

# **Optimierung von Trainingsdaten für semantische Segmentierung**

Christopher Pasda

April 7, 2021

## **Contents**

<b>1 Einleitung</b>	<b>2</b>
<b>2 Projekt SE Perception</b>	<b>3</b>
2.1 Problemstellung . . . . .	5
<b>3 Analyse gängiger Labeltools</b>	<b>6</b>
3.1 CVAT (Computer Vision Annotation Tool) . . . . .	6
3.2 Labelme . . . . .	9
3.3 Makesense.ai . . . . .	11
3.4 Matlab Image Labeler . . . . .	12
<b>4 Programm "LabelTool"</b>	<b>13</b>
4.1 Grundlagenforschung . . . . .	13
4.1.1 Die projektive Ebene . . . . .	13
4.1.2 Berechnung der Hilfslinie . . . . .	14
4.1.3 Der Catmull-Rom Spline . . . . .	15
4.2 Analyse des Annotation Tool's von Herrn Mario Hoffmann . . . . .	15
4.2.1 Informelle Beschreibung und Komplexitätsanalyse . . . . .	15
4.3 Requirements . . . . .	19
4.4 Use Cases . . . . .	19
4.5 Flussdiagramm . . . . .	19
4.6 Implementierungen . . . . .	19
4.7 Ergebnisse und Qualitätskontrolle . . . . .	28
<b>5 Fazit</b>	<b>28</b>

# **1 Einleitung**

Für das Forschungsprojekt von Herr Prof. Dr.-Ing. Carsten Thomas, in Zusammenarbeit mit der Firma BOSCH, zum Thema Schienenerkennung für automatisiertes Fahren unter der Verwendung eines neuronalen Netzwerkes, soll der Prozess der Erstellung von Trainingsdaten optimiert werden, damit auch eigene Trainingsdaten erstellt werden können. Dies ist notwendig, da existierende Datensätze nicht für kommerzielle Zwecke verwendet werden können. Ziel der Arbeit ist die Entwicklung eines Prototyps, welcher den Prozess des Labelns<sup>1</sup> eines Bildes optimiert. Dazu werden im ersten Teil verschiedene Tools analysiert und dazu eine systematische Übersicht der notwendigen Requirements für die Entwicklung eines eigenen Programms erstellt. Darauf aufbauend wird das "Annotation Tool" von Herrn Mario Hoffmann analysiert und weitere Requirements aus Use Cases abgeleitet. Im zweiten Teil wird das Programm, basierend auf einer Idee von Herr Thomas, vorgestellt und diskutiert. Abschließend folgt eine Qualitätskontrolle und das Fazit der Arbeit.

WAS IST SEMANTISCHE GEMENTIERUNG?!?!

---

<sup>1</sup>Labeln beschreibt den Prozess, bei dem jeder Klasse eines Bildes verschiedene Farbcodes zugeordnet werden, um sie so unterscheiden zu können.

## 2 Projekt SE Perception

MEhr Infos zu Bosch und dem neuen Schienending

Die vorliegende Arbeit wurde im Zusammenhang des Projektes SE Perception zwischen der Hochschule für Technik und Wirtschaft Berlin und der Firma BOSCH realisiert. In diesem Projekt wird ein Convolutional Neural Network dazu verwendet automatisch Schienen erkennen zu können. Um ein solches Netzwerk effektiv nutzen zu können, muss dieses vorab trainiert werden. Dazu werden Beispielbilder, welche eine Abbildung der jeweiligen Situation die erkannt werden soll darstellen, entsprechend markiert und daraus Ground Truth Daten<sup>2</sup> erstellt. Dies sind uint8 Bilder, welche den jeweiligen Klasseninteger in einem Pixel gespeichert haben. Daraus entsteht ein Grayscale-Bild, welches ein Neuronales Netzwerk verstehen kann. Darauf wird im späteren Verlauf näher eingegangen.

Um solche Ground Truth Daten erstellen zu können, muss ein Bild vorab gelabelt werden. Dazu werden interessante Bereiche (ROI = region of interest) pixelweise in der jeweiligen Klassenfarbe markiert.



Figure 1: Beispiel Pixelmaske

---

<sup>2</sup>Ground Truth eine gebräuchliche Terminologie, die in verschiedenen Bereichen häufig verwendet wird, um sich grundsätzlich auf jede Art von Information zu beziehen, die durch direkte Beobachtung bereitgestellt wird

Danach werden diese Bilder in grayscale umgewandelt und die markierten Pixel mit der jeweiligen Klassen-ID befüllt. So entsteht aus einem RGB mit 3 Bytes in jedem Pixel (rot, grün, blau) ein Bild mit nur einem Byte pro Pixel.



Figure 2: Beispiel Ground Truth

Mit diesen Trainingdsaten kann nun ein Netzwerk trainiert werden. Im Ergebnis kann durch eine "Heatmap" gezeigt werden, mit welcher Wahrscheinlichkeit das Neuronale Netzwerk die Schienen erkennt. Dabei wird durch ein ROS-Knoten eine Daten-Verbindung zwischen einem Video und dem Neuronalen Netzwerk geknüpft, wodurch die Algorithmen anhand des Bild-Inputs die Schienen erkennen können.

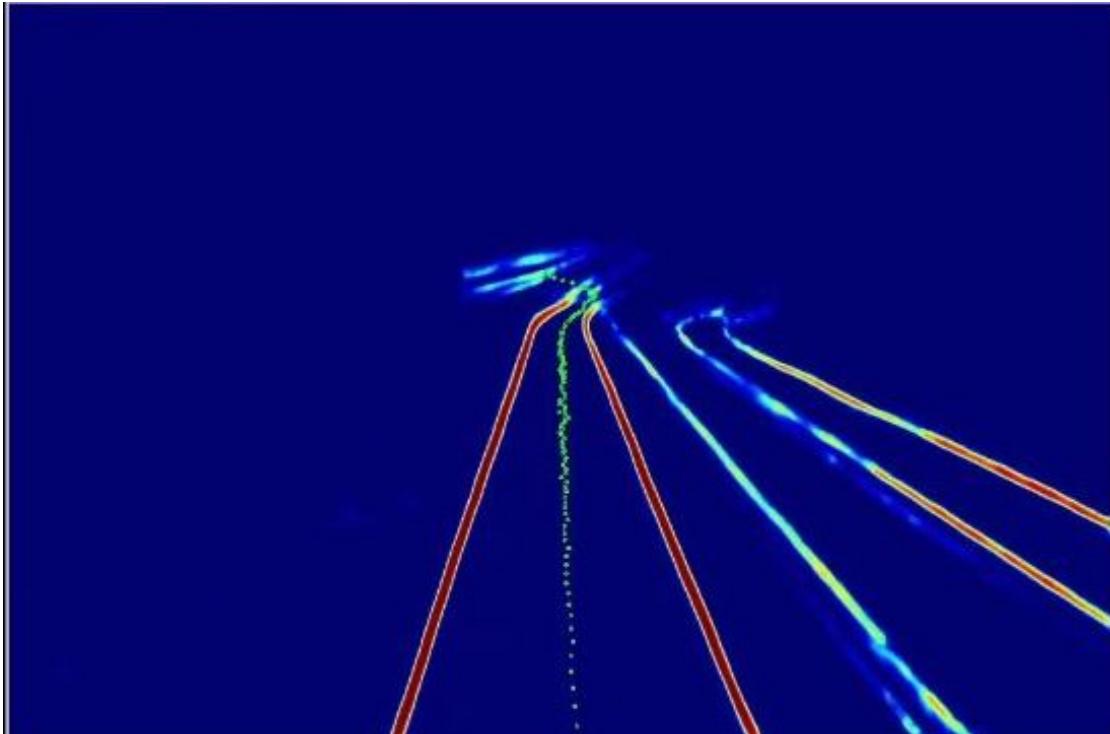


Figure 3: Wahrscheinlich der Erkennung von Schienen durch das Netzwerk

Die Güte der Erkennung hängt dabei direkt von der Anzahl und Qualität der Trainingsdaten ab. Der momentan verwendete Datensatz "Railsem19" umfasst 8500 Bilder und äquivalent viele Labels. Davon werden circa 70% für das Training und 30% der Bilder für das Testen des Netzwerkes benötigt. Es wird deutlich, dass eine große Anzahl an Bildern für ein Training gebraucht wird.

## 2.1 Problemstellung

Dadurch, dass der "Railsem19" Datensatz nicht für kommerzielle Zwecke genutzt werden darf und der allgemeine Aufwand ein Bild händisch zu labeln sehr groß ist, entstand die Notwendigkeit nach einem Programm, welches den Prozess des Labelns weitestgehend automatisiert. Dazu wurden vorab gängige Labeltools auf etwaige unterstützende Funktionen und allgemeine Funktionalitäten untersucht.

Eventuell Bilder labeln und zeit messen ?!

Projektive Ebene beschreiben doch wo?

IDEE VON HERRN THOMAS aufzeigen <https://www.uniserv.com/unternehmen/blog/detail/article/gr-truth-ohne-datenqualitaet-kein-machine-learning/>

## **3 Analyse gängiger Labeltools**

In dieser Analyse werden vier gängige Labeltools auf ihre Funktionalitäten und Features in einer Pro- und Contra-Liste analysiert, was sowohl der Ableitung von Requirements, als auch der Übersicht über verschiedene automatisierende Features dient. Am Ende dieses Kapitels wird ein Fazit gezogen, welches dieser Programme sich für eine Erweiterung eignen würde, oder ob ein eigener Ansatz gewählt wird.

### **3.1 CVAT (Computer Vision Annotation Tool)**

**NOCH LIZENZEN NENEN** Bei CVAT handelt es sich um ein Webapp von Intel, welche am 19.06.2018 für den PC veröffentlicht wurde. Besonders daran ist, dass neben dem Label-Tool auch ein Projektplaner verfügbar ist, bei dem verschiedene Jobs verteilt werden können. So lassen sich Label-Projekte sehr gut überblicken und strukturieren.

#### **Pro:**

- einfach zu installieren und skalieren - läuft in einer webapp (auf einem Docker)
- gute Zusammenarbeit in Teams möglich durch Tasks
- gute Übersicht über Labels
- Werkzeuge funktionieren sehr gut und ohne Probleme
- verschiedenste Outputmöglichkeiten

#### **Contra:**

- einen Task bereitzustellen erfordert Probieren
- nicht sehr intuitiv am Anfang da sehr umfangreich
- WebApp läuft nur auf Chrome
- AI-Tool nicht zu gebrauchen

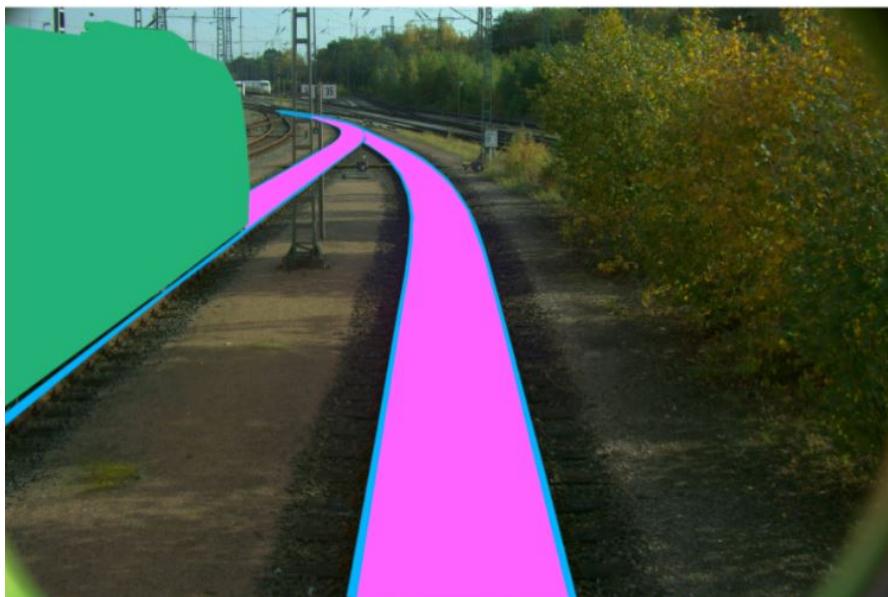


Figure 4: Gelabeltes Bild in CVAT, Aufwand: ca. 20 Minuten

CVAT bietet dem User eine Reihe an AI-Tools, welche unter dem Strich für das Labeln von Schienen leider nicht zu gebrauchen sind. Es gibt sogenannte "Interactors", welche eine äußere Umrandung von Objekten semiautomatisch erkennen sollen. Dabei werden vier Punkte gesetzt und Objekte gelabelt, welche sich in diesem Viereck befinden. Leider funktioniert diese Erkennung bei Schienen schlecht, da das Bild rund um die Schienen zu viele verschiedene Strukturen zeigt. Da dies häufig bei Schienen auftritt, kann die Verwendung von "Interactors" ausgeschlossen werden.

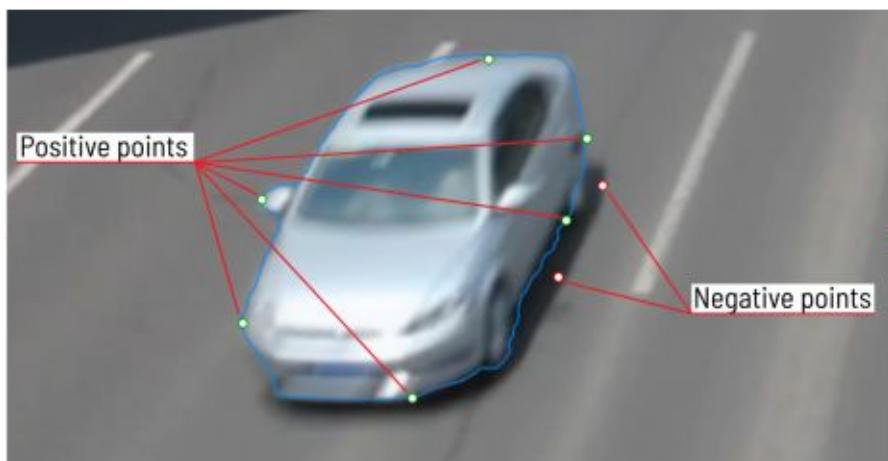


Figure 5: Interactor in CVAT Beispiel



Figure 6: Interactor in CVAT Schiene

Eine weitere Möglichkeit stellen "Detectors" dar. Das sind "supported DL models", welche automatisch aus einem Frame generiert werden. Diese eignen sich aber nur für die vorgefertigten Label-Kategorien, in denen sich leider bisweilen keine Schienen finden lassen. Die letzte Kategorier der AI-Tool stellen die "OpenCV tools" dar. Darunter findet sich eine "intelligent Scissor", welche ähnlich funktioniert wie die "Interactors". Dabei werden rund um das Objekt Punkte gesetzt und das Tool zeichnet die Umrandung automatisch. Die Erkennung des Randes funktioniert hier deutlich besser, jedoch müssen immernoch viele Punkte gesetzt werden, da der Rand nur in einer begrenzten Entfernung vom letzten Punkt erkannt werden kann. Demnach bietet diese Unterstützung kaum Zeitersparnis, da fast gleich viele Punkte, wie beim normalen Labeln gesetzt werden müssen und es trotzdem Schwächen bei der Erkennung der Umrandung gibt.

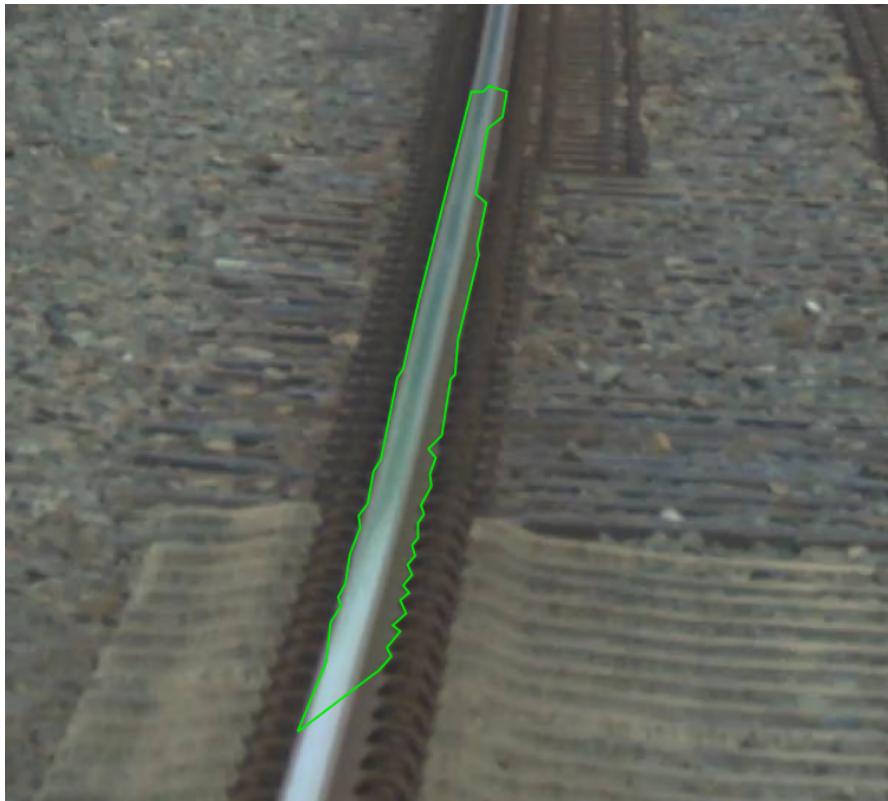


Figure 7: intelligent Scissors in CVAT

Zusammenfassend ist zu sagen, dass CVAT ein performantes gutes Programm darstellt, welches jedoch keine sinnvollen AI-Tools für das Labeln von Schienen mitbringt.

Github: <https://github.com/openvinotoolkit/cvat/>

### 3.2 Labelme

Labelme ist ein 2018 veröffentlichtes grafisches open-source Annotation-Tool von Kentaro Wada, welches in Python geschrieben wurde. Das Programm setzt eine Anaconda-Installation auf dem Computer voraus und kann über die Shell bedient werden. Das besondere daran ist, dass sich das Programm auch über Python-Code steuern lässt, sodass einfache Scripts geschrieben werden können, um Trainingsdaten zu erzeugen und zu konvertieren. Es bietet eine Vielzahl an verschiedenen Output-Formaten für die Ground Truth-Daten an.

#### Pro:

- github open source auf Python basierend
- Trainings Daten können direkt verschieden konvertiert werden

- einfaches Interface mit gut strukturierten Inhalten
- viele verschiedene Formate für Labels (z.B: COCO JSON, Pascal VOC XML, YOLO Keras TXT)
- Funktionen können über Prompt aufgerufen werden (programmierbarkeit)

**Contra:**

- Installation etwas umständlich
- Bildbearbeitungsmöglichkeiten sehr langsam
- Anaconda-Installation wird benötigt
- keine AI-Tools zur Unterstützung

Auf den ersten Blick scheint Labelme ein Programm zu sein, welches sich für die Zwecke des unterstützten Labelns erweitern lassen könnte, jedoch biete es so keine Vorteile gegenüber dem händischen Labeln mit CVAT. Der Prozess des Labelns eines Bildes ist für einen ungeübten Benutzer auch hier recht zeitaufwändig. Der Aufwand belief sich auf ca. 15 Minuten pro Bild, wobei es stark davon abhängt, wie viele Schienen gelabelt werden müssen.

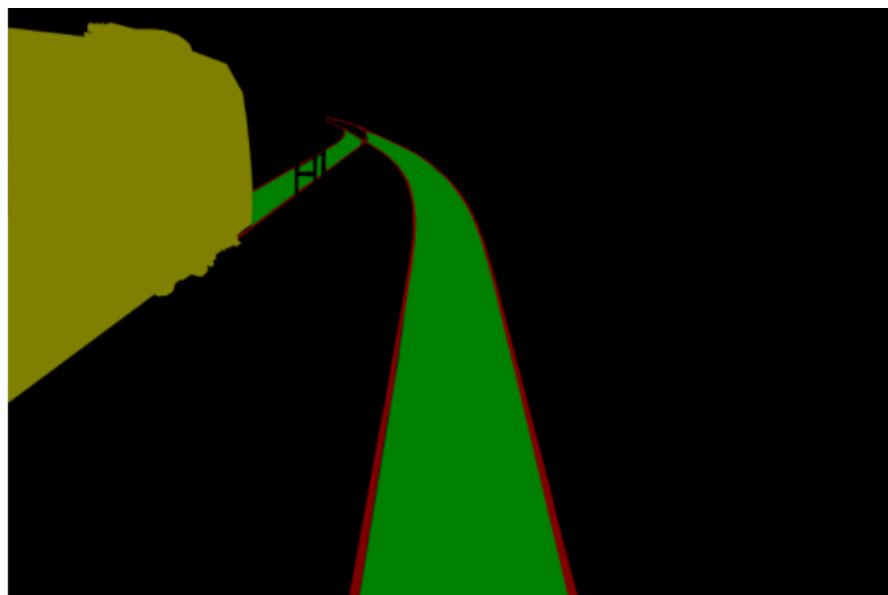


Figure 8: Gelabeltes Bild mit Labelme

Github: <https://github.com/wkentaro/labelme>

### 3.3 Makesense.ai

Makesense.ai ist eine WebApp, welche unter <https://makesense.ai> erreichbar ist oder über die Shell installiert werden kann. Es wurde von Piotr Skalski geschrieben und untersteht der "GNU General Public License". Das Tool bietet verschiedene AI-Funktionalitäten und Automatisierungen von repetitiven Aufgaben. Dabei verspricht es, dass es mit dem Single Shot MultiBox Detector, welcher ein "single deep neural network" darstellt und mit dem COCO Datensatz<sup>3</sup> trainiert wurde, Label im Bild zu erkennen und vorzuschlagen.

#### Pro:

- WebApp
- gutes übersichtliches Interface
- für sehr einfaches Labeling (rectangles) gut zu gebrauchen
- gute Übersicht über die Labels

#### Contra:

- schlechte Bedienweise
- Bewegung beim Setzen der Polygone sehr hakelig
- Labels können nur in VGG Json exportiert werden
- keine direkte Erstellung von Trainingsdaten

Auch hier können die AI-Tools für die genannten Zwecke nicht überzeugen. Zwar funktioniert die Erkennung mit dem COCO SSD, jedoch nur für Bilder, welche den trainierten Kategorien unterstehen. So kann ein Bild von einem Hund erkannt werden. Jedoch wird das Label nur mit einem Rechteck markiert, was für das Labeln der Schienen ungeeignet ist. In einem Beispielbild mit Schienen konnten keine Label vom Programm erkannt werden. Das lässt darauf schließen, dass das Model nur bestimmte Kategorien von Labels erkennen kann. Ein großer Vorteil ist, dass Label direkt über das COCO JSON-Format importiert werden können. Diese Eigenschaft wird sich zu einem späteren Zeitpunkt noch positiv auf das Projekt auswirken.

---

<sup>3</sup>Common Objects in Context (COCO)

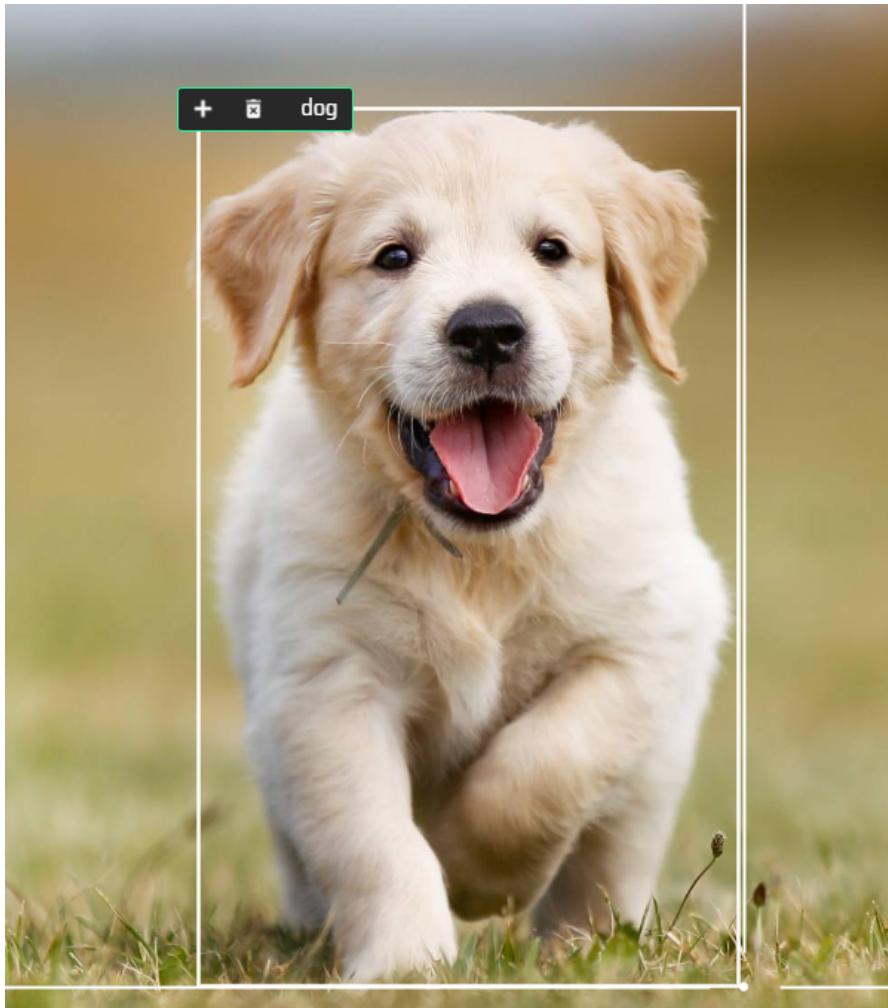


Figure 9: Erkanntes Bild vom SSD Model

Github: <https://github.com/SkalskiP/make-sense>

### 3.4 Matlab Image Labeler

Der Matlab Image Labeler ist eine App des Programms Matlab. Auch dieses Tool setzt auf AI-Unterstützung. So gibt es "Smart-Polygon" und eine "Assisted Freehand" Funktion. Sonst unterscheidet sich das Programm nicht von den anderen gängigen Labeltools.

**Pro:**

- Installation durch Matlab
- Bewegung im Bild problemlos
- intuitives Interface mit vielen Möglichkeiten

- gute Übersicht über die Labels

**Contra:**

- Sehr Ressourcenlastig und teilweise langsam
- AI-Tools funktionieren nicht entsprechend
- Import nur in .Mat-Format
- Export der Daten nur in Ground Truth

## 4 Programm "LabelTool"

### 4.1 Grundlagenforschung

#### 4.1.1 Die projektive Ebene

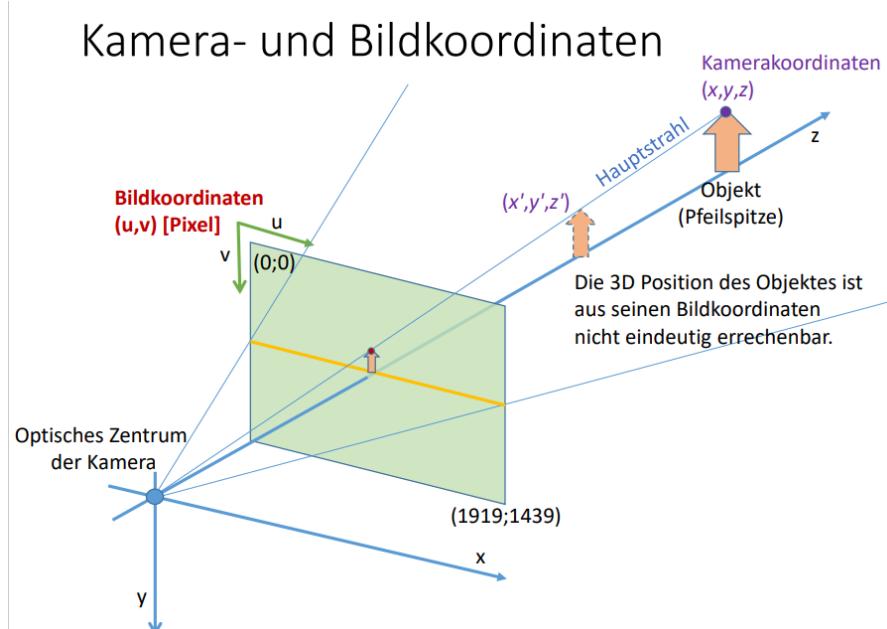


Figure 10: Kamera- und Bildkoordinaten

Die projektive Ebene findet ihren Ursprung in der Malerei der italienischen Renaissance. Dort versuchte man die Realität, insbesondere die Architektur, korrekt in Gemälden widerzuspiegeln. Dabei ging es vor allem um projektive Effekte im unendlichen, wie der Schnitt zweier parallel verlaufender Geraden im imaginären Fluchtpunkt unter projektiver Abbildung auf eine Ebene. Die Grundlage projektiver Räume ist durch homogene Koordinaten gegeben, deren Haupteigenschaft die Invarianz (Unveränderlichkeit) gegenüber Skalierungen ist. Diese werden durch  $(n+1)$ -dimensionale Vektoren beschrieben.

Die projektiven Punkte eines Raumes können sich als Strahlen vorgestellt werden, welche durch den Ursprung eines 3D-Koordinatensystems verlaufen. Alle auf dem Strahl befindlichen Punkte sind mit einer Multiplikation eines geeignet Skalierungsfaktors ineinander überführbar und somit äquivalent. Dabei existieren für parallel zur jeweils gewählten Ebene verlaufende Strahlen keine Repräsentanten. Im Sonderfall der parallel zur xy-Ebene gelegenen Ebene  $E : z = 1$  ergeben sich alle Repräsentanten aus den geschnittenen Strahlen durch Skalierung der dritten Vektorkomponenten auf 1. Diese Arbeit liegt dem Modell der Lochkamera zugrunde, wobei das Projektionszentrum der Kamera im Ursprung eines 3D-Kamerakoordinatensystems liegt (siehe Abbildung 1). Die optische Achse fällt mit der z-Achse des Koordinatensystems zusammen. Im Abstand der Bildweite( focal length)  $f \geq 0$  vom optischen zentrum befindet sich die Bildebene, auf die die 3D-Welt zweidimensional abgebildet wird ( $z = 1$ ). Die Bildebene ist dabei stets parallel zur x-y-Ebene des 3D-Koordinatensystems. Der Hauptpunkt an dem die optische Achse die Bildebene durchstößt, liegt demnach an Position  $(px, py, 1)$  auf der Ebene. Die Projektion eines beliebigen 3D-Punktes(Objekt) des Kamerakoordinatensystems auf die Bildebene ergibt sich in diesem Modell als Schnittpunkt des Richtungsstrahls vom optischen Kamerazentrum zum Punkt der Kamerakoordinaten( $x, y, z$ ) mit der Bildebene. Dabei ist noch unberücksichtigt geblieben, dass die beiden Koordinatenachsen der Bildebene in der Regel jeweils unterschiedlich und darüber hinaus unabhängig vom 3D-Koordinatensystem der Kamera skaliert sind. Um die Bildkoordinaten in Pixelkoordinaten korrekt umrechnen zu können, müssen somit bei der Abbildung in x- und y-Richtung jeweils spezifische Skalierungsfaktoren berücksichtigt werden. Durch das Wissen um die Kameraparamter (Brennweite, focal lenght) der entsprechenden Kameramatrix und der Kameraposition(Translations- und Rotationsmatrix) können die Weltkoordinaten (entsprechen Größen der realen Welt) in Bildkoordinaten (Pixel) umgerechnet werden. Dazu wird hintereinander eine Transformation von Welt  $\rightarrow$  Kamera und Kamera  $\rightarrow$  Bild durchgeführt.

#### FORMEL HIER

Das bedeutet, die Bildkoordinaten  $p_{img}$  entsprechen einer Multiplikation der Kameramatrix  $K$ , mit der Rotationsmatrix  $R-t$  und den Weltkoordinaten  $p_{world}$ . Bei allen Variablen handelt es sich um 3d-Vektoren. So können im Umkehrschluss auch Bild- zu Weltkoordinaten problemlos umgerechnet werden.

#### 4.1.2 Berechnung der Hilfslinie

Der exakte Abstand zwischen den beiden Endpunkten des Ar-Track wird berechnet, indem der Mittelpunkt (x-y-Koordinaten) des Mauszeigers an der jeweiligen Position genommen und in Weltkoordinaten umgerechnet wird. In der echten Welt verändert sich die Breite des Ar-tracks nicht, weshalb wir diesen einfach mit einem Spaltenvektor mit 3 Dimensionen nach links (linke Schiene) und nach rechts (rechte Schiene) verschieben können. Danach werden diese Punkte wieder in Bildkoordinaten umgerechnet, wodurch wir wieder die perspektivische Sicht erhalten. In allen Berechnungen wird angenommen, dass wir geradeaus schauen und nichts im Wege ist. Damit sind die Dimensionen nicht

mehr variabel und eine Umrechnung ist möglich.

#### 4.1.3 Der Catmull-Rom Spline

Da die Endpunkte der Hilfslinie auf die jeweils linke und rechte Schiene Abbilden, können diese Punkte dazu genutzt werden einen Catmull-Rom Spline zu zeichnen, welcher den Verlauf der Schiene markiert. Ein Catmull-Rom Spline ist eine stückweise definierte polynomiale Funktion mit mindestens vier Punkten. Nach dem Setzen des vierten Punktes, werden Punkte 1 und 4 als Stützpunkte herangezogen und zwischen Punkten 2 und 3 ein Spline gezogen. Durch die Stützpunkte kann der Verlauf relativ genau interpoliert werden. Möchten wir einen Spline durch  $k$  Punkte ziehen, dann brauchen wir insgesamt  $k+2$  Stützpunkte. Das Programm speichert die Punkte auf den Schienen, während der Benutzer die "Centerpoints" setzt und zeichnet ab dem vierten Punkt den ersten Spline auf der Schiene.

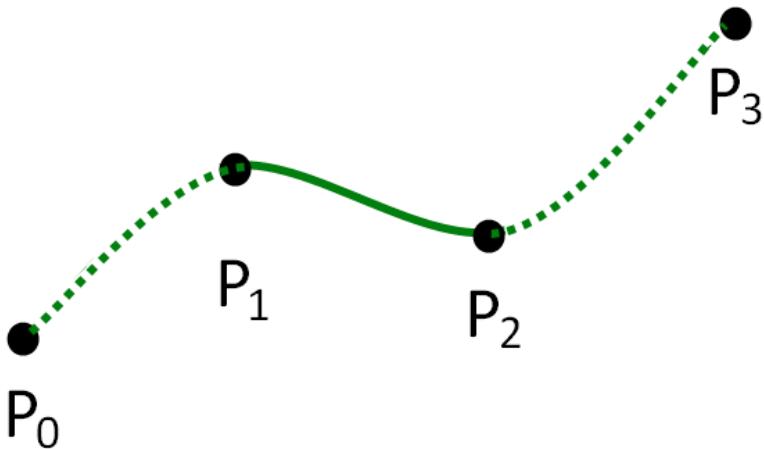


Figure 11: Catmull-Rom Spline

### 4.2 Analyse des Annotation Tool's von Herrn Mario Hoffmann

Das vorliegende "Annotation Tool" von Herrn Mario Hoffmann wurde im Rahmen des Projekts SE Perception zum qualitativen Labeln von Mittelpunkten einer Schiene erstellt. Dazu sollten immer gleich viele Mittelpunkte in einem festen Abstand auf einem Bild markiert werden. Diese Daten wurden dafür verwendet ein Neuronales Netzwerk auf die Erkennung von Mittelpunkten zu trainieren.

#### 4.2.1 Informelle Beschreibung und Komplexitätsanalyse

##### Installation:

- OpenCV, Version, bevorzugt 3.4.7 und gcc Toolchain installieren

- git clonen
- In das Verzeichnis wechseln: cd Videoplayer
- Build directory erstellen und in Dieses wechseln: mkdir -p build cd build
- CMake konfigurieren und Anwendung kompilieren: cmake .. make

Das Programm "Annotation Tool" bietet dem Nutzer die Möglichkeit einen Ordner von Bildern und eine Kamera-Matrix einzulesen und dann realitätsgetreu durch Abbildung der projektiven Ebene Mittelpunkte von Schienen markieren zu können. Das Programm wurde in C++ unter Verwendung der OpenCV Bibliothek realisiert. Der grobe Ablauf kann folgendermaßen dargestellt werden:

1. der User startet das Programm über den Terminal (Linux) mit folgendem Befehl:  
./videoplayer -label -yaml="Pfad zur yaml-Datei" "Pfad zu den Bildern"
2. das Programm öffnet sich und zeichnet das Bild samt Hilfslinie
3. der User wählt zwischen "x", "y" und "c" für Mittelschiene, linke Schiene oder rechte Schiene
4. der User bewegt den Mauszeiger über die Schiene und versucht mit der Hilfslinie die Mitte der Schiene abzuschätzen
5. der User setzt mit "d" entlang der Equidistant (gleicher Abstand) Linien die Punkte in der Mitte der Schiene
6. der User speichert mit "s" die Mittelpunkte in einer yaml-Datei
7. der User wechselt mit "n" zum nächsten Bild
8. der User beendet mit "q" das Programm

Diese Funktionalitäten wurden durch verschiedene Bibliotheken realisiert, welche vorher installiert werden müssen. Verwendete Bibliotheken:

- OpenCV, Version 3.4.7 Algorithmen für die Bildverarbeitung und Computer Vision
- nlohmann JSON für Modern C++

Neben diesen Bibliotheken wurden noch weitere Headerdateien eingebunden, um Funktionalitäten aus dem Videoplayer und die Berechnungen der Matritzen nutzen zu können:

- "Videoplayer.h"
- "Camera.h"

Das Programm besteht im Ausgangszustand aus einer Header-File, in der die Datentypen und Funktionen deklariert werden und einer .cpp-Datei, in der die Logik implementiert wurde. Es besteht aus 425 Zeilen Code und eine Fehlerbehandlung ist nur rudimentär vorhanden. Zum speichern der Punkte in Vektoren wird ein Struct verwendet, welches den Namen "LabelImage" trägt. Das Programm liest aus der Command-Zeile per Commandline-Parser die Eingabebefehle. Beim Ausführen wird zunächst die Labelmanual angezeigt. Danach werden die angegebenen Bilder eingelesen und zu jedem Bild ein "LabelImage" erstellt. Nach dem Einlesen startet das Labeltool und liest die Kamera-Matrix und die yaml-Datei für eventuell existierende Punkte ein. Die Hauptfunktion? stellt eine while-Schleife dar, welche mit jedem Durchlauf das eingelesene Bild in ein neues Mat-Objekt kopiert, damit die Hilfslinie immer auf einem neuen Bild gezeichnet wird und somit der Effekt entsteht, dass es sich dabei um eine dynamisch-bewegbare Linie handelt. Weiterhin wird in dieser Schleife der User-Input verwertet und die Punkte gezeichnet.

Das Programm kann in folgenden Aufbau aufgegliedert werden:

- Filehandling - Einlesen der Bilder, yaml-Dateien und Kamera-Matrix
- Inputhandling - Verwertung von Usereingaben
- Punkte speichern und löschen - Speichern der Punkte in Vektoren des Structs "LabelImage"
- Hilfslinie anzeigen - zeichnen der Hilfslinie am Punkt der Maus
- Berechnung der Equidistant-Linien - exakte Abstände zwischen Horizont und ersten Punkt auf der z-Achse
- Output generieren - schreiben der yaml-Datei mit OpenCV Filestorage

**Filehandling:** Das Filehandling übernehmen die Funktionen:

- void readImageList(const path, lblList)
- readJson(lblImages)
- Camera(yamlFileName) aus "Camera.h" Include

Die Funktion readImageList(const path, lblList) bekommt als Input einen Pfad und einen Vektor in dem die "LabelImages" gespeichert werden. Danach wird mit einem Directory-Iterator über die Bilder im Pfad iteriert, jedes "LabelImage" nach dem jeweiligen Bild benannt und danach das "LabelImage" auf dem Vektor gespeichert. Die readJson(lblList) bekommt als Input die Liste der gespeicherten "LabelImages" und sucht in dem gleichen Directory nach einer Datei, welche den selben Namen aber als Dateiendung ".yaml" trägt. Danach werden mit der OpenCV Filestorage Klasse die Punkte aus der yaml-Datei in den Vektoren der "LabelImages" gespeichert. Die Camera(yamlFileName)

Funktion bekommt lediglich den Pfad der yaml-Datei und lädt diese Kamera-Matrix durch den Constructor unter Verwendung der OpenCV Filestorage Klasse.

**Inputhandling:** Das Inputhandling wird durch eine einfach Switch-Case-Struktur realisiert.

**Punkte speichern und löschen:** Diese Funktionalitäten wurden durch folgende Funktionen realisiert:

- void savePoint(Point , LabelImg, equidistantY)
- void removeClosestPoint(LabelImg)

Zum Speichern eines Punktes wird der savePoint(Point , LabelImg, equidistantY) Funktion die momentane Mausposition als OpenCV Datentyp Point<sup>4</sup>, das jeweilige "LabelImage" und die Equidistant-Punkte übergeben. Beim Speichern wird abgefragt, wie weit entfernt der Cursor sich von den Equidistant-Linien befindet und sollten dies 10 Pixel sein, dann wird der Punkt automatisch auf dieser Linie gesetzt und auf den Vektor Equidistantpoints des "LabelImage" gespeichert. Danach wird mit einem Switch-Case überprüft, welche Schiene angewählt ist und entsprechend der Punkt auf dem Vektor gespeichert. Gelöscht werden die Punkte durch eine Abfrage in einem Switch-Case, welche Schiene angewählt ist. Danach wird ein Pointer auf den Adressspeicher des jeweiligen Vektors initialisiert. Sollte der momentane Mauspunkt näher als einem bestimmten Schwellenwert sein, dann wird dieser Adressspeicher wieder freigegeben.

**Hilfslinie anzeigen:** Diese Funktionalitäten wurden durch folgende Funktionen realisiert:

- void drawArTrack(img, LabelImg)

In dieser Funktion werden die momentanen Koordinaten des Mauszeigers genommen und von Pixel- in Weltkoordinaten umgerechnet(Beschrieben in Kapitel: Grundlagenforschung). Dieser Punkt wird dann um jeweils  $(-1435/2)^5$  nach links und  $(1435/2)$  nach rechts verschoben. Dort werden diese Punkte wieder in Pixelkoordinaten umgerechnet, sodass durch die Open CV circle() und line() Funktionen die Hilfslinie gezeichnet werden können. Da in der while-Schleife bei jedem Durchlauf auf einem neuen Bild gezeichnet wird, entsteht der Effekt einer sich frei bewegenden Linie.

**Berechnung der Equidistant-Linien :** Diese Funktionalitäten wurden durch folgende Funktionen realisiert:

- void getEquidistantY(image)

---

<sup>4</sup>Template Klasse für 2D-Punkte mit x- und y-Koordinate

<sup>5</sup>1435mm ist der exakt genormte Abstand zwischen der linken und rechten Schiene

Diese Funktion errechnet anhand eines gegebenen Horizonts und dem ersten sich im Bild befindlichen Punktes auf der z-Achse(Beschrieben in Grundlagenforschung) die Equidistant-Linien. Dafür wird zunächst eine z-Achse definiert, welche den ersten im Bild sichtbaren Punkt auf der z-Achse in Weltkoordinaten umgerechnet. Danach werden mit einer Forschleife und einem in Weltkoordinaten festgelgten Horizonts und einer festen Anzahl an Linien y-Punkte errechnet.

```
for (int i = 0; i <= numLines; ++i ){
    y = horizontPoint - (i * ((horizontPoint - z)/numLines));
    Point3d HWorld = Point3d(0, 0, y);
    horizonLine = kam.world2pixel(HWorld);
    eq.push_back(horizonLine.y);
}
```

**Output generieren:** Diese Funktionalitäten wurden durch folgende Funktionen realisiert:

- void writeJson(lblList)

Diese Funktion bekommt die Liste mit den "LabelImages" und iteriert über diese mit einer Forschleife. Danach wird für jedes "LabelImage" ein Open CV Filestorage geöffnet und die Punkte werden in die Datei geschrieben.

### 4.3 Requirements

### 4.4 Use Cases

### 4.5 Flussdiagramm

### 4.6 Implementierungen

In diesem Abschnitt sollen die eigenen Implementierungen im Kontext der abgeleiteten Anforderungen beschrieben.

#### Filehandling:

Anforderung LH	Anforderung PH	Name
LH 1.1	PH 2.3	bool readJson(vector LabelImg)
LH 2	PH 3	bool readJson(vector LabelImg)
LH 2	PH 3	void drawPolygons(Mat img, vector labels, bool filled)
LH 2	PH 3	void drawPolygon(Mat img, Polygonpoints, bool filled, Scalar color)
LH 2	PH 3	void createCatMullRomPolygon(Mat img, LabelImg, bool filled)
LH 2	PH 3	void DrawCatmullRomSegment()

## **PH 2.3 Das System muss Schienenpunktinformationen einlesen können :**

Um diese Funktion zu realisieren wurden folgende Funktionen implementiert:

- bool readJson(vector LabelImg)

readJson(vector LabelImg)):

Die Funktion zum Einlesen der yaml-Datei für die Punkte auf den Schienen bekommt den Vektor mit den jeweiligen eingelesenen "LabelImages" übergeben. Es wurde eine Abfrage beim Start des Programm implementiert, welche die Auswertung einer Boolean beinhaltet, ob eine yaml-Datei mit Schienenpunkten gelesen wurde. Sollte dies nicht der Fall sein, dann wird eine Mittelpunkt-Datei eingelesen damit garantiert werden kann, dass die alten existenten Mittelpunkt-Datei genutzt werden können. Da die Funktion zum Einlesen vorhanden war, wird hier nur auf die neue Funktion zum Einlesen der Schienenpunkte eingegangen. Dazu wurde die OpenCV Filestorage Klasse verwendet , welche zunächst abfragt, ob die Filestorage geöffnet wurde. Danach wurden Filenodes gesetzt, welche nach den jeweiligen Stichworten "ego track", "left neighbor" und "right neighbor" in der yaml-Datei als Anker haben, um darauf zugreifen zu können. Die yaml-Dateien haben dabei folgendes Format:

```

{"left neighbors":[{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]},
 [{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]},
  [{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]}],
 [{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]}]]}

{"right neighbors":[{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]},
 [{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]},
  [{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]}],
 [{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]}]]}

{"ego track":{"centerpoint":[[x,y],[x1,y1],[x2,y2]],
  "leftrail":[[x,y],[x1,y1],[x2,y2]],
  "rightrail":[[x,y],[x1,y1],[x2,y2]]}}

```

Figure 12: Aufbau yaml-Datei

Es können dabei belieb viele linke und rechte Nachbarn in der Datei vorhanden sein, jedoch nur ein "Ego Track". Über diese Filenodes wird nun mit einer For-Schleife iteriert und ein neuer Filenode gesetzt, welcher in der nächsten Ebene der yaml-Datei auf "ego track", "leftrail" und "rightrail" zugreift. Da es sich bei diesem Filenode um einen normalen Vektor mit den entsprechenden Punkten handelt, kann auch über diesen problemlos iteriert werden (Abbildung XX). In dieser Schleife wird ein neuer Punkt von OpenCV-Datentyp Point erstellt. Da die Punkte in der yaml-Datei hintereinander mit x- und y-Koordinate gespeichert sind, erhöht sich der Schleifeniteratator um  $+2$ . Diese Punkte werden dann auf den entsprechenden Vektor des Structs "label" gespeichert.

### **PH 3 Das System muss aus eingelesenen Informationen Schienenmarkierungen generieren:**

Um diese Funktion zu realisieren wurden folgende Funktionen implementiert:

- bool readJson(vector LabelImg)
- void drawPolygons(Mat img, vector labels, bool filled)
- void drawPolygon(Mat img, Polygonpoints, bool filled, Scalar color)
- void createCatMullRomPolygon(Mat img, LabelImg, bool filled)

- void DrawCatmullRomSegment()

Um aus den eingelesenen Punkten eine Schienenmarkierung generieren zu können, wird in der Funktion `readJson(vector<LabelImg>)` vor dem Einlesen der jeweiligen Schiene ein neues Struct "Label" mit dem Namen der Schiene erstellt. Dieses Struct enthält folgende Attribute:

- int id
- int categoryId
- vector<Point> trackbedPolygon
- vector<Point> rightRailPolygon
- vector<Point> leftRailPolygon
- vector<Point> rCatMullPoint
- vector<Point> lCatMullPoint
- vector<Point> centerPoin
- Scalar color
- Scalar trackbedColor

Die Schienenmittelpunkte, die dazugehörige Farbe und Kategorie-Id werden mit dem einlesen der Punkte in dem Struct gespeichert. Die "Label"-Structs werden in dem neu erstellen Vector "labels" des "LabelImage" gespeichert.

```
FileNode centerpoint = ego[j]["centerpoint"];
lbl = label(id, 1, "ego track", CENTER_COLOR, CENTERTRACKBED_COLOR);
std::cout << "created new Label from yaml: " << lbl.labelName << "id: " << lbl.id << std::endl;
```

Figure 13: Beispiel create label

createCatMullRomPolygon(Mat img, LabelImg, float steps):

Diese Punkte können nun dazu genutzt werden, mit der Funktioin `createCatMullRomPolygon(Mat img, LabelImg, float steps)`, welche das momentane Bild, das Struct "LabelImg" und eine Anzahl an Interpolationsschritten zwischen zwei Punkten übergeben bekommt, die Polygon-Label zu generieren. Dazu wird mit einer For-Schleife über den Vector "Labels" vom "LabelImage" iteriert. Dieser Vector beinhaltet die beim Einlesen erstellten "label"-Structs. Mit einer weiteren For-Schelife bekommt man den Zugriff auf die gespeicherten Schienenmittelpunkte, welche vorab sortiert werden, damit ein dynamisches Einfügen von Punkten möglich gemacht werden kann.

void DrawCatmullRomSegment():

Mit einer For-Schleife wird nun über die entsprechenden Schienenmittelpunkt-Vektoren, iteriert und die Funktion DrawCatmullRomSegment() darauf angewendet. Diese Funktion bekommt das auf dem zu zeichnende Bild, das Struct LabelImage und vier Punkte aus dem Vektor zwischen denen Interpoliert werden soll, mit der Anzahl der jeweiligen Interpolationsschritte übergeben. Diese Funktion berechnet dann ein Catmull-Rom Spline (Beschrieben in Grundlagen) durch die vier Punkte, mit einem Abstand der Interpolationspunkte entsprechend der Anzahl an Steps, welche übergeben wurden. Diese Interpolationspunkte werden dann von Pixel in Weltkoordinaten umgerechnet und auf der x-Achse exakt um  $(67/2)$ mm nach links und nach rechts verschoben. Die 67mm repräsentieren in diesem Fall die im Kapitel "Grundlagen" beschriebene Breite des Schienenkopfes. Als Ergebnis bilden nun die linken und rechten Interpolationspunkte zwischen den vier Catmull-Rom Punkten genau auf den äußeren Rand einer Schiene ab. Abschließend werden diese in Pixelkoordinaten zurückgerechnet und auf Hilfsvektoren des Labelimages gespeichert, sollten diese sich im Bild befinden (Es kann sein, dass Punkte außerhalb des Bildbereiches markiert werden, da die Messlinie in vielen Fällen nicht gerade ist und deshalb teilweise aus dem Bild rausragt). Diese Hilfsvektoren repräsentieren nun den linken und rechten äußeren Rand einer Schiene. Um daraus ein Polygon zu erstellen, müssen diese Punkte nun in richtiger Reihenfolge in einem Vektor gespeichert werden. Da das gewählte Drittanbieter-Tool makesense.ai das COCO Json-Format verwendet, müssen die Punkte der Reihenfolge nach wie sie gezeichnet werden sollen, auch gespeichert werden. Dazu wird der Vektor mit den Punkten des rechten Randes invertiert und mit dem Vektor des linken Randes konkatiniert und in dem entsprechenden Polygon-Vektor des Labels gespeichert.

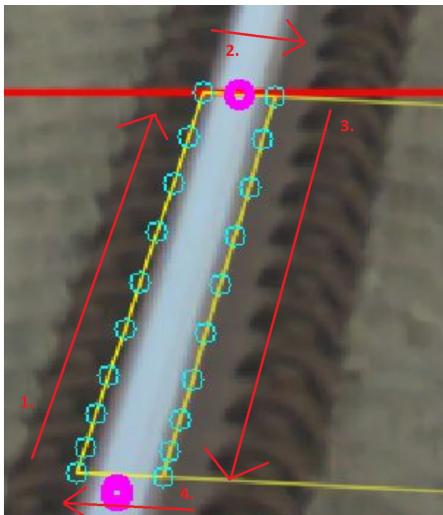


Figure 14: Reihenfolge der Polygonpunkte

drawPolygons(Mat img, vector<labels>, bool filled):

Damit die Label auf dem Bild gezeichnet werden können, wurde die Funktion draw-

Polygons(Mat img, vector labels, bool filled) implementiert. Diese Funktion bekommt das Bild auf dem gezeichnet werden soll, den Vector "labels" des Structs "LabelImage" und eine Boolean übergeben, ob das Polygon gefüllt werden soll oder nicht. In der Funktion wird über die einzelnen Polygon-Vektoren iteriert, welche durch die Funktion createCatmullRomPolygon() generiert wurden. Für jeden dieser Vektoren wird die Funktion drawPolygon(Mat img, vector labels, bool filled, Scalar color) aufgerufen. Diese bekommt das aktuelle Bild, den Vektor der Polygonpunkte, eine Boolean ob das Polygon gefüllt werden soll und eine Farbe für das Label übergeben. In der Funktion Polygonpunkte auf einem Vektor von Vektoren gespeichert und eine Abfrage ausgeführt, ob das Polygon gefüllt werden soll, wenn nicht, dann wird die OpenCV Funktion polylines() aufgerufen, welche die Polygonpunkte miteinander verbindet. Soll das Polygon gefüllt werden, wird die OpenCV-Funktion fillpoly() aufgerufen, welche den Vektor von Vektoren der Polygonpunkte übergeben bekommt.

#### **Punkte speichern und löschen:**

Anforderung LH	Anforderung PH	Name
LH 3	PH 4	void createNewLabel(LabelImg)
LH 3	PH 4.1	void createNewLabel(LabelImg)
LH 3	PH 5	label selectLabel(LabelImg, int lblIdx)
LH 3.1	PH 5.1	void saveCatMullPoint(selectedLabel, LabelImg, img)
LH 3.1	PH 5.1	void removeClosestCatMullPoint(selectedLabel)
LH 3.1	PH 5.1	void deleteLabelFromActiveLabels(LabelImg)
LH 3.1	PH 5.2	void drawGaugeLine(img, LabelImg, label)

#### **PH 4 Das System muss beliebig viele Schienenmarkierungen generieren können:**

createNewLabel(LabelImg lblImg):

Um beliebig viele Nachbar erstellen zu können, wurde die Funktion createNewLabel(LabelImage) implementiert. Diese Funktion vom Typ void bekommt ein LabelImg übergeben. Zuerst wird abgefragt, ob der Vektor labels des LabelImage-Structs leer ist. Sollte dies der Fall sein, dann wird ein neues Label für den "ego track" generiert. Ist der Vektor nicht leer überprüft diese Funktion den momentan Mauspunkt auf dem Bildschirm. Sollte dieser sich links neben dem "ego track" befinden, dann wird ein Label linker Nachbar erstellt und vice versa ein rechter Nachbar. Diese Label werden abschließend auf dem labels-Vektor des Labelimg gespeichert. Damit wurde auch PH 4.1 "Das System muss zwischen rechtem und linkem Nachbar unterscheiden können" erfüllt.

#### **PH 5 Das System muss zwischen den jeweiligen Schienenmarkierungen wechseln können**

selectLabel(LabelImg lblImg, int lblIdx)):

Damit der Benutzer die Möglichkeit hat auszuwählen, welche Schiene er markieren möchte, wurde die Funktion `selectLabel(LabelImg lblImg, int lblIdx)` implementiert. Die Funktion vom Typ `label` bekommt das `LabelImage` und die globale Variable `lblIdx`, welches den momentanen Index des ausgewählten Labels darstellt, übergeben. Dieser Index wird über den Input der Zahlen "1-9" vom User gesteuert. Zuerst findet eine Abfrage statt, ob der Vektor `labels` vom `LabelImage` leer ist. Ist dies nicht der Fall, dann wird überprüft, ob der Index kleiner als die momentan Größe des Vektors `labels` ist. Ist dies der Fall, den gibt die Funktion das `label` mit dem jeweiligen Index wieder. Andernfalls wird ausgegeben, dass dieses Label nicht existiert. Ist der Vektor `labels` leer, dann wird ein neues `label` erstellt und dieses zurückgegeben.

#### **PH 5.1 Das System muss die angewählten Schienenmarkierungen verändern können:**

`saveCatMullPoint(label lbl, LabelImg lblImg, Mat img):`

Diese Funktion soll die Punkte auf der Mitte der linken und rechten Schiene speichern. Dazu bekommt sie das aktuell ausgewählte Label, das `LabelImage` und das Bild übergeben, auf dem momentan gezeichnet wird. Zunächst wird abgefragt, ob der User das Mittelpunkt-Labeling oder das Seiten-fokussierte-Labeling nutzt. Danach wird entsprechend beim Mittelpunkt-Labeling der jeweilige Mittelpunkt in Weltkoordinaten umgerechnet und um  $(1502/2)$ mm nach links und rechts auf der x-Achse verschoben. 1502mm entstehen daraus, dass man die Spurweite von 1435 mit der Schienenbreite addiert. (Mittelpunkt Schiene =  $67\text{mm}/2$ , für beide Schienen  $(67\text{mm}/2)^*2$ ). Diese Punkte werden wieder in Pixelkoordinaten umgerechnet und in dem entsprechenden Vektor der jeweiligen Schiene gespeichert. Ist das Seiten-fokussierte-Labeling aktiv, dann wird zunächst die momentane Mausposition als linker Punkt auf der Schiene gesehen und von Pixel- in Weltkoordinaten umgewandelt. Danach wird dieser um 1502mm auf der x-Achse verschoben, um auf die andere Schiene abzubilden. Ist die globale Variable "switch sides" wahr, dann wird der rechte Punkt fokussiert. Dieses vorgehen ist notwendig, da die Messlinie oft nicht komplett gerade ist (Roll-Winkel) und deshalb können einseitig nicht alle Punkte gesetzt werden. Sollte die ersten Punkte für ein neues Schienenlabel gesetzt werden, dann wird der selbe Punkt um 1 auf der y-Achse verschoben wiederholt gespeichert, damit nur zwei weitere Punkte für den Catmull-Rom Spline gesetzt werden müssen, bis dieser gezeichnet wird.

`removeClosestCatMullPoint(label lbl):`

Diese Funktion bekommt das ausgewählte `label` übergeben. Zunächst werden Pointer auf den Adressspeicher der Vektoren allokiert, aus denen entsprechende Punkte gelöscht werden soll. Wenn der jeweilige Vektor nicht leer ist, dann wird geprüft welcher Punkte auf der linken oder rechten Schiene dem Mauszeiger am nächsten ist. Danach wird der Index erfasst und sollte der Mauszeiger weniger als 100 Pixel vom nächsten Punkt im Vektor sein, dann wird der Adressspeicher freigegeben. Anhand des Indexes wird danach der Punkt auf der anderen Seite gelöscht und am Ende der Mittelpunkt.

void deleteLabelFromActiveLabels(LabelImg lblImg):

Der Funktion wird das LabelImage übergeben. Zunächst wird über den globalen Vektor "activeRails" iteriert, welcher die aktiven Labels als Tupel mit der jeweiligen id hält. Wenn der Mauszeiger weniger als 20 Pixel von der x- und y-Koordinate des oberen Punktes der Linie, welche für das Label mit der jeweiligen Farbe steht, entfernt ist, dann wird das Label mit der korrespondierenden id aus dem Vektor labels des LabelImages gelöscht.

### **PH 5.2 Das System muss die Länge der Messlinie anpassen können:**

Hier reinschreiben jo

#### **Output generieren:**

Anforderung LH	Anfoderung PH	Name
LH 4	PH 6	bool readJson(vector LabelImage)
LH 5	PH 7	void writeJson(LabelImg)
LH 6	PH 8.1	void createPixelMask(LabelImg , Label)
LH 6	PH 8.2	void createGroundTruth(LabelImg, gTruth)
LH 6	PH 8.3	void writeJsonPolygon(LabelImg, img)

### **LH 6 Das System muss prüfen, ob Segmentierungsinformationen vorhanden sind:**

Um die Abfrage nach vorhandenen Segmentierungsinformationen gewährleisten zu können, wurde Funktion readJson(vector LabelImage) ein Rückgabewert vom Typ Boolean gegeben, welcher True ist, wenn Segmentierungsinformationen vorhanden sind und gelesen werden können.

### **PH 7 Das System muss vorhandene Markierungen von Schienen- und Mittelpunkten wiederholen:**

writeJson(LabelImg lblImg):

Diese Funktion schreibt die Vektordaten der Catmull-Rom-Punkte der Schienen und Mittelpunkte in eine yaml-Datei, welche, wie in PH 3 beschrieben wurde, wieder eingelesen werden kann. Es wird die nlohmann-Bibliothek verwendet, welche den Datentyp json bereistellt. Damit können Vektoren mit diesem Datentyp erstellt werden, welche das entsprechende Format berücksichtigen. Zunächst prüft die Funktion, ob ein Ordner mit dem namen "yaml-files" vorhanden ist. Wenn nicht, dann wird dieser erstellt. Danach wird über die Vektoren der labels des LabelImages iteriert und einzelne Punkte im json Format auf den Vektoren gespeichert. Es wird für jede der Schienen(linker Nachbar, rechter Nachbar und Ego Track) ein weiteres Json-Objekt erstellt, welches die jeweiligen Json-Vektoren der Schienen zugewiesen bekommt. So kann interativ eine Struktur erstellt werden. Es wird jede äußere Klammer einer Json-Datein als ein Json-Objekt

gesehen, welches weitere Vektoren des json-Typen beinhalten kann.

### **PH 8.1 Das System muss eine Pixelmaske generieren können:**

void createPixelMask(LabelImg , Label):

Diesse Funktion ist simpel gehalten und fragt lediglich ab, ob der Ordner "Pixelmask" im Verzeichnis vorhanden ist und wenn nicht, dann wird dieser erstellt. Danach wird das Label, welches vom Typ Mat ist und vom Programm neben dem Bild erstellt wurde, um darauf die Markierungen zu zeichnen, einfach unter dem jeweiligen Namen des Bildes in dem Ordner gespeichert.

### **PH 8.2 Das System muss aus der Pixelmaske Ground Truth Daten erstellen können:**

void createGroundTruth(LabelImg, gTruth):

Dieser Funktion wird das Label (hier gTruth genant) übergeben und das jeweilige LabelImage. Auch diese FUNKtion überprüft, ob sich ein Ordner mit dem Namen "gTruth" im Verzeichnis befindet und wenn nicht, dann wird dieser erstellt. Danach wird das übergebene Label mit den Markierungen in grayscale umgewandelt und damit in das uint8-Format, wobei jedes Pixel nur einen anstatt 3 Byte Informationen trägt. Die jeweiligen Farben der Label(Schiene und Schienenbett) wurden in grayscale analysiert und festgehalten. Danach wird über die Reihen und Zeilen des grayscale-Bildes iteriert und abgefragt, ob ein Pixel den jeweiligen Integer für die korrespondierende Farbe enthält. Sollte dies der Fall sein, dann wird das Pixel mit dem dem neuen Klasseninteger befüllt, welcher in diesem Fall für die Klasse steht und somit dem neuronalen Netzwerk verständlich wird. Für die neuen Klasseninteger würde der Railsem 19-Datensatz erweitert:

- linker Nachber Klasseninteger: 20
- Mittelschiene Klasseninteger: 21
- rechter Nachber Klasseninteger: 22

### **PH 8.3 Das System muss eine Vektorpolygon-Datei zur Verwendung bei Drittanbietern generieren:**

writeJsonPolygon(LabelImg lblImg, Mat image):

Diese Funktion wurde auf der selben Grundidee, wie in PH 7 beschrieben, aufgebaut. Es werden Json-Objekte benutzt und das jeweilige Format in eine Datein speichern zu können, nur dass in diesem Fall die Polygonpunkte der Labels in den jeweiligen Datein gespeichert werden. Hierzu wurde das COCO Dataset Format gewählt, um bei dem Drittanbieter makesense.ai die Label importieren zu können. Das COCO Dataset Format zeigt sich als besonders einfach und schnell zu erweitern. Der Grudbaufbau gestaltet sich folgendermaßen:

```
{
    "info": {...},
    "licenses": [...],
    "images": [...],
    "categories": [...],
    "annotations": [...],
}
```

Unter Info werden die grundlegenden "high-level" Informationen über das jeweilig Dataset definiert. Darunter fallen der Name, das Datum und andere beschreibende Parameter. Licences beinhaltet eine Liste von Bilder-Lizenzen, welche eventuell verwendet wurden. Unter Images wird eine Liste mit allen Bildern angelegt, welche verwendet wurden um daraus Label zu erstellen. Dabei wird jedes Bild mit dem Namen, der Größe, der id und anderen Informationen beschrieben. Categories beinhaltet eine Liste der jeweiligen Label-Kategorien, wobei es sich in unserem Fall lediglich um Schiene und Schienenbett handelt. Interessant wird es bei den annotations. Hier werden die wichtigen Informationen für die jeweilige Markierung im Bild gespeichert. Dabei handelt es sich um eine Liste der jeweiligen individuellen Markierungen des Datasets, wobei hintereinander x und y-Koordinaten gespeichert werden. Jede dieser Markierungen hält zu dem Informationen über die korrespondierende Kategorie, dem dazugehörigen Bild, eine eigene id und eine Abfrage, ob die Markierungen sich überdecken. Beispiel:

```
"annotations": [
{
    "segmentation": [[510.66,423.01,511.72,420.03,...,510.45,423.01]],
    "area": 702.1057499999998,
    "iscrowd": 0,
    "image_id": 289343,
    "bbox": [473.07,395.93,38.65,28.67],
    "category_id": 18,
    "id": 1768
},]
```

## 4.7 Ergebnisse und Qualitätskontrolle

## 5 Fazit