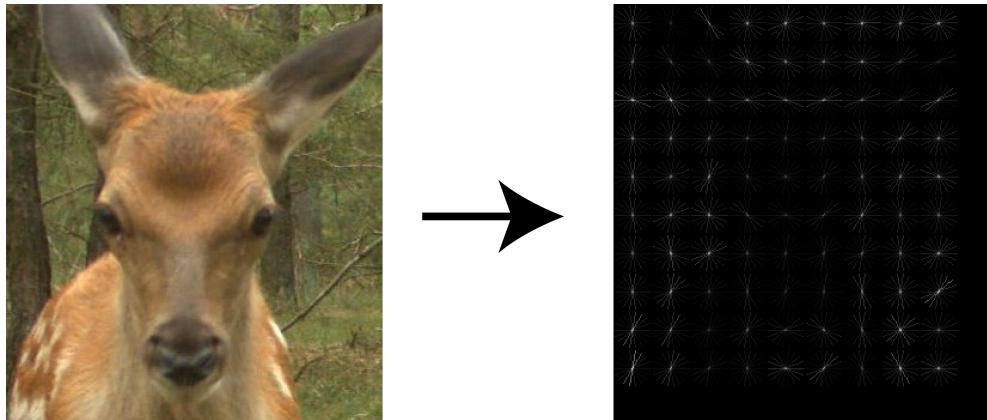


Abbildung 8: Exemplarischer Bildausschnitt (links) und die visuell aufbereitete Repräsentation als HOG-Deskriptor (rechts). In der HOG-Repräsentation ist es immer noch möglich das Reh zu erkennen.

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 (ROI) bereits bestimmt wurden. Details zur ROI Bestimmungen werden in Kapitel 3 gegeben. Eine Übersicht der Arbeitsweise wird in Abbildung 9 gezeigt. Dabei gibt es zwei Phasen: 1. Training und 2. Klassifikation. Die Trainings Phase 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 wurde die ROI nur als Graustufenbild 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 wurde empirisch als der mit den Ergebnissen bestimmt (siehe Kapitel 6.3.2). 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 Vektor 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 6.3.2 beschrieben.

Mithilfe der so trainierten SVM kann dann in der 2. Phase die Klassifikation 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 6.3 beschrieben.

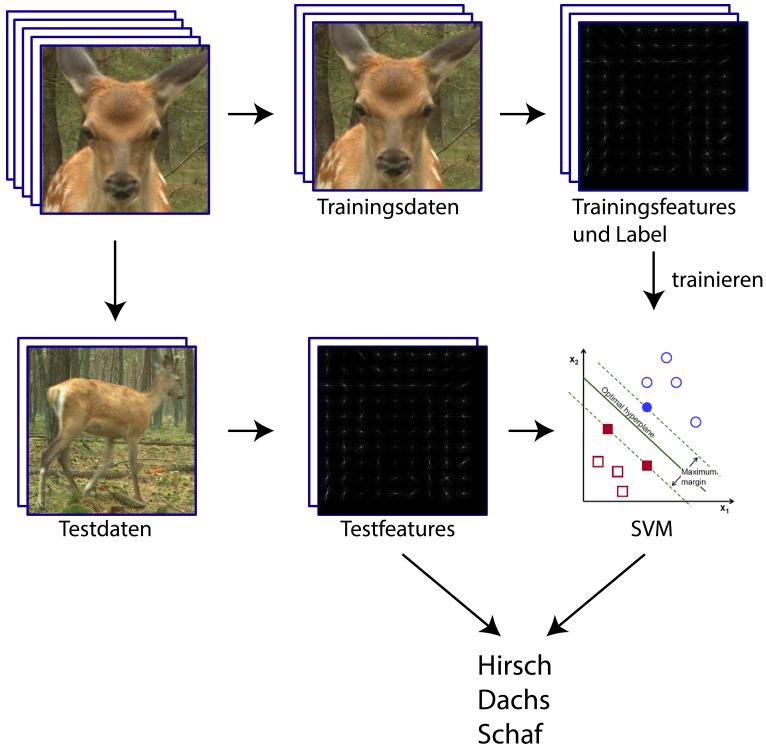


Abbildung 9: Schaubild zur Klassifizierung mit einer SVM. Der obere Pfad repräsentiert die Trainingsphase und der untere Pfad die Testphase. (Das SVM Bild (rechts unten) wurde aus der OpenCV Dokumentation übernommen [SVM1])

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 Szenen benutzt, um 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.

Die Features aus jedem der insgesamt 21 Bins werden anschließend über die Lösung eines Optimierungsproblems den Features 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 der Datenbank berechnet und anschließend mithilfe von Clustering Repräsentanten bestimmt. Bei

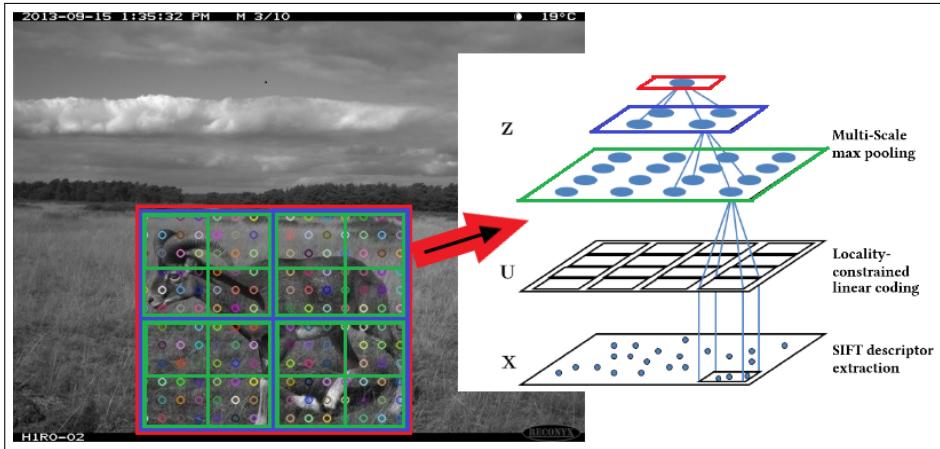


Abbildung 10: Architektur des Algorithmus mit Veranschaulichung der Spatial Pyramid auf SIFT-Features und Pooling. Im Anschluss werden der L_0 - (rot), die vier L_1 - (blau) und 16 L_2 -Codes (grün) konkateniert - angelehnt an [Yang et al. 09].

einigen Verfahren wird das Codebook auch online weiter optimiert, worauf wir jedoch verzichten. 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 mit größerem 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 beispielsweise 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 effizientem 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.

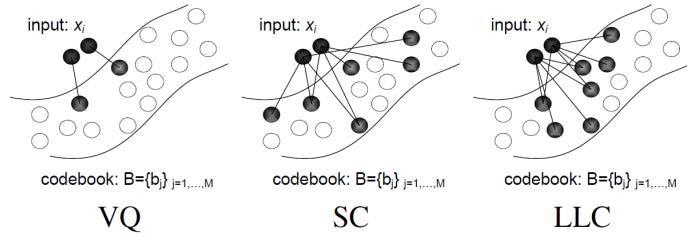


Abbildung 11: Vergleich der drei Kodierungsstrategien Vector Quantization, Sparse Coding und Locality-constrained Linear Coding [Wang et al. 10].

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. Unglücklicherweise kommt es dabei zu schlechten Zuordnungen, wenn es beispielsweise keine Feature im Codebook gibt, das zu f ähnlich ist (1) oder wenn zwei recht ähnliche Features f und g unterschiedlichen Features in B zugeordnet werden (2). Fehler dieser Art nennt man *Quantisierungsfehler*.

Eine deutliche Verbesserung stellte das Verfahren von [Yang et al. 09]. 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 Dünnsbesetztheit (*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 widerspiegeln.

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. $d_i = \exp\left(\frac{\text{dist}(x_i, B)}{\sigma}\right) \in \mathbb{R}^M$ hingegen 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:

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

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

[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 ein 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 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 aber 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 dabei Möglichkeit dar, die Informationen aller Kodierungen aus C in einem Deskriptor zu bündeln und die aussagekräftigsten Informationen dabei zu erhalten. Aufgrund der Assoziativität der Pooling-Operationen ist es nicht nötig alle 21 Bins einzeln zu kodieren. Das erspart einen Menge 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 *Multiscale 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 normalization: $c_{out} = c_{in} / \sum_j c_{in}(j)$
- ▷ l^2 normalization: $c_{out} = c_{in} / \|c_{in}\|_2$

5.2.4 Klassifizierung mit Support Vector Machines

Die LLC-Codes lassen sich mit einer Support Vector Machine (SVM) 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 einen 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 Pixel 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 dem 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 (s. 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 mit Hilfe 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.

6 Evaluierung

-hinweis auf laufzeit und klassifizierung -pca nicht testbar, da keine ground-truth bilder

6.1 Daten

-hoge veluwe -kamerafallenbilder, sequenzen -größe -spezies -DDD/DDD+

6.2 Laufzeit Spatial Pyramid Matching

-sequentiell -multiprocessing -numba -hauptproblem: lösung des gleichungssystems -ausblick:
umsetzung mit tensorflow

6.3 Experimentelle Optimierung und Auswertung 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 Tagesbilder von Dachs und Damhirsch verwendet.

6.3.1 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 automatische ROI-Selektion mit PCA auszuschließen. Später wurde die SVM jedoch mit automatisch selektierten ROIs trainiert und klassifiziert.

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 bestimmt. 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 angewandt.

Ein weiteres Problem der geringen Menge der Daten für den Dachs war, dass bei prozentualer Selektion der Trainingsdaten der Großteil aus Damhirschkörpern 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.

6.3.2 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 wie viele 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 waren für die Kostenfunktion $C = 12,5$ 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 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.

6.3.3 Ergebnisse

Die in Kapitel 6.3.2 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

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	$\bar{p} = 0.833, p_{min} = 0.763, p_{max} = 0.930, \sigma = 0.032$

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.

6.4 Güte der Klassifizierungstechnike

-HOG: auf DDD und DDD+ -SPM: -random search cv mit fünf folds -scipy reciprocal -versch. Größen
-SIFT Güte auf DDD -LBP Güte auf DDD -SIFT Güte auf DDD+ -kombination von SIFT und LBP auf DDD+

7 Fazit

Das Ziel des ersten Teils dieser Arbeit war es einen automatischen Detektor und Klassifizierer für die Damhirsch und Dachs Datenbank zu erstellen. Eine Schwierigkeit dieser Datenbank ist es, dass die Bildermenge, insbesondere für den Dachs, sehr gering ist. Dadurch ist es nicht möglich einen Klassifikator mit vielen Bildern zu trainieren. Um dieses Problem abzuschwächen wurde die Trainingsdatenmenge durch Spiegelung der Bilder künstlich verdoppelt. Außerdem war es ein Ziel eine Präzision von über 80 % zu erreichen. Mit dem von uns in Kapitel 4 gezeigtem Ansatz konnte dieses Ziel erreicht werden. Es wurde eine durchschnittlichen Präzision von über 83 % erreicht. Ein entscheidender Faktor dabei war die Wahl der besten experimentell bestimmten Parameter. Ein besonderer Vorteil der hier verwendeten Methode ist die hohe Geschwindigkeit mit der das Modell trainiert werden kann. Eine weitere Erhöhung der Geschwindigkeit könnte zusätzlich erreicht werden, wenn zur Umsetzung eine Hardware nähere Sprache wie C++ verwendet würde. Für diese konzeptionelle Arbeit ist das Resultat allerdings bereits als sehr gut einzurordnen. Der Klassifizierer eignet sich jedoch nicht für eine Klassifizierung von mehr als 2 verschiedene Klassen, da in diesem Experiment eine zu geringe Präzision erreicht wurde.

-kurzbeschreibung der verfahren und ergebnisse -ausblick: -tensorflow: schnellere laufzeit -> training mit größeren codebooks auf mehr daten -ensemblemethoden zur besseren kombination von sift und clbp: soft voting (SVC, statt LinearSVC), boosting

Anleitung

Im folgendem wird eine kurze Anleitung zur Verwendung des Programms gegeben.

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) und Numba (Version 0.43.0).

Sequenzierung

PCA

Klassifizierung mit HOG

SPM

Literatur

- [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.
- [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.
- [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.
- [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).

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)