Hochschule Wismar
University of Technology, Business and Design
Fakultät für Ingenieurwissenschaften

Forschungsprojekt
Videozusammenfassung und Klassifikation

von: Patrice Matz

Betreuer: Prof. Dr. Litschke

Datum: Januar 2021

Abstrakt

In dieser Arbeit wird ein Verfahren zur Videozusammenfassung und -klassifizierung erläutert. Das Verfahren basiert auf der Extraktion von Änderungen zwischen Einzelbildern und der Aggregation dieser in Ebenen.

Es wird ein objektorientierter und komponentenweise parallelisierter Ansatz verfolgt und in der Programmiersprache Python 3 unter Zuhilfenahme von der Frameworks NumPy, OpenCV und Tensorflow implementiert.

Inhalt

Ziel

Es wird ein Algorithmus angestrebt mit welchem alle relevanten Bewegungen aus einem Video extrahiert werden. Stehen einzelne Kontouren zeitlich oder räumlich im Zusammenhang sind diese zu Ebenen zu aggregieren.

Diese Ebenen sind anschließend entsprechend des Inhaltes zu klassifizieren, sodass eine Suche nach bestimmten Ereignissen in einem Video folgen könnte. Die Suche selbst wird nicht betrachtet, stattdessen soll eine Liste von Objekten aus dem COCO Datensatz in Ebenen identifiziert und die Ebenen anschließend mit diesen annotiert werden.

1 Konzept

In diesem Kapitel werden die Architektur, Datenstrukturen, Klassifizierungsansätze und das zugrundeliegende Konzept erläutert.

Grundsätzlich ist es möglich zwei Bilder einer statischen Kamera miteinander zu vergleichen und somit Änderungen zu finden. Dies ist ein häufig eingesetztes Verfahren der Bild- und Videoverarbeitung. Werden aufeinander folgende Bilder eines Videos miteinander verglichen und die Unterschiede nacheinander abgespielt wirkt dies auf einen Zuschauer als würden alle Änderungen in einem Video abgespielt werden. Logisch besteht aber keine Verbindung zwischen den einzelnen Konturen, somit ist keine Filterung oder weitere Verarbeitung möglich.

Weitere Verarbeitungsschritte könnte die Feststellung von zeitlichen oder räumlichen Mustern, die Klassifizierung und Annotierung von Bewegungen oder ähnliches sein. Um dies zu ermöglichen muss eine logische Verbindung zwischen den einzelnen Konturen hergestellt werden. In Abb. 1 ist der Datenfluss zwischen den Komponenten vereinfacht dargestellt. Die logische Verbindung zwischen Konturen wird über die Aggregation in Ebenen oder Layer realisiert. Eine extrahierte Kontur wird mit den letzten n Konturen aller Layer verglichen. Die Layer dürfen hierbei nicht älter als x Frames sein. Gibt es eine Überschneidung zwischen der aktuellen Kontur und einem existierenden Layer wird die Kontur dem Layer hinzugefügt. Gibt es weitere Überschneidungen werden alle gefundenen Layer kombiniert. Dies kann als transitive Assoziation interpretiert werden.

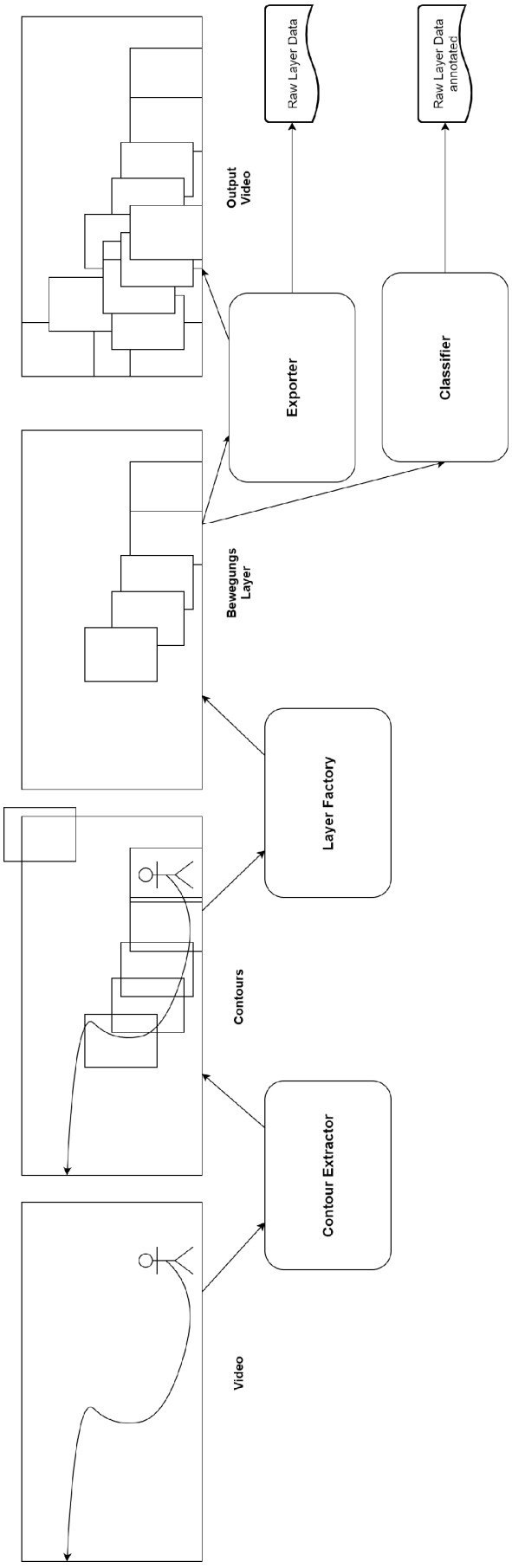

Abb. 1: Vereinfachter Datenfluss

1.1 Architektur

Dieses Unterkapitel dient der Herleitung der genutzten Architektur. Das Grundprinzip wird an einem vereinfachten Modell mit drei Komponenten erläutert. Anschließend wird eine erweiterte Architektur mit sechs parallelisierten Komponenten vorgestellt.

1.1.1 Drei Komponenten Modell

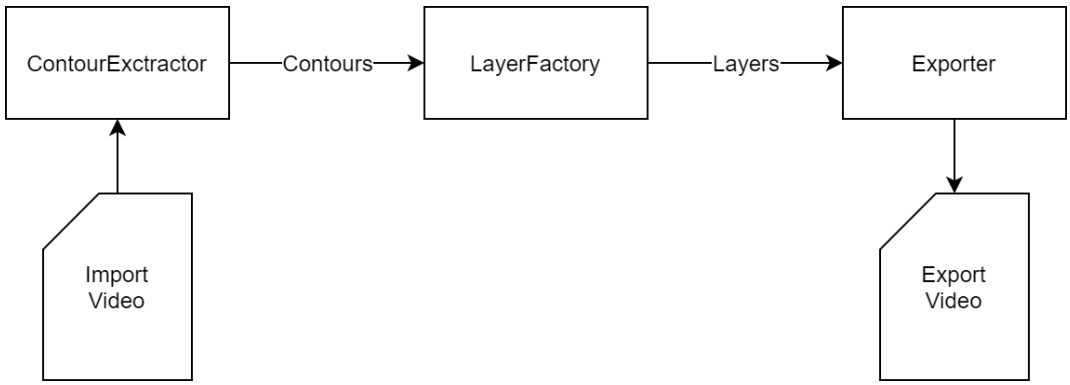

Abb. 2: Vereinfachte Architektur mit drei Komponenten

Wie in Abb. 2 dargestellt besteht die grundlegende Architektur aus drei Komponenten. Jede Komponente entspricht einer Klasse. Jede Klasse hat genau eine Aufgabe. Das „ContourExtractor“-Objekt extrahiert Konturen, diese dienen als Input einer Funktion des „LayerFactory“-Objektes, die Layer Factory wiederum produziert eine Menge von Layer Objekten, welche vom Exporter in ein Video gerendert werden. Auf die verwendeten Algorithmen wird in Kapitel 1.3 eingegangen.

Diese Architektur ist vollständig sequenziell. Somit ist die Verarbeitung eines einzigen Videos sehr zeitaufwendig, da zur Verfügung stehende Ressourcen nicht vollständig ausgelastet werden.

1.1.2 Sechs Komponenten Modell

Um die Ressourcenauslastung zu erhöhen und die Verwaltung der Layer zu verbessern werden zwei neue Komponenten eingefügt und bestehende Komponenten parallelisiert.

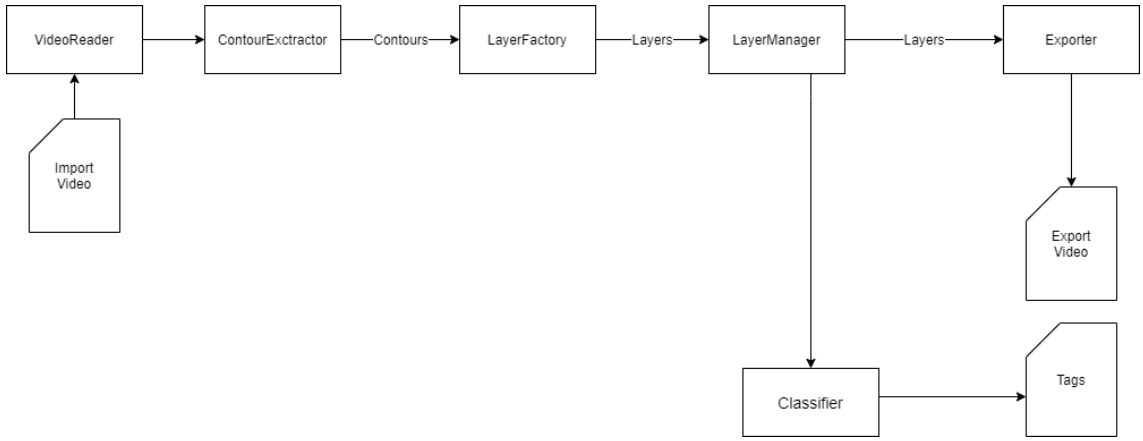

Abb. 3: Architektur mit sechs Komponenten

Die neuen Komponenten sind das „VideoReader"-Objekt und das „LayerManagement"-Objekt. Das LM-Objekt enthält eine Menge von Classifier-Objekten, welche jeweils ein Classifier-Interface Implementieren.

Das VR-Objekt spawnt einen neuen Thread, welcher das Video dekodiert und die dekodierten Frames in einem Buffer speichert. Auf den hierzu verwendeten Algorithmus wird in Kapitel 1.3.1 eingegangen.

Das LM-Objekt wurde eingefügt, um Funktionen zu aggregieren, die alle Layer betreffen, wie z.B. das Klassifizieren oder Rausfiltern von Layern mit geringer Qualität.

1.2 Datenstrukturen

Der verwendete Algorithmus benötigt zwei komplexe Datenstrukturen, diese werden in diesem Unterkapitel erläutert.

1.2.1 Konturen

Aus jedem Einzelbild können eine Menge von Konturen extrahiert werden, diese müssen mit einer Zuordnung des Bildes aus welchem sie entnommen wurden abgelegt werden. Hierzu wird ein Dictionary verwendet, welches die „Frame Number" als Key und eine Liste von Konturen als Value speichert. Die Konturen werden nicht mit ihrem

Inhalt gespeichert, da dies große Mengen an Hauptspeicher benötigen würde. Stattdessen werden die X- und Y-Koordinaten, die Breite und Höhe gespeichert.

Daraus resultiert eine Datenstruktur welche wie folgt aussehen könnte.

```
Konturen = {

0: [(10, 10, 20, 50), (100, 100, 100, 50), (150, 150, 80, 35), …],

1: [(12, 9, 20, 50), (105, 104, 100, 50), (150, 160, 80, 35), …],

…

}
```

Listing 1: Datenstruktur extrahierter Konturen

1.2.2 Ebenen

Extrahierte Konturen sollen zu Ebenen zusammengefasst werden. Jede Ebene enthält eine Bewegung. Auch wenn mehrere Bewegungen zum selben Zeitpunkt stattfinden wird für jede dieser Bewegungen eine separate Ebene erstellt. Dies erleichtert das Bereinigen der Ebenen von Störungen, da Störungen somit meist in einer eigenen Ebene liegen und somit die Qualität der andere Ebenen nicht beeinflussen. Als hochwertig wird eine Ebene bezeichnet, wenn sie möglichst wenig Störungen und Unterbrechung beinhaltet und der Inhalt relevant ist. Bewegende Blätter z. B. sind irrelevant, eine Person oder ein Hund die nachts über ein Grundstück laufen sind hingegen relevant.

Ebenen sind komplexere Datenstrukturen mit spezifischen Funktionen, daher wird ein Layer Objekt erstellt, dieses kann in Listing 2 gefunden werden.

```
Class Layer:

      startFrame = None

      lastFrame = None

      length = None

      data = []

      bounds = []

      __init__()

      __len__()

      add()
```

Listing 2: Datenstruktur Layer

Eine Ebene / Layer ist eine Sammlung von zusammenhängenden Konturen, die Datenstruktur spiegelt dies wider. Das „Bounds“ Array ähnelt dem Konturen Dictionary, nur werden die Schlüssel durch den Index ersetzt. Zusätzlich werden nur überlappende Konturen gespeichert.

1.3 Algorithmen

Die zentralen Algorithmen dieses Forschungsseminares werden in diesem Unterkapitel erläutert. Im Kapitel 2 Implementierung wird auf die Details der Implementierung eingegangen.

1.3.1 Konturenextraktion

Die Konturenextraktion basiert auf dem Vergleich zweier Bilder und dem Speichern der Unterschiede. Dieses Verfahren ist genauer als nötig und Störungsanfällig bei Bewegungsrauschen und Helligkeitsänderungen. Daher wird eine Reihe weiterer Schritte eingefügt.

Algorithmus zur Extraktion von Konturen:

1. Berechnen des Durchschnitts der vorherigen X Bilder
2. Bilder herunterskalieren
3. Farbraum nach Graustufen konvertieren
4. Gaußschen Weichzeichner anwenden
5. Unterschied der Graustufen berechnen
6. Binarisierung
7. Dilatation
8. OpenCV::findContours()
9. Konturen nach Größe filtern
10. Speichern der Konturen in einer Datenstruktur

1.3.2 Ebenenaggregation

Die Ebenenaggregation ist einfach verglichen mit der Konturenextraktion. Es wird über alle Layer iteriert, ist das Layer älter als die Toleranz zulässt wird es übersprungen. Anschließend werden die Konturen der letzten fünf Frames des jeweiligen Layers mit der aktuellen Kontur verglichen, gibt es eine Überlappung wird die Kontur in das Layer eingefügt.

Kann eine Kontur zu mehr als einem Layer hinzugefügt werden, wird sie dem ersten Layer hinzugefügt und allen nachfolgenden Layern mit einander kombiniert. Somit ist eine wird eine transitive Assoziation eingefügt.

1.4 Klassifizierung

Die Klassifizierung von Bildern und Videos hat in den letzten Jahren immer mehr an Bedeutung gewonnen. Nicht zuletzt aufgrund von Fortschritten im Bereich selbstfahrender Fahrzeuge, Gesichtserkennung, Bildverarbeitung oder Robotik. Hierbei scheinen Machine Learning Ansätze am vielversprechendsten. Aus diesem Grund werden sie auch in diesem Projekt verwendet. Tensorflow ist ein beliebtes Framework, welches sowohl in der Forschung als auch in der Wirtschaft eingesetzt wird, daher wird es auch in diesem Projekt eingesetzt.

Auch wenn neuronale Netze in dieser Arbeit bevorzugt werden bedeutet dies nicht das andere Klassifizierungsansätze ungeeignet sind, besonders in Kombination mit anderen Ansätzen können die Annotierungen deutlich aussagekräftiger werden. Ein NN basierter Klassifikator mag ein Layer mit dem Tag „Truck“ annotieren, ein weiterer simplerer Klassifikator könnte die primäre Farbe des Objektes feststellen. Das Tagging von Ebenen kann mit beliebig vielen Klassifikatoren erfolgen, somit wäre anschließend eine deutliche genauere Filterung möglich.

1.4.1 Neuronale Netze

Neuronale Netze können, neben anderen Verfahren, zur Klassifizierung von z. B. Bildern eingesetzt werden. Neuronale Netze bestehen aus einer Eingabeschicht, einer versteckten Schicht und einer Ausgabeschicht, wobei die verstecke Schicht Null, bis N Schichten enthalten kann. Jede dieser Schichten enthält Nodes, welche mit Nodes folgender und / oder vorangegangener Schichten verbunden sind. Die Struktur eines einfaches Neuronales Netz ist in Abb. 2 dargestellt.

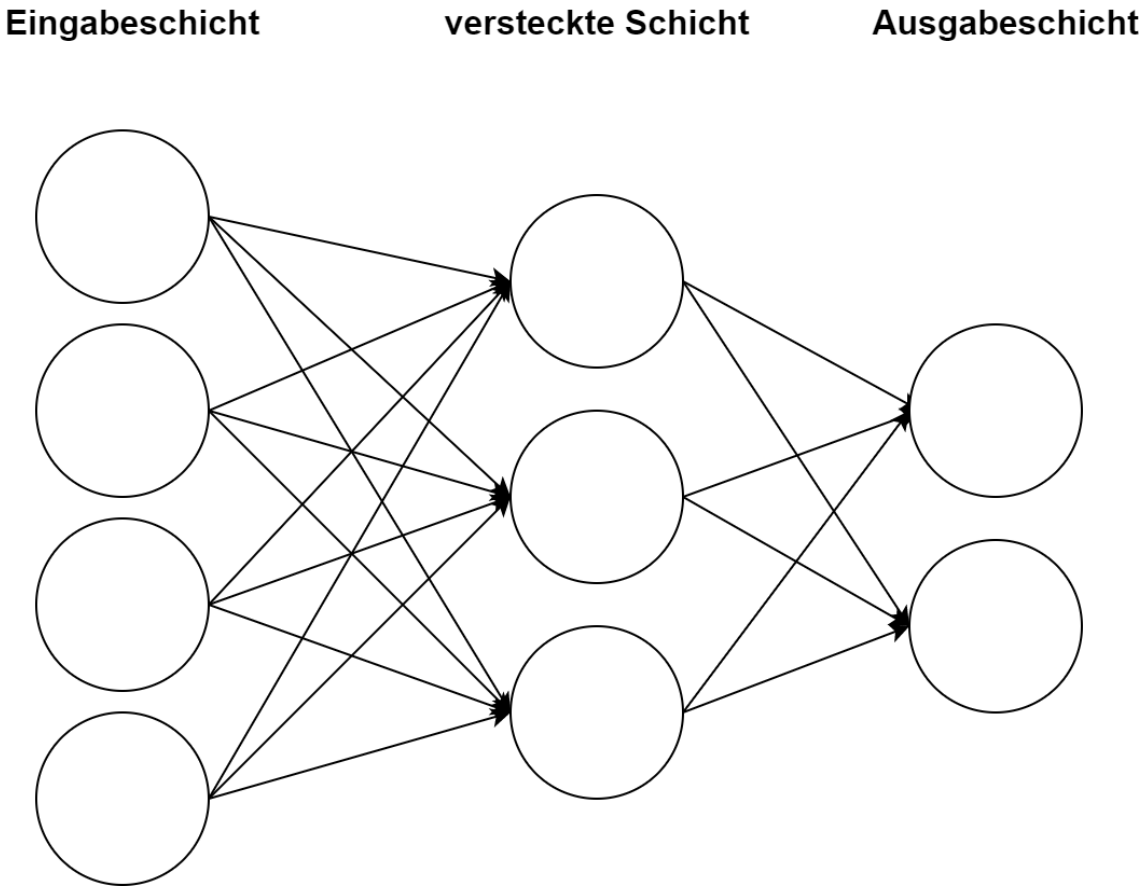

Abb. 4: Neuronales Netz mit einer versteckten Schicht

Das Verhalten eines Neuronalen Netzes und der Umgang mit Eingaben hängt von einer Vielzahl von Faktoren ab, unteranderem der Anzahl der versteckten Schichten, der Anzahl der Nodes pro Schicht, der Anzahl der Trainingszyklen, der Art, Anzahl und Qualität der Trainingsdaten und weiteren. Neuronal Netze sind ein komplexer Themenbereich, der in dieser Arbeit nicht tiefgründig untersucht werden kann.

Im Rahmen dieser Arbeit werden neuronale Netze als eine Implementierung des „Classifier-Interfaces“ betrachtet. Es wird ein vortrainiertes neuronales Netz verwendet, welches mit Bildern des COCO-Datensatzes (Common Objects in Context) trainiert wurde. Das bedeutet, dass Bildausschnitte als Eingabe eines Neuronalen Netzes dienen und die Ausgabe als Index für eine Klasse des COCO-Datensatzes interpretiert wird. Da das neuronale Netz somit als Black-Box betrachtet wird ist ein tiefes Verständnis von neuronalen Netzen nicht für das Verständnis dieser Arbeit nötig.

2 Implementierung

In diesem Kapitel wird die Implementierung der bisher vorgestellten Konzepte erläutert. Der Aufbau dieses Kapitels ähnelt hierbei der Processing Pipeline. Es wird in Kapitel 2.2 mit der synthetischen Generierung eines Validierungsvideos begonnen. Anschließend wird die Implementierung der Konturenextraktion erläutert. Anschließend folgt ein Unterkapitel zur Aggregation von Konturen zu Layern mithilfe einer Layer Factory und der Verwaltung und Klassifikation dieser mithilfe des Layer Managers und Classifiern. Abschließend wird auf das Objekt zum Exportieren der Videos und der erzeugten Daten eingegangen.

2.1 Vorgehen

Die Qualität der Videoverarbeitung und besonders die Videozusammenfassung kann schwer quantifizierbar sein, da die Ideallösung nicht immer bekannt ist. Daher wurde mit der Implementierung eines Tools zur synthetischen Generierung eines Videos mit definierten Ereignissen begonnen. Mithilfe der so erzeugten Videos konnten erzeugt Ergebnisse und somit auch das verwendete Verfahren validiert werden, da bereits bekannt ist wie viele Ereignisse mit welcher Dauer zu extrahieren sind.

Videos liegen aufgrund ihres hohen Speicherbedarfs meist in komprimierter Form vor. Sollen ein Video verarbeitet oder analysiert werden bedeutet dies daher oft, dass die Einzelbilder erst dekomprimiert bzw. decodiert werden müssen. Der Zeitaufwand für das Decodieren eins Videos, Frame für Frame ist nicht vernachlässigbar. Daher wird ein „VideoReader“-Objekt geschaffen, welches die Dekodierung der Videos in einen eigenen Thread auslagert und die dekodierten Frames in einem Buffer für die weitere Verarbeitung ablegt.

Somit sind die Voraussetzungen für eine effiziente und zielgerichtete Arbeit and der eigentlichen Video-Zusammenfassung Komponente geschaffen.

2.2 Videosynthetisierung

Die Videosynthetisierung erfolgt mit dem Programm „gen.py“. Das Programm verfügt über kein CLI, da es in der Programmiersprache Python geschrieben ist können aber Einstellungen geändert werden, ohne dass eine erneute Kompilierung erfolgen muss.

In dem Programm können die Auflösung der Videos, die FPS, die Länges des Videos und die Anzahl der Ereignisse spezifiziert werden. Es ist nur ein Ereignis gleichzeitig sichtbar. Die Zeit zwischen Ereignissen ergibt sich als Länge des Videos / Anzahl der Ereignisse.

Ein Ereignis besteht aus dem Erscheinen eines Rechtecks mit zufälliger Dimensionierung, an einer zufälligen Position und der Bewegung dieses in eine Zufällige aber konstante Richtung, zusätzlich wird dem Rechteck eine zufällige Farbezugewiesen. So können die Rechtecke und die Farbtreue nach der Video-Zusammenfassung besser validiert werden.

Die Implementierung kann in Listing 3 eingesehen werden.

```
def genVideo():
    writer = imageio.get_writer(outputPath, fps=fps)
    writer.append_data(np.zeros(shape=[1080, 1920, 3], dtype=np.uint8))
    writer.append_data(np.zeros(shape=[1080, 1920, 3], dtype=np.uint8))

    for i in range(numberOfEvents):
        objectWidth = (5 + random.randint(0, 5)) * xmax / 100
        objectHeight = (10 + random.randint(-5, 5)) * ymax / 100

        objectX = random.randint(0, xmax)
        objectY = random.randint(0, ymax)

        objectSpeedX = random.randint( 1 ,5 )
        objectSpeedY = random.randint( 1, 5 )
        color = getRandomColorString()

        for j in range(int(fps*length*60 / numberOfEvents)):
            objectX -= objectSpeedX
            objectY -= objectSpeedY

            objectShape = [
                (objectX, objectY),
                (objectX + objectWidth, objectY + objectHeight)
            ]
            img = Image.new("RGB", (xmax, ymax))
            img1 = ImageDraw.Draw(img)
            img1.rectangle(objectShape, fill = color)
            writer.append_data(np.array(img))

    writer.close()
```

Listing 3: Programm zur Generierung eines synthetischen Videos

2.3 VideoReader Objekt

Wie bereits in Kapitel 2.1 erwähnt dient der Videoreader dem Handling des Dekodierens eines Videos in einem separaten Thread. Der Videoreader ist hierbei wenig mehr als ein Wrapper um das OpenCV VideoCapture Objekt und einen Buffer in Form einer Queue. Die Konfiguration erfolgt mit einem „Configuration“-Objekt, welches alle einstellbaren Parameter der Applikation enthält, das selbe Objekt wird von allen Komponenten verwendet.

Es existieren zwei Funktionen für das Befüllen des Buffers, die „readFrames“-Funktion und die „readFramesByList“-Funktion. Die erstgenannte Funktion liest alle Frames eines Videos in den Buffer, diese Funktion ist für die Konturenextraktion besonders relevant. Die zuletzt genannte Funktion hingegen ist für den Export des Videos relevant, da hier eine List von Frames aus welchen Bewegungen extrahiert wurden übergeben werden kann. Somit müssen nur relevante Frames dekodiert werden.

2.4 Konturenextraktion

Die Konturenextraktion wurde entsprechend der Beschreibung in Kapitel 1.3.1 implementiert und wurde in Abb. 5 dargestellt.

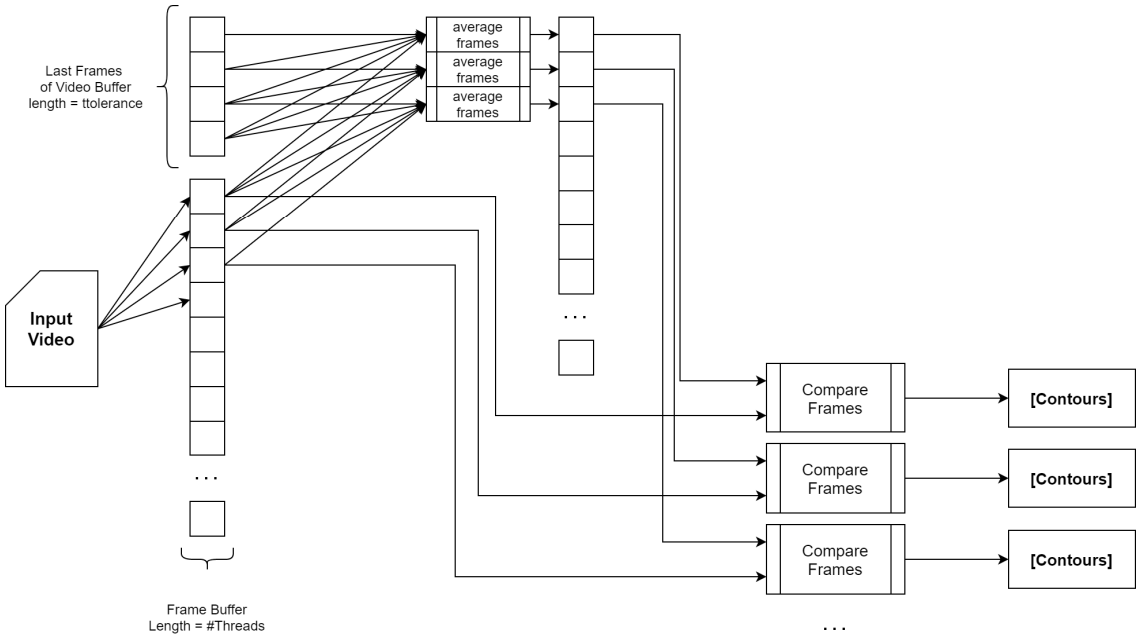

Abb. 5: Daten und Datentransformation während der Konturenextraktion

Um eine höhere Verarbeitungsgeschwindigkeit zu erzielen wurde diese Komponente parallelisiert. Um Konturen extrahieren zu können muss für jedes Bild ein Vergleichbild gewählt oder erstellt werden. In diesem Fall wird der Durchschnitt der

vorherigen 10 Bilder gebildet und der Unterschied zu diesem berechnet. Die Durchschnittsbildung mit der „average frame“-Funktion läuft hierbei parallel zu der Funktion „compare frames“, obwohl das Durchschnittsbild natürlich eine Vorbedingung für die Berechnung des Durchschnittes ist. Um eine lose Kopplung zu ermöglichen kommt hier ein weiterer Buffer zum Einsatz. Die „compare frames“-Funktion wartet solange bis beide nötigen Bilder in den jeweiligen Buffern vorliegen, um die Unterschiede zu berechnen und die Konturen zu extrahieren. Die Implementierung dessen kann im folgenden Listing eingesehen werden.

```
    def getContours(self, data):
        frameCount, frame = data
        # wait for the reference frame,
        # which is calculated by averaging some previous frames
        while frameCount not in self.averages:
            time.sleep(0.1)
        firstFrame = self.averages.pop(frameCount, None)

        gray = self.prepareFrame(frame)
        frameDelta = cv2.absdiff(gray, firstFrame)
        thresh = cv2.threshold(frameDelta, threashold, 255, cv2.THRESH_BINARY)[1]
        thresh = cv2.dilate(thresh, None, iterations=10)
        cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
                                cv2.CHAIN_APPROX_SIMPLE)
        contours = []
        masks = []
        for c in cnts:
            ca = cv2.contourArea(c)
            (x, y, w, h) = cv2.boundingRect(c)
            if ca < self.min_area or ca > self.max_area:
                continue
            contours.append((x, y, w, h))

        if len(contours) != 0 and contours is not None:
            self.extractedContours[frameCount] = contours
```

Listing 4: Konturenextraktion in Python

Die Funktion „prepareFrame()“ in Zeile 9 des Listing bündelt die Vorverarbeitungsschritte, die für jeden Frame nötig sind. Das sind die Herunterskalierung, das Konvertieren in ein Graustufenbild und die Anwendung des Gaußschen Weichzeichners. In Kapitel 1.3.1 entspricht dies den Schritten 2 – 4.

Als Algorithmus für die Konturenfindung wird „`cv2.CHAIN_APPROX_SIMPLE`“ genutzt, dieser wird empfohlen, da er sehr Speichereffizient ist, indem nicht alle Randpunkte sondern nur relevante Eckpunkte gespeichert werden.

2.5 Ebenenaggregation

Die Ebenenaggregation kann in zwei Stufen unterteilt werden, die Ebenenerzeugung und das Ebenenmangement. Die Ebenenerzeugung ist selbsterklärend. Das Ebenenmanagement bündelt Funktionen zum Filtern und Klassifizieren der Ebenen. Die Ebenenerzeugung könnte in das Ebenenmanagement integriert werden, darauf wurde aber verzichtet, um die einzelnen Objekte handhabbar zu halten.

2.5.1 Ebenenerzeugung

Die Ebenenerzeugung wurde entsprechend des Algorithmus in Kapitel 1.3.2 implementiert, die Implementierung kann in der Datei „LayerFactory.py“ eingesehen werden.

2.5.2 Ebenenmanagement

Das Layermanagement wird durch das LayerManager-Objekt implementiert. Dieses Objekt wendet eine Reihe von Meta-Filtern auf die Layer an und kann anschließend eine Reihe von Classifiern auf diese Layer Anwenden. Die Classifier die angewandt werden können werden in Kapitel 2.6 erläutert.

Es können Filter für die maximale und minimale Layer-Längen festgelegt werden, die Idee hierbei ist das eine „Bewegung“ die mehr als z. B.10 Minuten wahrscheinlich nicht relevant, sondern nur ein wandernder Schatten oder eine Hauptstraße mit ständiger Bewegung. Ähnliches gilt für sehr kurze Layer, ein Layer mit einer Länge einer halben Sekunde wird wahrscheinlich keine relevante Bewegung sein. Diese Längen können angepasst werden, um den Vorstellungen des Nutzers bzw. dem Einsatzzweck zu entsprechen. Zusätzlich werden alle dünn besetzten Layer gelöscht. Als dünn besetzt wird ein Layer bezeichnet, wenn mindestens 20% der Konturen Null sind, da bei der Ebenenerzeugung eine „Time Tolerance“ angewandt wird kann es dazu kommen, dass ein Teil der Konturen einer Ebene Leer ist. Aus der Beobachtung während der Entwicklung geht hervor, das dies oft bei Laubbäumen vorkommt, da die Bewegung der Blätter durch den Wind sporadisch aufgenommen wird.

Durch die angewandten Filteroperationen kann die Anzahl der Layer erheblich reduziert werden, wodurch die rechenintensiven Klassifizierungsoperation um ein Vielfaches beschleunigt werden können.

2.6 Classifier

Wie in dem ersten Kapitel erläutert können mehrere Classifier nacheinander oder parallel auf die Layer angewandt werden, durch die Kombination des Outputs der einzelnen Classifier ist es möglich sehr präzise Filter zu erstellen. Um dies zu ermöglichen wurde ein Interface geschaffen, welches von allen Classifiern implementiert werden muss. Das die Implementierung wird als Black-Box gehandhabt.

Im Rahmen dieser Arbeit wurde zwei NN basierte Ansätze untersucht. Ein CPU basierter Ansatz unter Zuhilfenahme von OpenCV mit einem YOLOv4 Netz und ein GPU basierter Ansatz, welcher mit Tensorflow implementiert wurde und auf einem unspezifizierten NN basiert, welches auf dem COCO Datensatz trainiert wurde.

2.6.1 Interface

Das Classifier-Interface wurde so genannt, um Verwirrung beim Leser zu vermeiden. Python besitzt und benötigt keine Interfaces, stattdessen können „Abstract Base Classes“ verwendet werden. Diese sind den Java Interfaces funktional sehr ähnlich.

```
from abc import ABC, abstractmethod

class ClassifierInterface(ABC):
    @abstractmethod
    def tagLayer(self, imgs):
        """takes filled contours of one frame, returns list (len(), same
           as input) of lists with tags for corresponding contours"""
        pass
```

Listing 5: Classifier Interface

Dieses Interface sollte implementiert werden, da es aber sehr simpel ist, ist dies nicht zwingend nötig, solange der implementierte Classifier eine Funktion „tagLayer()“ besitzt.

2.6.2 OpenCV

Die OpenCV Implementierung ist deutlich simpler und einfacher in der Installation der Abhängigkeiten als die Tensorflow Implementierung.

2.6.3 Tensorflow

2.7 Export

Das Exporter-Objekt besitzt 3 Funktionen für den Export von Daten. Eine zum Exportieren der „Raw“-Daten und zwei zum Exportieren von Videos. Die extrahierten Konturen, Layer und Klassifikationen können separat exportiert werden, somit ist es möglich die Konturenextraktion, Layeraggregation, Klassifikation oder Videoexport unabhängig voneinander durchzuführen, wo sie andernfalls voneinander abhängig wären.

Die Layer können entweder nacheinander oder überlagert exportiert werden, bei vielen extrahierten Layern kommt es schnell dazu, das einige Layer von anderen überlagert werden, hier könnte eine Verzögerung eingefügt werden, so dass die Reihenfolge beibehalten wird, es aber nicht mehr zu einer Überlagerung kommt. Aus Zeitgründen wurde hierauf verzichtet. Alternativ können die Layer nacheinander exportiert werden, so ist zu jedem Zeitpunkt nur ein Layer sichtbar. Bei Videos mit mehreren Gleichzeitigen Bewegungen kann es hier aber dazu kommen das die „Zusammenfassung“ länger ist als das originale Video. Hier ist auf die Zielorientierung zu achten, sollen generelle Trends festgestellt werden oder einige Zeitbereich besonders genau analysiert werden?

```
    def exportRawData(self, layers, contours, masks):
        with open(self.config["importPath"], "wb+") as file:
            pickle.dump((layers, contours, masks), file)

    def importRawData(self):
        print("Loading previous results")
        with open(self.path, "rb") as file:
            layers, contours, masks = pickle.load(file)
        return (layers, contours, masks)
```

Listing 6: RAW Import- und Export-Funktion

Ex- und Import der „Raw“-Daten erfolgt mit der Python Bibliothek „Pickle“. Mit Pickle ist es möglich beliebige Python Objekte als Binärdatei zu speichern und später wieder einzulesen. In Listing 6 können Im- und Export Implementierung eingesehen werden. Im Programmcode liegen diese Funktion im Impoerter- bzw. Exporter-Objekt.

Die Implementierung des Videoexports ähnelt sich in beiden Funktionen, der Unterschied liegt hauptsächlich in dem Objekt über das iteriert wird. In beiden Fällen muss das Originalvideo mithilfe des VideoReader-Objektes eingelesen werden. Die Konturen der Ebenen müssen aus diesem Video ausgeschnitten und in das zu exportierende Video eingefügt werden. Wobei hier auf die Konvertierung der Koordinaten für die Eckpunkte zu achten ist, da die Extraktion der Konturen auf herunterskalierten Bildern erfolgte. Die Koordinaten der Eckpunkte müssen mit einem Faktor multipliziert werden, welcher sich durch das Teilen der Breite des Originalvideos durch die Breite des herunterskalierten Videos ergibt.

Werden die Layer nacheinander Exportiert, ist der Algorithmus relativ einfach. Pro Layer wird eine Liste von Frames erstellt, die aus dem Originalvideo ausgelesen werden müssen, diese wird anschließend an den VideoReader übergeben. Aus den eingelesenen Frames werden die Konturen des Layers ausgeschnitten und anschließend in ein Bild eingefügt, diese Bilder werden mit der ImageIO (FFMPEG) Bibliothek zu einem Video zusammengefügt und gespeichert.

Werden die Layer überlagert exportiert, ist der Algorithmus ähnlich. Allerdings wird damit begonnen die Länge des Längsten Video zu ermitteln, anschließend wird ein Array mit dieser Anzahl Elementen erstellt. Anschließend wird über jedes Layer iteriert und die Konturen des jeweiligen Layers an Index X werden im Export-Array ebenfalls an Index X eingefügt. Das so erstellte Array wird anschließend zu einem Video zusammengefügt.

Der Code für diese Funktionen ist zu umfangreich, um ihn hier einzufügen, er kann aber in „Exporter.py“ eingesehen werden.

3 Auswertung

In diesem Kapitel sind Benchmarks und Beobachtungen zu finden, welche den Lesefluss an anderer Stelle negativ beeinflusst hätten.

3.1 Benchmarks

Die Implementierung und die Konturenextraktion im Besonderen, ist stark parallelisiert. Es wurden eine Reihe von Konfigurationen erprobt, um eine gute Verteilung der Ressourcen auf Funktionen zu finden.

Alle Versuche wurden mit einem 1080p 30 FPS Video auf einem Computer mit einer Ryzen 3700X 8 Kern 16 Thread CPU und 32 Gb 3200 MHz RAM durchgeführt.

In Abb. 6 sind die Ergebnisse zweier Versuchsreihen zu finden. Es wurde ein Stacked-Bar-Plot der benötigten Zeiten pro Komponente angefertigt. Mit jeder Messung wurde das Verhältnis der Threads für die Konturen Extraktion variiert. Während der Konturen Extraktion laufen zwei parallele Prozesse parallel. Sowohl die Durchschnittsbildung als auch der Bildvergleich sind parallelisiert. Im unteren Plot in Abb. 6 wurde die Anzahl der Threads für die Durchschnittsbildung erhöht, wobei die Anzahl der Threads für den Bildvergleich bei eins blieb. Im oberen Plot wurde die Anzahl der Threads für die Durchschnittsbildung auf 15 gesetzt und die Anzahl der Threads für den Bildvergleich in Zweierschritten erhöht.

Aus diesen Versuchen geht hervor das die Anzahl der Physischen und logischen Kerne gut ausgenutzt werden kann, wobei die Ergebnissteigerung bei Erhöhung der Thread Anzahl über die Anzahl der Physischen Kerne hinaus abnimmt. Ebenso zeigt sich das die Durchschnittsbildung den höheren Rechenaufwand erzeugt und eine Optimierung dieser Funktion scheinbar den größten Effekt hätte.

Es wird auch deutlich das 2 Threads ausreichend scheinen um ein Video schneller als in Echtzeit verarbeiten zu können. Da 600 Sekunden des Videos in ca. 520 Sekunden verarbeitet werden können, wenn je ein Thread pro Funktion reserviert wird.

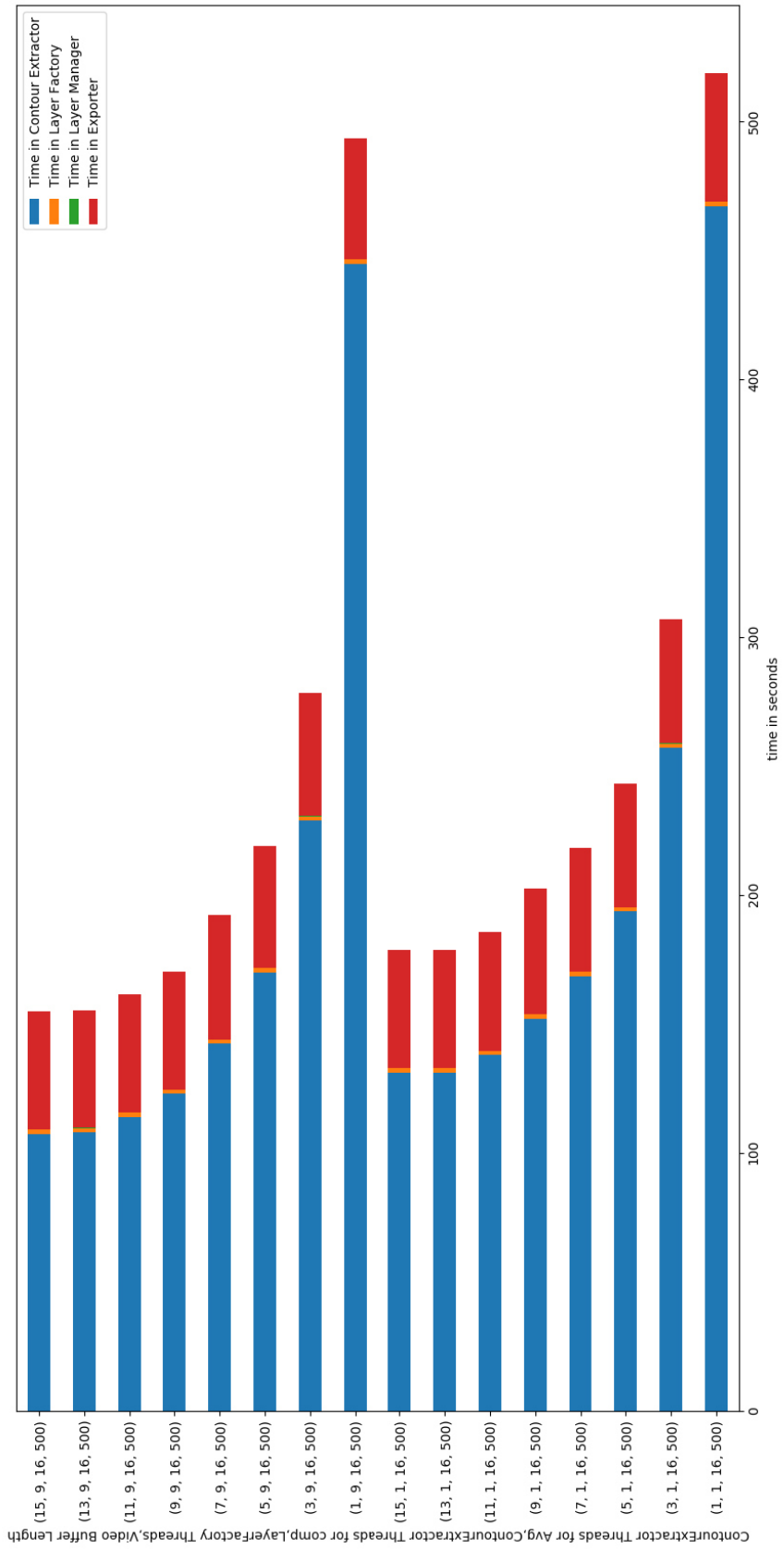

Abb. 6: Benchmark Ergebnisse

3.2 Beobachtungen

Zu Beginn dieses Projektes wurden Versuche mit verschiedenen Farbräumen durchgeführt. Zu diesem Zeitpunkt war das Vergleichsbild des Konturenextraktors nicht ein Durchschnitt der vorangegangenen Bilder, sondern das erste Bild des Videos. Natürlich war bekannt das dieses Vorgehen nur bei kurzen Videos Erfolg bringen kann, da langfristige Änderungen, wie Lichtveränderungen und Schattenbewegung für zu große Änderungen sorgen würde. Die Auswirkungen der Schattenbewegung waren größer als erwartet, daher wurden andere Farbräume erforscht, die einen eigenen Kanal für Helligkeitswerte vorsehen. Hierbei wurde festgestellt, dass der Farbunterschied durch unterschiedliche Farbtemperaturen im direkten Sonnenlicht und im Schatten groß genug waren, um die positiven Effekte des neuen Farbraumes zu negieren. Der Farbunterschied in den verbleibenden Kanälen war ähnlich groß wie der Helligkeitsunterschied bei einem einfachen Graustufen Bild. Somit brachte die Konvertierung in einen neuen Farbraum keine feststellbaren Vorteile, erhöhte aber den rechnerischen Aufwand, da zwei statt einem Kanal genutzt wurden. Im speziellen wurden die Farbräume „LAB“ und „YUV“ untersucht.

Theoretisch ist eine Durchschnittsbildung vorheriger Bilder im Konturenextraktor nicht nötig, wenn im Vorhinein bereits bekannt wäre in welchen Zeitbereichen besonders viele Bewegungen stattfinden. Als Vergleichsbild könnte in vielen Fällen (ausgenommen stundenlange Bewegungen) ein Bild unmittelbar vor dem Beginn dieser Änderungen genutzt werden. Hierfür könnte eine Analyzer Komponente eingeführt werden, die diese Analyze vor der Verarbeitung des Videos durchführt. Leider ist die Dekodierung des Videos relativ Zeitintensiv. Es wurde festgestellt, dass die Durchschnittsbildung nur wenig länger dauert, aber deutlich fehlertoleranter ist. Besonders nützlich ist das die Durchschnittsbildung gut mit starkem Bildrauschen und ähnlichen Artefakten, wie Blätterbewegungen an einem Baum, umgehen kann.

4 Zusammenfassung

Videozusammenfassung ist eine Technologie mit vielfältiger Anwendung besonders in der Videoüberwachung. In dieser Arbeit wurde ein CPU basierter Ansatz beschreiben und implementiert. Dieser Ansatz unterscheidet sich entscheidend von dem von BriefCam und Prof. Shmuel Peleg patentierten Ansatz (PELEG, 2012) und stellt somit einen neuen Stand der Technik dar.

Durch den modularen Entwurf der Implementierung sollte es relativ einfach sein Komponenten auszutauschen und für neue Einsatzzwecke anzupassen, sollte die breite Konfiguration nicht ausreichen. Dank der durchgeführten Benchmarks können Rückschlüsse auf die zu erwartende Geschwindigkeit auf bestehender Hardware gemacht werden.

Die Klassifikation der Events ist nicht ideal und könnte in der Zukunft verbessert werden, indem geeignetere Neuronale Netze oder einfachere und mehr Klassifikatoren eingesetzt werden. Ebenso wäre ein Mechanismus zum Filtern und selektiven Sichten der Layer sinnvoll.

Insgesamt kann dieses Projekt als Erfolg gewertet werden.

5 Literaturverzeichnis

PELEG, S. (August 2012). *https://www.cs.huji.ac.il*. Von https://www.cs.huji.ac.il/~peleg/patents/EP1955205.pdf abgerufen

6 Abbildungsverzeichnis

7 Listingverzeichnis