

**Diplomarbeit**

**Flugzeugerkennung in Satellitenbildern**

**Anwendung von Kaskaden-Klassifizierern nach**

**Viola und Jones**



Felice Serra  
Mtrklnr.: 2944562  
[felice.serra@gmx.de](mailto:felice.serra@gmx.de)  
Wilhelm-Schickard-Institut für Informatik  
Lehrstuhl für Graphisch-Interaktive Systeme  
Prof. Dr. Schilling

SoSe 2015



## Inhaltsverzeichnis

<b>1. Übersicht</b>	<b>1</b>
<b>2. Studienarbeit zur Flugzeugerkennung</b>	<b>2</b>
2.1. Motivation . . . . .	2
2.2. Methode . . . . .	2
2.3. Ergebnis . . . . .	4
<b>3. Theorie</b>	<b>6</b>
3.1. Viola-Jones-Methode . . . . .	6
3.2. Aufbau des Kaskaden-Klassifizierers . . . . .	6
3.3. Merkmalsbeschreibung . . . . .	7
3.3.1. Haar-ähnliche Merkmale . . . . .	8
3.3.1.1. Beschleunigte Berechnung mit Integral-Bildern . . . . .	9
3.3.2. Local Binary Pattern (LBP) . . . . .	10
3.4. Vor- und Nachteile . . . . .	12
<b>4. Training</b>	<b>14</b>
4.1. Vorbereitung . . . . .	14
4.1.1. Google-Static-Maps-API . . . . .	14
4.1.2. Negative . . . . .	14
4.1.3. Positive . . . . .	15
4.1.4. Die Methode <i>opencv_createsamples</i> . . . . .	16
4.1.5. Anwendung der <i>opencv_createsamples</i> -Methode . . . . .	17
4.2. Training des Klassifizierers . . . . .	19
<b>5. Anwendung</b>	<b>21</b>
5.1. Verwendete Software, Programmiersprachen und Computerspezifikation	21
5.2. Implementation - Ablauf der Objekterkennung . . . . .	21
5.3. Die Methode <i>detectMultiScale</i> . . . . .	24
<b>6. Auswertung</b>	<b>28</b>
6.1. Das Testset . . . . .	28
6.2. Einfluss der Rotation der positiven Samples . . . . .	29



6.3. Parameterwahl im Training . . . . .	31
6.3.1. Vergleich von Haar-ähnlichen Merkmalen und <i>Local-Binary-Pattern</i> . . . . .	31
6.3.1.1. Visualisierung der Merkmale . . . . .	33
6.3.2. Einfluss der Stufenanzahl . . . . .	34
6.4. Aus Fehlern lernen . . . . .	35
<b>7. Optimierung der Laufzeit</b>	<b>37</b>
7.1. Großflächiges Suchen . . . . .	37
7.2. Implementierung in C++ . . . . .	38
7.3. Verwendung einer größeren Zoomstufe . . . . .	38
7.4. Anpassung der Rotation . . . . .	38
7.5. Anpassung der maximal Größe . . . . .	39
7.6. Der Skalierungsfaktor . . . . .	40
7.7. Skalierungsfreie-Detektion . . . . .	41
7.8. Zusammenfassung . . . . .	43
7.8.1. Speicherleck . . . . .	45
<b>8. Fazit</b>	<b>46</b>
<b>Literaturverzeichnis</b>	<b>47</b>
<b>A. Anhang</b>	<b>i</b>
A.1. Eidesstattliche Erklärung . . . . .	i
A.2. Adresse . . . . .	i



## 1. Übersicht

Das Ziel dieser Diplomarbeit ist es, eine Methode zur Objekterkennung von Flugzeugen in Satellitenbildern zu erarbeiten. Zur Anwendung kommt hier ein, von Viola und Jones entwickelter Algorithmus zur Objekterkennung [Viola und Jones 2001]. Dabei wird ein Kaskaden-Klassifizierer eingesetzt. Dieser wird trainiert mit *Local-Binary-Pattern* und Haar-ähnlichen Merkmalen.

Im Folgenden wird zunächst ein Überblick über die Theorie der verwendeten Algorithmen zur Objekterkennung gegeben. Anschließend wird die Erstellung der Trainingsdaten, das Trainieren verschiedener Klassifizierer und deren Anwendung beschrieben. Die erhaltenen Daten werden ausgewertet und analysiert.

Zuerst wird versucht eine möglichst hohe Treffer-Genauigkeit zu erreichen. Um eine großflächige Suche auf den Kartendaten von Google-Maps zu ermöglichen, wird daraufhin die Laufzeit der Objekterkennung noch optimiert. Abschließend wird das Ergebnis zusammengefasst und diskutiert.

Das Projekt ist in Java geschrieben und nutzt die OpenCV-Bibliothek zur Bildbearbeitung und Objekterkennung. Auch die Methode von Viola und Jones findet sich in dieser Bibliothek.

Es soll an dieser Stelle auch erwähnt sein, dass dieses Projekt keinerlei militärischen Bezug hat. Es dient lediglich als Fingerübung zum Erwerb von Fachwissen im Bereich der Objekterkennung.



## 2. Studienarbeit zur Flugzeugerkennung

### 2.1. Motivation

Dieser Diplomarbeit geht eine Studienarbeit vom selben Autor voraus. Das Thema ist: *Anwendung des SURF-Algorithmus - Flugzeugerkennung in Satellitenbildern*. Sie diente als Motivation für diese Diplomarbeit. Die in der Studienarbeit erhaltenen Ergebnisse waren schlecht und so unbefriedigend, dass ein weiterer Versuch zur Flugzeugerkennung in Satellitenbildern, im Rahmen dieser Diplomarbeit, unternommen wurde.

### 2.2. Methode

In der Studienarbeit werden zur Objekterkennung die sogenannten SURF-Merkmale (*Speeded-Up-Robust-Features*) [Bay u. a. 2006] eingesetzt. Diese werden für das Objekt berechnet und dann mit den Merkmalen der Szene, dem Bild, in dem gesucht wird, verglichen. Dabei kommt, im Gegensatz zur Diplomarbeit, kein Lern-Algorithmus zur Anwendung. Der große Vorteil der SURF-Merkmale ist ihre Rotations-Invarianz. Das heißt, ein Merkmal kann beliebig rotiert werden und kann danach immer noch als das Selbe identifiziert werden. Die SURF-Merkmale besitzen daher auch eine Ausrichtung, die ausgelesen und verglichen werden kann. Diese Merkmale sind außerdem auch skalierungs-invariant. Das bedeutet, sie können auch nach Vergrößerungen oder Verkleinerungen noch identifiziert werden. Da Flugzeuge auf Satellitenbildern in unterschiedlichen Größen und in unterschiedlichen Flugrichtungen vorliegen, sollten die SURF-Merkmale für die Flugzeugerkennung geeignet sein. Sie werden wie folgt berechnet. Zuerst wird nach sogenannten *Points of Interest* gesucht:

”*Points of Interest* sind markante Punkte eines Bildes, wie z.B. Ecken, Blobs (zusammenhängende Bereiche gleicher Intensität) oder T-Kreuzungen von Kanten. Bei der Suche nach den *Points of Interest* wird ein Blob-Detektor basierend auf der Hesse-Matrix angewandt. Dabei werden Blob-ähnliche Strukturen dort gesucht, wo die Determinante der Hesse-Matrix am größten ist.” [Serra 2015]



Danach werden die SURF-Deskriptoren berechnet:

”Zuerst wird eine kreisförmige Region um den *Point Of Interest* nach markanten Blobs abgesucht. Mit diesen Informationen wird die Orientierung des *Point Of Interests* festgelegt. Im zweiten Schritt wird ein quadratischer Bereich gebildet, der der vorher festgelegten Orientierung folgt. Dieser Bereich wird wieder in kleinere Rechtecke unterteilt und der SURF-Deskriptor wird erzeugt, indem man *Haar-Wavelet-Responses* [Haar 1910] in x- und y-Richtung berechnet. Die Aufsummierung der einzelnen *Haar-Wavelet-Responses* in einem Vektor ergibt den SURF-Deskriptor des *Point Of Interests*.” [Serra 2015]

Um mit den SURF-Merkmalen Objekterkennung zu betreiben, werden zuerst die Merkmale des gesuchten Objektes berechnet. Diese werden dann mit den Merkmale in der Szene verglichen. Dabei wird aus der Differenz beider Merkmals-Deskriptoren eine Abstands-Metrik berechnet, die angibt wie ähnlich die beiden Merkmale sind. Nun wird versucht diejenige Transformation zu finden, die die Objektmerkmale auf die Szenemerkmale abbildet und dabei die geringste Differenz aufweist. Über schreitet diese Differenz eine gewisse Schranke, so gilt das Objekt als nicht gefunden. Andernfalls kann aus der Transformations-Abbildung Position und Ausrichtung des gesuchten Objekts in der Szene bestimmt werden.

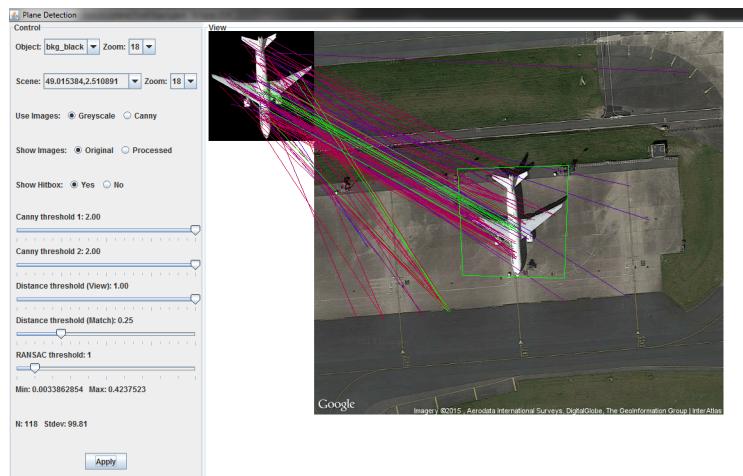


Abbildung 2.1.: Studienarbeit: Graphische Benutzeroberfläche.

Getestet wurde die Methode in 3 verschiedenen Zoomstufen (16, 17, 18) an Graustufenbildern und an Gradientenbildern. Das sind Bilder, die als Ergebnis einer Kan tenerkennung entstehen. Dabei wird nach großen Unterschieden im Graustufenwert zweier benachbarter Pixel gefiltert. Die dabei verwendete Methode ist der Canny Algorithmus [Canny 1986], der ebenfalls in OpenCv implementiert ist.

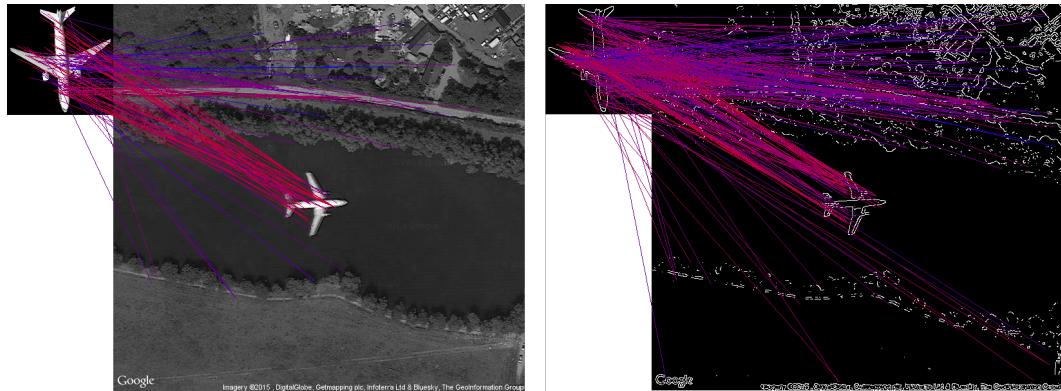


Abbildung 2.2.: Studienarbeit: Graustufenbild und Gradientenbild im Vergleich.  
Merkmalspaare sind durch eine Linie verbunden.

Es wurden außerdem unterschiedliche Hintergrundfarben für das Objekt getestet. Das Flugzeug wurde auf einem schwarzen, weißen und grauen Hintergrund platziert. Um die Einflüsse aller involvierten Parameter zu untersuchen, wurde eine graphische Benutzeroberfläche erstellt. In dieser ließen sich die Schwellwerte der benutzten Algorithmen beliebig einstellen, verschiedene Zoomstufen, Hintergrundfarben und Szenen auswählen, sowie zwischen Graustufen und Gradientenbildern wählen. Dadurch wurde es möglich, eine optimale Wahl für alle Parameterwerte zu finden.

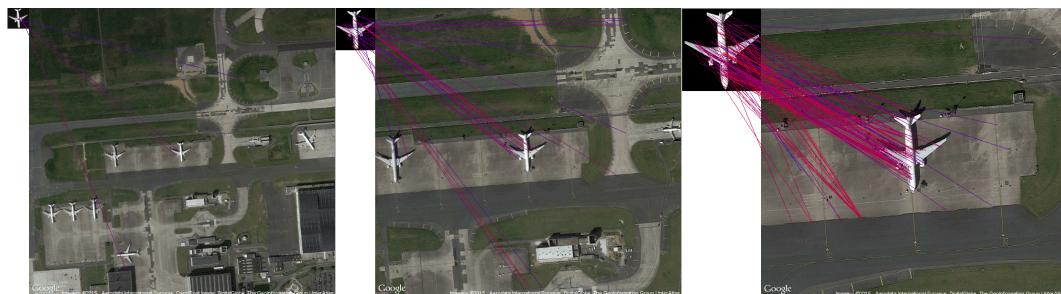


Abbildung 2.3.: Studienarbeit: Zoomstufe 16, 17 und 18 im Vergleich. Merkmalspaare sind durch eine Linie verbunden.

### 2.3. Ergebnis

Zusammenfassend lässt sich sagen, dass die SURF-Merkmale, so wie sie hier eingesetzt wurden, ungeeignet sind für die Flugzeugerkennung. Ein Flugzeug lässt sich in erster Linie durch seine kreuzförmige Form und seine typischen Proportionen, das Verhältnis von Spannweite zu Rumpflänge, erkennen. Von allen SURF-Merkmalen, die für das Flugzeug gebildet werden, beschreiben nur sehr wenige, diese abstrakte Form des Objekts. Die meisten Merkmale beziehen sich auf kleinere Details, die



zur Flugzeugerkennung ungünstig sind. Außerdem werden die Merkmale auf einem gleichfarbigen Hintergrund gebildet, was zusätzlich die Vergleichbarkeit mit realen Beispielen erschwert. Ein Auszug aus dem Fazit der Studienarbeit lautet:

”Es konnten nur zwei Flugzeuge in den insgesamt 16 Szenen gefunden werden. Dieses schlechte Ergebnis kann nicht auf eine suboptimale Wahl der Parameter zurückgeführt werden. Es liegt vielmehr in der Natur der Objekte und der Art der Merkmals-Extraktion des SURF-Algorithmus. Ein Flugzeug wird vor allem durch seine Form und durch spezifische Proportionen zwischen Rumpf und Tragfläche definiert. Der SURF-Algorithmus sucht nach markanten Merkmalen in verschiedenen Skalierungsebenen. Dabei geben nur die großflächigen Merkmale auf höchster Ebene diese, für ein Flugzeug, spezifischen Merkmale der Form wieder. Merkmale auf niedrigeren Ebenen beziehen sich auf Details die am Flugzeugrumpf gefunden werden, wie z.B. ein Logo oder Antennenvorrichtungen. Diese Merkmale unterscheiden sich aber bei verschiedenen Flugzeugtypen und sind daher nicht geeignet für die Objekterkennung.”  
[Serra 2015]



## 3. Theorie

### 3.1. Viola-Jones-Methode

Mit dem Aufkommen von digitalen Foto- und Videokameras entwickelten Paul Viola und Michael Jones 2001 eine Methode zur Objekterkennung in digitalen Bildern [Viola und Jones 2001]. Ihr Hauptaugenmerk lag dabei auf einer Möglichkeit der Gesichtserkennung in Echtzeit. Die Methode erlangte, wegen ihrer Effizienz, weltweite Popularität. Sie ist die erste Methode mit der es gelang Gesichter in Echtzeit zu erkennen. Sie nutzt den AdaBoost-Algorithmus zum Training eines Kaskaden-Klassifizierers, kombiniert mit Haar-ähnlichen Merkmalen, eine Weiterentwicklung der bis dahin gängigen Haar-Wavelets. Eine Implementierung findet sich in OpenCv in den dort zur Verfügung stehenden Methoden für Bildbearbeitung und maschinelles Sehen.

In dieser Diplomarbeit wird die Viola-Jones-Methode genutzt um einen Klassifizierer für Flugzeuge in Satellitenbildern zu trainieren und anzuwenden. Im Folgenden wird die allgemeine Funktionsweise der Methode beschrieben.

### 3.2. Aufbau des Kaskaden-Klassifizierers

Im Allgemeinen versteht man unter einer Kaskade einen mehrstufigen Wasserfall. Angelehnt an dieses Bild, verdanken Kaskaden-Klassifizierer ihren Namen ihrem stuifenweisen, kaskadenähnlichen Aufbau. Dabei entspricht jede Stufe einem sogenannten starken Klassifizierer (engl. *strong classifier*). Diese werden in Reihe geschaltet und bilden zusammen den eigentlichen Klassifizierer.

Ein starker Klassifizierer ist eine Linearkombination gewichteter sogenannter schwacher Klassifizierer (engl. *weak classifier*). Ein schwacher Klassifizierer behandelt nur ein einzelnes Merkmal, z.B. ein Haar-ähnliches Merkmal an einer bestimmten Position. Da ein Merkmal allein nicht sehr aussagekräftig ist, wird ein solcher Klassifizierer *schwach* genannt. Erst in Kombination mit weiteren schwachen Klassifizierern, wird



### 3. Theorie

---

daraus ein starker Klassifizierer.

Ein schwacher Klassifizierer  $h_i$  kann wie folgt beschrieben werden:

$$h_j(\mathbf{x}) = \begin{cases} -s_j & \text{if } f_j < \theta_j \\ s_j & \text{sonst} \end{cases}$$

Wobei  $s_j \in \pm 1$ .  $f_j$  ist das zugewiesene Merkmal. Die Polarität von  $s_j$  und der Schwellwert  $\theta_j$  werden während des Trainings bestimmt. In Linearkombination mit anderen schwachen Klassifizierern ergibt sich dann ein starker Klassifizierer  $h$ :

$$h(\mathbf{x}) = \text{sign} \left( \sum_{j=1}^M \alpha_j h_j(\mathbf{x}) \right)$$

Die Gewichtungen  $\alpha_j$  der schwachen Klassifizierer werden ebenfalls im Training ermittelt.

Die Grundidee der in Reihe geschalteten Klassifizierer ist es, mit einem sehr allgemein gehaltenen Klassifizierer zu beginnen. Dieser wird so gewählt, dass er eine 100% Trefferquote hat. Die Falsch-Positiv-Rate liegt dadurch meist noch sehr hoch bei circa 40%. Dieser Mechanismus bewirkt, dass schon am Anfang viele der eingehenden Fenster verworfen werden können. Da in den meisten Anwendungen, das gesuchte Objekt nur in einem Bruchteil der zu überprüfenden Fenstermenge liegt, wird dadurch der gesamte Prozess erheblich beschleunigt. Nun wird mit jeder Stufe der Mechanismus wiederholt und so die Klassifikation stufenweise verfeinert, bis am Ende eine gewünschte Treffer- und Falsch-Positiv-Rate erreicht ist. Die Trefferquote eines Klassifizierers ist immer abhängig von der Trefferquote des vorangegangenen Klassifizierers:

$$F = \prod_{i=1}^K f_i$$

Dasselbe gilt für die Falsch-Positiv-Rate:

$$D = \prod_{i=1}^K d_i$$

### 3.3. Merkmalsbeschreibung

Um ein Objekt in einem Bild zu erkennen, muss definiert werden, was dieses Objekt von anderen Objekten unterscheidet. So wird nach objektspezifischen Formen, Farben und Kontrasten gesucht. Diese charakteristischen Eigenschaften werden Merk-



### 3. Theorie

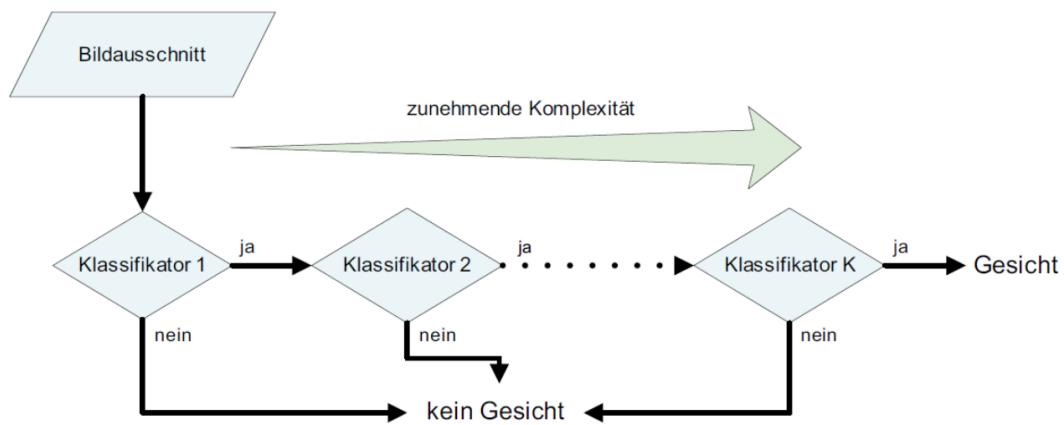


Abbildung 3.1.: Ablauf der Objekterkennung im Kaskaden-Klassifizierer. [Manuel Walter 2011]

male (engl. *Features*) genannt. Ein Teil der Objekterkennung besteht darin diese Merkmale mathematisch zu erfassen und zu beschreiben. Eine gute Merkmalsbeschreibung sollte schnell zu berechnen sein und robust sein gegenüber leichten Abweichungen, wie z.B. Helligkeitsunterschiede, Rotationen oder Ähnliches. Zwei Arten der Merkmalserzeugung werden in diesem Projekt verwendet und in diesem Kapitel besprochen: Die Haar-ähnlichen Merkmale und die *Local Binary Pattern*.

#### 3.3.1. Haar-ähnliche Merkmale

Haar-ähnliche Merkmale wurden benannt nach ihrer Ähnlichkeit zu den *Haar-Wavelets* [Haar 1910]. Sie dienen der Erzeugung von bildspezifischen Merkmalen zur Objekterkennung. Entwickelt wurden sie für den Algorithmus zur Gesichtserkennung von Viola und Jones. Sie werden beschrieben durch rechteckige, schwarz-weiße Masken,

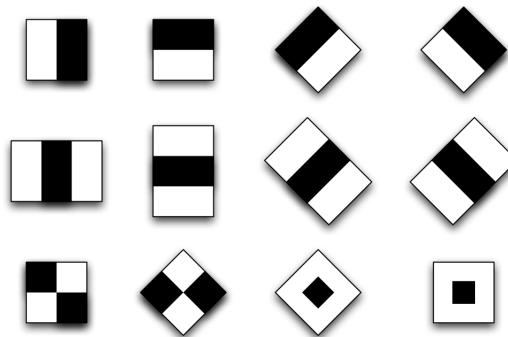


Abbildung 3.2.: Beispiel-Masken für Haar-ähnliche Merkmale. [Wikipedia 2015b]

die über einen Bereich des Bildes gelegt werden. Die Intensitätswerte der Pixel innerhalb der schwarzen und weißen Regionen werden getrennt aufsummiert. Danach



### 3. Theorie

wird die Differenz beider Summen berechnet. Diese Summe ist charakteristisch für den Bildbereich und die verwendete Maske. Je höher der Wert ist, desto ähnlicher sind der Bildbereich und die verwendete Maske.



Abbildung 3.3.: Ein Beispiel aus der Gesichtserkennung: Der Bereich um die Augen ist deutlich dunkler als der darunter. Die verwendete Haar-Maske liefert also einen hohen Wert. [Wikipedia 2015b]

Da ein einzelnes Haar-ähnliches Merkmal für die Objekterkennung wenig aussagekräftig ist, werden meist mehrere unterschiedliche Masken gewählt. Jede Maske reagiert sensitiv auf eine bestimmte Art von Merkmal, z.B. eine horizontale Linie. Um auch Merkmale wie z.B.  $45^\circ$  Winkel zu erfassen, entwickelten Lienhart und Maydt um  $45^\circ$  gedrehte Masken [Lienhart und Maydt 2002]. Es wurde ebenfalls versucht daraus eine Rotationsinvariante Version der Haar-ähnlichen Merkmale zu konstruieren. Diese Variante scheiterte jedoch daran, dass in der Praxis für zeitkritische Anwendungen meist Bilder mit niedriger Auflösung verwendet werden. Dies führt oft zu Rundungsfehlern bei Rotationen und zu schlechten Ergebnissen.

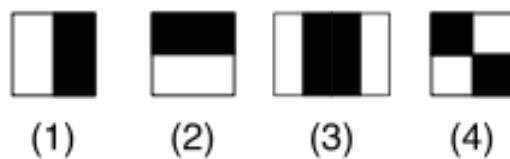


Abbildung 3.4.: Diese vier Masken für Haar-ähnliche Merkmale finden Verwendung in der Gesichtserkennung von Viola und Jones. [Wikipedia 2015b]

#### 3.3.1.1. Beschleunigte Berechnung mit Integral-Bildern

Um die Berechnung der Haar-ähnlichen Merkmale zu beschleunigen, werden sogenannte Integralbilder eingesetzt. Sie erlauben ein schnelles Berechnen von Summen von Pixelwerten in einer rechteckigen Region. Ein Integralbild wird aus dem ursprünglichen Bild erzeugt. Dabei wird iterativ jedem Pixel ein Wert zugewiesen. Ein Wert an den Koordinaten  $(x, y)$  im Integralbild entspricht der Summe der Pixel in



### 3. Theorie

1	1	1
1	1	1
1	1	1
1	1	1

Originalbild

1	2	3
2	4	6
3	6	9
4	8	12

Integralbild

Abbildung 3.5.: Beispiel für ein Integralbild.

einem Rechteck vom Ursprung bis zum Punkt  $(x, y)$  im Originalbild. Das so entstandene Integralbild kann nun dazu genutzt werden, die rechteckigen Regionen der Haar-Masken zeit-effizient aufzusummieren. Die Summe der Pixel in einem Rechteck kann damit in nur drei Additionen berechnet werden. Anstatt also für jede Haar-Maske jedes Mal einzelne Pixelwerte aufzusummieren, wird zu Beginn einmal das Integralbild berechnet und dieses wird dann dazu genutzt, in nur drei Additionen pro rechteckiger Region, die Summe zu berechnen. Dabei braucht es nur 4 Speicherzugriffe auf die Eckpunkte der Region (Siehe Abbildung 3.6). Dies beschleunigt den Prozess erheblich.

Bei der iterativen Erstellung des Integralbildes kann außerdem auf schon berechnete Regionen zurückgegriffen werden, sodass die Summe nicht für jeden Pixel komplett neu berechnet werden muss. So wird auch dieses Verfahren beschleunigt.

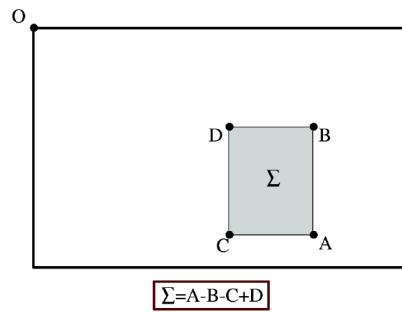


Abbildung 3.6.: Mit Hilfe des Integralbildes lässt sich die Summe der Pixel innerhalb des Rechtecks in nur drei Additionen berechnen.

### 3.3.2. Local Binary Pattern (LBP)

Die *Local Binary Pattern* (LBP) sind wie die *Haar-like Features* eine Technik zur Merkmalsbeschreibung von Bildpunkten [Ojala u. a. 1996]. Die LBP-Merkmale lassen sich im Vergleich zu den Haar-Merkmalen einfacher und schneller berechnen. Allerdings liefern sie im Gegenzug auch eine weniger detaillierte Beschreibung der Bild-



### 3. Theorie

punkte (Siehe Kapitel: Vergleich von Haar-ähnlichen Merkmalen und *Local-Binary-Pattern*, S.31).

Der Algorithmus arbeitet auf Graustufenbildern. Die einfachste Form eines LBP-Merkals lässt sich folgendermaßen berechnen:

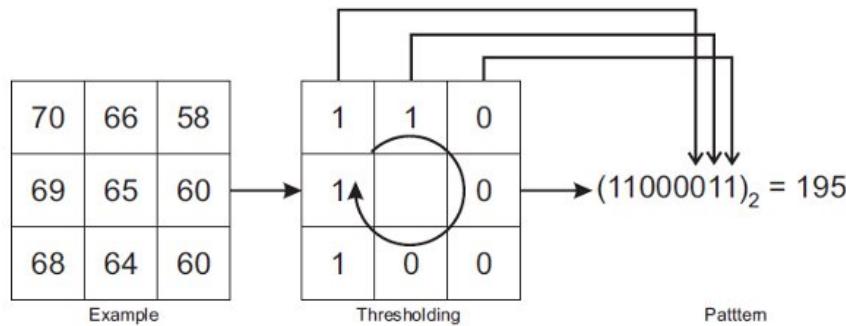


Abbildung 3.7.: Beispiel für die Berechnung eines einfachen *Local Binary Patterns* (LBP). Der Vergleich mit den acht Nachbarn liefert einen 8-Bit Vektor, der in eine Dezimalzahl umgerechnet wird. [Silva u. a. 2013]

Der Wert des aktuellen Bildpunktes wird mit den Werten aller benachbarter Bildpunkte verglichen. Man startet links-oben mit dem ersten Nachbarn. Ist dessen Graustufen-Wert höher als der des aktuellen Bildpunktes, so wird dem Merkmals-Vektor eine 1 angehängt. Ist der Wert niedriger, wird eine 0 hinzugefügt. Das Ganze wird so im Uhrzeigersinn fortgeführt, bis alle benachbarten Bildpunkte verglichen wurden. Der so entstandene Vektor wird als binäre Schreibweise einer Dezimalzahl aufgefasst und umgerechnet. Die dadurch erhaltene Zahl ist das LBP-Merkmal des Bildpunktes.

Nun kann mit den erhaltenen Dezimalzahlen der Bildpunkte ein Histogramm des Bildes erzeugt werden. Darin wird die Häufigkeit einzelner Zahlen abgetragen. Optional kann dieses Histogramm auch noch normalisiert werden. Zwei unterschiedliche Bilder können so über die Ähnlichkeit ihrer Histogramme verglichen werden.

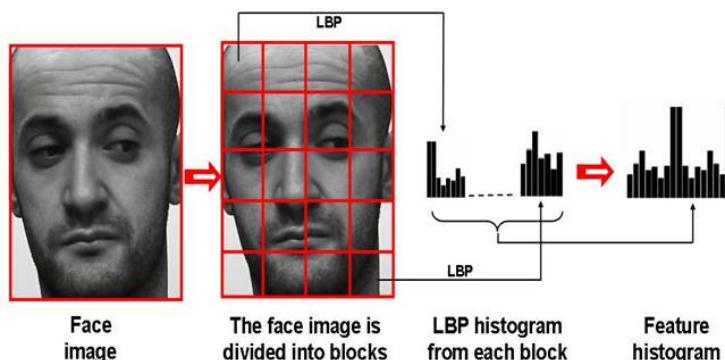


Abbildung 3.8.: Gesichtserkennung mit Hilfe von *Local Binary Pattern* (LBP) und lokalen Histogrammen. [advancedsourcecode 2015]



### 3. Theorie

Da in der Histogramm-Form, für die Objekterkennung wichtige, räumliche Informationen verloren gehen, empfiehlt es sich, das Bild in einzelne lokale Histogramme zu unterteilen. Diese können dann zu einer großen Merkmalsbeschreibung zusammengefasst werden.

Erwähnenswert ist außerdem, dass die allgemeine Form der LBP sich nicht nur auf die direkten Nachbarn eines Bildpunktes beschränkt. Die Idee wird auf eine kreisförmige Umgebung erweitert. Dabei wird ein  $LBP(P,R)$  mit den Parametern  $P$  und  $R$  beschrieben.  $P$  gibt die Anzahl der zu vergleichenden Bildpunkte an und damit auch die Länge des entstehenden Vektors.  $R$  definiert den Radius des Kreises auf dem die zu vergleichenden Bildpunkte liegen.

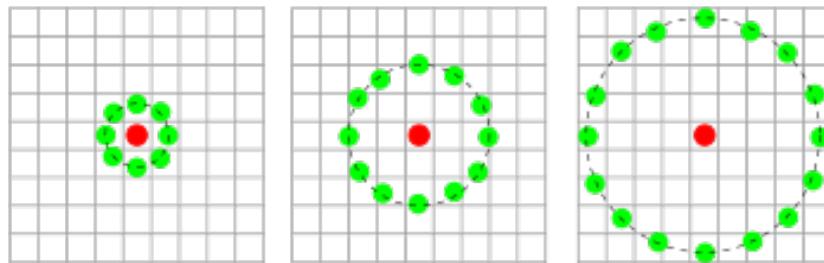


Abbildung 3.9.: Beispiel für allgemeine  $LBP(P,R)$  mit  $P$  Nachbarn und Radius  $R$ .  
[Wikipedia 2015a]

## 3.4. Vor- und Nachteile

Die Viola-Jones-Methode hat einige Vorteile. Einer davon ist, dass sie nicht nur auf Gesichter anwendbar ist, sondern ein Werkzeug für eine allgemeine Objekterkennung bietet. So eignet sie sich zum Beispiel auch zur Flugzeugerkennung in diesem Projekt. Ein weiterer Vorteil ist ihre sehr schnelle Merkmalsberechnung und die effiziente Merkmalsauswahl. Dadurch wird eine Objekterkennung in Echtzeit ermöglicht. Das kommt auch diesem Projekt zu Gute. Es ist zwar nicht Ziel der Arbeit eine zeitkritische Anwendung zu entwickeln, jedoch müssen, für eine großflächige Suche nach Flugzeugen in beliebigen Gebieten, sehr viele Bilder bearbeitet werden.

Bei dieser Methode muss außerdem das Bild nicht in unterschiedlichen Skalierungen berechnet werden, wie bei anderen Methoden, da statt dessen die Merkmale skaliert werden. Dies trägt ebenfalls zur schnellen Berechenbarkeit bei.

Weiterhin von Vorteil ist, dass diese Methode skalierungs-invariant ist. So lassen sich leichter Flugzeuge unterschiedlicher Größe, bedingt durch Flugzeugtyp und Flughöhe, erkennen.



Der wohl größte Nachteil der Viola-Jones-Methode ist, dass sie nicht rotationsinvariant ist. Weicht der Winkel des gesuchten Objekts im Bild zu sehr von dem des gelernten Objekts ab, wird dieses nicht als solches erkannt. Das ist bei der Flugzeugerkennung ein großes Problem, da Flugzeuge in allen Flugrichtungen erkannt werden sollen. Hier muss daher das Bild rotiert werden oder es müssen mehrere Klassifizierer eingesetzt werden oder eine Kombination von beidem.

Ein weiterer Nachteil der klassischen Methode, ist die Verwendung der Haar-ähnlichen Merkmale. Diese sind zwar sehr gut geeignet um horizontale und vertikale Merkmale zu beschreiben, eignen sich aber nicht gut, wenn diese in einem  $45^\circ$  Winkel vorliegen. Wenn also ein Klassifizierer für nicht horizontal oder vertikal ausgerichtete Flugzeuge gebraucht wird, sollte auf die Haar-ähnlichen Merkmale verzichtet werden.

Ebenfalls ein Problem sind die verschiedenen Helligkeitsunterschiede im Bild.

Die Methode funktioniert für helle Flugzeuge auf dunklem Hintergrund. Ist das Flugzeug allerdings dunkler als der Hintergrund, wird es nicht erkannt. Dies tritt auf, bei einer dunklen Flugzeug-Lackierung oder bei ungünstigen Landschaften und Lichtverhältnissen.

Dass bei diesem Algorithmus, iterativ, kleine Fenster über das Bild gelegt werden, hat den ungünstigen Nebeneffekt, dass ein und das selbe Objekt möglicherweise mehrmals erkannt wird. So entstehen mehrere mögliche Hitboxen um das Objekt, die eine Nachbearbeitung erfordern, um den Treffer genau identifizieren zu können.



## 4. Training

### 4.1. Vorbereitung

#### 4.1.1. Google-Static-Maps-API

Mit Hilfe der Google-Static-Maps-API lassen sich Satellitenbilder an gewünschten Positionen mit unterschiedlichen Größen und Zoom-Stufen generieren. Dazu können gewünschte Parameter in einer URL-Anfrage spezifiziert werden. Der Parameter *center* gibt die Bildmitte an, *zoom* die Zoom-Stufe, *size* die Bildgröße, *format* das Format der ausgegebenen Datei, *maptype* die Art der Kartenansicht. Der letzte Parameter *key* ist optional. Diesen Schlüssel erhält man durch Anmeldung bei der *Google Developer Console*. Die Anmeldung ist kostenlos und hat den Vorteil, dass bis zu 25000 Kartenanfragen pro Tag getätigt werden können. Bei Zugang ohne Produktschlüssel ist das Kontingent deutlich niedriger. Es sind noch weitere Parameter definierbar, doch sind diese, für die Zwecke dieses Projektes, nicht relevant.

```
String imageUrl = "http://maps.googleapis.com/maps/api/staticmap?" +
    "center=49.015384,2.510891" +
    "&zoom=17" +
    "&size=640x600" +
    "&format=png32" +
    "&maptype=satellite" +
    "&key=xxxxxxxxxxxxxxxxxxxxxx";
```

#### 4.1.2. Negative

Um einen Klassifizierer zu trainieren braucht man Negativ- und Positiv-Beispiele. Die Negativ-Beispiele wurden aus 1000 Bildern mit zufälligen Koordinaten über Deutschland generiert. Als Zoomstufe wurde 17 gewählt, der selbe Wert, wie bei den Positiv-Beispielen. Die Größe beträgt 640x600, der größtmögliche Wert in einer Abfrage. Als Datentyp wurde das jpg-Format gewählt. Dieses Format komprimiert die Bilddaten. Das hat den Vorteil eines geringeren Speicherbedarfs und bietet damit eine einfachere und schnellere Bearbeitung und Verwaltung der Bilder. Der Nachteil, der damit verbundenen Informationsverlust, wurde dafür in Kauf genommen. 1000 Bilder in einem unkomprimierten Format, wie z.B. *png*, belegen ca. 400 Mb und als



*jpg* nur ca. 70 Mb.

Alle zufällig generierten Negativbeispiele wurden einzeln von Hand auf mögliche Flugzeuge überprüft, um keine positiven Beispiele mit negativen zu mischen. Dabei wurde kein Flugzeug gefunden. Allerdings wurde beim Durchschauen bemerkt, dass für bestimmte Koordinaten kein Bild von Google-Maps generiert werden konnte, da Karteninformationen für diesen Bereich fehlen. Diese Bilder wurden alle aus der Negativ-Liste entfernt. Es ergab sich dadurch eine Gesamtzahl von 894 Negativbeispielen.

#### 4.1.3. Positive

Die Positiv-Beispiele wurden von Hand gesucht. Dabei lag der Fokus darauf, möglichst viele fliegende Flugzeuge zu finden. Dies gestaltete sich relativ schwierig. Die meisten Objekte konnten in den An- und Abflugschneisen großer Flughäfen entdeckt werden. Allerdings konnten so nur insgesamt 14 Flugzeuge im Flug gefunden werden. Um weitere Positivbeispiele zu erhalten wurden Bilder von parkenden oder fahrenden Flugzeugen gewählt. Diese sind leicht zu finden, indem man verkehrsreiche Flugplätze absucht. Als Zoomstufe wurde der Wert 17 gewählt. Auf Zoomstufe 18 ist zwar das Flugzeug deutlich größer und detailreicher, allerdings ist der Bildausschnitt auch vier mal so klein. Im Hinblick auf eine großflächige Suche, ist es wünschenswert, möglichst große Bildausschnitte bearbeiten zu können. Bei Zoomstufe 16 ist das Objekt dann wiederum zu klein. Ein kleines Flugzeug misst auf dieser Stufe ca. 20x20 Pixel. Es wurde deshalb hauptsächlich mit Beispielen der Zoomstufe 17 gearbeitet. In Kapitel 7 wurde versuchsweise die Zoomstufe 16 getestet.



Abbildung 4.1.: Die drei unterschiedlichen Zoomstufen 16, 17, und 18 von links nach rechts.

Es stellte sich die Frage nach der Anzahl und dem Verhältnis zwischen positiven und negativen Beispielen. Beides hängt stark davon ab in welchem Szenario man sich befindet. Versucht man ein Objekt auf einem immer gleich-bleibenden Hintergrund zu erkennen, so braucht man wenig Negativ-Beispiele und mehr Positiv-Beispiele.



Versucht man jedoch, ein immer identisch aussehendes Objekt auf einem stark variierenden Hintergrund zu finden, dreht sich das Verhältnis um. Im Fall der Flugzeugerkennung in Satellitenbildern hat man es mit stark variierenden Hintergründen zu tun. Das Objekt Flugzeug variiert ebenfalls, je nach Model, Flughöhe, Wetter und Schattenwurf können deutliche Unterschiede zwischen zwei Objekten entstehen. Es wurde daher ein Verhältnis von 1:2 angepeilt, mit 500 Positiv-Beispielen und 894-Negativ-Beispielen. Da keine 500 Positiv-Beispiele von, sich im Flug befindenden, Flugzeugen gefunden werden konnten, mussten diese künstlich erzeugt werden. Dazu diente die in OpenCv zur Verfügung stehende Methode *opencv\_createsamples*.

#### 4.1.4. Die Methode *opencv\_createsamples*

Die Methode *opencv\_traincascade* zum Trainieren eines Klassifizierers benötigt die Positiv-Beispiele in einer sogenannten vec-Datei. Diese Datei wird mit der Methode *opencv\_createsamples* erstellt. Diese Methode hat unterschiedliche Operations-Modi, die sich je nach Art der Parameterübergabe unterscheiden. Sie wird klassischer Weise in der Kommandozeile der Eingabeaufforderung aufgerufen. Um jpg- oder png-Bilder in vec-Dateien umzuwandeln kann folgende Form genutzt werden:

```
opencv_createsamples -info info.txt -vec output.vec -w 40 -h 40 -num 10
```

*-info* gibt an wo die Info-Datei zu finden ist. In dieser Textdatei muss der Pfad des jpg-Bildes hinterlegt werden. Dazu muss die Anzahl der im Bild befindlichen Treffer angegeben werden und die Position und Abmessungen ihrer Hitboxen.

*-vec* benennt vec-Datei die erzeugt wird.

*-w* und *-h* definieren Breite und Höhe der zu erzeugenden Samples.

*-num* gibt die Anzahl der zu erzeugenden Samples an.

In diesem Beispiel werden also 10 Samples mit den Abmessungen 40x40 in der Datei output.vec gespeichert. Um sich den Inhalt einer vec-Datei in Bildern anzeigen zu lassen, wird folgender Befehl verwendet:

```
opencv_createsamples -vec vec-file.vec -w 40 -h 40
```

Dabei müssen Breite und Höhe den Abmessungen der Samples in der vec-Datei entsprechen.

Die Methode *opencv\_createsamples* kann aber noch mehr. So ist es möglich, mit ihr weitere künstliche Samples aus Positiv-Beispielen zu generieren. Dazu muss ein Positiv-Beispiel ausgeschnitten und auf schwarzem Hintergrund platziert werden. Das Objekt wird dann auf ein zufälliges Bild aus einem Pool von Beispiel-Hintergründen gelegt. Außerdem kann es zufällig in mehrere Richtungen rotiert



werden und in der Helligkeit verändert werden. Dies kann beliebig oft wiederholt werden, um eine Sammlung künstlich erzeugter Positiv-Beispiele zu generieren. Der dafür zuständige Befehl lautet:

```
opencv_createsamples -img example.png -bg bg.txt -num 10 -vec output.vec -w 40 -h 43  
-maxxangle 0 -maxyangle 0 -maxzangle 0.2 -maxidev 0
```

*-img* ist der Dateiname des Objekts auf schwarzem Hintergrund.

*-bg* ist eine Textdatei, in der die Dateinamen der Hintergrundsbilder vermerkt sind.

*-num* ist die Anzahl der zu erzeugenden Samples.

*-vec* ist der Name der ausgegebenen vec-Datei.

*-w* und *-h* geben die Maße der zu erzeugenden Samples an.

*-maxxangle*, *-maxyangle* und *-maxzangle* beschreiben eine Rotation um die x-, y- oder z-Achse. Diese Rotation ist zufällig und wird begrenzt durch den angegebenen Wert. Dieser wird in Radian angegeben.

*-maxidev* beschreibt eine zufällige Variation der Helligkeit, wobei der angegebene Wert die maximale Abweichung angibt.

So werden also in diesem Beispiel aus einem Positiv-Beispiel zehn künstliche Samples auf zufälligen Hintergründen generiert. Diese sind um die z-Achse um maximal 0.2 Radian (ca. 11°) zufällig gedreht.

#### 4.1.5. Anwendung der *opencv\_createsamples*-Methode

Zur Vorbereitung wurden mit GIMP (Version 2.8.14) von Hand 50 Flugzeuge ausgeschnitten und mit der Nase nach oben auf einem schwarzen Hintergrund platziert. Dabei wurde darauf geachtet, ein möglichst breites Spektrum an Flugzeuggrößen und Flugzeugtypen abzudecken. Die Flugzeuge wurden alle so platziert, dass sie das Bild möglichst komplett ausfüllen. Das heißt, das Flugzeug stößt oben, unten, links und rechts an den Rand des Bildes. Die so entstandenen Samples haben daher alle unterschiedlichen Größen und Proportionen.



Abbildung 4.2.: Beispiele für positiv Samples unterschiedlicher Modelle und Größen.

Diese 50 Samples wurden dann an die *opencv\_createsamples*-Methode übergeben, um daraus 500 Positiv-Beispiele zu generieren. Dazu diente folgender Befehl:



```
opencv_createsamples -img example.png -bg bg.txt -num 10 -vec output.vec -w 40 -h 43  
-maxxangle 0 -maxyangle 0 -maxzangle 0.2 -maxidev 0
```

Dieser wurde 50-mal in eine Batch-Datei kopiert und umgeschrieben, um für jedes Positiv-Beispiel 10 künstliche Samples zu generieren. Die dabei verwendeten Hintergrundbilder, auf die die Objekte zufällig platziert werden, sind die 894 Negativ-Beispiele. Dadurch entstehen 500 Samples mit den Maßen 40x43. Diese Proportion wurde deshalb gewählt, da die Flugzeuge im Schnitt etwas höher als breit sind. Es wurde damit versucht, ein Flugzeug möglichst allgemein zu beschreiben. Ein Flugzeug, dass diesen Proportionen überhaupt nicht entspricht, kann vom Klassifizierer später nur sehr schwer erkannt werden. In Kapitel 7 wird noch eine weitere Variation der Sample-Erstellung diskutiert, bei der die Proportion des Original-Objekts erhalten bleibt.

Die Fläche von 40x43 Pixeln ist ein Kompromiss zwischen dem, durch die Skalierung entstehenden, Informationsverlust und der Zeit, die der Klassifizierer nachher für das Training benötigt. In der Regel werden, z.B. für die Gesichtserkennung, mit Abmessungen um die 20x20 Pixel trainiert. Je größer die Trainingssamples, desto länger die Trainingsdauer und der benötigte Arbeitsspeicher während des Trainings. Samples größer als 40x43 erwiesen sich als nicht praktikabel.

Das größte Positiv-Beispiel misst 195x113 Pixel. Es muss daher um das ca. 5-fache herunter-skaliert werden. Das kleinste Positiv-Beispiel misst 30x32 Pixel. Insgesamt 5 kleine Flugzeuge mussten auf die 40x43 hoch-skaliert werden, der Rest wurde herunter-skaliert.

Der Parameter zur Einstellung der Helligkeitsabweichung wurde auf 0 gesetzt. Dieses Projekt beschränkt sich darauf helle Flugzeuge auf dunklerem Grund zu erkennen. Wird die Helligkeit der Flugzeuge zufällig variiert, so steigt die Zahl der Positiv-Beispiele, bei denen das Flugzeug dunkler ist als der Hintergrund. Durch die Platzierung auf einem zufälligen Hintergrund ist dies schon ohne Helligkeits-Variation möglich und ein eher unerwünschter Nebeneffekt der künstlichen Generierung.

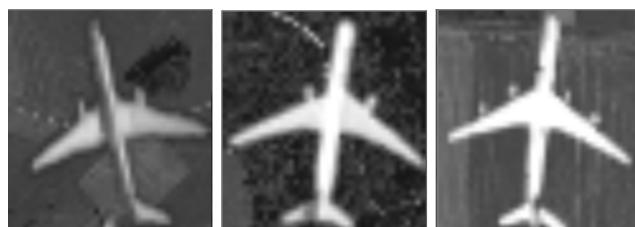


Abbildung 4.3.: Beispiele für die von *opencv\_createsamples* erzeugten vec-Samples.  
Das Objekt wird jeweils leicht gedreht und auf einem zufälligen Hintergrund platziert.



## 4.2. Training des Klassifizierers

Zur Erstellung eines Kaskaden-Klassifizierers nach Viola und Jones, steht in OpenCv die Methode `opencv_traincascade` zur Verfügung. Sie wird klassischerweise in der Eingabeaufforderung aufgerufen. Dabei können benutzerdefinierte Einstellung durch Parameterübergabe vorgenommen werden. Ein Beispiel:

```
opencv_traincascade -data data -vec vec/pos-all.vec -bg bk.txt -numPos 450 -numNeg 894 -  
numStages 12 -featureType LBP -w 40 -h 43 -bt GAB -minHitRate 0.995 -maxFalseAlarmRate  
0.3
```

Die verwendeten Parameter haben folgende Bedeutung:

- **-data**

Der Ordner in dem die erzeugten Klassifizierer-Dateien abgelegt werden.

- **-vec**

Die vec-Datei, die die positiven Samples enthält.

- **-bg**

Eine txt-Datei in der die Dateinamen der Negativ-Beispiele aufgeführt sind.

- **-numPos**

Die Anzahl der positiven Samples, die zum Training einer Stufe genutzt werden sollen. Hier werden nicht die kompletten 500 positiven Samples angegeben, da sonst der Trainingsvorgang scheitern kann. Für jede Stufe wird die spezifizierte Anzahl an Samples verwendet. Wird ein Sample in einer Stufe nicht als positiv erkannt, fällt es weg und wird für das Training der nächsten Stufen nicht mehr genutzt. Es wird stattdessen ein neues Sample aus der vec-Datei hinzugefügt. Kann kein neues Sample gefunden werden, da alle Samples schon benutzt wurden, so endet das Training in einem Fehler. Deshalb wurden hier 450 statt 500 Samples angegeben. Das bedeutet folglich auch, dass meistens nicht alle 500 Samples zum Training genutzt wurden.

- **-numNeg**

Die Anzahl der negativen Samples, die zum Training einer Stufe genutzt werden sollen. Hier kommen die 894 Negativ-Beispiele zum Einsatz.

- **-numStages**

Die Anzahl an Stufen, aus der der Klassifizierer bestehen soll. Jede Stufe entspricht einem *strong-classifier*. Hier wurde mit einer Stufenanzahl zwischen 6 und 13 experimentiert. Je kleiner die Anzahl, desto mehr Falsch-Positive liefert



der Klassifizierer. Je Größer die Anzahl, desto schärfer wird der Klassifizierer und die Anzahl der Falsch-Positive sinkt. Dabei steigt aber die Anzahl der *Misses* (verpasste Treffer).

Ebenso gilt, je mehr Stufen, desto höher die Trainingsdauer.

- **-featureType**

Hier kann die Art der Merkmale gewählt werden. Zur Verfügung stehen *Local-Binary-Pattern* und Haar-ähnliche Merkmale. Beide Typen wurden in dieser Arbeit getestet.

- **-w** und **-h**

Die Größe, der in der vec-Datei spezifizierten Samples. Hier wurde hauptsächlich mit einer Abmessung von 40x43 Pixeln gearbeitet.

- **-bt**

Die Art des Trainings-Algorithmus. Hier lässt sich wählen zwischen Discrete AdaBoost, Real AdaBoost, LogitBoost und Gentle AdaBoost. In dieser Arbeit wurde ausschließlich der Typ Gentle AdaBoost verwendet.

- **-minHitRate**

Hier kann die gewünschte minimale Trefferquote pro Stufe angegeben werden. Es wurden Werte zwischen 0.95 und 0.999 ausprobiert. Die besten Ergebnisse lieferte der Wert 0.999.

- **-maxFalseAlarmRate**

Hier kann die gewünschte maximale Falsch-Alarm-Rate pro Stufe angegeben werden. Um dem Prinzip des Kaskaden-Klassifizierers zu folgen, sollte der Wert hierfür nicht zu niedrig gehalten werden. Je kleiner der Wert, desto höher die Anzahl an *weak-classifiern* pro Stufe. Es wurden Werte zwischen 0.1 und 0.5 getestet. Die meisten hier vorgestellten Klassifizierer wurden mit 0.3 trainiert.



## 5. Anwendung

### 5.1. Verwendete Software, Programmiersprachen und Computerspezifikation

Das Projekt ist in Java implementiert. Als Entwicklungsumgebung diente das Eclipse-SDK (Version 3.5 - JavaSE 1.6). Die Algorithmen zur Objekterkennung und Bildbearbeitung stammen aus der OpenCv-Bibliothek (Version 2.4.9).

Das Projekt wurde als Test zur Laufzeit-Optimierung ebenfalls in C++ unter Microsoft-Visual-Studio-Community-2015 implementiert.

Des weiteren wurde Python (Version 2.7.9) genutzt, um das Skript *mergevec* zur Vereinigung mehrerer vec-Dateien auszuführen.

Die Satellitenbilder wurden mithilfe der Google-Static-Maps-API erstellt.

Zur Bildbearbeitung wurden drei verschiedene Programme genutzt. GIMP (Version 2.8.14) wurde verwendet, um die Flugzeuge von Hand auszuschneiden, senkrecht zu drehen und auf schwarzem Hintergrund zu platzieren.

Für eine Stapelverarbeitung mehrerer Bilder, zur Konversion des Datentyps oder zur Größenänderung, wurde XnView (Version 1.97.8) verwendet.

IrfanView (Version 4.38) wurde verwendet um mehrere Dateien nach einem bestimmten Schema umzubenennen.

Für die Statistik und die Erstellung der Diagramme diente Microsoft Excel 2003.

Die schriftliche Ausarbeitung erfolgte in Texmaker (3.2.2).

Computerspezifikation: Intel Core i5 CPU M540 2.53 GHZ, 4GB RAM, Windows 7 Ultimate, Service Pack 1, 64-Bit, NVIDIA GeForce GTX 280M.

### 5.2. Implementation - Ablauf der Objekterkennung

Die Implementierung erfolgte in Java. Dazu wurde die aktuelle Version OpenCv-2.4.9 eingebunden. Die Methode *opencv-traincascade* erzeugt eine Datei *cascade.xml* in der alle gelernten Daten gespeichert sind. Sie enthält alle Informationen, die zur Objekterkennung nötig sind. Um einen Klassifizierer zu benutzen, wird also zuerst die dementsprechende XML-Datei geladen.



```
CascadeClassifier detector = new CascadeClassifier("classifier/cascade.xml");
```

In diesem Projekt wurden 25 Bilder als Test-Datensatz verwendet. Für jedes Bild wird die Methode *rotatingDetection* aufgerufen.

```
for(int i = 0; i<25; i++){
    String fname = i +".png";
    rotatingDetection(folder+fname, outFolder+fname, detector, 1);
}
```

Diese ist dafür zuständig das Bild in gewünschten Schritten um 360° Grad zu drehen und dabei in jedem Schritt das gedrehte Bild nach Objekten zu durchsuchen. Dies ist nötig, da sowohl die Haar-ähnlichen Merkmale, als auch die *Local-Binary-Pattern* nicht rotations-invariant sind und der Klassifizierer mit nach oben zeigenden Objekten trainiert wurde.

Zuerst wird das gewünschte Bild von der Festplatte in den Speicher geladen. In OpenCv werden alle Bilder als Matrix behandelt und daher im Datentyp *Mat* gehalten.

```
Mat img = Highgui.imread(filename);
```

Danach wird ein leeres, schwarzes 900x900-Pixel großes Bild erstellt, in welchem das geladene Bild mittig platziert wird. Würde das Bild in seiner Originalgröße gedreht werden, so würde Information am Rand abgeschnitten werden, da die Bild-Diagonale immer länger ist als die Breite und die Höhe des Bildes. Durch die 900x900-Matrix kann das Bild in der Mitte verlustfrei rotiert werden.

```
Mat big_img = new Mat(new Size(900,900),CvType.CV_8UC3);
img.copyTo(big_img.submat(new Rect(129, 149, img.cols(), img.rows())));
```

Bevor die Rotationen durchgeführt werden, wird eine Liste für die entstehenden Treffer-Kreise erstellt. Die Ausgabe der Detektionsfunktion ist eine Liste von Rechtecken. Da aber das Bild oft gedreht wird und die resultierenden Rechtecke im Originalbild dann ebenfalls rotiert gezeichnet werden müssten, wurde hier mit Kreisen gearbeitet. Diese ergeben ein deutlich übersichtlicheres Bild der Treffer und deren Ausrichtung.

Des Weiteren wird ein Container für das rotierte Bild initialisiert und die Variable berechnet, die die maximale Anzahl an Rotationen angibt und als Abbruchbedingung für die Rotationsschleife dient. Möchte man z.B in 2° Grad Schritten drehen, so muss man  $360/2 = 180$  mal rotieren.



```
ArrayList<int[]> circles = new ArrayList<int[]>();
Mat big_img_rot = new Mat();
int rotations = (int) 360/angle;
```

Nun beginnt die Rotationsschleife in der das Bild schrittweise gedreht und dann detektiert wird. Anzumerken ist hierbei, dass das Originalbild immer wieder, um iterativ größer werdende Schritte, gedreht wird. Es wäre auch möglich, das Bild in jedem Schritt um eine feste Gradanzahl zu drehen. Dabei würde immer wieder ein bereits gedrehtes Bild weiter gedreht. Eine Drehung bedeutet aber immer einen gewissen Teil an Informationsverlust durch Rundungen von dabei entstehenden Gleitkommazahlen. Würde man das gedrehte Bild wieder weiter drehen, würde sich der Informationsverlust potenzieren. Durch Drehung des Originalbildes hingegen, kann die Verfälschung durch Rundungen minimal gehalten werden.

Die Matrix *rot* liefert die gewünschte Rotationsmatrix, die dann mit dem Befehl *warpAffine* auf das Originalbild angewendet wird. Der Punkt (449, 449) beschreibt dabei die Bildmitte um die gedreht wird.

```
for(int i =0; i<rotations;i++){
    Mat rot = Imgproc.getRotationMatrix2D(new Point(449, 449), i*angle, 1);
    Imgproc.warpAffine(big_img, big_img_rot, rot, new Size(900,900));
```

Der Klassifizierer liefert die Methode *detectMultiScale* zur Objekterkennung. Siehe: [Die Methode *detectMultiScale*, Seite 24] für eine Erklärung der einzelnen Parameter. Diese wird nun auf das rotierte Bild angewendet. Ausgegeben wird eine Liste von Rechtecken, die die gefundenen Treffer beschreibt.

```
MatOfRect detections = new MatOfRect();
detector.detectMultiScale(big_img_rot, detections, 1.1, 3, 0, new Size(40,43),new Size(125,125));
```

Diese Liste wird in einer for-Schleife ausgelesen, um die Rechtecke in Kreise umzurechnen und der Kreis-Liste hinzuzufügen.

```
for (Rect rect : detections.toArray()) {
    int cx = (int) rect.x + (int) rect.width/2;
    int cy = (int) rect.y + (int) rect.height/2;
    int cr = (int) rect.height/2;
    int ca = i*angle;
    int[] c = new int[]{cx,cy,cr,ca};
    circles.add(c);
}
```



## 5. Anwendung

Ein Kreis wird beschrieben durch ein Array (x-Position, y-Position, Radius, Rotationswinkel). Der Mittelpunkt des Kreises entspricht dem des gefundenen Rechtecks. Der Umfang entspricht der Höhe des Rechtecks. Die Ausrichtung wird durch eine Linie deutlich gemacht. Diese Linie zeigt nach oben, in Richtung der Flugzeugnase, da in den Trainings-Samples das Flugzeug immer mit der Nase nach oben ausgerichtet wurde.

Nachdem alle Rotationen ausgeführt wurden und das Bild somit einmal um 360° gedreht wurde, ist die Liste der gefundenen Treffer komplett. Da sich die Koordinaten der einzelnen Treffer allerdings noch auf das entsprechend gedrehte Bild beziehen, müssen diese jetzt noch umgerechnet werden, damit die Treffer-Kreise an die richtige Position im Originalbild gezeichnet werden können.

```
for(int j = 0; j < circles.size(); j++){
    int[] crcl = circles.get(j);
    Point p1 = getRotatedPoint(crcl[0],crcl[1],crcl[3]);
    Point p2 = getRotatedPoint(crcl[0],crcl[1]-crcl[2],crcl[3]);
    Core.circle(big_img, p1, crcl[2], new Scalar(0,255,0));
    Core.line(big_img, p1, p2, new Scalar(0,255,0));
}

Highgui.imwrite(output, big_img);
```

Die Methode *getRotatedPoint* rechnet die gedrehten Punktkoordinaten in die entsprechenden Koordinaten im Originalbild um. Dabei ist der Winkel in Radian angegeben. Der Punkt (449, 449) beschreibt wieder den Bildmittelpunkt, um den gedreht wurde.

```
double rx = Math.cos(angleR) * (x-449) - Math.sin(angleR) * (y-449) + 449;
double ry = Math.sin(angleR) * (x-449) + Math.cos(angleR) * (y-449) + 449;
```

Optional können die Treffer vor dem Zeichnen noch gruppiert werden. Das Ergebnis ist, dass pro gefundenem Objekt nur ein einzelner Kreis gezeichnet wird. Da aber mehrere Kreise pro Objekt einen besseren Eindruck von dem vermitteln, was während der Objekterkennung abgelaufen ist, wurde auf eine Gruppierung der Treffer meistens verzichtet.

Abschließend wird das entstandene Bild auf die Festplatte geschrieben.

```
Highgui.imwrite(output, big_img);
```

### 5.3. Die Methode *detectMultiScale*

Die Methode *detectMultiScale* wird von einem Klassifizierer (Typ *CascadeClassifier*) aufgerufen. Sie dient dazu, die vom Klassifizierer gestellten Merkmale und Grenz-



## 5. Anwendung

werte auf ein Bild anzuwenden, um darin ein oder mehrere gesuchte Objekte zu finden. Sie kann mit unterschiedlicher Parameterzahl aufgerufen werden. die einfachste Form ist:

```
detectMultiScale(Mat Image, MatofRect Hits)
```

*Image* spezifiziert das Bild, in dem gesucht wird und *Hits* gibt die Liste an, in der die gefundenen Rechtecke gespeichert werden sollen.

Um jedoch eine benutzerdefiniertere Suche ausführen zu können, wird folgende Form der Parameterübergabe angeboten:

```
detectMultiScale(Mat Image, MatofRect Hits, double scaleFactor, int minNeighbors, int flags,  
Size minSize, Size maxSize)
```

### • scaleFactor

Der Algorithmus schiebt ein Fenster iterativ, pixel-weise über das gesamte Bild. Bei jedem Schritt werden die Merkmale innerhalb dieses Fenster ausgewertet. Ein Durchlauf ist beendet, wenn das Fenster einmal über das gesamte Bild geschoben wurde. Der Skalierungsfaktor *scaleFactor* gibt an, um welchen Wert die Fenstergröße in jedem Durchlauf hoch-skaliert werden soll. Per default ist der Wert auf 1.1 eingestellt. Das heißt das Fenster wird nach jedem Durchlauf um 10% größer skaliert. Das Fenster kann nicht kleiner-skaliert werden, da der Wert für den *scaleFactor* immer größer als Eins sein muss.

Wichtig ist hier, dass in Wirklichkeit nicht das Fenster vergrößert wird, sondern das Bild verkleinert wird. Die neue Bildgröße errechnet sich also durch Bildgröße/Skalierungsfaktor. Hier unterscheidet sich der OpenCv-Algorithmus von der, von Viola und Jones vorgestellten Methode. Diese sieht vor die Fenstergröße und damit auch die Merkmale zu skalieren.

### • minNeighbors

Nachdem alle Fensterskalierungen durchlaufen wurden, werden die gesammelten Treffer-Rechtecke mit der Methode *groupRectangles* gruppiert. Dabei werden Rechtecke zusammengefasst, die nahe bei einander liegen, sogenannte Nachbarn. Dies gilt für zwei Rechtecke gleicher oder ähnlicher Größe, deren Position sich nicht oder nur leicht unterscheidet. Somit werden Treffer auf unterschiedlichen Skalierungen und ähnlichen Positionen zusammengefasst. Dies ist sinnvoll, da ein Objekt meist in mehreren Fenstern erkannt wird. Objekte, die wenige bis keine Nachbarn besitzen, werden damit aussortiert. Der Parameter *minNeighbors* gibt an, wie viele Nachbarn ein Objekt mindestens besitzen muss, um als Treffer ausgegeben zu werden. Per default ist dieser Wert auf 3 gesetzt.



- **flags**

Dieser Parameter bezieht sich auf eine veraltete Form eines Klassifizierers, die nicht mehr aktuell ist. In diesem Projekt wurde die neue Version der Kaskaden-Klassifizierer verwendet. Der Parameter hat deshalb keinen Einfluss auf den Algorithmus. Er diente ursprünglich dazu, dem Algorithmus anzuzeigen, ob das Bild oder ob die Fenstergröße skaliert werden soll.

- **minSize**

Hiermit kann die minimale Größe der gesuchten Objekte angegeben werden. Der Algorithmus fängt erst an zu detektieren, wenn die minimale Größe in die aktuelle Fenstergröße passt. Wichtig ist hier, dass das Bild nicht hochskaliert werden kann. Die Fenstergröße kann also nicht verkleinert, sondern nur vergrößert werden. Die minimale Fenstergröße entspricht der Größe der im Training gelernten Samples.

- **maxSize**

Der Parameter *maxSize* gibt die maximale Größe an, die ein Objekt haben darf. Fenstergrößen größer als *maxSize* werden nicht mehr berechnet.



*5. Anwendung*

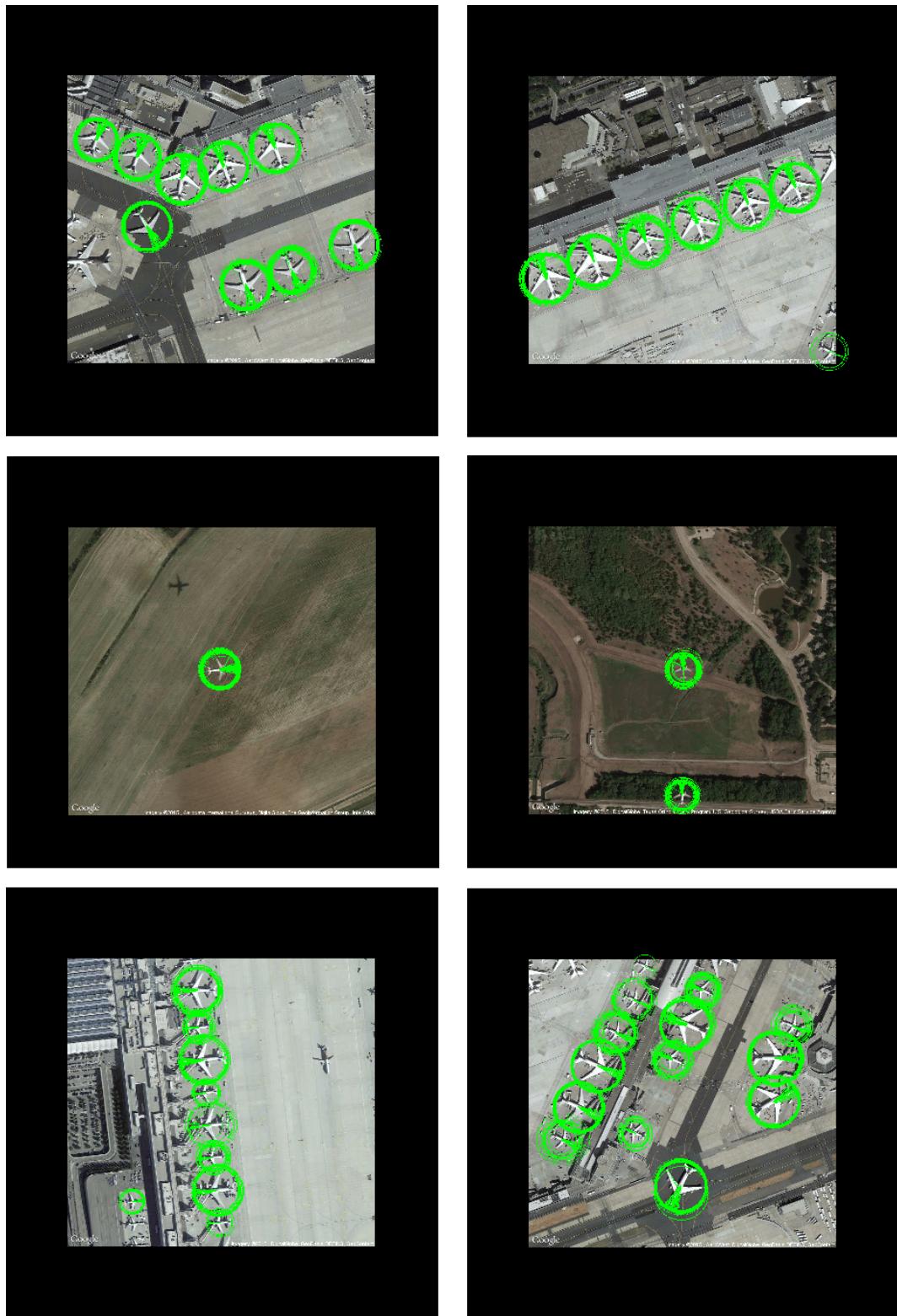


Abbildung 5.1.: Sechs Beispiele für das Ergebnis der Objekterkennung.



## 6. Auswertung

Es wurden mehrere Klassifizierer trainiert, um den Einfluss aller definierbaren Parameter zu testen. Anfangs lag der Fokus darauf, eine möglichst hohe Trefferquote bei einer möglichst niedrigen Falsch-Positiv-Rate zu erzielen. Der Faktor Laufzeit spielte dabei keine Rolle.

Die variablen Parameter lassen sich in drei Gruppen teilen. Die Einstellungen bei der Erzeugung der positiven Samples, die Parameter die der Methode *opencv-traincascade* beim Training übergeben werden und die Schrittgröße, um die bei der Detektions-Anwendung gedreht wird.

### 6.1. Das Testset

Um die Leistung der Klassifizierer bewerten zu können und untereinander vergleichen zu können, wurde ein festes Set von Beispiel-Bildern gewählt. Dieses besteht aus 25 Bildern und enthält insgesamt 94 zu erkennende Flugzeuge. Dabei sind sowohl fliegende Flugzeuge, als auch stehende zu finden. Die Stehenden befinden sich immer auf einem Flugplatz. Dabei sind Flugzeuge unterschiedlicher Modelle und Größe zu finden. Alle Klassifizierer wurden mit diesen Bildern getestet. Die Bilder werden im Folgenden auch als Testset bezeichnet.



Abbildung 6.1.: Drei Beispiel-Bilder aus dem Testset.



## 6.2. Einfluss der Rotation der positiven Samples

Um den Einfluss der zufälligen Rotation um die z-Achse bei der Erstellung der positiven Samples zu testen, wurden mehrere Klassifizierer mit unterschiedlichen Werten trainiert. Dabei wurden die Werte 0, 0.1, 0.2, 0.3 und 0.4 verwendet. Ein Wert von 0.2 bedeutet, das Flugzeug wird entweder nach rechts oder nach links um maximal 0.2 Radian (ca. 11 Grad) gedreht. Es entsteht so also eine Schwankung von maximal ca. 22 Grad. Da während der Detektion das Bild gedreht werden muss, ist es hier interessant herauszufinden, wie groß die Schritte der Drehung sein dürfen ohne das Genauigkeit verloren geht.

Um diesen Einfluss zu testen, wurde nach einer Möglichkeit gesucht, die Spannweite eines Klassifizierers zu ermitteln. Spannweite bedeutet hier, wie sensitiv der Klassifizierer auf gedrehte Objekte reagiert und ab welchem Winkel ein Objekt nicht mehr erkannt wird. Im ersten Versuch wurde dafür eine Rotationstabelle erstellt, ein Bild, in dem ein Flugzeug auf schwarzem Hintergrund in verschiedenen Winkeln abgebildet ist. So sollte man ablesen können, ab welchem Winkel das Objekt nicht mehr erkannt wird.

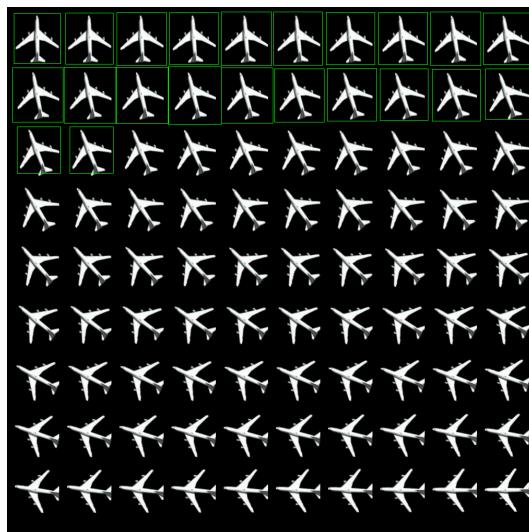


Abbildung 6.2.: Die Rotationstabelle zeigt an, um welchen Winkel das Objekt gedreht werden kann, bis es nicht mehr erkannt wird. Das Objekt wird hier in 1-Grad-Schritten gedreht.

Leider erwies sich diese Rotationstabelle als schlechter Indikator für die Spannweite eines Klassifizierers. Wendet man die erhaltene Spannweite auf das Testset an, so schneidet der Klassifizierer deutlich schlechter ab. Dies liegt wahrscheinlich daran, dass das künstlich erstellte Beispiel auf schwarzem Hintergrund nicht den Objekten im realen Testset entspricht.

Die Spannweite eines Klassifizierers wurde daraufhin immer dadurch geprüft, dass



## 6. Auswertung

die Bilder aus dem Testset um verschiedene Schrittgrößen gedreht wurden. Anschließend wurde ausgezählt, wieviele Objekte entdeckt werden konnten.

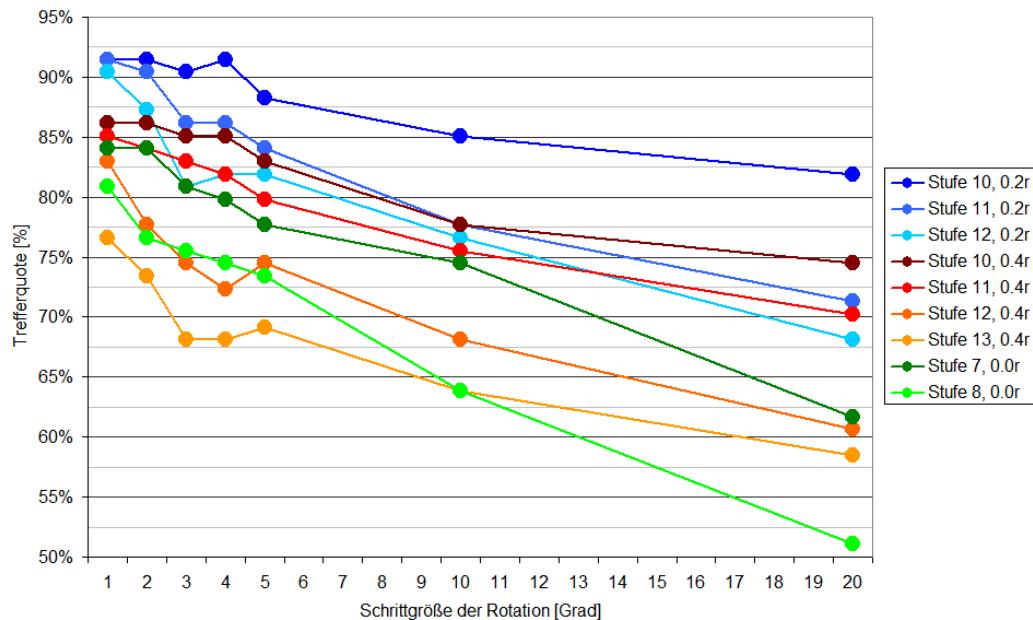


Abbildung 6.3.: Einfluss des Trainingswinkels und der Stufenanzahl auf die Trefferquote.

Das Ergebnis ist, dass es praktisch nicht möglich ist, die Schrittgröße zu erhöhen ohne dabei an Genauigkeit zu verlieren. Dennoch gibt es Klassifizierer, die besser als andere auf große Schritte reagieren.

Zur einfacheren Beschreibung wird hier eine Abkürzung definiert:

Ein R02-Klassifizierer ist ein Klassifizierer, dessen positive Samples bei der Erzeugung um maximal 0.2 Radian gedreht wurden.

Die Annahme war: Je größer der zufällige Winkel bei der Erstellung, desto größer die Spannweite des Klassifizierers. Dies konnte jedoch nur teilweise bestätigt werden. Der R02-Klassifizierer schnitt deutlich besser ab als der R0 und der R01. Allerdings nahm bei einem höheren Erstellungswinkel, die Genauigkeit wieder ab. So brachten R03 und R04 ein schlechteres Ergebnis.

Ein Grund dafür könnte sein, dass die Menge der Trainingssamples für die starke Variation zu gering ist, so dass nicht genug Samples pro Winkel entstehen. Mit einer größeren Menge an Sampels pro Winkel sollte sich auch ein besserer R03- oder R04-Klassifizierer trainieren lassen.

Betrachtet man nicht die Spannweite, sondern lediglich die Trefferquote bei der Drehung in 1-Grad-Schritten, so lässt sich feststellen, dass auch hier der R02 die besten Ergebnisse erzielt. Es wurde zunächst angenommen, dass der R0 die besten Ergeb-



nisse bei 1-Grad-Schritten liefert, da bei diesem alle Samples den selben Winkel( $0^\circ$ ) haben und so die maximale Anzahl an Samples pro Winkel vorliegt. Wie sich aber herausstellte, ist der R0 einer der schlechteren Klassifizierer. Er ließ sich nicht über eine Anzahl von 8 Stufen trainieren. Der Vorgang bricht ab mit der Meldung: "Angegebene maximale Falsch-Positiv-Rate erreicht." Die angegebene Rate lag bei 0.3. Dies zeigt, dass es mit nur wenigen Merkmalen und wenigen Stufen möglich ist, die positiven Samples von den negativen zu trennen. Diese Trennung funktioniert zwar im Trainingsset, in dem alle Objekt exakt gleich ausgerichtet sind, scheitert jedoch im Testset, da hier wahrscheinlich größere Schwankungen auftreten.

Der R02 hingegen ist schon mit größeren Schwankungen trainiert worden und wurde dadurch gezwungen, andere, komplexere Merkmale zur Trennung der Daten zu verwenden. Dies erklärt die bessere Trefferquote.

Abschließend lässt sich sagen, dass ein Winkel von 0.2 Radiant bei der Erstellung der Trainings-Samples die besten Klassifizierer zur Folge hatte.

### 6.3. Parameterwahl im Training

Die Parameter, die maßgeblich das Trainingsresultat beeinflussen, sind: Die Art der Merkmale, die Anzahl der Stufen, die maximale Falsch-Positiv-Rate und die minimale Trefferquote.

Die minimale Trefferquote pro Stufe sollte möglichst niedrig gewählt werden, so dass alle Positiv-Beispiele auch als Treffer erkannt werden. Hier wurde 0.999 als Wert gewählt.

Die maximale Falsch-Positiv-Rate pro Stufe sollte etwas höher gewählt werden. Je kleiner der Wert, desto mehr Merkmale werden pro Stufe verwendet. Da vor allem die ersten Stufen eines Kaskaden-Klassifizierers nur ein grober Filter sind und schnell zu berechnen sein sollten, sollte die Merkmals-Anzahl in den ersten Stufen möglichst niedrig gehalten werden. Hier brachte ein Wert von 0.3 die besten Ergebnisse.

#### 6.3.1. Vergleich von Haar-ähnlichen Merkmalen und Local-Binary-Pattern

Der Vorteil der *Local-Binary-Patterns* ist eindeutig ihre schnelle Trainingsdauer. Einen Klassifizierer auf die selbe Stufenanzahl zu trainieren dauert mit Haar-ähnlichen Merkmalen um ein Vielfaches länger. In Abbildung 6.4 sieht man, wie das Training auf 9 Stufen mit *Local-Binary-Patterns* gerade mal 8 Minuten braucht, wo hingegen die Haar-ähnlichen Merkmale über sechs Stunden Training erfordern. Aus diesem Grund wurden für dieses Projekt hauptsächlich die *Local-Binary-Patterns* genutzt. Insgesamt wurden im Laufe der Arbeit 84 verschiedene Klassifizierer trainiert. Dies



Abbildung 6.4.: Vergleich der Trainingsdauer zweier 9-stufiger Klassifizierer mit Haar-ähnlichen Merkmalen und *Local-Binary-Pattern*.

wäre mit Haar-ähnlichen Merkmalen ein enormer zeitlicher Aufwand gewesen. Der Informationsgehalt eines Haar-ähnlichen Merkmals ist deutlich höher als der eines *Local-Binary-Patterns*. Der Wert eines *Local-Binary-Patterns* gibt nur ein ungefähres Bild der Helligkeitsunterschiede zum Mittelpunkt wieder. Die Haar-ähnlichen Merkmale dagegen können, durch ihre unterschiedlichen Formen, genauere Merkmale, wie z.B. streifen-ähnliche Strukturen o.Ä., beschreiben. Trotzdem zeigt sich, dass die *Local-Binary-Patterns* mit dem richtigen Training eine vergleichbare Trefferquote erzielen können. In Abbildung 6.5 wird ein 11-stufiger-LBP-Klassifizierer mit einem 9-stufigen-Haar-Klassifizierer verglichen. Die Trefferquote ist identisch. Die Falsch-Positiv-Rate bei den Haar-ähnlichen Merkmalen ist noch sehr hoch. Dieser Klassifizierer sollte noch weitere Stufen trainiert werden.

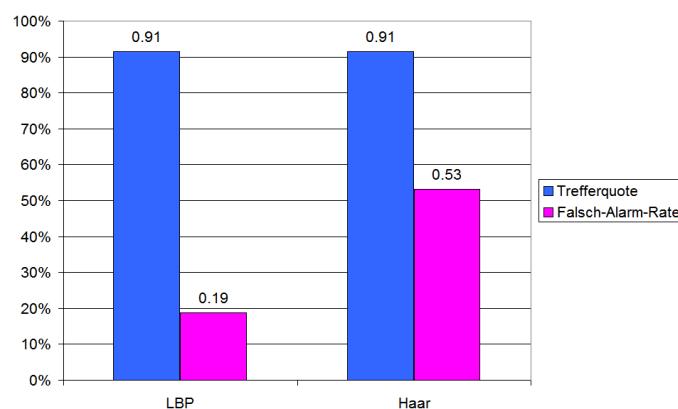


Abbildung 6.5.: Vergleich zwischen einem 11-stufigen-LBP-Klassifizierer und einem 9-stufigen-Haar-Klassifizierer.



### 6.3.1.1. Visualisierung der Merkmale

Die XML-Datei, die einen Klassifizierer beschreibt, konnte ausgelesen werden, um die einzelnen Merkmale jeder Stufe zu visualisieren. Dabei wurde zum einen jede Stufe eines mit *Local-Binary-Pattern* trainierten 11-stufigen-Klassifizierers sichtbar gemacht (Abbildung 6.6). Jedes der farbigen Rechtecke steht für ein Block-*Local-Binary-Pattern*. Es ist in neun gleichgroße Blöcke geteilt, deren Helligkeitsverhältnis zum Mittel-Block den Wert des Merkmals ergibt. Auf jeder Stufe werden je drei bis 5 Merkmale geprüft. Man sieht, wie in Stufe 1 mit drei charakteristischen Merkmalen ein Flugzeug grob erfasst wird. In jeder weiteren Stufen kommen weitere Merkmale hinzu, um die Beschreibung des Objekts zu präzisieren.

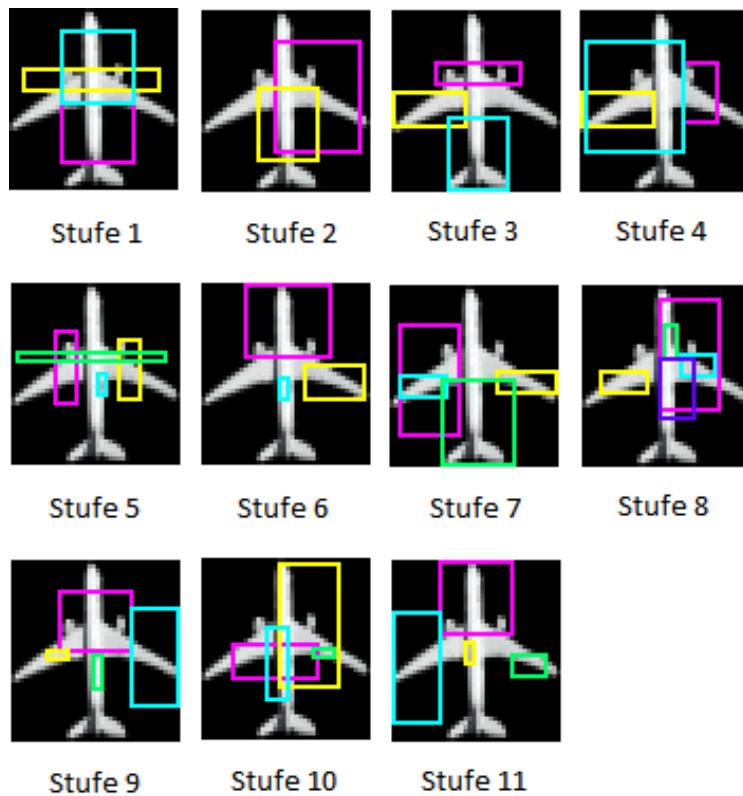


Abbildung 6.6.: Visualisierung der *Local-Binary-Patterns* in jeder Stufe eines 11-stufigen-Klassifizierers.

Die Visualisierung eines 9-stufigen-Klassifizierers mit Haar-ähnlichen Merkmalen ist in Abbildung 6.7 dargestellt. Der Klassifizierer hat bis zu 11 Merkmalen pro Stufe, so dass eine Darstellung wie bei den *Local-Binary-Pattern* zu unübersichtlich ist. Stattdessen wurde nur die letzte Stufe des Klassifizierers visualisiert. Dabei entsprechen die grünen und blauen Rechtecke, den schwarzen und weißen Flächen der Haar-ähnlichen Merkmale. Deren Summe wird voneinander abgezogen und liefert



den Wert des Merkmals. Man sieht, wie alle Formen der Merkmale zum Einsatz kommen. Die Größe variiert ebenfalls stark. So besteht das Merkmal in Bild 11 nur aus drei Pixeln. Das Merkmal in Bild 2 hingegen prüft den linken Flügel in einem großen Block.

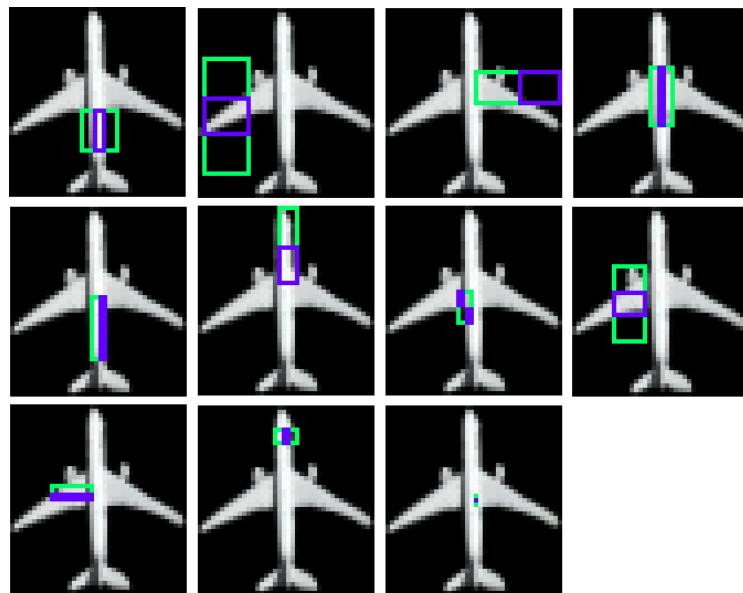


Abbildung 6.7.: Visualisierung der Haar-ähnlichen Merkmale in der letzten Stufe eines 9-stufigen-Klassifizierers.

### 6.3.2. Einfluss der Stufenanzahl

Einer der wichtigsten Parameter für das Training ist die Stufenanzahl. Je weniger Stufen ein Klassifizierer hat, desto allgemeiner ist er und desto höher ist die Falsch-Positiv-Rate. Je mehr Stufen trainiert werden, desto präziser wird der Klassifizierer und die Falsch-Positiv-Rate sinkt. Mit ihr sinkt allerdings auch irgendwann die Trefferquote, da der Klassifizierer dann zu sehr auf die Trainingssamples spezialisiert wird und Objekte außerhalb der Trainingsmenge nicht mehr erkennt. Es ist hier also wichtig, den Punkt zu finden, an dem die Trefferquote anfängt zu sinken und dort eine Stufe zu wählen mit annehmbarer Falsch-Positiv-Rate. Die optimale Stufenanzahl variiert zwischen den unterschiedlichen Klassifizierern. Für die *Local-Binary-Pattern* lag diese meist zwischen 10 und 13 Stufen.



## 6. Auswertung

---

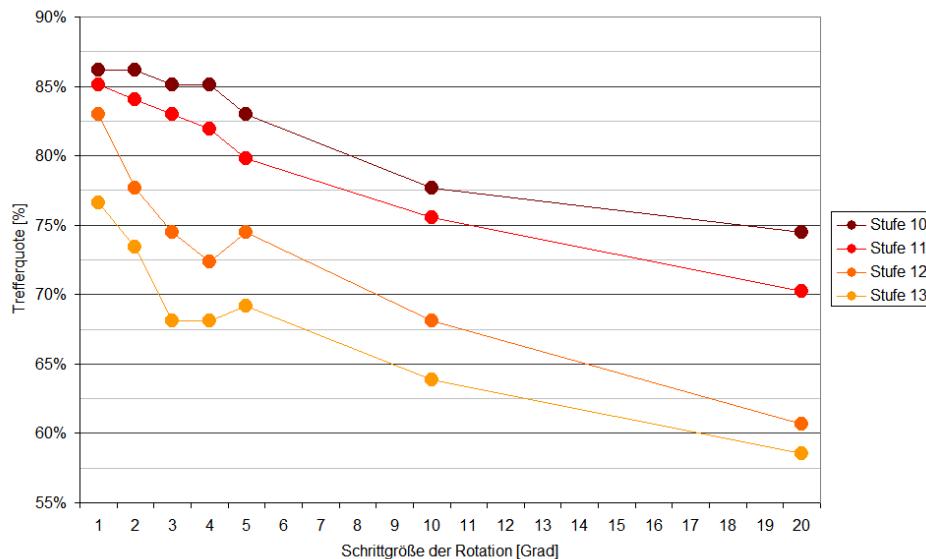


Abbildung 6.8.: Einfluss der Stufenanzahl auf die Trefferquote.

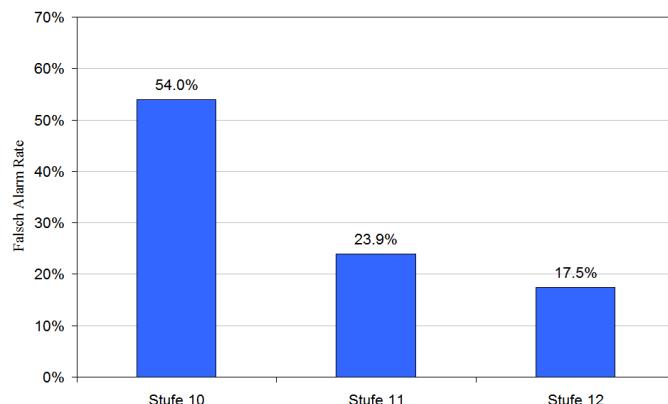


Abbildung 6.9.: Einfluss der Stufenanzahl auf die Falsch-Positiv-Rate.

## 6.4. Aus Fehlern lernen

Um die Trefferquote zu steigern und die Falsch-Positiv-Rate zu senken, kann eine Art Bootstrap-Methode verwendet werden. Dabei wird der Klassifizierer einmal auf das Testset angewendet. Die als positiv bewerteten Bilder werden von Hand in Treffer und Falsch-Positive unterteilt. Die Falsch-Positiven werden dann den Negativ-Beispielen aus dem Trainings-Set hinzugefügt.

Ebenso können nicht erkannte Objekte aus dem Testset ausgeschnitten werden und als Positiv-Beispiele dem Trainings-Set hinzugefügt werden. Mit diesen neu erzeugten Trainingsdaten kann dann ein neuer Klassifizierer gelernt werden. Dieser ist meist deutlich genauer als sein Vorgänger. Dieser Vorgang kann theoretisch solange



wiederholt werden, bis eine gewünschte Genauigkeit erreicht ist.

Der Vorteil dieser Methode ist es, dass der Klassifizierer robuster und präziser wird, hauptsächlich dadurch, dass er ein größeres Trainings-Set bekommt, das speziell darauf ausgelegt ist vergangene Fehler zu korrigieren.

Der Nachteil der Methode ist allerdings, dass der Klassifizierer mit Daten aus dem Testset trainiert wird, was dem eigentlichen Sinn des Testsets widerspricht. Um diese Methode korrekt anzuwenden, muss also jedes mal ein neues, unbenutztes Testset gewählt werden. Dies war bei diesem Projekt nur schwer möglich, hauptsächlich durch den Mangel an guten Beispiel-Bildern von fliegenden Flugzeugen. Die Methode wurde dennoch einmal getestet, indem man einen neuen Klassifizierer aus den Fehlern des Vorgängers lernen ließ. Wie man in Abbildung 6.10 sieht, brachte dies den erwarteten Erfolg. Die Trefferquote stieg deutlich und die Falsch-Positiv-Rate sank.

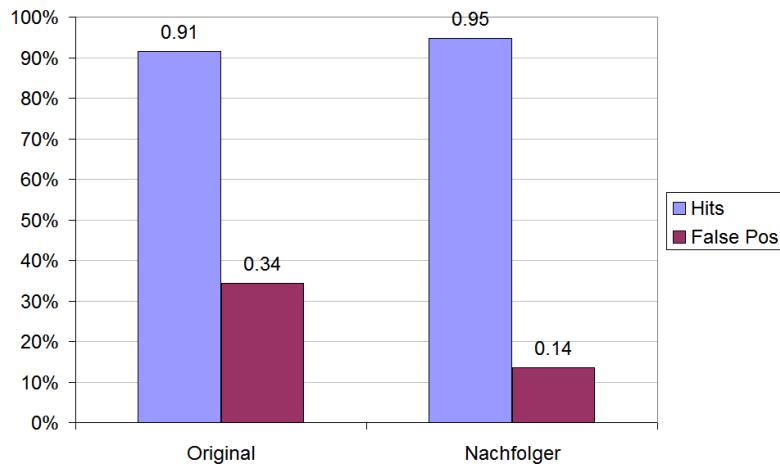


Abbildung 6.10.: Verbesserung der Trefferquote und Falsch-Positiv-Rate durch Bootstrap-Methode.



## 7. Optimierung der Laufzeit

### 7.1. Großflächiges Suchen

Ein Ziel dieser Arbeit war es, mit dem trainierten Klassifizierer, in den Google-Kartendaten großflächig nach Flugzeugen zu suchen und diese dann auf einer Übersichtskarte anzusehen. Dazu wurde ein kleines Fall-Beispiel erstellt. Es wurde eine graphische Benutzeroberfläche programmiert, die es ermöglicht in einem definierten Bereich um den Flughafen Frankfurt am Main nach Flugzeugen zu suchen und deren Positionen zu visualisieren. Der Bereich um den Flugplatz besteht aus 150x150

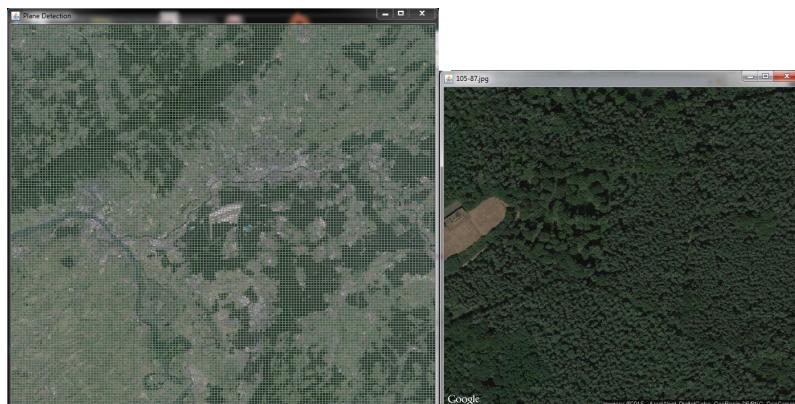


Abbildung 7.1.: Die Übersichtskarte(links) des Gebiets um Frankfurt am Main ist in kleine Quadranten unterteilt. Mit einem Klick in den Bereich wird das entsprechende Fenster(rechts) geladen.

Einzelbildern. Diese 22500 Bilder müssen schrittweise durchsucht werden. Um den Zeitaufwand dieses Vorhabens abzuschätzen wurde eine Laufzeitanalyse vorgenommen.

Dabei wurde zunächst festgestellt welche Operation wie viel Zeit beansprucht. Es stellte sich heraus, dass die meiste Zeit aus Detektion besteht. Ein Detektions-Aufruf kostet ca. 110 ms. Die Drehung des Bildes ist im Vergleich deutlich schneller: ca. 7 ms pro 1-Grad-Drehung. Das Ergebnis ist, dass der Algorithmus ca. 40 Sekunden pro Bild benötigt, wenn das Bild in 1-Grad-Schritten gedreht wird. Das entspricht einer Zeit von ca. 250 Stunden für die großflächige Suche im gesamten Bereich. Das ist deutlich zu hoch um praktikabel zu sein. Motiviert dadurch, wurde nun versucht die Laufzeit des Algorithmus zu verkürzen.



## 7.2. Implementierung in C++

Im Allgemeinen gilt C++ als die bessere Wahl für zeitkritische Anwendungen, da in Java die, dort vorgeschaltete Java-Virtual-Machine, die Programmausführung etwas verlangsamen kann. Testweise wurde der Algorithmus, der ein einzelnes Bild rotiert und durchsucht in C++ implementiert. Das Ergebnis zeigte keinen erkennbaren Unterschied in der Laufzeit. Dies liegt möglicherweise daran, dass die OpenCv-Bibliothek ohnehin schon in C++ geschrieben ist.

## 7.3. Verwendung einer größeren Zoomstufe

Ein Bild aus Zoomstufe 17 passt viermal in ein Bild aus Zoomstufe 16. Mit einem Klassifizierer für diese Zoomstufe könnte die Laufzeit pro Quadratmeter Kartenfläche also deutlich verkürzt werden. Allerdings sind die Objekte auf niedriger Zoomstufe auch entsprechend kleiner. Es wurde versucht, einen Klassifizierer mit Trainingsdaten in Zoomstufe 16 zu trainieren. Die Ergebnisse waren leider ungenügend. Zu viele kleine Flugzeuge konnten nicht erkannt werden.

Eine weitere Idee war es, den Zoomstufe-16-Klassifizierer nur auf eine niedrige Anzahl an Stufen zu trainieren, so dass alle Flugzeuge gefunden werden. Dann sollte das Bild in vier Unterfenster unterteilt werden, entsprechend den vier kleineren Zoomstufe-17-Bildern. Würde im großen Bild ein Treffer gefunden werden, so müsste man näher zoomen und das entsprechende Zoomstufe-17-Bild untersuchen. Würde in einem Unterfenster des großen Bildes kein Treffer gefunden, so könnte man sich das Untersuchen dieses Bildes sparen.

Leider konnte kein Klassifizierer trainiert werden, mit dem man diese Idee umsetzen konnte. Die Falsch-Positiv-Rate war immer zu hoch und führte dazu, dass jedes mal alle Unterfenster untersucht werden mussten.

## 7.4. Anpassung der Rotation

Offensichtlich besteht ein großer Zusammenhang zwischen der Schrittgröße der Rotation und der Laufzeit. Je größer die Schritte, desto weniger Bild-Rotationen müssen berechnet werden und desto weniger Detektions-Aufrufe werden getätigt. Da aber die Trefferquote mit steigender Schrittgröße sinkt, muss hier ein Kompromiss zwischen Laufzeit und Genauigkeit getroffen werden. Siehe dazu Kapitel: Zusammenfassung Seite 43.



Die Berechnung der Bild-Rotation benötigt zwar, im Vergleich zur Detektion, wenig Zeit, kann aber auch optimiert werden. Normalerweise wird die Rotation berechnet, indem die Bild-Matrix mit einer Rotations-Matrix multipliziert wird. Nun kann aber z.B eine Drehung um 180° auch durch eine Spiegelung ausgedrückt werden. Das funktioniert, da die gesuchten Flugzeuge spiegel-symmetrisch sind. Eine Spiegelung ist schneller zu berechnen als eine Rotation, da dabei nur Indizes vertauscht werden und keine Matrix-Multiplikation stattfindet.

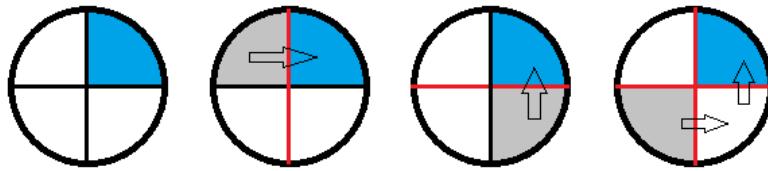


Abbildung 7.2.: Jede Rotation größer als 90° kann durch eine Spiegelung ersetzt werden.

Die Idee ist also, das Bild nur schrittweise um 90 Grad zu drehen und in jedem Schritt drei Spiegelungen durchzuführen (Abb. 7.2). Eine Spiegelung um die Horizontale, eine Spiegelung um die Vertikale und eine Spiegelung um beide Achsen decken den kompletten Bereich ab. Somit werden aus ursprünglich 360 Rotationen dann 90 Rotationen und 3x90 Spiegelungen. Das verringert die Laufzeit um 1.5 Sekunden pro Bild.

## 7.5. Anpassung der maximal Größe

Für die Detektion im Bild ist die Methode *detectMultiScale* zuständig. Dieser kann als Parameter eine maximale Objektgröße übergeben werden. Die zu untersuchenden Fenster, die über das Bild geschoben werden, werden solange hoch-skaliert, bis die Fenstergröße die gewünschte maximale Objektgröße überschreitet. Dann bricht der Algorithmus ab. Durch eine Begrenzung der Fenstergröße nach oben, können also Iterationen mit großen Fenstern eingespart werden und damit verbessert sich die Laufzeit. Als Anhaltspunkt wurde das größte Flugzeug im Testset gewählt. Darauf wurden nochmal 10 Pixel addiert, was eine maximale Objektgröße von 120x125 Pixeln ergab. Dies ergibt eine Zeittersparnis von ca. 24 ms pro Detektion und ca. 6.5 Sekunden pro Bild.



## 7.6. Der Skalierungsfaktor

Ein weiterer Parameter, der der Methode *detectMultiScale* übergeben werden kann, ist der Skalierungsfaktor. Er gibt an, um welchen Wert die Größe des Fensters, das über das Bild geschoben wird, im nächsten Durchlauf skaliert werden soll. Per default liegt der Wert bei 1.1. Das heißt, das Fenster wird nach jedem Durchlauf um 10% vergrößert. Erhöht man diesen Wert, so verringert sich die Anzahl der Durchläufe und damit auch die Laufzeit.

Wird der Wert allerdings zu hoch gewählt, leidet die Genauigkeit. Es gibt Flugzeuge, die nur in einer bestimmten Fenstergröße erkannt werden. Wird diese Größe durch einen ungünstigen Skalierungsfaktor übersprungen, so wird dieses Flugzeug nicht erkannt. Es gilt also auch hier einen Kompromiss zwischen Laufzeit und Genauigkeit zu finden.

Um die Genauigkeit bei größeren Fenstergrößen zu erhöhen, wurde eine neue Art positiver Trainings-Samples entworfen. Bisher wurde so vorgegangen, dass das Flugzeug oben, unten, links und rechts an die Fensterbegrenzung stößt. Das führt dazu, dass ein Flugzeug nur dann erkannt wird, wenn es das komplette Fenster ausfüllt. Durchsucht man das Bild mit einem hohen Skalierungsfaktor, so passiert es, dass ein Flugzeug nicht erkannt wird, da es zu klein ist und nicht das Fenster ausfüllt. Um diese Flugzeuge zu erkennen, wurden die Trainingssamples so konstruiert, dass Flugzeuge in unterschiedlichen Skalierungen inmitten einer festen Fenstergröße platziert wurden.

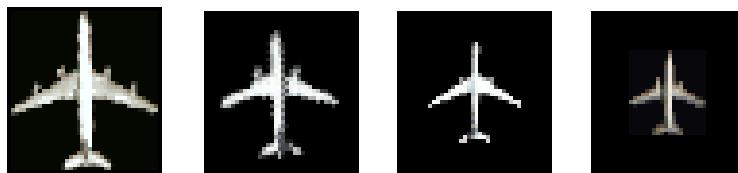


Abbildung 7.3.: Beispiel für die neuen positiven Samples.

Dies sollte zu einer höheren Genauigkeit bei einem größeren Skalierungsfaktor führen. Überraschenderweise war dies nicht der Fall. Die skalierten und die unskalierten Trainingssamples erbrachten in etwa dieselben Ergebnisse, wobei die unskalierten etwas besser abschnitten.

Ein Faktor, der hierbei eine wichtige Rolle spielt, ist der Parameter *minNeighbors*. Er gibt an, wie viele Nachbar-Treffer ein Treffer mindestens haben muss, um als solcher anerkannt zu werden. Da bei einem höheren Skalierungsfaktor insgesamt weniger Fenster entstehen, muss dieser Wert angepasst werden. Normalerweise zählen Fenster der nächsten oder der vorherigen Skalierungsstufe als Nachbarn. Ist der Skalierungsfaktor allerdings zu hoch, werden diese nicht mehr als Nachbarn gewertet. Um also keine Treffer durch einen falsch gewählten *minNeighbors* Parameter zu



verlieren, wurden die Werte 1 bis 3 getestet.

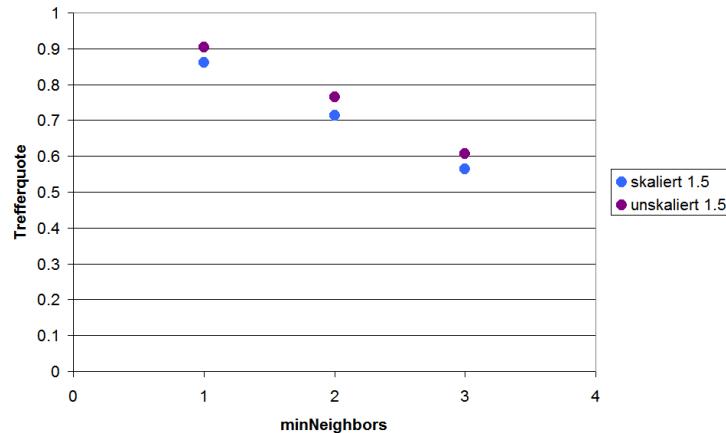


Abbildung 7.4.: Skalierte und unskalierte Trainingssample im Vergleich bei Skalierungsfaktor 1.5.

Eine weitere, hier nur angedachte, Idee ist, den Klassifizierer mit der höchsten Trefferquote heran zunehmen und für alle Treffer die Fenstergrößen, in denen sie gefunden werden zu notieren. Damit ließe sich die minimale Anzahl an Fenstergrößen feststellen, die benötigt werden, um alle Treffer zu finden. Das wäre somit auch die minimale Anzahl an benötigten Durchläufen.

## 7.7. Skalierungsfreie-Detektion

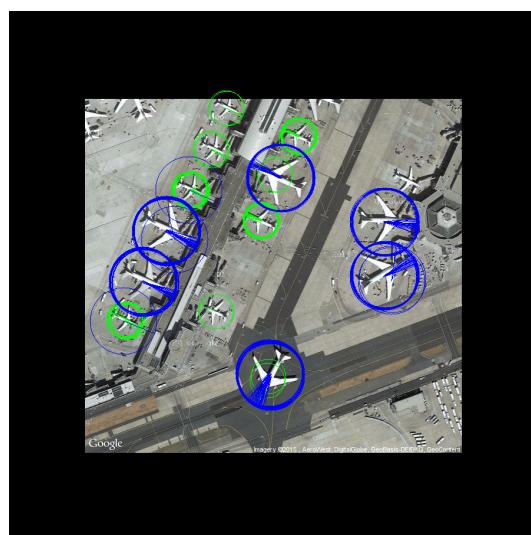


Abbildung 7.5.: Beispiel für das skalierungsfreie Klassifizieren.



Inspiriert durch die große Zeitersparnis bei hohen Skalierungsfaktoren, wurde ein Modell entwickelt, das ohne Fenster-Skalierung auskommt. Die Idee dabei ist, mit zwei festen Fenstergrößen einmal über das Bild zu laufen. Mit der einen Fenstergröße werden die größeren Flugzeuge erkannt und mit der anderen die kleineren. Dafür mussten wieder spezielle Trainingssamples erstellt werden, die auf dieses Modell zugeschnitten sind. Diesmal wurde auf den Einsatz von *opencv\_createsamples* zu Erstellung künstlicher Samples verzichtet und stattdessen eine selbst programmierte Routine verwendet.

Zuerst wurden die Positiv-Beispiele nach Größe in zwei Gruppen geteilt. Jedes Flugzeug höher als 60 Pixel wurde als großes Flugzeug eingeordnet. Das größte Flugzeug der großen Gruppe wurde so skaliert, dass es genau in ein 40x40 Fenster passt. Dieser Skalierungsfaktor wurde dann auf die restlichen großen Flugzeuge angewendet. Die erhaltenen Bilder wurden dann in 1-Grad-Schritten von -10 bis +10 Grad gedreht. So dass aus einem Bild 21 verschiedene gedrehte Bilder erzeugt wurden. Diese wurden dann auf zufällig ausgewählte Hintergründe aus Negativ-Beispielen gelegt.

Dasselbe Verfahren wurde auch bei den kleinen Flugzeugen angewandt, nur das diese in ein 30x30 Fenster skaliert wurden.

Anschließend wurden mit diesen positiven Samples zwei Klassifizierer trainiert. Jeweils einen für die große und einen für die kleine Gruppe. Die Methode *detectMultiScale* wurde dann mittels dem Skalierungsfaktor und der minimalen und maximalen Objektgröße, so angepasst das ein Klassifizierer, in der gelernten Fenstergröße, genau einmal das Bild durchläuft.

Dies liefert eine Laufzeit-Ersparnis von ca. 98 ms pro Detektion und 26 Sekunden pro Bild.

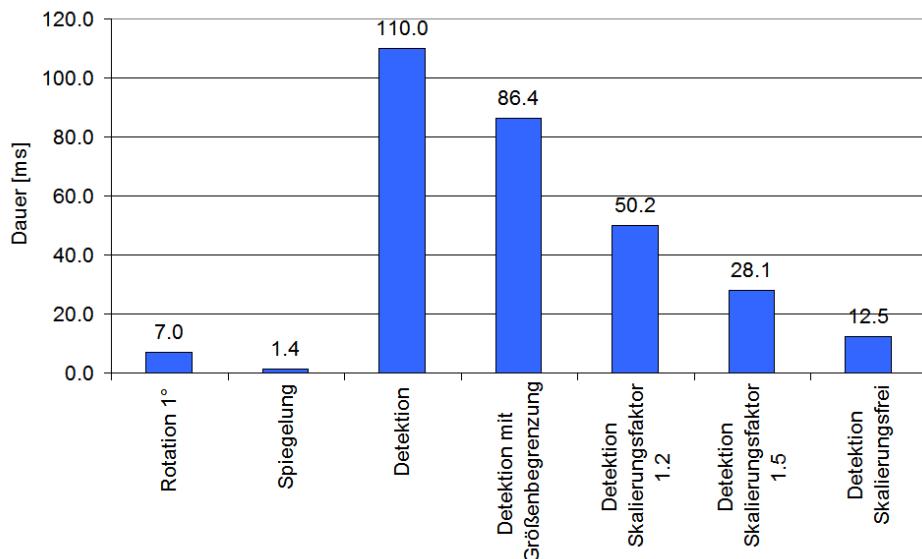


Abbildung 7.6.: Vergleich der Varianten zur Laufzeit-Optimierung.



Allerdings ist die Genauigkeit beider Klassifizierer nicht die Beste. Besonders der Klassifizierer der kleinen Gruppe machte Probleme. Er konnte nur durch die Bootstrap-Methode auf eine annehmbare Trefferquote gebracht werden.

Es wurden mehrere Klassifizierer-Kombinationen getestet. Die beste Trefferquote, die erreicht werden konnte, lag bei 91%.

## 7.8. Zusammenfassung

Bis auf die Implementation in C++ und den Versuch mit Zoomstufe 16 zu arbeiten, erreichen die vorgestellten Verfahren eine Verkürzung der Laufzeit. Nun wird analysiert, welches Verfahren den besten Kompromiss zwischen Laufzeit und Genauigkeit bietet. Dazu wurde Abbildung 7.7 erstellt.

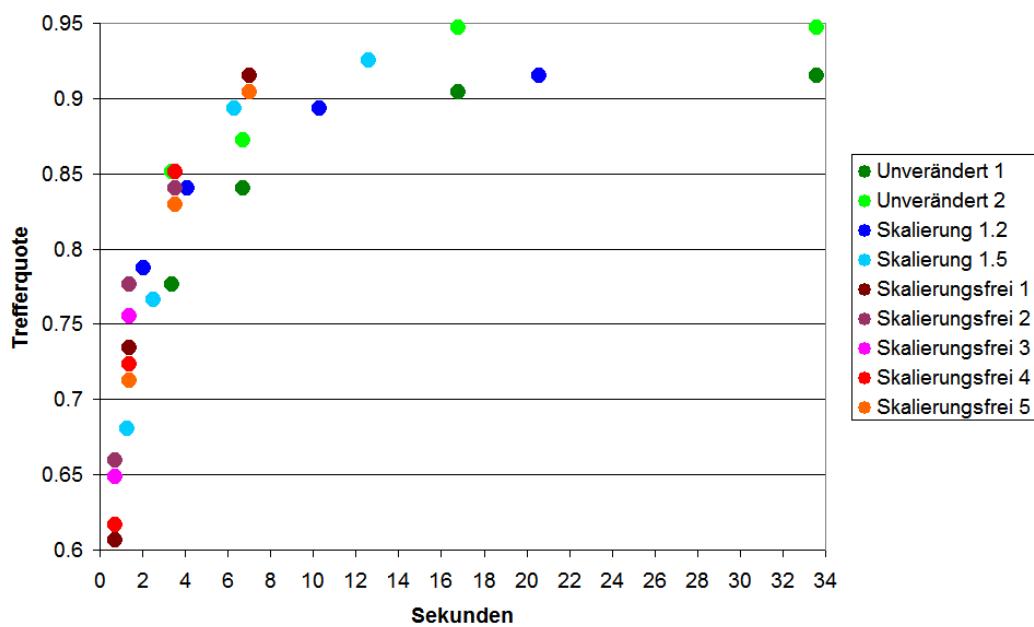


Abbildung 7.7.: Laufzeit pro Bild gegen Trefferquote.

Hier sind alle möglichen Verfahren abgetragen. Dabei bedeutet:

Unverändert 1 + 2: Klassifizierer ohne Laufzeit-Optimierung.

Skalierung 1.2 + 1.5: Skalierungsfaktor der Fenstergröße bei der Detektion.

Skalierungsfrei 1 - 5: Zwei Klassifizierer kombiniert, mit fester Fenstergröße bei der Detektion.

Jeder Klassifizierer ist hier mit vier Punkten dargestellt. Diese entsprechen den Schrittgrößen  $1^\circ$ ,  $2^\circ$ ,  $5^\circ$  und  $10^\circ$  in denen das Bild gedreht wird. Der Punkt für  $1^\circ$  befindet sich am weitesten rechts, da er die größte Laufzeit besitzt. Die anderen gehen ihm der Größe nach voraus.



Zu sehen ist, wie die unterschiedlichen Klassifizierer mit den unterschiedlichen Detektions-Verfahren abschneiden. So sieht man z.B., dass der Klassifizierer *Unverändert 2* bei einer Schrittgröße von  $1^\circ$  ca. 34 Sekunden pro Bild benötigt und eine Trefferquote von ca. 95% aufweist. Der selbe Klassifizierer mit einer Rotation in  $2^\circ$ -Schritten erreicht immer noch 95%, ist allerdings doppelt so schnell mit ca. 17 Sekunden pro Bild.

Die Klassifizierer *Skalierung 1.2* und *Skalierung 1.5* sind schon bei der 1-Grad-Drehung schneller als die unveränderten Klassifizierer, allerdings auch etwas unge nauiger.

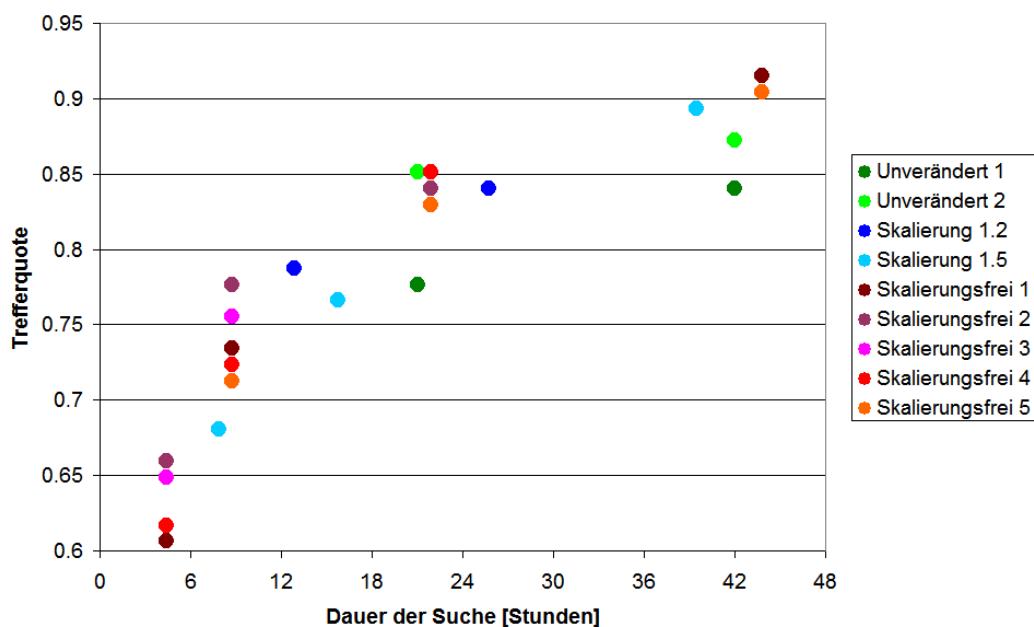


Abbildung 7.8.: Laufzeit pro Bild gegen Trefferquote.

Die skalierungsfreien Kalssifizierer sind die schnellsten, mit Werten bis zu unter einer Sekunde pro Bild. Diese weisen aber auch eine recht geringe Trefferquote auf. Welcher, der zur Verfügung stehenden Verfahren, nun eingesetzt werden soll für eine großflächige Suche, hängt davon ab wieviel Zeit zu Verfügung steht und wie wichtig eine hohe Trefferquote ist. Um für die Suche rund um den Flughafen Frankfurt eine Entscheidung zu treffen, wurde die Abbildung 7.8 erstellt. Diese beschreibt die selben Werte wie Abbildung 7.7, nur dass die Zeit in Stunden, für das Durchsuchen des gesamten Gebietes auf der x-Achse angezeigt wird. Diese Dauer wurde auf 48 Stunden begrenzt.

Für die großflächige Suche wurde nun ein Klassifizier-Verfahren ausgewählt, dass den Bereich innerhalb von 24 Stunden durchsuchen kann. Das schnellste und dabei genaueste Verfahren, war der unveränderte Klassifizierer 2, mit einer Drehung



des Bildes in 10°-Schritten. Er braucht geschätzt 21 Stunden und hat dabei eine Trefferquote von 85%.

### 7.8.1. Speicherleck

Nachdem die großflächige Suche mit dem gewünschten Klassifizier-Verfahren gestartet wurde, brach das Programm nach ca. 15 Minuten ab. Der Grund: Nicht genügend Arbeitsspeicher zur Verfügung. Es kann beobachtet werden, wie der Arbeitsspeicher nach und nach immer voller wird, bis 99% beansprucht sind. Danach bricht das Programm ab. Das deutet auf ein Speicherleck (engl. *Memory Leak*) hin. Dies tritt auf, wenn der Speicher nicht mehr verwendeter Variablen nicht zur weiteren Nutzung frei gegeben wird.

Normalerweise braucht das Programm nur einen geringen Teil an Arbeitsspeicher. Diesen belegen der geladene Klassifizierer, das geladene Bild, die entstehende Rotationsmatrix, das gedrehte Bild und die Liste, in der die Treffer gesammelt werden. All diese Variablen, sollten für das folgende Bild wieder frei gemacht werden. Mit Ausnahme des Klassifizierers. Dieser kann aber muss nicht mit jedem Bild neu geladen werden. In Java ist der sogenannte *Garbage Collector* eigentlich dafür zuständig, nicht mehr referenzierte Variablen frei zu geben. Um dennoch sicher zu gehen, dass das Speicherleck nicht im Java-Code zu finden ist, wurden alle benutzen Variablen, inklusive des Klassifiziereres, von Hand am Ende der Schleife auf *null* gesetzt. Der in OpenCv genutzte *Mat*-Datentyp hat eine *release()*-Funktion. Auch diese wurde angewandt. Nichts davon brachte irgendeine Veränderung.

Daher ist davon auszugehen, dass entweder im Code der OpenCv-Bibliothek, wahrscheinlich in der Methode *detectMultiScale*, ein Speicherleck vorliegt oder, dass bei der Kommunikation zwischen Java und der eingebundenen OpenCv-Bibliothek Probleme auftreten, die sich als Speicherleck äußern. Eine Lösung dieses Problems konnte im zeitlichen Rahmen der Diplomarbeit leider nicht mehr gefunden werden, so dass auf eine großflächige Suche verzichtet werden musste.



## 8. Fazit

Die von Viola und Jones entwickelte Methode zur Objekterkennung konnte mit Erfolg für die Flugzeugerkennung in Satellitenbildern umgesetzt werden. Die Kombination aus Kaskaden-Klassifizierern und *Local-Binary-Pattern* ermöglichte es bis zu 95% aller Flugzeuge im Testset zu erkennen, mit einer Falsch-Positiv-Rate von 14%. Dies ist ein erheblich besseres Ergebnis, als das was im Rahmen der Studienarbeit mit SURF-Merkmalen erzielt werden konnte.

Die größte Schwäche der *Local-Binary-Pattern*, so zeigte sich, ist ihre nicht vorhandene Rotations-Invarianz. Dies führt zu einer erhöhten Laufzeit, da es nötig ist das Bild schrittweise zu drehen.

Dem wurde durch verschiedene Verfahren zur Laufzeit-Optimierung entgegen gewirkt. So wurde eine Laufzeit von anfänglich 40 Sekunden pro Bild auf eine Laufzeit von 17 Sekunden pro Bild verringert, ohne dass dabei Verluste in der Genauigkeit entstanden. Das Verfahren konnte sogar soweit beschleunigt werden, dass ein Bild in weniger als einer Sekunde durchsucht werden konnte, dabei muss jedoch eine recht geringe Trefferquote von 60% in Kauf genommen werden.

Damit lässt sich sagen, dass diese Methode für eine Erkennung in Echtzeit ungeeignet ist. Auch für großflächige Suchen, die wegen eines Speicherlecks leider nicht getestet werden konnten, wäre eine kürzere Laufzeit wünschenswert.

Wegen ihrer langen Trainingsdauer, wurden die Haar-ähnlichen Merkmale nicht näher untersucht, aber es ist zu erwarten, dass mit diesen ein noch etwas besseres Ergebnis erzielt werden könnte.

Generell könnten mit mehr Zeit, einem größeren Trainingsset und einer höheren Rechenleistung noch mehr Fortschritte gemacht werden, sowohl in der Genauigkeit als auch in der Laufzeit. Man könnte z.B. die Ergebnisse großflächiger Suchen dazu nutzen, das Trainingsset nach und nach zu verbessern und damit schrittweise einen immer robuster werdenden Klassifizierer trainieren.

Des Weiteren wäre es interessant, einen Kaskaden-Klassifizierer mit rotations-invarianten Merkmalen zu kombinieren. Damit ließe sich die Flugzeugerkennung in Satellitenbildern möglicherweise noch weiter beschleunigen.



## Literaturverzeichnis

### advancedsourcecode 2015

ADVANCEDSOURCECODE: *High-Speed Face Recognition with Local Binary Patterns.* <http://www.advancedsourcecode.com/lbphistogram.asp>. Version: 2015. – [Online; Stand 1. Oktober 2015] 3.8

### Bay u. a. 2006

BAY, Herbert ; TUYTELAARS, Tinne ; GOOL, Luc V.: SURF: Speeded Up Robust Features. In: *Proceedings of the ninth European Conference on Computer Vision*, 2006 2.2

### Canny 1986

CANNY, J: A Computational Approach to Edge Detection. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 8 (1986), Juni, Nr. 6, S. 679–698. – ISSN 0162-8828 2.2

### Haar 1910

HAAR, A.: Zur Theorie der Orthogonalen Funktionensysteme. (German) [On the theory of orthogonal function systems]. 69 (1910), S. 331–371. – ISSN 0025-5831 (print), 1432-1807 (electronic) 2.2, 3.3.1

### Lienhart und Maydt 2002

LIENHART, R. ; MAYDT, J.: An extended set of Haar-like features for rapid object detection. In: *Image Processing. 2002. Proceedings. 2002 International Conference on* Bd. 1, IEEE, 2002. – ISBN 0-7803-7622-6, I-900-I-903 vol.1 3.3.1

### Manuel Walter 2011

MANUEL WALTER, Helmut K.: Seminar: Multi-Core Architectures and Programming. (2011) 3.1

### Ojala u. a. 1996

OJALA, Timo ; PIETIKÄINEN, Matti ; HARWOOD, David: A comparative study of texture measures with classification based on featured distributions. In: *Pattern Recognition* 29 (1996), Januar, Nr. 1, 51–59. [http://dx.doi.org/10.1016/0031-3203\(95\)00067-4](http://dx.doi.org/10.1016/0031-3203(95)00067-4). – DOI 10.1016/0031-3203(95)00067-4. – ISSN 00313203 3.3.2



### Serra 2015

SERRA: Anwendung des SURF-Algorithmus - Flugzeugerkennung in Satellitenbildern. In: *Studienarbeit* (2015) 2.2, 2.3

### Silva u. a. 2013

SILVA, Romuere ; AIRES, Kelson ; VERAS, Rodrigo ; SANTOS, Thia-  
go ; LIMA, Kalyf ; SOARES, AndrÃ.: Automatic Motorcycle Detecti-  
on on Public Roads. In: *CLEI Electronic Journal* 16 (2013), 12, 4  
- 4. [http://www.scielo.edu.uy/scielo.php?script=sci\\_arttext&pid=S0717-50002013000300004&nrm=iso](http://www.scielo.edu.uy/scielo.php?script=sci_arttext&pid=S0717-50002013000300004&nrm=iso). – ISSN 0717-5000 3.7

### Viola und Jones 2001

VIOLA, Paul ; JONES, Michael: Rapid object detection using a boosted cascade of simple features, 2001, S. 511–518 1, 3.1

### Wikipedia 2015a

WIKIPEDIA: Local binary patterns — Wikipedia, The Free Encyclopedia. [https://en.wikipedia.org/w/index.php?title=Local\\_binary\\_patterns&oldid=678249685](https://en.wikipedia.org/w/index.php?title=Local_binary_patterns&oldid=678249685). Version: 2015. – [Online; accessed 1-October-2015] 3.9

### Wikipedia 2015b

WIKIPEDIA: Viola-Jones-Methode — Wikipedia, Die freie En-  
zyklopÃ¤die. <https://de.wikipedia.org/w/index.php?title=Viola-Jones-Methode&oldid=139476346>. Version: 2015. – [Online; Stand 1.  
Oktober 2015] 3.2, 3.3, 3.4



## **A. Anhang**

### **A.1. Eidesstattliche Erklärung**

Ich, Felice Serra, Matrikel-Nr. 2944562, versichere hiermit, dass ich meine Studienarbeit mit dem Thema

*Flugzeugerkennung in Satellitenbildern Anwendung von Kaskaden-Klassifizierern  
nach Viola und Jones*

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Tübingen, den 1. Oktober 2015

---

FELICE SERRA

### **A.2. Adresse**

Felice Serra  
Schlachthausstraße 30  
72074 Tübingen  
E-Mail: felice.serra@gmx.de  
Tel.: 0178-8017472