DHBW
Duale Hochschule
Baden-Württemberg

Individuelle Verkehrsoptimierung durch Analyse von Verkehrskameras mittels Machine Learning

Studienarbeit

Für die Prüfung zum
Bachelor of Science

Studiengang Informatik
Studienrichtung Angewandte Informatik
Duale Hochschule Baden-Württemberg Karlsruhe

Von
Marco Herglotz (3538705), Simon Knab (2751614), Jonas Kümmerlin (5702574)

Abgabedatum:	14.05.2018
Kurs:	TINF15B4
Betreuer:	Dr. Christian Bomhardt

Eidesstattliche Erklärung

Erklärung gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2015. Wir versichern hiermit, dass wir diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen als Hilfsmittel verwendet haben.

Karlsruhe, den 13. Mai 2018

Marco Herglotz (3538705), Simon Knab (2751614), Jonas Kümmerlin (5702574)

Copyright-Vermerk

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Der Ursprung dieser Studienarbeit liegt in einem Problem unseres Betreuers Dr. Christian Bomhardt. Auf seinem täglichen Arbeitsweg nach Ettlingen steht er regelmäßig im Stau auf der A5. Alternative Fahrtrouten wären vorhanden, allerdings warnt der Verkehrsfunk häufig nur unzuverlässig, beziehungsweise mit zu großer Verzögerung über aufkommende oder bereits existierende Staus. Wie wir in Kapitel 3.4.2 feststellen konnte, ist dieser Eindruck unseres Betreuers durchaus gerechtfertigt.

Das Ziel ist es nun, eine Anwendung zu entwickeln, welche zuverlässiger und vor allem früher als der Verkehrsfunk über etwaige Staus auf der Autobahn warnt. Die Anwendung soll dabei auf einem Mobilgerät installierbar sein und den Benutzer rechtzeitig, das heißt spätestens vor der letzten Ausweichmöglichkeit, informieren. Da das Smartphone während der Fahrt nicht verwendet werden darf, muss die App den Benutzer per Sprachausgabe über die aktuelle Verkehrssituation an der nächsten Ausfahrt unterrichten.

1.2 Lösungsidee

Um schneller als der Verkehrsfunk auf Staus reagieren zu können, werten wir die Kameras der Straßenverkehrszentrale Baden-Württemberg aus. Die Webcams sind dabei meistens an Autobahnausfahrten angebracht und liefern alle 1 bis 5 Minuten ein Bild der Autobahn. Dabei sind pro Ausfahrt immer zwei Kameras angebracht, um Bilder der Straße in beide Fahrtrichtungen zu liefern.

Mittels Machine-Learning-Algorithmen entwickeln wir einen Klassifikator, welcher anhand der Webcam-Bilder überprüft, ob an der Ausfahrt aktuell Stau ist. Das Ergebnis dieser Klassifikation soll dem Benutzer des mobilen Endgeräts verfügbar gemacht werden. Die Bildanalyse wird hierfür auf einem Backend-Server kontinuierlich ausgeführt, um sicherzustellen, dass immer die aktuellsten Bilder und Ergebnisse zur Verfügung stehen.

Eine von uns entwickelte Android-App überwacht dann die Position des Benutzers, ruft die aktuellen Daten vom Backend-Server ab und warnt den Benutzer mittels Sprachausgabe rechtzeitig vor erkannten Staus. Dabei soll die Anwendung nicht dauerhaft aktiv sein, um den Akku des Smartphones nicht unnötig zu belasten.

2 Machine Learning

Was ist eigentlich maschinelles Lernen? Ein Computerprogramm lernt maschinell, wenn mit zunehmender Erfahrung die gemessene Fehlerrate an einer gegebenen Aufgabe sinkt. [Mitchell et al., 1997, S.2]

Das maschinelle Lernen ist ein weites Feld. In dieser Studienarbeit beschränken wir uns auf die Bildklassifizierung durch neuronale Netze, deren theoretische Grundlagen in diesem Kapitel erläutert werden.

2.1 Neuronale Netze

Künstliche Neuronale Netze (Artificial Neural Networks, kurz ANN) stellen ein generelles, pratikables Verfahren dar, um reellwertige, diskretwertige und vektorwertige Funktionen aus vorgegebenen Beispielen zu erlernen [Mitchell et al., 1997, S.81]. Künstliche neuronale Netze sind der Arbeitsweise des menschlichen Gehirns nachempfunden [Demuth et al., 2014, S.1-1].

2.1.1 Biologische Inspiration

Ein Gehirn in der Biologie ist aufgebaut aus vielen miteinander verbundenen Neuronen. Beim menschlichen Gehirn sind das etwa 10^{11} Neuronen, von denen jedes mit ca. 10^4 anderen Neuronen verbunden ist. Ein menschliches Gehirn ist sehr leistungsfähig: Obwohl jedes Neuron ca. 0,001s zum Schalten braucht, kann es eine andere Person optisch in nur 0,1s identifizieren [Mitchell et al., 1997, S.82] [Demuth et al., 2014, S.1-8].

Ein biologisches Neuron besteht aus

- einem Zellkörper, in dem die Signalverarbeitung stattfindet
- den *Dendriten*, die elektrische Impulse empfangen und in Richtung Zellkern weiterleiten
- einem *Axon*, das den verarbeiteten Impuls vom Zellkörper an andere Neuronen weiterleitet.

Die Verbindungsstelle zwischen dem Axon und den Dendriten einer anderen Zelle nennt man *Synapse*. Ein biologisches Gehirn lernt, indem Verbindungen zwischen den Neuronen neu erstellt oder verändert werden [Demuth et al., 2014, S.1-8].

Wie die Signalverarbeitung in einem biologischen Neuron funktioniert ist im Rahmen dieser Arbeit zu komplex und auch nicht relevant genug, um hier weiter behandelt zu werden.

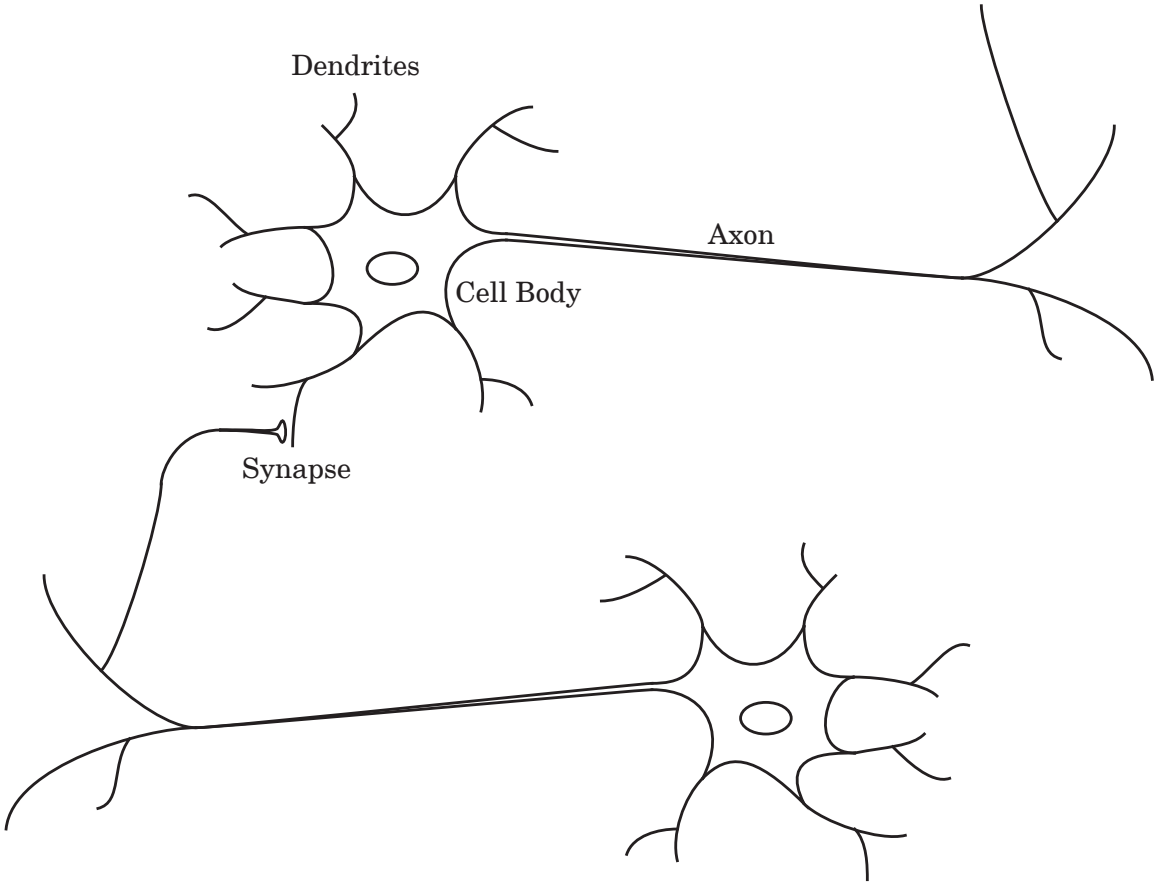
2.1.2 Modellierung eines künstlichen Neurons

Aus dem biologischen Neuron abgeleitet modellieren wir ein künstliches Neuron mit folgenden Komponenten: [Demuth et al., 2014, S.2-2ff]

- Einem *Eingabe*vektor $p \in \mathbb{R}^{n \times 1}$. Das sind die n Eingänge des Neurons.
- Einem *Gewichtungs*vektor $w \in \mathbb{R}^{1 \times n}$.
- Einem *Bias*-Wert $b \in \mathbb{R}$.
- Einer *Transferfunktion* $f : \mathbb{R} \mapsto \mathbb{R}$. In der Literatur wird diese auch als *Aktivierungsfunktion* bezeichnet (u.a. in [Karpathy, 2018], [Goodfellow et al., 2016]).

Der Ausgabewert a eines Neurons (der dann als Eingabewert für weitere Neuronen dient) ist definiert als $a = f(wp + b)$.

Abbildung 2.1: Schematischer Aufbau eines biologischen Neurons



Quelle: [Demuth et al., 2014, S.1-8]

Parameter und Hyperparameter Der Gewichtungsvektor und den Bias-Wert sind *Parameter*, die durch das Training erlernt werden. Die Transferfunktion wird nicht trainiert, sie ist ein *Hyperparameter*.

2.1.3 Wahl der Transferfunktion

Die Transferfunktion für das Neuron wird abhängig von dem vom neuronalen Netz zu lösenden Problem gewählt [Demuth et al., 2014, S.2-3]. Wichtig für das Training mit dem Backpropagation-Algorithmus (vgl. Kapitel 2.2.3) ist, dass die Transferfunktion differenzierbar ist (und das ist nach [Goodfellow et al., 2016, S.192] auch das einzig

wichtige Kriterium: Alle differenzierbaren Transferfunktionen funktionieren irgendwie, viele sogar gar nicht schlecht). In diesem Kapitel wollen wir kurz auf die zwei in der Bildverarbeitung populärsten Transferfunktionen eingehen

Log-Sigmoid Ein klassischerweise oft gewählte Transferfunktion ist die *Log-Sigmoid*-Funktion [Goodfellow et al., 2016, S.191]:

$$\text{logsig}(n) = \frac{1}{1 + e^{-n}}$$

Die Log-Sigmoid-Funktion ist einfach differenzierbar ($\frac{\partial \text{logsig}}{\partial n} = \text{logsig}(n) \cdot \text{logsig}(-n)$), aber sie beschränkt auch den Wertebereich der Ausgabe auf das Intervall $[0, 1]$, was das Training mit dem Gradientenabstiegsverfahren nicht wirklich fördert (vgl. [Goodfellow et al., 2016, S.191]).

Rectified Linear Units Neuere Literatur empfiehlt stattdessen die Verwendung der *Rectified Linear Unit*-Funktion (*ReLU*) [Goodfellow et al., 2016, S.171] [Glorot et al., 2011], weil lineare oder annähernd lineare Funktionen sich gut für das Gradientenabstiegsverfahren eignen:

$$\text{relu}(n) = \begin{cases} 0 & \text{für } x < 0 \\ n & \text{sonst} \end{cases}$$

Die ReLU-Funktion ist an der Stelle 0 nicht stetig differenzierbar. Das stört in der Praxis kaum, eher problematisch ist die Eigenschaft, dass für Eingabewerte kleiner 0 kein Gradient entsteht, der zur Veränderung der Parameter beiträgt. In besonders unglücklichen Fällen können dadurch ganze Regionen des Netzes absterben, wenn die Parameter nirgendwo mehr einen Gradienten herbekommen [Karpathy, 2018].

2.1.4 Anordnung der künstlichen Neuronen als Netz

Um aus einem einzelnen Neuron ein neuronales Netz zu machen, müssen die Neuronen verbunden werden. In einem künstlichen neuronalen Netz sind diese Verbindungen immer

fest vorgegeben (werden also nicht beim Lernprozess erzeugt, sondern nur die Gewichtung der Verbindung wird erlernt).

Anordnung als Graph Soll das Netz eine gewöhnliche mathematische Funktion berechnen, dann darf es in der Verbindung der Neuronen keinen Zyklus geben, d.h. die Neuronen bilden einen gerichteten azyklischen Graphen. Ein solches Netz wird als *Feed-Forward* Netz bezeichnet [Goodfellow et al., 2016, S.164].

Neuronale Netze mit Rückkopplungen (*recurrent neural networks*, kurz *RNN*s) sind möglich und nützlich für die Verarbeitung von Text und Sprache. Dazu wird das Netz an den Rückkopplungen zeitlich aufgerollt und der Lernalgorithmus dann auf den nun azyklischen, aufgerollten Graph angewendet [LeCun et al., 2015]. Für die Bildverarbeitung sind RNNs nicht relevant und werden deswegen hier nicht weiter behandelt.

Zusammenfassung als Layer Mehrere parallel arbeitende Neuronen, die alle dieselben Eingabewerte haben und je einen Ausgabewert produzieren, werden zusammengefasst als *Layer* [Demuth et al., 2014, S.2-10ff].

Ein Layer aus m Neuronen mit n Eingängen lässt sich als nur eine Funktion darstellen, dann bestehend aus

- Eingabevektor $p \in \mathbb{R}^{n \times 1}$
- Gewichtungs*matrix* $W \in \mathbb{R}^{m \times n}$
- Bias-Vektor $b \in \mathbb{R}^{m \times 1}$
- Transferfunktion $f : \mathbb{R}^m \mapsto \mathbb{R}^m$

mit dem m-elementigen Ausgabevektor $a = f(Wp + b)$.

Multilayer Perceptron Neuronale Netze bestehen in der Praxis aus mehreren hintereinander geschalteten Layern. Im einfachsten Falle, dem *Multilayer Perceptron*, fungieren die Ausgabevektoren der unteren Layer, auch *hidden layer* genannt, als Eingabevektor des nächsthöheren Layers. Das oberste Layer, *visible layer*, erstellt die Ausgabe des gesamten Netzes (vgl. [Goodfellow et al., 2016, S.6/S.165], [Demuth et al., 2014, S.2-12]).

2.2 Trainieren des Netzes

In diesem Kapitel wird erläutert, wie ein neuronales Netz aus gegebenen Eingabe- und Ausgabewerten eine Funktion erlernen kann. Diese Art des Lernens wird in der Literatur als *supervised learning* bezeichnet (vgl. [Goodfellow et al., 2016, S.103]). Das *unsupervised learning*, das ohne vorbereitetes Trainingsmaterial auskommt, wird im Rahmen dieser Arbeit nicht behandelt.

2.2.1 Lernfortschritt messen mit der Fehlerfunktion

Um die Leistung eines Netzes quantifizieren und den Lernfortschritt steuern zu können, wird eine *Fehlerfunktion* (*performance measure*) definiert. Die Fehlerfunktion beim supervised learning gibt für ein Eingabedatum an, wie gut die Ausgabe des Netzes von der zusammen mit der Eingabe definierten richtigen Ausgabe übereinstimmt. Die Definition der Fehlerfunktion ist im allgemeinen abhängig von der Aufgabe, die das neuronale Netz erfüllt [Goodfellow et al., 2016, S.101ff].

Ein Standardmaß für den Fehler ist die mittlere quadratische Abweichung MSE für alle m Ausgabewerte f_i des neuronalen Netzes und die für die Trainingsdaten erfassten Referenzwerte q_i (vgl. [Goodfellow et al., 2016, S.105]):

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (f_i - q_i)^2$$

2.2.2 Lernen mit dem Gradientenabstiegsverfahren

Das Verfahren Das Gradientenabstiegsverfahren (*Gradient Descent*) ist ein iteratives Verfahren, mit dem ein Minimum einer Funktion bestimmt werden kann. Es funktioniert folgendermaßen (nach [Goodfellow et al., 2016, S.80ff], ursprünglich [Cauchy, 1847]):

- Sei $f : \boldsymbol{x} \mapsto \mathbb{R}$ mit $\boldsymbol{x} = (x_1, x_2, ...)$ und einem beliebigen Startwert $\boldsymbol{x}^{(0)}$.
- Dann ist $x_k^{i+1} = x_k^i - \alpha \cdot \frac{\partial f}{\partial x_k}$ mit der Lernrate (*learning rate*) α (i.d.R. eine kleine Konstante).

Die Iteration wird solange fortgesetzt, bis sich $\boldsymbol{x}$ nicht mehr ändert. Das Gradientenabstiegsverfahren konvergiert immer, aber es wird nicht zwangsläufig das globale Minimum erreicht. Abhängig vom Startwert kann auch ein lokales Minimum berechnet werden.

Minimierung der Fehlerfunktion Mit dem Gradientenabstiegsverfahren kann ein neuronales Netz trainiert werden, indem die Fehlerfunktion minimiert wird: Dazu wird die Fehlerfunktion über alle Beispieldaten im Trainingsdatensatz gebildet und dann der Gradient für jeden zu trainierenden Parameter berechnet (wie erklären wir in Kapitel 2.2.3). Anhand dieses Gradienten wird dann über das Gradientenabstiegsverfahren der Parameter verändert.

Stochastisches Gradientenabstiegsverfahren Ein Problem bei der Verwendung des Gradientenabstiegsverfahrens zum Trainieren eines neuronalen Netzes ist der große Rechenaufwand, um die Fehlerfunktion über alle Werte im Trainingskorpus auszurechnen, besonders weil die für ein gutes Ergebnis notwendigen Trainingskorpora meist sehr umfangreich ausfallen.

Das Verfahren des *stochastischen Gradientenabstiegs* beruht auf der Erkenntnis, dass die Fehlerfunktion in aller Regel aus dem arithmetischen Mittel über alle Werte im Trainingskorpus besteht. Eine derartige Fehlerfunktion und deren Gradienten können auch durch eine zufällige Auswahl aus Trainingsbeispielen gut genug approximiert werden. (vgl. [Goodfellow et al., 2016, S.149], [Bottou and Bousquet, 2008]).

Beim stochastischen Gradientenabstiegsverfahren wird also in jedem Trainingsschritt eine zufällige Auswahl an Daten aus dem Trainingsdatensatz getroffen (oft als *Minibatch* bezeichnet) und damit die Fehlerfunktion und deren Gradienten berechnet, anhand deren danach die trainierbaren Parameter verändert werden.

2.2.3 Gradientenbildung mittels Backpropagation

Wird zum Training des neuronalen Netzes das Gradientenabstiegsverfahren (vgl. Kapitel 2.2.2) angewendet, so müssen die Gradienten der Fehlerfunktion für jeden zu trainierenden Parameter berechnet werden. Diesen Gradienten (numerisch oder symbolisch) für jeden Parameter direkt zu bilden, ist theoretisch möglich, in der Praxis jedoch viel zu aufwändig. Stattdessen macht man sich die aus der Analysis bekannte Kettenregel zu Nutze, um den Gradienten über den Berechnungsgraph zurück zu verteilen (vgl. [Goodfellow et al., 2016, S.200ff]). Die Kettenregel, wie sie in der Analysis gelehrt wird, sieht folgendermaßen aus:

$$[f(g(x))]' = f'(g(x)) \cdot g'(x)$$

bzw.

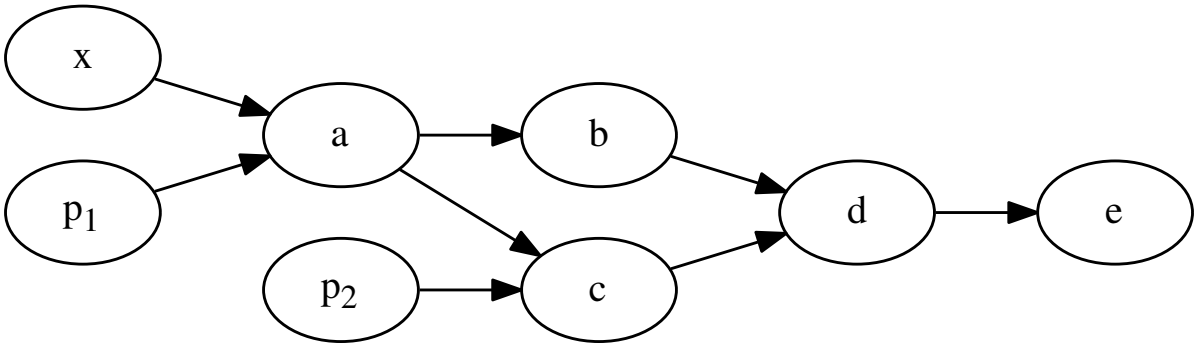
$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

Die Kettenregel wird nun auf einem *Berechnungsgraph* angewendet (ein neuronale Netz ohne Rückkopplungen kann als Berechnungsgraph interpretiert werden). Der Graph in Abbildung 2.2 zeigt beispielhaft die Definition einer Funktion e.

Anhand dieser Funktion wird erläutert, wie der Ausgabewert $e_{p_1,p_2}(x)$ sowie die Gradienten $\frac{\partial e}{\partial p_1} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial p1} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial a}\frac{\partial a}{\partial p1}$ und $\frac{\partial e}{\partial p_2} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial p_2}$ im Graph berechnet werden können.

Die Berechnung von $e(x)$ wird im Graph schrittweise von innen nach außen durchgeführt: Zuerst wird $a(x, p_1)$ berechnet, dann $b(a)$, $c(a, p_2)$, $d(b,c)$ und $d(e)$. Die Berechnungsreihenfolge entspricht einer topologischen Sortierung des Berechnungsgraphen.

Abbildung 2.2: Beispiel-Berechnungsgraph



Berechnungsgraph für die Funktion $e_{p_1,p_2}(x) = e(d(b(a(x, p_1)), c(a(x, p_1), p_2)))$

Quelle: Eigene Darstellung

Die Rückverteilung der Gradienten wird in genau umgekehrter Richtung durchgeführt: Zuerst wird $\frac{\partial e}{\partial d}$ berechnet, dann $\frac{\partial e}{\partial b} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$, $\frac{\partial e}{\partial c} = \frac{\partial e}{\partial d}\frac{\partial d}{\partial c}$, $\frac{\partial e}{\partial p_2} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial p_2}$, $\frac{\partial e}{\partial a} = \frac{\partial e}{\partial b}\frac{\partial b}{\partial a} + \frac{\partial e}{\partial c}\frac{\partial c}{\partial a}$ und schließlich $\frac{\partial e}{\partial p_1} = \frac{\partial e}{\partial a}\frac{\partial a}{\partial p_1}$.

Hat man über das Backpropagation-Verfahren die Gradienten der Fehlerfunktion an alle Parameter verteilt, können die Parameter nach dem Gradientenabstiegsverfahren modifiziert werden.

2.2.4 Trainingsfehler vermeiden

Das Training des neuronalen Netzes basiert auf der Fehlerfunktion *für den gegebenen Trainingsdatensatz*. Für den späteren Produktiveinsatz relevant ist jedoch, wie gut das Netz auf *bisher unbekannte Eingaben* reagiert. Wenn das Netz dabei schlecht abschneidet, spricht man von einem *Generalisierungsfehler* [Goodfellow et al., 2016, S.109].

Overfitting, Underfitting und Kapazität Bei der Übertragung des trainierten Netzes auf einen separat gesammelten Testdatensatz lassen sich zwei Arten von Trainingsfehlern erkennen (vgl. [Goodfellow et al., 2016, S.109]):

- **Overfitting** Das Netz hat einen sehr geringen Fehler auf dem Trainingsdatensatz, jedoch eine viel höhere Fehlerrate auf dem Testdatensatz. Das Netz hat also im

Trainingsdatensatz Muster erkannt, die sich nicht auf den Testdatensatz generalisieren lassen.

- **Underfitting** Das Netz konnte im Training keine gute Performance auf dem Trainingsdatensatz erreichen. Es ist nicht fähig, die gewünschte Aufgabe mit akzeptabler Fehlerrate durchzuführen.

Das Potential zum Under- und Overfitting wird durch die Kapazität (*capacity*) des Netzes bestimmt. Die Kapazität misst die im Netz trainierbaren Parameter und drückt aus, wie kompliziert die vom Netz berechnete Funktion überhaupt sein kann. Ein Netz mit ungenügender Kapazität muss zwangsweise zu Underfitting führen, eine zu hohe Kapazität ist ein Risikofaktor für Overfitting [Goodfellow et al., 2016, S.110].

Regularisierung Als Regularisierung (*regularization*) wird jede Modifikation des Lernalgorithmus bezeichnet, die den Generalisierungsfehler verkleinert, ohne den Trainingsfehler negativ zu beeinflussen [Goodfellow et al., 2016, S.117].

Der einfachste Weg um Overfitting vorzubeugen, ist die Verwendung von mehr Trainingsdaten. Hat man nicht genügend davon zur Verfügung, lohnt es sich oft, zusätzliche Daten aus den vorhandenen Trainingsbeispielen durch Anwendung von Transformationen, gegenüber denen das Ergebnis des Netzes invariant sein soll, zu generieren [Goodfellow et al., 2016, S.237]. Für einen Bildklassifizierer wäre z.B. ein zufälliges Vergrößern, Verkleinern, Verschieben oder Drehen angemessen.

Dropout-Regularisierung Eine gute Regularisierungsmethode, die ohne Veränderungen der Eingaben auskommt, ist die *Dropout-Regularisierung* [Srivastava et al., 2014]. Bei der Dropout-Regularisierung werden während des Trainings zufällig innere Knoten und deren Verbindungen im Netz verworfen (in der Praxis werden keine Knoten entfernt, sondern deren Ausgabe mit 0 multipliziert). Die verworfenen Knoten können als zusätzliches Rauschen im Netz interpretiert werden, oder als Kombination vieler unterschiedlicher neuronaler Netze. Dropout erreicht mit minimalem Rechenaufwand bessere Ergebnisse als andere Regularisierungsverfahren [Srivastava et al., 2014]. Es kann auf fast jedes Netz und jede Lernmethode (inkl. das stochastische Gradientenabstiegsverfahren) angewendet werden [Goodfellow et al., 2016, S.262].

2.3 Convolutional Neural Networks zur Bildklassifizierung

Die Aufgabe einer Bildklassifizierung besteht darin, ein gegebenes Bild einer vorgegebenen Menge an Klassen zuzuordnen. Ein Bildklassifizierer ist eine Funktion, die für ein Eingabebild eine Wahrscheinlichkeitsverteilung über die vorgegebenen Klassen ausgibt.

State-of-the-Art in der Klassifizierung von Bildern sind aktuell Convolutional Neural Networks mit vielen Faltungslayern [LeCun et al., 2015]. Seit dem Durchbruch durch [Krizhevsky et al., 2012] gehören Convolutional Neural Networks (kurz CNN, oder auch ConvNets) zu den Standardwerkzeugen in der digitalen Bildverarbeitung und erzielen mit fortschreitender Forschung und Rechnerleistung immer bessere Erfolge in der Objektklassifizierung (z.B. [Krizhevsky et al., 2012], [Zeiler and Fergus, 2014], [Szegedy et al., 2015]).

2.3.1 Motivation

Ein großes Problem bei der Verarbeitung von Bildern ist die Menge der zu verarbeitenden Daten. Um ein Bild der Größe 320x240 Pixel ($= 320 \cdot 240 \cdot 3 = 230400$ Werte) durch ein Layer an Neuronen zu verarbeiten, die mit jedem Eingabepixel verbunden sind, wäre eine Gewichtungsmatrix mit so vielen Spalten wie Bildpixeln vonnöten. Diese Menge an zu trainierenden Parameter übersteigt das Platzangebot einer handelsüblichen Grafikkarte und ist ein Risikofaktor für Overfitting (vgl. [Karpathy, 2018]).

Ein Convolutional Neural Network besteht aus einer Menge von Faltungslayern (Kapitel 2.3.3) kombiniert mit Max-Pooling-Layern (Kapitel 2.3.4) und danach ein oder mehrere vollverbundene Layer (vgl. Kapitel 2.1.4) zur Errechnung der Ausgabe. Ein Netz zur Bildklassifizierung gibt eine mit der Softmax-Funktion (Kapitel 2.3.5) normalisierte Wahrscheinlichkeitsverteilung für die definierten Bildklassen aus.

Die Faltungslayer haben im Vergleich zu einem vollverbunden Layer viel weniger zu trainierende Parameter, die Max-Pooling-Layer reduzieren die zu verarbeitende Datenmenge für nachfolgende Schichten.

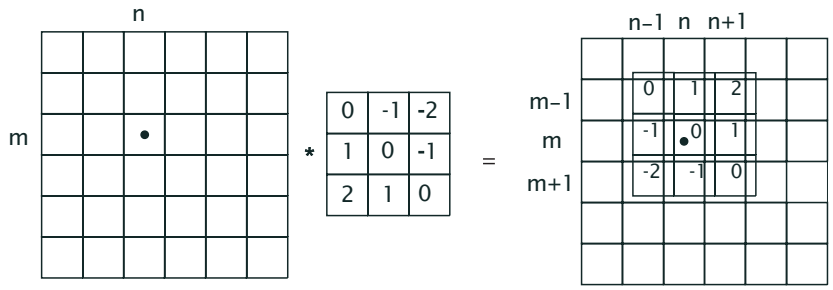
2.3.2 Faltung

Die *Faltung* ist eine aus der Bildverarbeitung bekannte Operation zur Verarbeitung eines k-dimensionalen Bildes (bzw. einer gleichartigen Datenstruktur). Im diskreten Raum ist die Faltung zweier Funktionen f und $g : \mathbb{C}^k \mapsto \mathbb{C}$ definiert als:

$$(k * f)(x_1, x_2, ..., x_k) = \sum_{i_1=-\infty}^{\infty} \sum_{i_2=-\infty}^{\infty} ... \sum_{i_k=-\infty}^{\infty} k(i_1, i_2, ..., i_k) f(x_1 - i_1, x_2 - i_2, ..., x_k - i_k)$$

In der Praxis der Bildverarbeitung ist die Funktion k, der *Faltungskern*, nur auf einem begrenzten Intervall $[u, v]$ ungleich 0, weswegen die Summen nur von u nach v gebildet werden müssen (vgl. [Jähne, 2013, S.127, S.307]). Ein graphisches Beispiel für eine Faltung ist in Abbildung 2.3 zu sehen.

Abbildung 2.3: Faltung im zweidimensionalen Raum mit 3x3-Faltungskern



Quelle: [Jähne, 2013, S.309]

Randproblem Wenn die Faltung für einen Bildpunkt am Rand berechnet wird, gibt es das Problem, dass die Berechnung die Werte von nicht existierenden Bildpunkten außerhalb des

Bildes benötigt. Für dieses sogenannte *Randproblem* gibt es eine Reihe an Lösungsansätzen (vgl. [Jähne, 2013, S.306], [Karpathy, 2018], [Goodfellow et al., 2016, S.343]):

- Keine Faltung für Randpunkte berechnen (Matlab-Terminologie: *valid convolution*). Das Bild schrumpft dadurch.
- Nicht existierende Werte durch 0 ersetzen (Matlab-Terminologie: *same*).
- Extrapolation des Randes durch Kopieren des Randwerts oder Fortsetzung des Gradienten.

2.3.3 Convolutional Layer

Ein Faltungslayer interpretiert die Eingänge als k-dimensionales Bild, der Faltungskern hat eine feste Größe und besteht aus trainierbaren Parametern.

Ein bildverarbeitendes Faltungslayer arbeitet mit einem dreidimensionalen Eingangsbild (Breite, Höhe, Farbkanäle) und erzeugt ein dreidimensionales Ausgangsbild (Breite, Höhe, Ausgangskanäle). Dabei können sowohl die erzeugte Breite und Höhe vom Eingangsbild abweichen (z.B. etwas kleiner um das Randproblem zu umgehen), als auch die Anzahl Kanäle. Die durchgeführte Operation entspricht dann nicht mehr der oben gezeigten Faltungsoperation, sondern kann z.B. folgendermaßen aussehen [Goodfellow et al., 2016, S.342]:

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m,k+n} K_{i,l,m,n} + B_i$$

beschreibt die Kreuzkorrelation (entspricht einer Faltung mit punktgespiegeltem Faltungskern) einer Bildfunktion $V_{l,j,k}$ (der Bildpunkt an Stelle j,k im Kanal l) mit einem vierdimensionalen Faltungskern $K_{i,l,m,n}$ (Gewichtungsfaktor an Stelle m,n von Eingangskanal l zu Ausgangskanal i), dazu ein Bias-Wert B_i (Skalar für Ausgabekanal i) zu einer neuen Bildfunktion $Z_{i,j,k}$ (Bildpunkt an Stelle j,k im Kanal i).

[Karpathy, 2018] interpretiert die Ausgabe der Faltung nicht als 3D-Bild, sondern als Kombination von mehreren 2D-Bildern, die aus der Faltung des Eingabebilds mit mehreren dreidimensionalen Faltungskernen entstehen. Mathematisch macht das keinen Unterschied.

Neuronale Interpretation Ein Faltungslayer kann man sich vorstellen als eine Menge an Neuronen, die als Gitter angeordnet sind, jeweils nur mit ihrer Nachbarschaft verbunden sind (statt mit allen Neuronen des vorherigen Layers), und sich trainierbare Parameter teilen [Karpathy, 2018].

Implementierung als Matrixmultiplikation Zur Beschleunigung der Ausführung auf einer Grafikkarte wird die Faltung als Matrixmultiplikation ausgedrückt. Dazu werden die Punkte aus der X/Y-Ebene seriell angeordnet [Karpathy, 2018]:

- Alle Bildpunkte werden nach ihren X- und Y-Koordinaten serialisiert. Aus Punkt j,k wird Punkt n; $\hat{Z}_{i,n} = Z_{i,j,k}$ ist der Ausgabepunkt n im Ausgabekanal i.
- Die Nachbarschaft der Eingabepunkte wird aus den X-, Y-Koordinate und dem Kanal serialisiert: $\hat{V}_{n,r}$ ist der r-te Wert in der Nachbarschaft von Punkt n.
- Der Faltungskern wird für jeden Ausgabekanal serialisiert: $\hat{K}_{i,r}$ ist der r-te Wert im Faltungskern für den Ausgabekanal i. Dabei muss so serialisiert werden, dass $\hat{K}_{i,r} \cdot \hat{V}_{n,r} = K_{i,l,x,y} \cdot V_{l,j+x,k+y}$ (also der r-te Wert im serialisierten Faltungskern ist der, der bei der Faltung mit dem r-ten Wert in der serialisierten Nachbarschaft multipliziert wird).

Daraus ergibt sich dann folgende Berechnungsformel in Matrixschreibweise:

$$\begin{pmatrix} \hat{Z}_{0,0} & \dots & \hat{Z}_{0,n} \\ \vdots & \ddots & \vdots \\ \hat{Z}_{i,0} & \dots & \hat{Z}_{i,n} \end{pmatrix} = \begin{pmatrix} \hat{K}_{0,0} & \dots & \hat{K}_{0,r} & B_0 \\ \vdots & \ddots & \vdots & \vdots \\ \hat{K}_{i,0} & \dots & \hat{K}_{i,r} & B_i \end{pmatrix} \begin{pmatrix} \hat{V}_{0,0} & \dots & \hat{V}_{n,0} \\ \vdots & \ddots & \vdots \\ \hat{V}_{0,r} & \dots & \hat{V}_{n,r} \\ 1 & \dots & 1 \end{pmatrix}$$

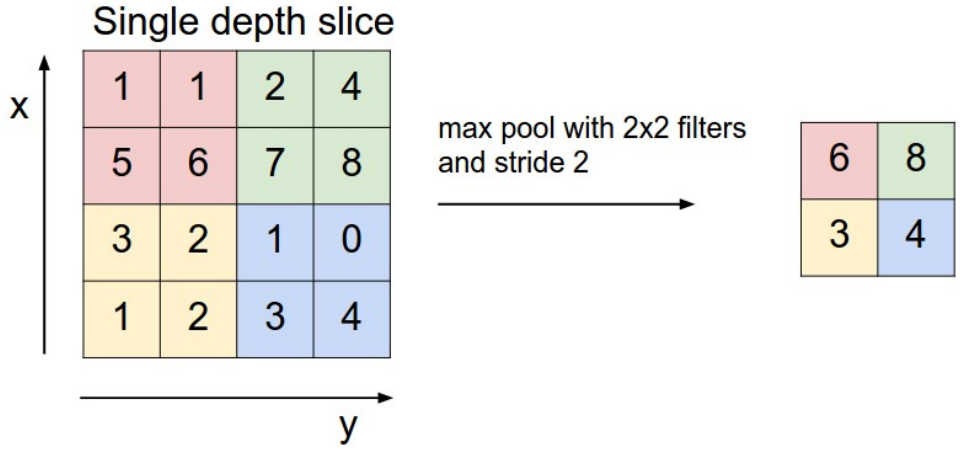
2.3.4 Max-Pooling Layer

Ein Max-Pooling-Layer hat das primäre Ziel, die räumliche Ausdehnung des Bilds und damit die Anzahl Parameter für weitere Verarbeitungsschritte zu reduzieren. Es wird

in einem Convolutional Neural Network in aller Regel direkt hinter einem Faltungslayer angeordnet [Karpathy, 2018].

Visuell betrachtet fasst ein Max-Pooling-Layer mehrere Pixel zusammen, und übernimmt nur den maximalen Pixelwert als Wert des Ausgabebilds (vgl. Abbildung 2.4).

Abbildung 2.4: Max-Pooling visuell erklärt



Quelle: [Karpathy, 2018]

Die formale Definition ist wie folgt: Aus einer 2D-Eingabe $X : \mathbb{R}^{n \times m}$ erzeugt das Max-Pooling-Layer eine 2D-Ausgabe $M : \mathbb{R}^{(\frac{n-w}{s}+1)\times(\frac{m-h}{s}+1)}$ mit den beim Netzentwurf festzulegenden Hyperparametern w,h: Größe des Pooling-Bereichs und s: Schrittweite (*stride*).

Für ein Ausgabepixel des Max-Pooling-Layers gilt dann:

$$M_{x,y} = \max\left(\sum_{i=0}^{w-1}\sum_{j=0}^{h-1} X_{xs+i,ys+j}\right)$$

Training Das Max-Pooling-Layer hat keine trainierbaren Parameter. Der Gradient (vgl. Kapiel 2.2.3) wird zurückverteilt an den Eingabewert, der als Maximum ausgewählt wurde.

Alternativen Die Verkleinerung des Bildes kann statt durch ein Max-Pooling-Layer alternativ durch eine Schrittweite im Faltungslayer ersetzt werden (d.h. es wird einfach nur jeder n-te Ausgabepunkt der Faltung berechnet). [Springenberg et al., 2014] kamen zum Schluss, dass das genauso gut funktioniert.

2.3.5 Softmax-Ausgabe und Fehlerfunktion

Am Ende eines Netzwerks zur Bildklassifizierung steht die Einteilung in erkannte Bildklassen. Typischerweise hat das letzte Layer je zu erkennender Klasse genau einen Ausgabewert. Jeder dieser Ausgabewerte symbolisiert (mehr ist besser, jedoch ohne festgelegte Skalierung), wie gut das Bild zu der zugehörigen Klasse passt.

Die Softmax-Funktion Die Ausgabewerte des letzten Layers werden über die *Softmax*-Funktion in eine Wahrscheinlichkeitsverteilung umgerechnet (d.h. derart normalisiert, dass jeder Wert zwischen 0 und 1 liegt und alle Werte zusammen 1 ergeben). Die Definition der Softmax-Funktion ist wie folgt [Karpathy, 2018] [Goodfellow et al., 2016, S.181]:

$$\text{softmax}_i(f_i) = \frac{e^{f_i}}{\sum_j e^{f_j}}$$

softmax_i ist die normalisierte Wahrscheinlichkeit für die Klasse i, errechnet aus dem Ausgabewert f_i des Netzes für diese Klasse und den Ausgabewerten f_j für jede Klasse j.

Softmax-Fehlerfunktion Ein angemessenes Fehlermaß für eine Wahrscheinlichkeitsverteilung ist die *Kreuzkorrelation* (*cross-entropy*, in der Literatur teilweise auch als *negative log-likelihood* [Goodfellow et al., 2016, S.131] bezeichnet). Die Kreuzkorrelation H zweier Wahrscheinlichkeitsverteilungen p und q ist ein Maß dafür, wie gut die geschätzte Wahrscheinlichkeitsverteilung q die wahre Wahrscheinlichkeitsverteilung p approximiert [Karpathy, 2018]:

$$H(p, q) = -\sum_x p(x) \log q(x)$$

Wird einem Eingabebild immer genau eine Klasse zugeordnet, so vereinfacht sich die Wahrscheinlichkeitsverteilung p dahingehend, dass sie für die richtige Klasse genau 1 und sonst gleich 0 ist. Daraus ergibt sich, kombiniert mit der Softmax-Funktion, folgendes vereinfachtes Fehlermaß [Karpathy, 2018] [Goodfellow et al., 2016, S.182]:

$$E = -\log \frac{e^{f_i}}{\sum_j e^{f_j}} = \log \left(\sum_j e^{f_j} \right) - f_i$$

f_i ist der Ausgabewert des Netzes für die eigentlich korrekte Klasse, f_j sind jeweils die Ausgabewerte für die Klasse j.

3 Stauerkennung auf Bildern der Verkehrskameras

Das theoretischen Rüstzeug aus dem vorherigen Kapitel nutzen wir nun, um die Kamerabilder der Straßenverkehrszentrale auszuwerten. Dazu klassifizieren wir jedes Bild in die zwei Kategorien *Stau* und *fließend*.

Als Framework für das neuronale Netz haben wir TensorFlow [Abadi et al., 2016] gewählt. Ausschlaggebend für die Entscheidung für TensorFlow war, dass TensorFlow bereits seit Jahren als zuverlässiges Framework etabliert ist, sowie viel und gute Dokumentation und weitere Hilfsmaterialen dazu existieren. Es enthält praktischerweise auch alle im vorangegangenen Theoriekapitel besprochenen Elemente, um ein neuronales Netz zur Bildklassifizierung aufzubauen.

3.1 Auswahl und Vorverarbeitung der Kamerabilder

Für die Studienarbeit berücksichtigt wurden die Verkehrskameras an den Anschlussstellen Bruchsal, Karlsruhe Nord, Karlsruhe Mitte, und Ettlingen. Die Kamera am Dreieck Karlsruhe war im Zeitraum, in dem die Trainings- und Validierungsdaten gesammelt wurden, außer Betrieb und ist deswegen nicht berücksichtigt worden.

Zwei Richtungen Alle diese Kameras sind doppelt vorhanden, je eine pro Fahrtrichtung. Wir sehen auf der Kamera also jeden Stau einmal gegen die Fahrtrichtung („von vorne“) und einmal mit der Fahrtrichtung („von hinten“). In der Auswertung (vgl. Kapitel 3.4.2) hat sich bestätigt, dass die Verarbeitung beider Kamerabilder sich positiv auf das Ergebnis auswirkt.

Maskierung Um den Klassifikator nicht unnötig zu verwirren, wird jedes Kamerabild mit einer *Maske* überdeckt, die alles außer die zu analysierende Fahrbahn überdeckt. Die verwendeten Masken wurden jeweils von Hand in einem Bildbearbeitungsprogramm erstellt und werden automatisiert über die Kamerabilder gelegt. Die Maskierung der Gegenfahrbahn verbessert das Ergebnis der Klassifizierung, der Unterschied ist aber weniger groß als wir erwartet hatten (vgl. Ergebnis von Netz 2 ohne Maske im Kapitel 3.4.2).

In Abbildung 3.1 ist beispielhaft gezeigt, wie ein Stau auf der Verkehrskamera und in den für die Klassifizierung maskierten Bildern aussieht.

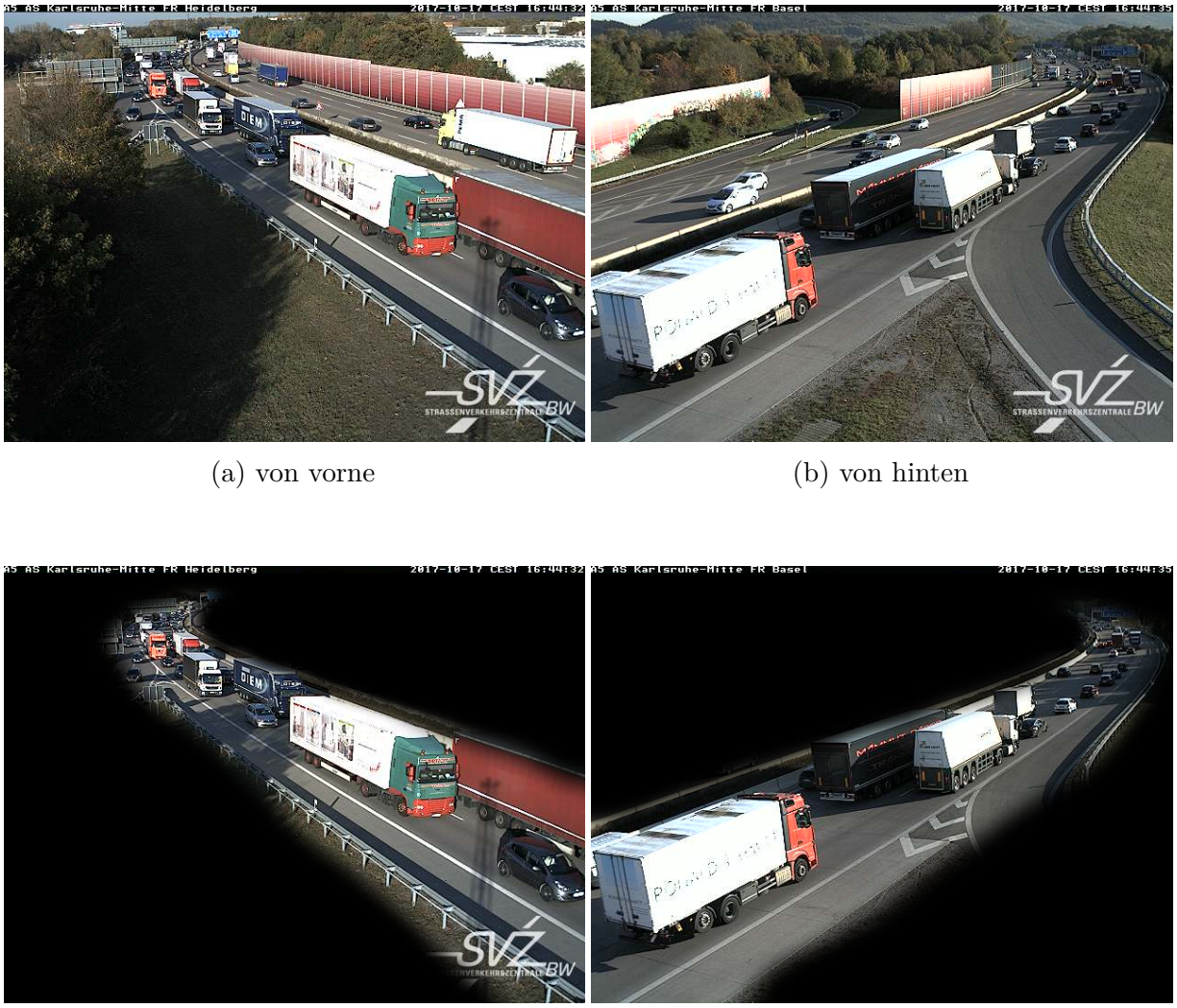
Zuverlässigkeit Die Kameras arbeiten nicht immer so zuverlässig, wie wir das gerne hätten. Das Aktualisierungsintervall der Kamerabilder variiert im Regelfall zwischen einer und fünf Minuten, in einem Fall ist uns allerdings ein seit Stunden nicht mehr aktualisiertes Bild aufgefallen. Manchmal werden abgeschnittene JPEG-Dateien verteilt. Leider häufiger werden nicht verfügbare Kamerabilder durch ein „außer Betrieb“-Standbild (Abbildung 3.3) ersetzt. Sowohl veraltete als auch nicht verfügbare Bilder werden vom Backend-Server (vgl. Kapitel 4) erkannt und für die Stau-Auswertung ignoriert, im Trainings- und Validierungsdatensatz haben wir sie manuell aussortiert.

Bildqualität bei Nacht Die Verkehrskameras haben große Schwierigkeiten, bei wenig Licht noch gute Bilder zu erfassen. Auf einem Nachtbild mit wenigen bis gar keinen Fahrzeugen ist praktisch nichts zu erkennen, erst bei vielen Fahrzeugen reicht das Scheinwerferlicht aus, um ein akzeptables Bild zu erzeugen (vgl. Abbildung 3.4). Praktische Schwierigkeiten hat das unerwarteterweise nicht hervorgerufen, die trainierten neuronalen Netze haben die Erkennung eines Staus in der Dunkelheit erfolgreich gelernt.

3.2 Netzentwurf

Zur Klassifizierung der Bilder wurden mehrere unterschiedliche neuronale Netze entworfen und getestet. Hier vorgestellt werden 5 von uns entworfene und trainierte Netze, die einen signifikant unterschiedlichen Aufbau haben. Das sind nicht alle von uns trainierten und getesteten Netzarchitekturen, sondern die fünf, die im Test die besten Ergebnisse erzielten

Abbildung 3.1: Kamerabilder eines Staus in Karlsruhe Mitte



(a) von vorne

(b) von hinten

(c) von vorne, maskiert

(d) von hinten, maskiert

Quelle: Straßenverkehrszentrale Baden-Württemberg / Eigene Darstellung

Abbildung 3.3: Eine Kamera außer Betrieb

Quelle: Straßenverkehrszentrale Baden-Württemberg

Abbildung 3.4: Kamerabilder bei Dunkelheit

(a) fließend (b) Stau

Quelle: Straßenverkehrszentrale Baden-Württemberg

und uns für die Studienarbeit relevant schienen. In Ermangelung weiterer Kreativität sind die Netze durchnummeriert, die Reihenfolge spiegelt jedoch nicht die Erstellungsreihenfolge oder die Leistungsabstufung wieder.

3.2.1 Netz 1: Convolutional Neural Network mit zwei Faltungslayern

Das erste Netz setzt zwei Faltungslayer zur Verarbeitung der Kamerabilder ein. Es werden beide vorhandenen Kamerabilder für eine Stelle getrennt in den Faltungslayern verarbeitet und dann in einem vollverbundenen Hidden Layer zusammengeführt (vgl. Abbildung 3.6).

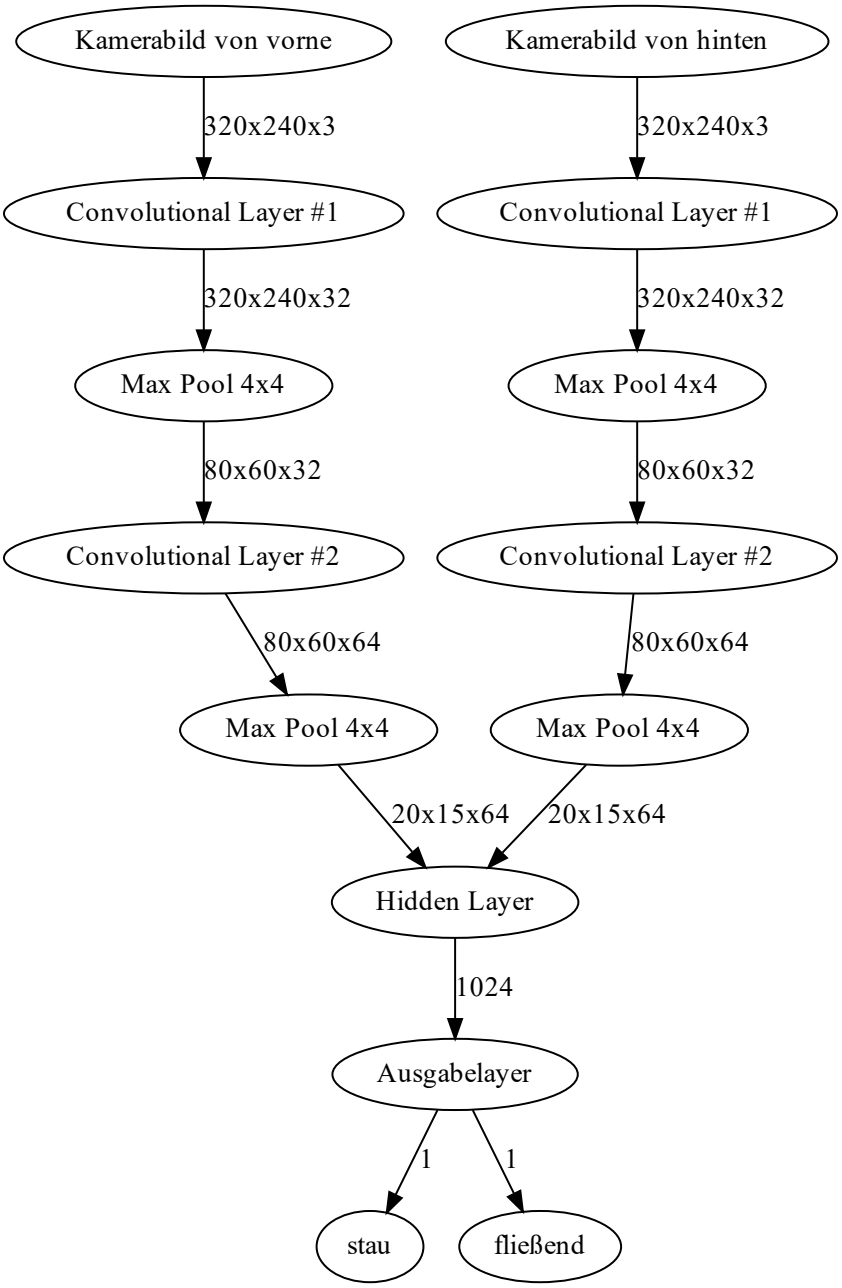
Ausgegeben wird eine Wahrscheinlichkeitsverteilung für die zwei Bildklassen „fließend“ und „Stau“. Als Aktivierungsfunktion wird immer die ReLU-Funktion (vgl. Kapitel 2.1.3) verwendet.

Zum Training wird vor dem Ausgabelayer noch ein Dropout-Layer mit Dropout-Wahrscheinlichkeit $p = 0{,}4$ eingefügt. Das Dropout-Layer ist ein Ansatz zur Umsetzung der Dropout-Regularisierung (vgl. Kapitel 2.2.4). Es hat so viele Aus- wie Eingabewerte und jeder Ausgabewert ist entweder zufällig 0 oder der mit $\frac{1}{1-p}$ skalierte Eingabewert (sodass die Summe der Werte unabhängig vom Dropout gleich bleibt). Der Dropout wird nur während des Trainings ausgeführt, nach abgeschlossenem Training wird das Dropout-Layer deaktiviert und reicht die Eingabewerte unverändert weiter [TensorFlow, 2018].

Trainiert wird für 100000 Trainingsschritte mit je 10 Bildern pro Minibatch. Als Fehlermaß kommt die Softmax-Fehlerfunktion zum Einsatz (vgl. Kapitel 2.3.5). Für das Gradientenabstiegsverfahren (vgl. Kapitel 2.2.2) wurde eine Lernrate von $\alpha = 0{,}045$ gewählt, die mit der Zeit exponentiell fällt ($\alpha = 0{,}045 \cdot 0{,}96^{\frac{\text{steps}}{1000}}$).

Damit das Netz noch auf die zum Training verwendete Grafikkarte (eine Nvidia Geforce GTX 670) passt, arbeitet das Netz nur mit auf die halbe Größe (320x240 Pixel) verkleinerten Bildern.

Abbildung 3.6: Netzarchitektur: CNN mit zwei Faltungslayern



Die Knoten sind Layer im Netz, die Kantenbeschriftungen geben die Dimensionen des Ausgabe-/Eingabevektors an

Quelle: Eigene Darstellung

3.2.2 Netz 2: Convolutional Neural Network mit nur einem Faltungslayer

Das zweite Netz ähnelt im Aufbau dem ersten, macht aber einen anderen Kompromiss um auf die Grafikkarte zu passen: Die Bilder werden in voller Größe verarbeitet (also 640x480 Pixel), dafür gibt es nur noch ein Faltungslayer.

Die Architektur ist in Abbildung 3.7 dargestellt. Alle Parameter sind genau wie bei Netz 1, jedoch haben sich 50000 Trainingsschritte als völlig ausreichend herausgestellt. Die Evaluierung (vgl. Kapitel 3.4.2) ergab, dass das Entfernen eines Faltungslayers ein schlechter Kompromiss ist.

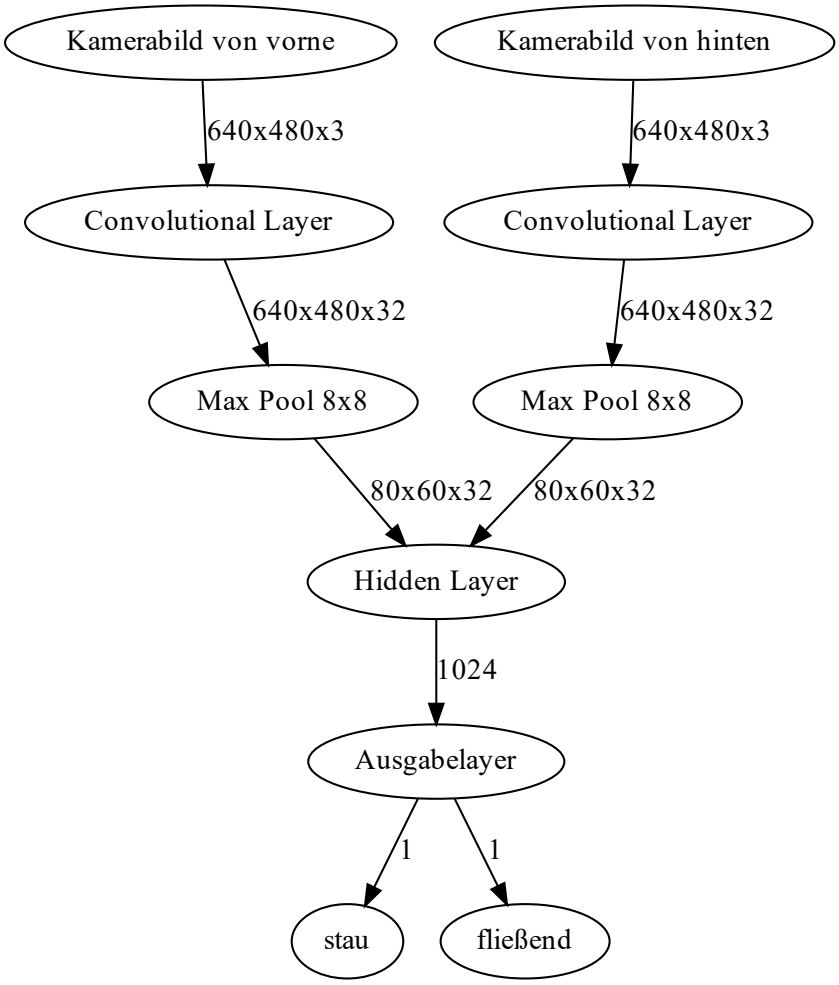
3.2.3 Netz 3: Transferlernen mit Inception-V3

Als weiterer Ansatz wurde das vortrainierte Netz Inception-V3 [Szegedy et al., 2015] mit einem neuen Hidden- und Ausgabelayer ausgestattet und trainiert.

Transfer Learning Das Verfahren, die trainierten Hidden Layers eines Netzes weiter zu verwenden und nur ein neues Ausgabelayer zu trainieren bezeichnet man als *Transferlernen* (*Transfer Learning*) [Goodfellow et al., 2016, S.534ff]. Das Transferlernen funktioniert, wenn das Netz zum Erfüllen des ursprünglichen Lernauftrags Muster erkannt hat, die auch für den neuen Anwendungsfall relevant sind. Man spart sich die für ein aufwendiges CNN notwendige Trainingszeit, die mehrere Wochen auf mehreren GPUs betragen kann [Karpathy, 2018]. Die Effektivität von Transferlernen für Bildklassifizierungsaufgaben kann erstaunlich hoch sein [Pan and Yang, 2010] [Razavian et al., 2014].

Inception-V3 Das Netz Inception-V3 [Szegedy et al., 2015] ist trainiert auf den Daten des Bildklassifizierungs-Wettbewerbs ImageNet [Russakovsky et al., 2015] von 2012 und erreichte dort Verbesserungen im Vergleich zum damaligen Gewinner „AlexNet“ [Krizhevsky et al., 2012] bei geringerem Rechenaufwand als konkurrierende Netzarchitekturen. Der Aufbau von Inception-V3 ist in Abbildung 3.8 dargestellt.

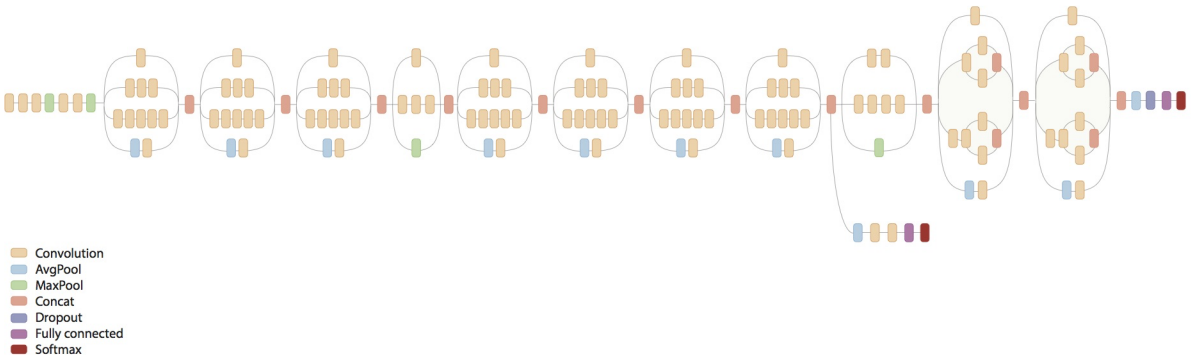
Abbildung 3.7: Netzarchitektur: CNN mit einem Faltungslayer



Die Knoten sind Layer im Netz, die Kantenbeschriftungen geben die Dimensionen des Ausgabe-/Eingabevektors an

Quelle: Eigene Darstellung

Abbildung 3.8: Architektur von Inception-V3



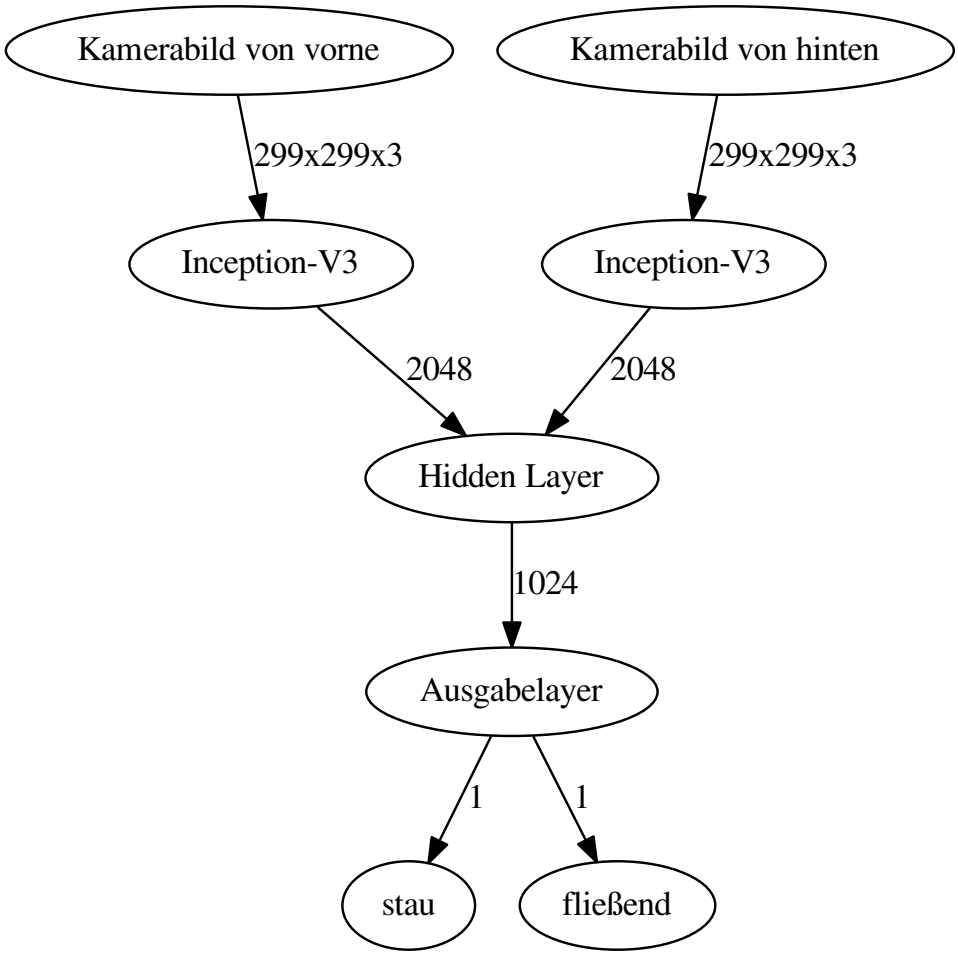
Quelle: `https://github.com/tensorflow/models/tree/master/research/inception`, Abgerufen am 22.04.2018

Umtrainieren von Inception-V3 zur Stauerkennung Zur Stauerkennung binden wir die Hidden Layers des vortrainierte Inception-V3-Netz[1] anstelle eigener Faltungslayer in unser Netz ein. Abbildung 3.9 zeigt den Aufbau des Netzes, trainiert werden lediglich das Hidden Layer und das Ausgabelayer in 100000 Trainingsschritten zu je 100 Bildern bei einer Lernrate von 0,045, die alle 1000 Schritte exponentiell verringert wird ($\text{rate} = 0{,}045 \cdot 0{,}96^{\frac{\text{steps}}{1000}}$). Die Ausgabe des Inception-V3-Layers wird dabei nicht bei jedem Trainingsschritt erneut berechnet, sondern wird zu Beginn für alle Bilder im Trainingsdatensatz einmalig berechnet. So wird das Training erheblich beschleunigt und kann in wenigen Minuten auf einer einzigen Grafikkarte durchgeführt werden.

Die Auswertung in Kapitel 3.4.2 und 3.4.3 hat ergeben, dass dieser Ansatz für unsere Aufgabe messbar schlechter funktioniert als einfache vollständig selbst trainierte Netze. Vermutlich unterscheiden sich die Bilder der Verkehrskameras zu sehr vom ImageNet-Datensatz, sodass die von Inception-V3 gefundenen Merkmale nicht optimal für unsere Klassifizierungsaufgabe geeignet sind.

[1] Wir verwenden die mit dem Tensorflow-Framework veröffentlichte Version: `http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz`

Abbildung 3.9: Netzarchitektur: Transferlernen mit Inception-V3



Die Knoten sind Layer im Netz, die Kantenbeschriftungen geben die Dimensionen des Ausgabe-/Eingabevektors an

Quelle: Eigene Darstellung

3.2.4 Netz 4 und 5: Nur ein Kamerabild verwenden

Die Netze 4 und 5 sind aufgebaut und trainiert wie Netz 1, jedoch verwenden sie nur das vorder- bzw. rückseitige Bild (vgl. Abbildung 3.10).

Durch diese Netze wurde die Hypothese überprüft, dass die Verwendung beider Bilder ein Gewinn für die Klassifizierung ist. Die Auswertungsergebnisse (vgl. Kapitel 3.4.2 und 3.4.3) haben das klar bestätigt.

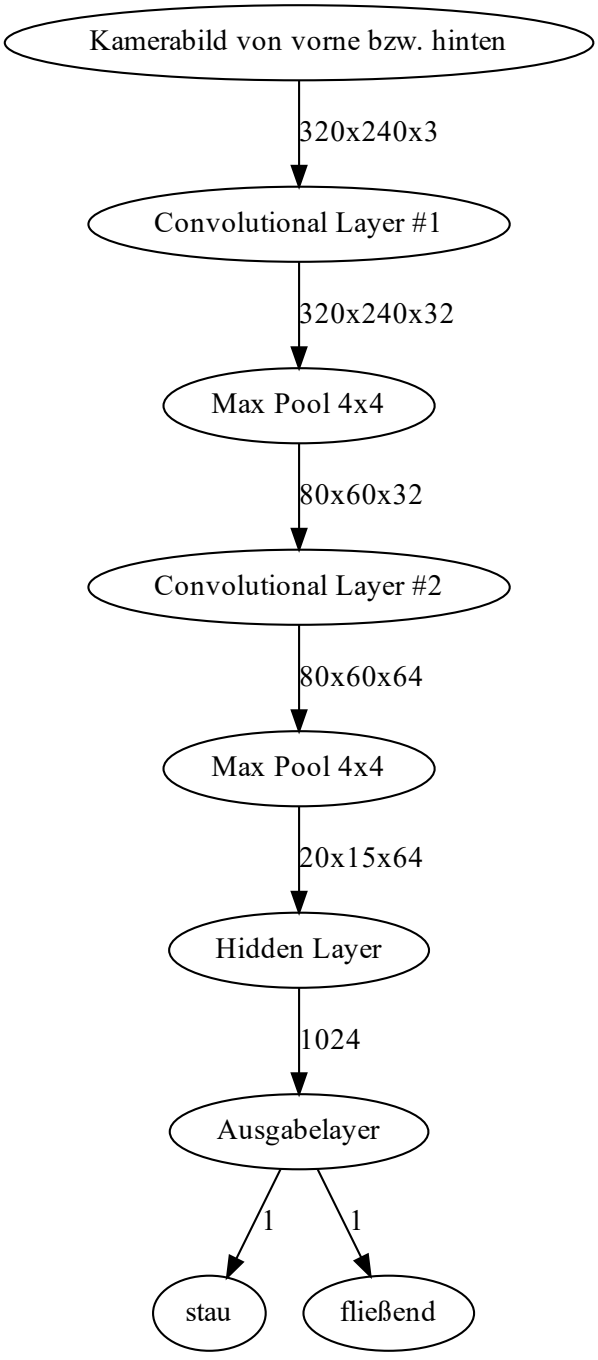
3.3 Training

Zum Training der im vorherigen Abschnitt gezeigten neuronalen Netze haben wir eine Menge an Kamerabildern[2] manuell klassifiziert. Die Bilder wurden zwischen 15.10.2017 und Weihnachten 2017 von der Straßenverkehrszentrale abgerufen.

Der Trainingsdatensatz besteht aus 4897 Bildern zum Trainieren und 2445 weiteren Bildern zur Validierung (die nicht im Trainingsprozess verwendet werden). Für jede Kameraposition sind mehrere hundert Bilder mit Stau und mit flüssigem Verkehr enthalten, lediglich für die Anschlussstelle Bruchsal sind nur insgesamt 63 Bilder mit Stau im Beobachtungszeitraum aufgetreten. Deswegen ist die tatsächliche Leistung für diese Kamera ungewiss und die Evaluierungsergebnisse haben für Bruchsal nur eine begrenzte Aussagekraft.

[2] Download: `https://mega.nz/#!8Ncm0S5L!Kkrf8YJClnCGDckBSRF5aNuuogijtPsBxQsjYJzraBQ`

Abbildung 3.10: Netzarchitektur: CNN mit zwei Faltungslayer für nur ein Bild



Die Knoten sind Layer im Netz, die Kantenbeschriftungen geben die Dimensionen des Ausgabe-/Eingabevektors an

Quelle: Eigene Darstellung

3.4 Evaluierung

3.4.1 Kennzahlen

Accuracy Meist wird ein Klassifizierer nach der Genauigkeit (*Accurracy*) seiner Klassifizierung bewertet. Die Genauigkeit gibt an, wie viele der gewünschten Bilder richtig klassifiziert wurden:

$$\text{Accuracy} = \frac{\text{Anzahl richtig klassifizierter Elemente}}{\text{Gesamtzahl der Elemente}}$$

In der zeitlichen Auswertung erweist sich die Genauigkeit als schlechter Messwert, denn durch die (erfreulich) niedrige Prävalenz von Staus hat ein Klassifizierer, der niemals einen Stau findet, immer noch eine sehr hohe Genauigkeit ohne in der Praxis nützlich zu sein (vgl. Auswertung im Kapitel 3.4.3).

Weitere Kennzahlen Deswegen haben wir zur Evaluierung der trainierten neuronalen Netze einige weitere Kennzahlen herangezogen, die dem Fachgebiet des Information Retrieval entspringen (vgl. [Manning et al., 2008, S.142f]).

Diese Kennzahlen sind ursprünglichen auf ein Suchsystem definiert, lassen sich jedoch gut auf unsere Stauerkennung übertragen. Zur Berechnung weiterer Kennzahlen muss jedes von der Stauerkennung klassifizierte Bild in eine der folgenden Kategorien eingeteilt werden:

- **True Positive** Das Netz hat einen Stau erkannt, wo auch ein Stau war
- **True Negative** Das Netz hat fließenden Verkehr erkannt, wo der Verkehr fließend war
- **False Positive** Das Netz hat einen Stau erkannt, obwohl fließender Verkehr war
- **False Negative** Das Netz hat fließenden Verkehr erkannt, obwohl Stau war

Aus diesen 4 Kategorien lassen sich dann folgenden Kennzahlen errechnen [Manning et al., 2008, S.142ff]:

Precision Gibt an, wie viele der gefundenen Staus auch tatsächlich richtig waren:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall Gibt an, wie viele der tatsächlich existierenden Staus gefunden wurden:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Precision und Recall sind oft gegensätzlich: Ein Klassifizierer, der häufig auf Stau plädiert, wird einen sehr guten Recall haben und eine schlechte Precision, während ein zurückhaltender Klassifizierer vielleicht eine perfekte Precision bei gleichzeitig schlechtem Recall hat. So fällt beispielsweise die Karte der Straßenverkehrszentrale in der Auswertung in Kapitel 3.4.2 durch eine sehr gute Precision auf, der Recall lässt jedoch zu wünschen übrig. Für die Stauerkennung wollen wir natürlich beide Kennzahlen optimieren: Bei zu niedrigem Recall wird der Benutzer nicht zuverlässig genug vor einem Stau gewarnt, bei einer zu niedrigen Precision wird unser Benutzer unnötig oft von der Autobahn geleitet.

F1 Score Verrechnet Precision und Recall zu einem Wert:

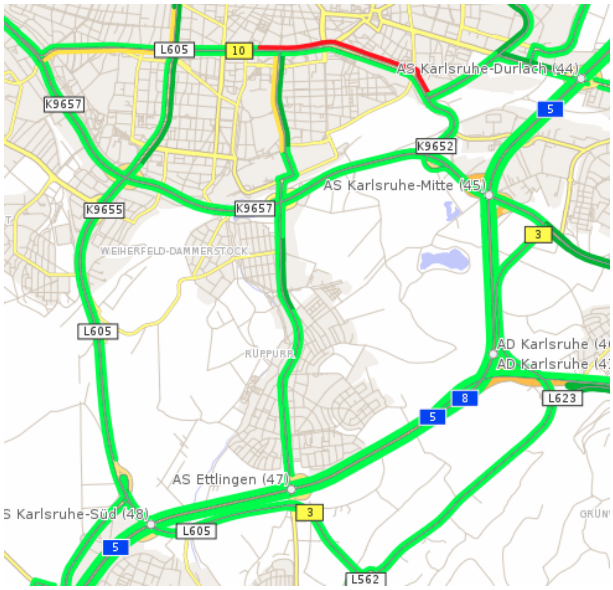
$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Der F1-Wert ist gut geeignet, um einen Kompromiss zwischen Precision und Recall auszuloten. Wir verwenden den F1-Wert als maßgebliches Kriterium für die Qualität unserer Stauerkennung.

3.4.2 Evaluierung am Validierungsdatensatz

Alle in den vorherigen Abschnitten beschriebenen Netze wurden mit den Validierungsbildern aus dem Trainingsdatensatz geprüft. Als Basis-Vergleichswerte dient eine Auswertung der Farbmarkierung in der Karte der Straßenverkehrszentrale (rot markierte Straße entspricht Stau, vgl. Abbildung 3.11). Netz 2 wurde zusätzlich mit unmaskierten Bildern trainiert, um die Nützlichkeit der Maskierung (vgl. Kapitel 3.1) zu überprüfen.

Abbildung 3.11: Ausschnitt aus der Verkehrskarte der Straßenverkehrszentrale



Quelle: `https://svz-bw.de`, abgerufen am 10. Mai 2018

Das Ergebnis der Auswertung ist in Abbildung 3.12 zu sehen. Die Ergebnisse der schlechten Netze sind bewusst abgeschnitten, um die feinen Unterscheidungen in den erfolgreichen Klassifizierern besser hervorzuheben. Alle trainierten Netze (bis auf Netz 4) haben hier sehr gut abgeschnitten.

3.4.3 Evaluierung an realen Tagesverläufen

Der Trainingsdatensatz mit bunt zusammengewürfelten Bildern ist für den Praxiseinsatz nicht unbedingt repräsentativ. Um den Praxiseinsatz besser zu simulieren, haben wir für 10 Tage alle Bilder vollständig manuell klassifiziert. Diese Kollektion aus dem Zeitraum

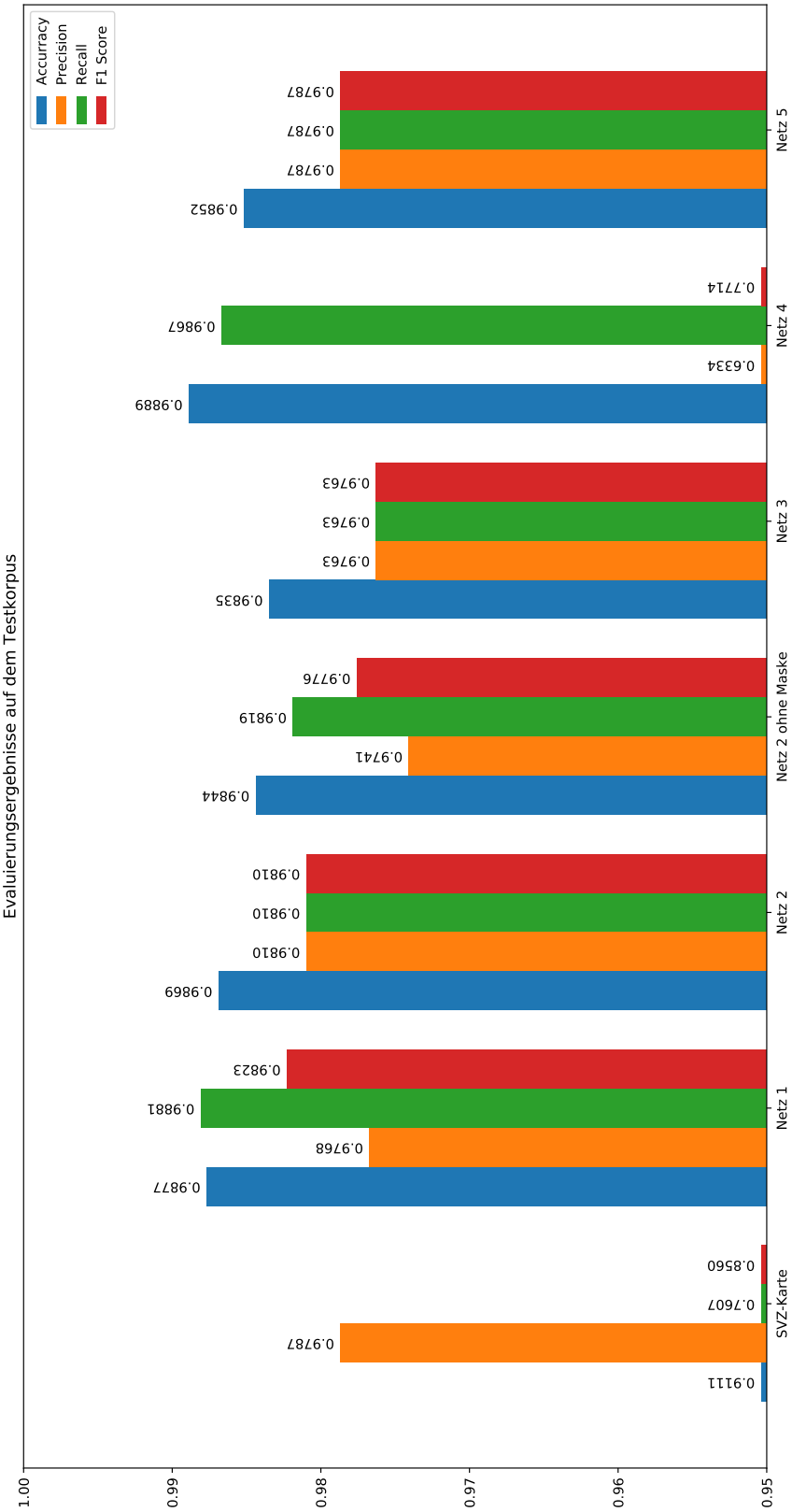


Abbildung 3.12: Evaluierungsergebnisse auf dem Validierungsdatensatz

Quelle: Eigene Darstellung

vom 6.11.2017 bis einschließlich 15.11.2017 enthält genau 2090 Kamerabilder mit Staus und 109468 mit fließendem Verkehr[3]

Die ungefilterte Anwendung aller von uns trainierten neuronale Netze fällt durch eine sehr hohe Rate an false positives auf. Aus stichprobenhaftigen Untersuchungen lässt sich die Hypothese ableiten, dass das Netz kurzzeitige Verdichtungen des Verkehrs als Stau erkennt, selbst wenn auf dem vorherigen und nächsten Bild eindeutig kein Stau zu sehen ist (und das Bild demnach auch in der händischen Klassifizierung nicht als Stau markiert wurde), oder kurzzeitige Bewegung im Stau bereits als flüssigen Verkehr interpretiert, obwohl auf dem vorherigen und nachfolgendem Bild der Stau klar erkennbar ist. Abhilfe schaffen aus der Statistik bekannte Verfahren zur Glättung einer Zeitreihe. Wir haben eine exponentielle Glättung der Stauwahrscheinlichkeit ($\bar{x_i} = \alpha x_i + (1 - \alpha) x_{i-1}$) angewendet.

Das Ergebnis für die trainierten Netze 1-5 und die Karte der Straßenverkehrszentrale als Referenz ist in Abbildung 3.13 zu sehen. Dort sind die Kennzahlen Accuracy, Precision, Recall und der F1-Wert in Abhängigkeit des Glättungsfaktors α dargestellt. Der Eintrag bei $\alpha = 1$ entspricht dem Ergebnis ohne Glättung.

Aus dem Schaubild ergibt sich, dass die Glättung nicht nur sehr sinnvoll ist, sondern auch durch alle Netze hinweg bei einem Glättungsfaktor um $0{,}2 < \alpha < 0{,}3$ einen optimalen F1-Wert erreicht. Am besten abgeschnitten hat das Netz 1 mit einem F1-Wert von über 0,95 bei einem Glättungsfaktor $\alpha = 0{,}3$. Dieses Netz wird deswegen auch im Produktiveinsatz verwendet.

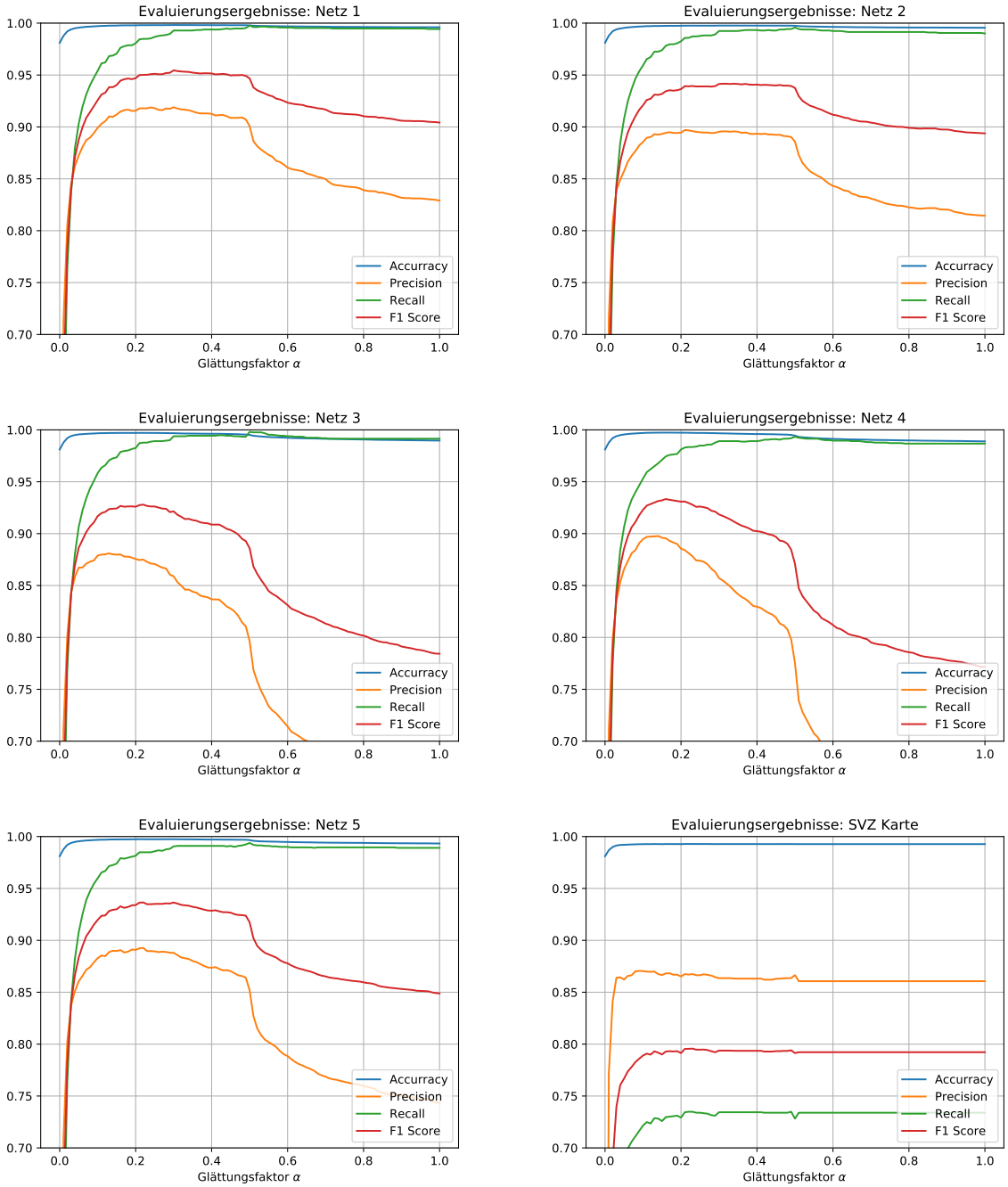
Ein Nachteil der Glättung ist, dass das neuronale Netz einen Stau potentiell einige Minuten zu spät erkennt. Die Zahl von false positives reduziert sich im Test jedoch drastisch (d.h. die Precision steigt, manchmal auf Kosten des Recalls), sodass dies als ein sehr guter Kompromiss erscheint.

3.4.4 Evaluierung für einzelne Ausfahrten

Das Netz 1 wurde zusätzlich noch einzeln für jede Kameraposition ausgewertet, einmal für die Bilder im Validierungsdatensatz aus Kapitel 3.4.2 und einmal für die Tagesverläufe

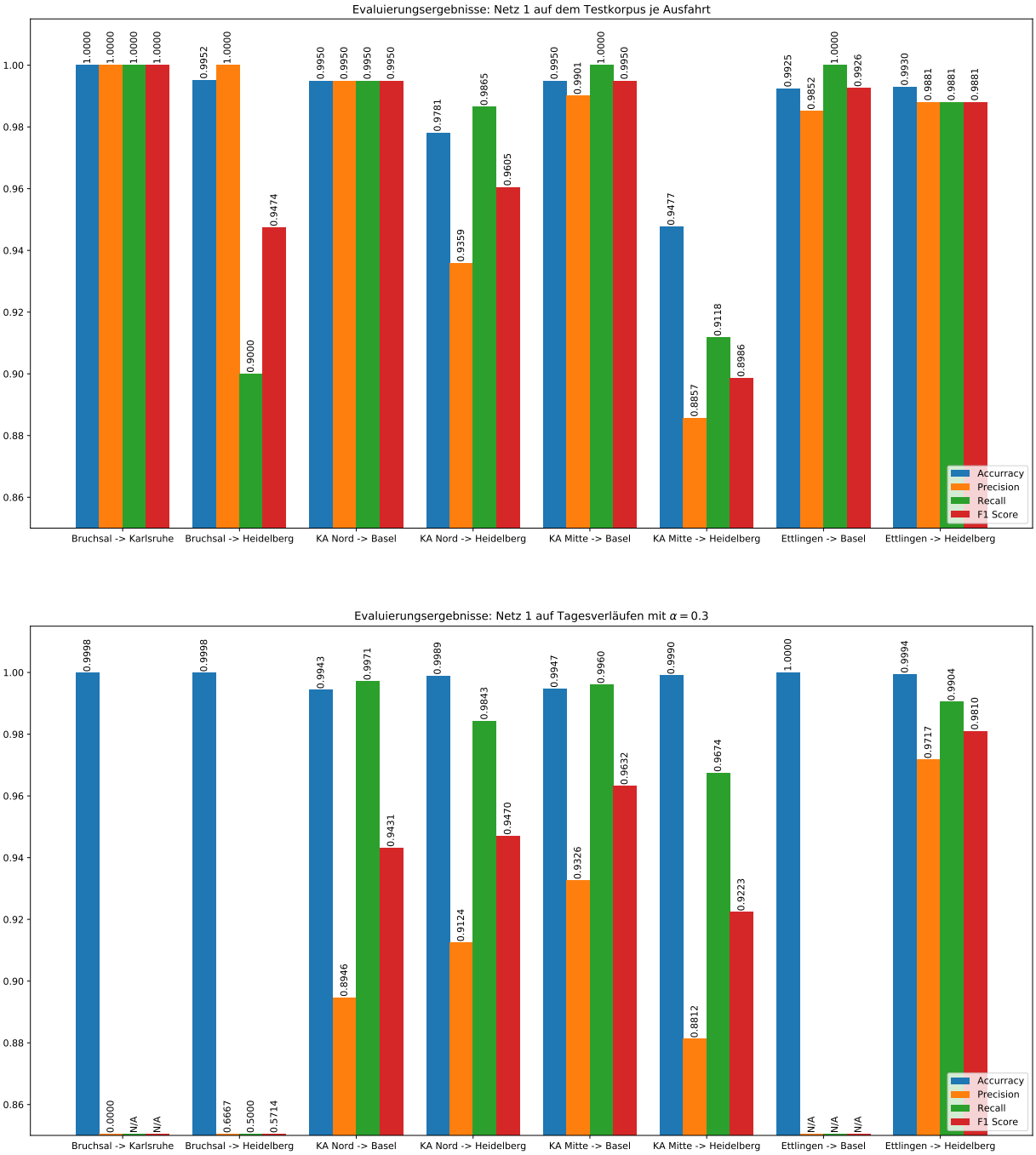
[3]Download: `https://mega.nz/#!oM1zySpZ!v9iPpbP5OQ30gX-Owtv3JM2S4wymWiQ21M4Z30OkoB0`

Abbildung 3.13: Evaluation mit exponentieller Glättung



Quelle: Eigene Darstellung

Abbildung 3.14: Evaluation aufgeschlüsselt nach Ausfahrt



`N/A` bedeutet, dass hier eine Division durch 0 erfolgt wäre, weil kein einziger Stau an der Stelle im Zeitraum beobachtet bzw. erkannt wurde.

Quelle: Eigene Darstellung

aus Kapitel 3.4.3 mit einem Glättungsfaktor $\alpha = 0.3$. Das Ergebnis ist in Abbildung 3.14 zu sehen.

Aus der ersten Teilauswertung ergibt sich, dass das Netz an den Anschlussstellen Karlsruhe Nord, Karlsruhe Mitte und Bruchsal Fahrtrichtung Heidelberg nicht so gut funktioniert wie an den anderen Kameras. Die zweite Teilauswertung ist hier nur begrenzt aussagekräftig, weil in Bruchsal Fahrtrichtung Basel und in Ettlingen Fahrtrichtung Basel im Auswertungszeitraum kein Stau aufgetreten ist, in Bruchsal Fahrtrichtung Heidelberg nur genau 4 Bilder mit Stau (von denen das Netz nur zwei erkannt hat).

4 Backend

4.1 Datenmodell

In diesem Kapitel wird das Datenmodell des Backends beschrieben. Eine grafische Repräsentation in Form eines Entity-Relationship-Modells (ERM) finden Sie in Abbildung 4.1

CameraSpotConfigEntity Diese Entitäten sind die Grundlage, auf denen sämtliche Services operieren. Eine `CameraSpotConfigEntity` repräsentiert eine Kamera mit sämtlichen hierfür nötigen Attributen. Als Primärschlüssel wird wie für fast alle Entitäten eine UUID verwendet, da diese garantiert eindeutig ist und daher extern festgelegt werden kann. In unserem Fall werden diese Entitäten in der Datenbank angelegt und stellen alle von der Anwendung unterstützten Kameras dar. Als weitere Attribute enthält diese Entität die Ids der Kameras, welche verwendet werden um die entsprechenden Bilder von der Straßenverkehrszentrale Baden-Württemberg abzurufen, einen Namen für die Darstellung in der Android App, die `ImageMaskEntity`s für Vorder- und Rückseite, sowie den Längen- und Breitengrad der Kameraposition. Als Fremdschlüssel sind eine `StreetEntity`, eine Menge von `ExitSpotConfigEntity`s als letzte, alternative Ausfahrt vor dieser Kamera und eine `CameraSpotConfigEntity`, welche die Kamera für die andere Fahrtrichtung repräsentiert, vorhanden.

StreetEntity Eine `StreetEntity` stellt eine Autobahn dar und ist sehr einfach aufgebaut. Der Primärschlüssel ist eine UUID und das einzige andere Attribut der Name der Autobahn. Die Idee hinter dieser Entität und der zugrundeliegenden Information ist eine mögliche Sortierung der Kameras in der Android App.

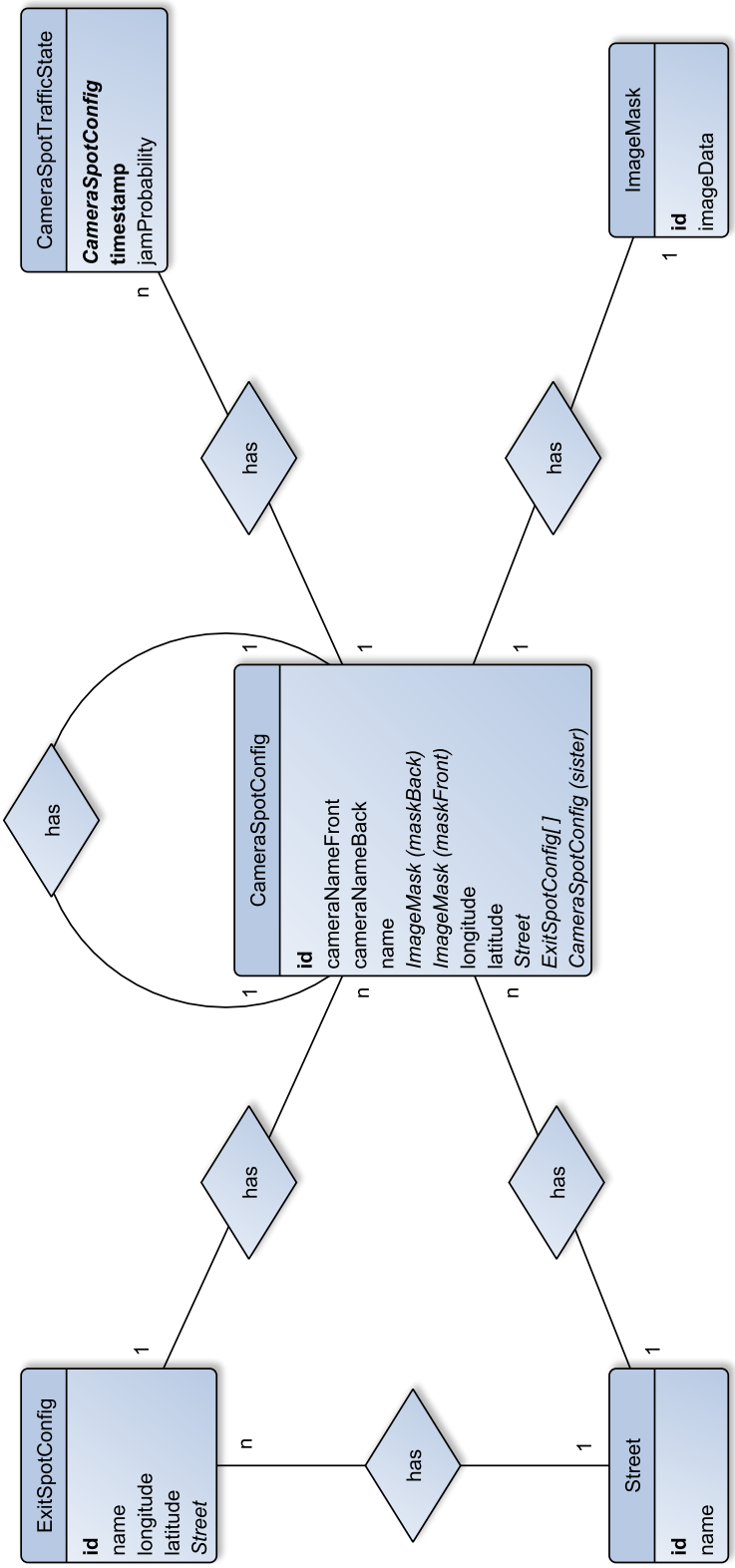


Abbildung 4.1: Entity-Relationship-Modell

Quelle: Eigene Darstellung

ExitSpotConfigEntity Diese Entität stellt eine Autobahnausfahrt dar. Auch hier wird als Primärschlüssel eine UUID verwendet. Die anderen Attribute sind der Name der Ausfahrt, der Längen- und Breitengrad der Ausfahrt, sowie ein Fremdschlüssel auf die `StreetEntity`, welche die Autobahn, auf der diese Ausfahrt liegt, darstellt.

CameraSpotTrafficStateEntity Eine `CameraSpotTrafficStateEntity` repräsentiert einen klassifizierten Eintrag in der Datenbank. Der Primärschlüssel ist ein zusammengesetzter Schlüssel aus dem aktuellen Timestamp und der `CameraSpotConfigEntity`, welcher dieses Ergebnis zugeordnet wird. Das einzige andere Attribut ist die Stauwahrscheinlichkeit, welche von dem neuronalen Netz zurückgeliefert wurde. Diese Entität ist die einzige, welche Bewegungsdaten in der Datenbank repräsentiert und welche zur Laufzeit des Systems automatisch angelegt wird. Alle anderen Entitäten werden einmal in der Datenbank eingepflegt und von der Anwendung selbst nur lesend verarbeitet.

ImageMaskEntity Diese Entität ist ebenfalls sehr einfach aufgebaut. Der Primärschlüssel ist der Name der Ausfahrt inklusive der Fahrtrichtung und das einzige Attribut die Maske als Blob.

4.2 Architektur

4.2.1 Microservices

Idee Die Idee der Microservice Architektur ist es, eine Anwendung als System mehrerer Services aufzubauen. Diese werden jeweils in einem eigenen Prozess ausgeführt und sind damit unabhängig von anderen Services. Die Kommunikation wird häufig über das Internet abgebildet, zum Beispiel über Remote Procedure Calls oder klassisch über HTTP Requests und Responses. Hierfür definiert jeder Service eine klare API, die dann von den anderen angesprochen werden kann. [Lewis and Fowler, 2014] Der Ansatz von Microservices verfolgt das Single Responsibility Principle, welches von Robert C. Martin definiert wurde. Jede Klasse, bzw. jedes Modul oder in diesem Fall jeder Service sollte genau einen Grund haben, verändert zu werden. [Martin, 2009, S.138] Daraus ergibt sich implizit, dass jeder Service

nur eine Aufgabe übernehmen kann und damit in aller Regel einfach und übersichtlich gehalten ist.

Evolutionsfähigkeit Im Gegensatz zu einem monolithischen System, in welchem die gesamte Anwendung in einem einzigen Prozess untergebracht ist, sorgt die Microservice Architektur für eine klare Trennung der Zuständigkeiten der einzelnen Services. Durch diese Trennung ist der Einflussbereich einer Änderung nur sehr lokal und endet meist an den Grenzen des Microservices. Daraus ergeben sich einige Vorteile für die Evolution der Software. Da die Services unabhängig voneinander sind, können sie auch unabhängig voneinander weiterentwickelt werden. Solange die öffentliche Schnittstelle eines Services nicht verändert wird, kann dieser isoliert von allen anderen modifiziert und neu deployed werden. Auch ein komplettes Neuschreiben oder Austauschen eines Services ist denkbar. Im Falle des Monolithen müsste die gesamte Anwendung neu gebaut und ausgeliefert werden. [Lewis and Fowler, 2014]

Inhomogenität Ein Vorteil der Kommunikation über klare Schnittstellen und eines unabhängigen Mediums wie dem Internet ist die Möglichkeit, das System aus inhomogenen Services zusammenzusetzen, das heißt einzelne Services können mit grundverschiedenen Technologien und in verschiedenen Programmiersprachen implementiert sein. Die einzige Anforderung an den Service ist eine klare öffentliche Schnittstelle, welche sich an die Regeln zur Kommunikation des Systems hält. Die interne Implementierung des Services hat keinen Einfluss auf andere Services. In einem monolithischen System mit nur einem Prozess ist diese Möglichkeit nicht gegeben. [Newman, 2015, S.24f]

Testbarkeit Durch diese Trennung zwischen den Services in Kombination mit der geringen Größe und der Tatsache, dass nur eine Aufgabe pro Service implementiert sein soll, ergibt sich automatisch eine gute Testbarkeit des Services, da dessen Aufgabe isoliert getestet werden kann. Im Gegensatz dazu ist die Testbarkeit des Gesamtsystems durch die erhöhte Komplexität und die Kommunikation über Prozessgrenzen hinweg schwerer, als es in einem monolithischen System der Fall wäre.

Skalierbarkeit Ein weiterer großer Vorteil ist die horizontale Skalierbarkeit des Systems, welche mit der Trennung der Services einhergeht. Sollte eine bestimmte Funktion des Gesamtsystems nicht performant genug sein, kann genau der eine Service, welcher hierfür zuständig ist, mit mehreren Instanzen gestartet werden. Bei einem Monolithen ist die Duplizierung des gesamten Systems nötig, obwohl nur ein kleiner Teil davon ausgereicht hätte um die Performance sicherzustellen. [Newman, 2015, S.26f]

Fehlertoleranz Fehlertoleranz kann dadurch sichergestellt werden, dass Services fehlertolerant gegenüber dem Ausfall anderer Services sind. Sollte ein Service ausfallen, ist nur dessen Funktionalität und somit nur einen kleinen Teil des Gesamtsystems nicht verfügbar. Alle anderen Funktionen des Systems sind allerdings davon nicht betroffen. In einem monolithischen System sorgt der Ausfall einer Funktion für den Ausfall des Gesamtsystems, da die Anwendung, das heißt der eine Prozess, normalerweise terminiert. [Newman, 2015, S.26]

Netzwerk Traffic Ein Nachteil dieser Architektur ist der erhöhte Netzwerk Traffic im Vergleich zu einem monolithischen System, welches über in-memory Variablen Daten austauschen kann, da sämtliche Daten zwischen den Komponenten über ein Netzwerk übertragen werden müssen. Außerdem entsteht die Gefahr eines Netzwerk-Ausfalls zusätzlich zum Ausfall einzelner Services. [Lewis and Fowler, 2014]

REST REST steht für Representational State Transfer. Die Idee hinter REST ist die Fokussierung auf Ressourcen und deren Darstellungsformen. Im REST Umfeld wird daher keine Business-Funktionalität angesprochen, sondern nur Ressourcen abgerufen, angelegt, modifiziert oder gelöscht. Dabei ist jede Ressource über eine eindeutige URI identifiziert. Die URL, welche eine Methode des REST Services aufruft, wird Endpoint genannt. Mit Hilfe von Links wird dem Client der Zusammenhang der einzelnen Ressourcen vermittelt. [Dazer, 2012]

HTTP Wenn es auch nicht explizit vom REST-Architektur-Stil gefordert wird, so hat sich doch die Verwendung von HTTP als zugrundeliegende Übertragungsform etabliert. Das liegt in erster Linie daran, dass HTTP einige Parallelen zu REST aufweist, wie zum Beispiel

die Verwendung von URLs zur Identifikation von Ressourcen beziehungsweise Dateien, Links zur Navigation zwischen Inhalten oder die Verwendung der HTTP Verben `GET`, `POST`, `PUT` und `DELETE`, welche genau die von REST vorgesehenen Zugriffsmöglichkeiten auf Ressourcen abbilden. [Dazer, 2012]

Repräsentation REST ist unabhängig von der Darstellungsform der Ressourcen. Im Beispiel von HTTP-GET wird die Ressource vom Client über ihre eindeutige URI angesprochen und vom Server zum Client übertragen. Dabei schreibt REST keine Darstellungsform vor, sondern Server und Client müssen sich zum gegenseitigen Verständnis auf eine Darstellung einigen. Hierfür bieten sich die `accept` und `content-type` Header von HTTP an. Prinzipiell sind alle Darstellungsformen wie zum Beispiel png, gif oder pdf bis hin zu plain text denkbar. Die häufigsten Darstellungsformen sind dabei HTML zur Darstellung von Websites im Browser und XML oder JSON zur Übertragung von Datenobjekten. [Dazer, 2012]

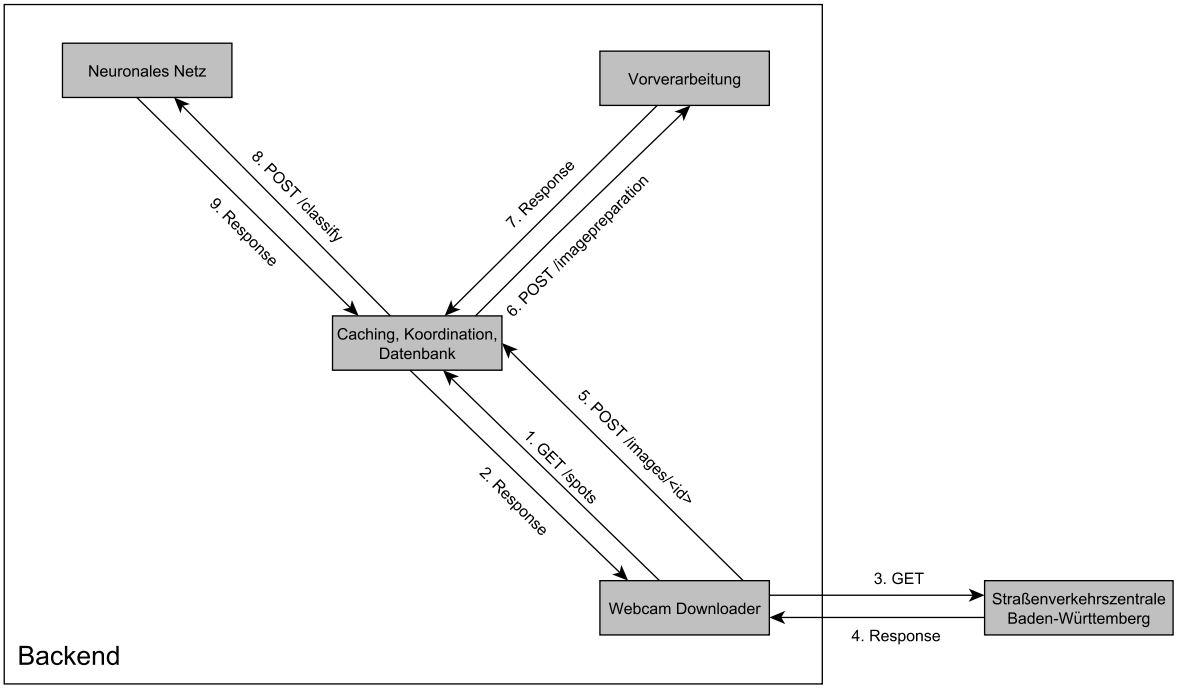
Zustandslosigkeit Ein Server, welcher seine Ressourcen über eine REST-Schnittstelle zur Verfügung stellt, sollte möglichst `stateless`, das heißt zustandslos, sein. Der Hauptgrund hierfür liegt in der Möglichkeit URLs abzuspeichern und wiederzuverwenden oder das Ergebnis einer Anfrage in einem Cache abzulegen. Die Abbildung von Business-Funktionen ist durch die Reihenfolge der REST-Aufrufe gegeben, das heißt der Zustand muss vom Client gehalten werden, um diese richtige Reihenfolge sicherzustellen. [Dazer, 2012]

Hypermedia Die Idee hinter Hypermedia ist es, dem Client durch Links in der Antwort auf eine Anfrage mögliche weitere Schritte aufzuzeigen. Im Fall von HTTP-PUT könnte dies zum Beispiel der Link zu der neu angelegten Ressource sein. Durch diese Links wird der Client dabei unterstützt, valide Zustandswechsel zu finden und durchzuführen und damit Business-Funktionen abzubilden. [Dazer, 2012]

4.2.2 Implementierung

Für unsere Anwendung sind serverseitig 4 Services entstanden, welche untereinander über REST-Schnittstellen kommunizieren. Diese Schnittstellen wurden mit Hilfe eines

Abbildung 4.2: Ablauf für die Bildklassifizierung



Quelle: Eigene Darstellung

Frameworks umgesetzt. Weitere Informationen zu diesem Framework können in Kapitel 4.3 nachgelesen werden. Eine Grafik, welche die Services und deren Requests untereinander zeigt, ist in Abbildung 4.2 dargestellt.

Kamerabilder abrufen Der erste Service ist dafür zuständig, in regelmäßigen Abständen Bilder der Autobahnausfahrten von der Straßenverkehrszentrale Baden-Württemberg herunterzuladen. Welche Kameras dabei unter welcher URL verfügbar sind, wird aus der Datenbank abgerufen. Das zugrundeliegende Datenmodell ist in Kapitel 4.1 detailliert beschrieben. Die so erhaltenen Bilder werden allerdings nicht gespeichert, sondern über einen POST-Request an den nächsten Service weitergeleitet. Dieser POST-Request enthält dabei immer zwei zusammengehörige Bilder, welche einmal die relevante Fahrtrichtung von vorne und einmal von hinten zeigen. Diese Bilder werden in der weiteren Verarbeitungskette immer gemeinsam verarbeitet.

Dieser Service stellt selbst keine REST Endpoints zur Verfügung, sondern ruft nur andere Services und die Bilder der Straßenverkehrszentrale ab.

Datenbankzugriff, Caching und Koordination Der zweite Service dient in erster Linie der Koordination des Ablaufs und der Datenbankverwaltung. Im Gegensatz zur klassischen Microservice-Architektur verwenden wir nur eine Datenbank, auf welche allerdings nur von diesem Service aus zugegriffen wird. Alle anderen Services und die Android App verwenden einen REST Endpoint, um die entsprechenden Ressourcen abzufragen.

Dieser Service nimmt nun die Bilder entgegen, reichert den Request mit den Masken an und leitet diesen direkt an die Vorverarbeitung weiter. Das Ergebnis dieser Vorverarbeitung wird dann an das neuronale Netz zur Klassifizierung weitergegeben. Aus der daraus erhaltenen Stauwahrscheinlichkeit wird zusammen mit den beiden zugehörigen Bildern im Cache abgelegt um Datenbankzugriffe zu sparen. Außerdem wird für die Stauwahrscheinlichkeit ein Datenbankeintrag angelegt.

Der Service selbst stellt die folgenden REST Endpoints zur Verfügung:

- /images/<id>, POST: Nimmt die beiden zusammengehörigen Bilder (Sicht von vorne und von hinten) für die Kamera mit der gegebenen Id entgegen. Die Methode hat keinen Rückgabewert.
- /mask/<id>, GET: Gibt die Maske für die gegebene Id als `image/png` zurück.
- /mask/<id>, POST: Speichert die gegebene Maske unter der gegebenen Id in die Datenbank. Dieser Endpoint dient dem Hinzufügen weiterer Masken und ist mit Credentials gesichert.
- /spots/, GET: Gibt alle gespeicherten Kameras-Entitäten zurück.
- /spots/<id>, GET: Gibt die Kamera-Entität für die gegebene Id zurück.
- /spots/, POST: Speichert die gegebene Kamera-Entität in der Datenbank ab. Dieser Endpoint dient dem Hinzufügen weiterer Kameras und ist mit Credentials gesichert.
- /exit/, GET: Gibt alle gespeicherten Ausfahrten zurück.
- /exit/<id>, GET: Gibt die Ausfahrt für die gegebene Id zurück.

- /exit/, POST: Speichert die gegebene Ausfahrt in der Datenbank ab. Dieser Endpoint dient dem Hinzufügen weiterer Ausfahrten und ist mit Credentials gesichert.
- /street/, GET: Gibt alle gespeicherten Straßen zurück.
- /street/<id>, GET: Gibt die Straße für die gegebene Id zurück.
- /street/, POST: Speichert die gegebene Straße in der Datenbank ab. Dieser Endpoint dient dem Hinzufügen weiterer Straßen und ist mit Credentials gesichert.
- /info/html GET: Erzeugt die Debug Seite, welche alle aktuellen Bilder mit zugehöriger Stauwahrscheinlichkeit für alle Kameras anzeigt.
- /info/<id>/frontimg, GET: Gibt das Bild für die Frontalansicht der gegebenen Kamera zurück.
- /info/<id>/backimg, GET: Gibt das Bild für die Rückansicht der gegebenen Kamera zurück.
- /traffic/camera/<id>, GET: Gibt die aktuelle Stauwahrscheinlichkeit für die gegebene Kamera zurück.
- /traffic/bulk, GET: Gibt die Stauwahrscheinlichkeiten für die per Query Parameter übergebenen Kameras zurück. Wird verwendet, damit die Android App nicht pro Kamera einen eigenen Request senden muss.
- /traffic/bulk-front-image, GET: Gibt die Frontalbilder für die per Query Parameter übergebenen Kameras zurück. Wird verwendet, damit die Android App nicht pro Kamera einen eigenen Request senden muss.

Vorverarbeitung Wie in Kapitel 3.1 beschrieben, liefert das neuronale Netz bessere Ergebnisse, wenn auf dem Bild nur die relevante Fahrtrichtung sichtbar ist. Ein für die Vorverarbeitung zuständiger Service führt diese Maskierung durch. Das zu maskierende Bild wird zusammen mit der passenden Maske empfangen, die Maske über das Bild gelegt und dann zurück an den Aufrufer gesendet.

Die folgenden REST Endpoints sind für diesen Service verfügbar:

- /imagepreparation/, POST: Nimmt ein rohes Bild und die zugehörige Maske entgegen und legt die Bilder übereinander. Das Ergebnis wird als Response-Body zurückgesendet.

Klassifizierung Der nächste Service in der Verarbeitungskette ist das neuronale Netz zur Stauerkennung. Das neuronale Netz ist in Kapitel 3 detailliert beschrieben. Das neuronale Netz nimmt die beiden maskierten Bilder entgegen und gibt eine ermittelte Stauwahrscheinlichkeit an den Aufrufer zurück.

Dieser Service hat den folgenden Endpoint:

- /classify/, POST: Dieser Endpoint nimmt ein maskiertes Bild entgegen und leitet dieses zur Klassifizierung in das neuronale Netz. Des Ergebnis wird als Response-Body zurückgegeben.

4.3 Framework Evaluierung

Alle Microservices verwenden das gleiche Framework, um ihre REST-Schnittstellen zu definieren. Da pro Service ein eigener Prozess gestartet wird und jeder Service in einer eigenen Jar-Datei gepackt ist, sollte das Framework leichtgewichtig sein um die Dateien möglichst klein zu halten. Abgesehen davon, sollte das Framework einfach verwendbar sein, um den Einstiegsaufwand gering zu halten.

4.3.1 Javalin

Javalin ist ein Open-Source Framework für leichtgewichtige REST Schnittstellen. Die gesamte Konfiguration der REST Endpoints findet dabei über Java oder Kotlin Quellcode statt, sodass keine XML Dateien oder ähnliches benötigt werden. Javalin basiert stark auf Lambdas und sogenannten Fluent Interfaces, sodass eine deklarative, gut lesbare Konfiguration möglich ist. Ein Beispiel hierfür ist in Programmcode 4.1 zu sehen. [Javalin, 2018]

```
Javalin app = Javalin.create()
    .enableStandardRequestLogging()
    .enableDynamicGzip()
    .port(port)
    .start();

app.routes(() -> {
    path("users", () -> {
        get(UserController::getAllUserIds);
        post(UserController::createUser);
        path(":user-id", () -> {
            get(UserController::getUser);
            patch(UserController::updateUser);
            delete(UserController::deleteUser);
        });
    });
});
```

Programmcode 4.1: Beispiel für eine REST Schnittstelle mit Javalin

Endpoint-Definition Durch die Benutzung von Method References in Kombination mit Dependency Injection Mechanismen entsteht dabei eine vollständige Trennung zwischen Definition der REST Endpoints und der eigentlichen Anwendungslogik. Es werden keine zusätzlichen Klassen oder Annotationen im Quellcode benötigt. Die komplette Trennung ist zwar wünschenswert, birgt aber auch das Risiko, Methoden zu verändern, ohne direkt zu sehen, dass es sich um eine öffentliche Schnittstelle handelt.

Runtime Javalin wird direkt mit einer Server Runtime ausgeliefert. Beim Ausführen der Anwendung wird automatisch ein Jetty Server gestartet und die Endpoints unter der angegebenen Adresse und dem angegebenen Port zur Verfügung gestellt. Das Framework selbst hat durch seine geringe Größe nahezu keinen Einfluss auf die Leistung, sodass die Performance sich kaum von einer reinen Jetty Runtime unterscheidet. [Javalin, 2018]

Exception Handling Das Exception Handling ist mit Javalin sehr gut möglich und wird zentralisiert an einer Stelle festgelegt. Dabei werden Exception Mapper registriert, welche

jeweils für eine Exception Klasse (oder Subklassen davon) zuständig sind und das Mapping in eine geeignete HTTP Response übernehmen. Die eigentliche Anwendungslogik ist dann frei von Exception Handling. Geworfene Exceptions werden vom Framework gefangen und entsprechend der Exception Mapper verarbeitet. [JavalinDoc, 2018, Exception Mapping]

Dokumentation Die Dokumentation des Frameworks ist auf dem aktuellen Stand und ermöglicht es, sehr schnell eine lauffähige Anwendung zu erstellen, ist aber nicht ausführlich genug, wenn mehr als nur eine kleine Basis geschaffen werden soll. Das Framework hat außerdem aktuell nur eine kleine Anwendergruppe, sodass wenig bis keine Hilfe in Foren oder ähnlichem zu finden ist.

Modularität Javalin hat keinen modularen Aufbau, sodass immer die gesamte Bibliothek eingebunden werden muss, auch wenn der Großteil der Funktionen nicht benötigt wird. Da das Framework selbst allerdings sehr leichtgewichtig ist, ist dies vernachlässigbar.

Übersichtlichkeit Ein größeres Problem ist die Übersichtlichkeit bei wachsender Anzahl Endpoints. Ab einem gewissen Punkt (Anzahl oder Pfadlänge) werden die Fluent Interfaces eher zum Hindernis und die Lesbarkeit leidet deutlich. Da für unser Projekt allerdings nur einige wenige Endpoints benötigt werden, ist dies akzeptabel.

Testbarkeit Die Testbarkeit ist bei Javalin nicht gegeben und muss selbst implementiert werden. Zum Testen der Schnittstellen muss ein externer REST Client implementiert oder eingebunden werden, da Javalin selbst keinen Client bereitstellt.

4.3.2 Jersey

Auch Jersey stellt ein Open-Source Framework für REST Schnittstellen dar und implementiert den JAX-RS Standard. Dabei werden allerdings zusätzliche Features eingefügt, um das Definieren der Endpoints zu vereinfachen. Da Jersey JAX-RS implementiert, basiert das Framework stark auf Annotationen, sodass eine Trennung zwischen Anwendungslogik und Endpoint Definition nicht vollständig möglich ist. Dabei werden pro Endpoint in der

Regel drei oder mehr Annotationen pro Methode benötigt. Ein Beispiel für einen REST Endpoint ist in Programmcode 4.2 dargestellt. Der Einstieg in Jersey ist dank Maven Archetypes sehr einfach. [Jersey, 2018]

```
@Path("/somepath")
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response postMethod(Request request) {
    ...
}
```

Programmcode 4.2: Beispiel für eine REST Schnittstelle mit Jersey

Endpoint-Definition Die Definition der Endpoints findet per Annotation direkt an den aufgerufenen Methoden statt. Damit werden die Endpoints nicht wie bei Javalin zentral definiert, sondern über mehrere Klassen verteilt. Dies bringt den Vorteil, dass die Definition der Schnittstelle direkt an der aufgerufenen Methode ist und somit klar erischtlich ist, welche Methoden zur öffentlichen Schnittstelle gehören und deren Signatur damit nicht verändert werden sollte.

Runtime Jersey bietet Unterstützung für die meisten bekannten Server Runtimes wie Grizzly oder Jetty. Welche davon verwendet wird, wird über die Wahl der Dependencies festgelegt.

Exception Handling Das Exception Handling in Jersey ist durch klare Trennung gelöst. Dabei wird pro Exception Typ eine eigene Klasse benötigt, welche das Mapping auf eine HTTP Response übernimmt. Bei vielen Exceptions erzeugt das eine Menge Boiler Plate Code, sodass die Übersichtlichkeit leidet und die Gefahr von dupliziertem Quellcode steigt. Ein Nachteil ist das interne Exception Logging. Bei unseren Tests waren teilweise Exceptions nicht im Log auffindbar, haben die Funktionalität allerdings beeinträchtigt. Die Exceptions konnten nur durch Debugging gefunden werden. Fraglich ist dabei, ob es sich um einen Fehler im Framework oder eine fehlerhafte Konfiguration unsererseits handelt. [JerseyDoc, 2018]

Dokumentation Die Dokumentation ist auf dem aktuellen Stand und sehr ausführlich, allerdings teilweise ein wenig unübersichtlich. Da Jersey ein bekanntes Framework ist, ist die Anwendergruppe entsprechend groß. Daher ist die Unterstützung durch Community-Foren und andere Hilfen vorhanden.

Modularität Jersey ist mit einer modularen Architektur aufgebaut, sodass die benötigten Features durch verschiedene Dependencies über Maven eingebunden werden können. Außerdem entsteht dadurch eine sehr gute Erweiterbarkeit, sodass bereits sehr viele Module verfügbar sind, unter anderem der für dieses Projekt benötigte JSON Support.

Übersichtlichkeit Aufgrund der Menge an benötigten Annotation werden die Klassen, welche die Endpoints definieren teilweise ein wenig unübersichtlich. Die Anzahl der Endpoints hat darauf allerdings keinen Einfluss, sodass Jersey auch für große Anwendungen gut skaliert.

Testbarkeit Die Testbarkeit wird durch den bereits inkludierte REST Client in Kombination mit dem sehr schnellen Startup (in der Regel <1 Sekunde) gegeben. Durch diesen schnellen Startup ist es möglich, die Anwendung nach jeder Testklasse neuzustarten um sicherzustellen, dass andere Tests nicht durch eventuell vorhandene Daten gestört werden.

4.3.3 Entscheidung

Trotz dessen Nachteile fiel unsere Wahl letztendlich auf Jersey als Framework. Die ausschlaggebenden Merkmale waren dabei in erster Linie die einfache Testbarkeit und der modulare Aufbau, sodass nur die wirklich benötigten Features eingebunden wurden. Dadurch konnten die Jar-Dateien möglichst klein gehalten werden. Auch der mitgelieferte REST Client ist ein Vorteil von Jersey gegenüber Javalin, da die Services in der Microservice-Architektur miteinander kommunizieren müssen. Unsere Services müssen sowohl REST Endpoints zur Verfügung stellen, als auch selber anfragen können, sodass jeder Service einen REST Client benötigt.

5 Android App

5.1 Anforderungen

5.1.1 Funktionale Anforderungen

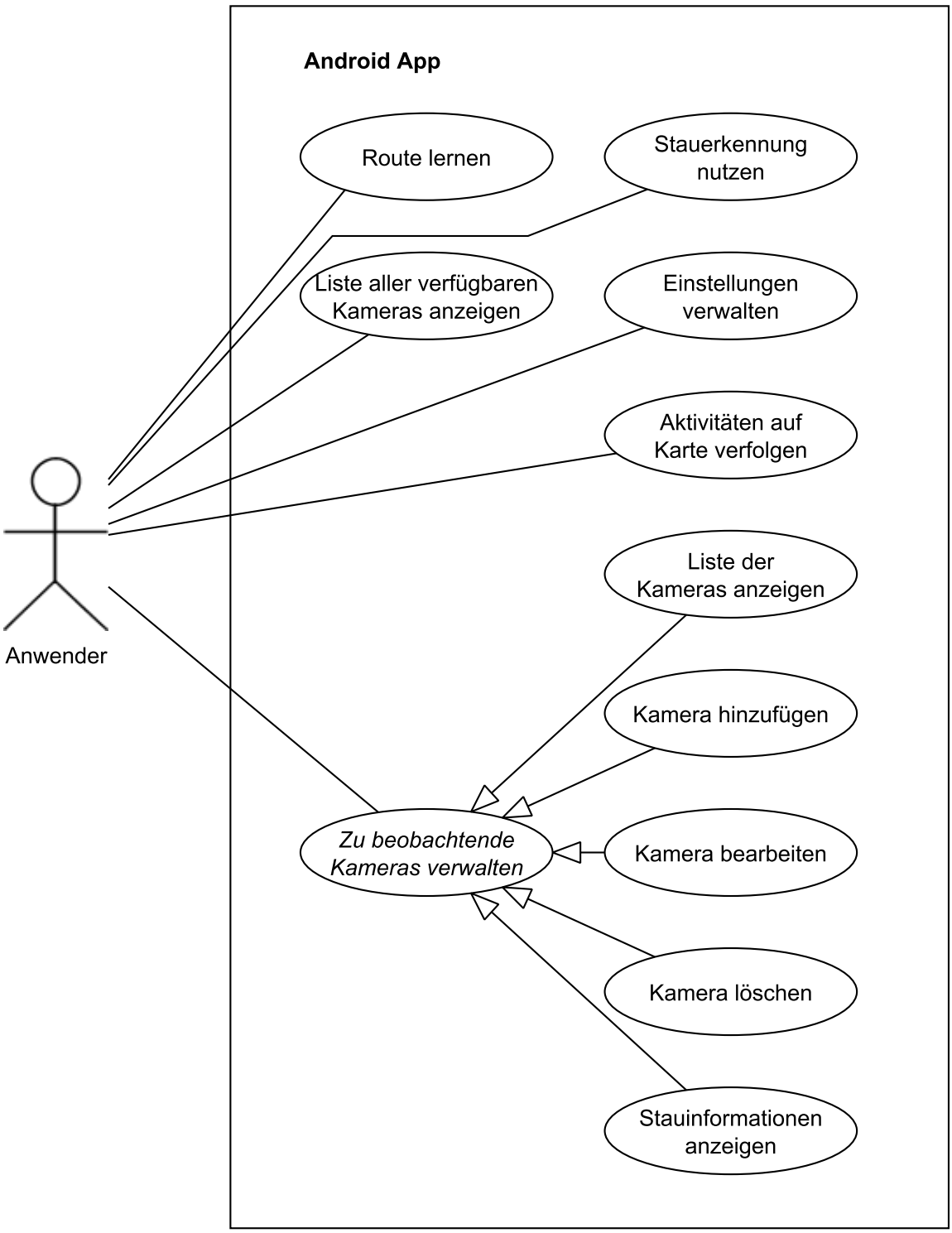
Um die in Kapitel 1.1 und 1.2 beschriebenen Probleme und Lösungsideen als Anforderungen bei der Entwicklung der Android App zu berücksichtigen, wurde ein Anwendungsfalldiagramm (engl. Use Case Diagram) erstellt (siehe Abbildung 5.1). Dieses visualisiert die Anwendungsfälle (engl. Use Cases) der App aus Sicht des Nutzers.

Im folgenden werden die einzelnen Use Cases näher beschrieben, um einen ersten Überblick über die Funktionalitäten der Android App zu geben.

Route lernen Der Anwender soll die Möglichkeit haben, die Anwendung eine Route erlernen zu lassen. Damit ist gemeint, dass der Nutzer eine für ihn relevante Strecke abfährt und die Android App die unterwegs passierten Kameras erkennt. Dabei sollen nur solche Kameras „gelernt“ werden, welche für die aktuelle Fahrtrichtung relevant sind.

Stauerkennung nutzen Beim Erreichen des Benachrichtigungspunktes einer zu beobachtenden Kamera (siehe Use Case „Zu beobachtende Kameras verwalten“) wird der Anwender über die Stauwahrscheinlichkeit an selbiger über eine Sprachnachricht informiert. Der Modus zur Stauerkennung ist nicht permanent aktiv und muss vom Anwender gezielt eingeschaltet werden.

Abbildung 5.1: Use Case Diagram der Android App



Quelle: Eigene Darstellung

Liste aller verfügbaren Kameras anzeigen Hier wird dem Anwender eine Liste aller Kameras angezeigt, die potentiell hinsichtlich eines Staus von ihm beobachtet werden können. Diese Liste wird beim Start der Android App mit dem Server abgeglichen.

Einstellungen verwalten Über das Einstellungsmenü kann der Nutzer sowohl den Geofence-Radius als auch das Polling-Intervall für die Positionsbestimmung manuell festlegen (siehe auch Kapitel 5.3.7). Die Einstellungen werden mit sinnvollen Default-Werten vorbelegt.

Aktivitäten auf Karte verfolgen Für die beiden Use Cases „Route lernen“ und „Stauerkennung nutzen“ soll die Möglichkeit bestehen, die Aktivität in Echtzeit auf einer Straßenkarte nachzuvollziehen. Auf dieser Karte sollen der aktuelle Standpunkt und die je nach Anwendungsfall beobachteten bzw. verarbeiteten Objekte (Kameras, Ausfahrten und/oder Benachrichtigungspuntke) eingezeichnet werden.

Zu beobachtende Kameras verwalten Hierbei handelt es sich um einen abstrakten Use Case, der die folgenden Use Cases „Liste der Kameras anzeigen“, „Kamera hinzufügen“, „Kamera bearbeiten“, „Kamera löschen“ und „Stauinformationen anzeigen“ gruppiert.

Liste der (zu beobachtenden) Kameras anzeigen Der Anwender soll sich eine Liste aller Kameras anzeigen lassen können, die er aktuell bezüglich eines aufkommenden Staus beobachtet.

(Zu beobachtende) Kamera hinzufügen Zusätzlich zu der Möglichkeit eine Route zu lernen, kann der Anwender eine Kamera manuell der Liste seiner zu beobachtenden Kameras hinzufügen. Hierzu wählt er aus der Liste aller Kameras (siehe Use Case „Liste aller verfügbaren Kameras anzeigen“) eine Kamera aus und legt auf einer Karte den Ort fest, an dem er über die Stauwahrscheinlichkeit an dieser Kamera benachrichtigt werden möchte. Zur Unterstützung werden die vorhergehende(n) Ausfahrt(en) und der Standpunkt der Kamera selbst als mögliche Einstiegspunkte für die Karte angeboten.

(Zu beobachtende) Kamera bearbeiten Nachdem eine Kamera zur Beobachtung angelegt wurde, kann der gewählte Benachrichtigungsort nachträglich geändert werden. Zusätzlich zu den in Use Case „(Zu beobachtende) Kamera hinzufügen“ beschriebenen Einstiegspunkten für die Karte wird hier noch der zuletzt gewählte Benachrichtigungspunkt als Auswahl angeboten.

(Zu beobachtende) Kamera löschen Eine zu beobachtende Kamera kann vom Nutzer jederzeit wieder gelöscht werden, wenn er keine Benachrichtigungen zu dieser Kamera mehr wünscht.

Stauinformationen anzeigen Auch wenn die Stauerkennung (siehe Use Case „Stauerkennung nutzen“) nicht eingeschaltet ist, soll der Anwender die Möglichkeit haben, sich für jede zu beobachtende Kamera die Stauwahrscheinlichkeit und das von dieser Kamera zuletzt gelieferte Bild anzeigen zu lassen.

5.1.2 Technische Anforderungen

Die folgenden technischen Anforderungen werden für die Android-App festgelegt:

Android SDK Version Natürlich wäre es angenehm, immer gegen die neueste Version des Android Software Development Kit (SDK) zu entwickeln (derzeit SDK-Version 27). Dadurch könnte man stets die aktuellen Funktionen der neuen Version nutzen. Allerdings würde damit auch die potentielle Zielgruppe der App stark beschränkt, da längst nicht jeder ein mobiles Endgerät besitzt, auf dem die aktuelle Android-Version lauffähig ist. In unserem Fall würden wir sogar ein Mitglied des Projektteams vom Testen der Anwendung ausschließen (SDK-Version 23). Die Wahl der minimalen geforderten SDK-Version stellt also immer ein Abwägen zwischen neuesten Features und der Größe der gewünschten Zielgruppe dar. Wir haben uns dafür entschieden, mit der Android App alle Geräte ab Android 5.0 „Lollipop“ (SDK-Version 21) aufwärts zu unterstützen. Dadurch können wir einige praktische Java 8 Features wie z.B. Lambdas nutzen. Leider bleiben uns andere Features wie z.B. Streams und Optionals verwehrt, da diese mindestens die SDK-Version 24 voraussetzen. [Google LLC, 2018i]

Unabhängige Services Die Dienste zum Lernen einer Route und zum Erkennen eines Staus sollen unabhängig von der Android App lauffähig sein. Weder das Ausschalten des Displays noch das Minimieren oder Schließen der App soll einen laufenden Dienst beeinträchtigen. Das ermöglicht dem Anwender einen maximalen Komfort, da er trotz aktiver Nutzung der Dienste andere Aktionen an seinem mobilen Endgerät durchführen kann. Das Ausschalten des Displays sorgt zudem für reduzierten Stromverbrauch und schont den Akku.

Minimierung der Paketgröße Das Herunterladen großer Apps belastet das Datenvolumen und schreckt potentielle Anwender möglicherweise davon ab, die Anwendung auszuprobieren. Zudem hat die Größe Auswirkungen auf den Stromverbrauch und die Startgeschwindigkeit der App. [Google LLC, 2018g] Aus diesen Gründen wird die Größe der Anwendung möglichst klein gehalten.

5.2 Wahl der Entwicklungsmethode

5.2.1 Programmiersprache

App-Entwicklung in Android verbinden die meisten Menschen automatisch mit Java. Dieser erste Eindruck wird bestätigt, wenn man sich auf die Such nach Tutorials zum Einstieg in Android begibt, denn die meisten dieser Anleitungen verwenden Java als Programmiersprache. Setzt man sich allerdings mit alternativen Möglichkeiten auseinander so stellt man fest, dass Code-Snippets in den offiziellen Guides von Android[1] immer für die Programmiersprachen Java und Kotlin angeboten werden. Schaut man sich bei den offiziellen Beispielprojekte[2] um, so findet man auch Beispielprojekte, die C++ als Grundsprache verwenden.

Entscheidung Wir haben uns dafür entscheiden, Java als Grundsprache für unsere Android App einzusetzen. Java wurde im Rahmen des Studiums schon ausführlich behandelt und alle Mitglieder des Teams sind mit dieser Sprache vertraut.

[1] Android Developers Guides: `https://developer.android.com/guide/`
[2] Android Developers Samples: `https://developer.android.com/samples/`

5.2.2 Layout

Ähnlich wie bei der Wahl der Programmiersprache tendiert die recht eindeutige Empfehlung der meisten Einstiegsliteratur zur Verwendung von XML-basierten Layouts. Die Layouts werden vorab statisch in XML-Dateien spezifiziert und anschließend durch Java-Klassen geladen und mit Funktionalität versehen. Dies ermöglicht eine klare Trennung zwischen Layout und Funktionalität. Auch die offiziellen Guides[3] und Beispielprojekte[4] von Google verwenden XML-basierte Layouts.

Aber es ist auch möglich, die Layouts zur Laufzeit durch Java-Code zu generieren. Hierdurch werden Layouts wesentlich dynamischer und anpassbarer. Bei der Entwicklung von Mobile Games oder bei Layouts, die sich im Laufe der Nutzung ständig ändern, bietet sich dieser dynamische Ansatz an. [Smyth, 2017, S.187ff]

Entscheidung Die meisten Use Cases dieser Anwendung (vgl. Kapitel 5.1.1) beschäftigen sich mit der Anzeige und Manipulation von Daten oder dem Nutzen von Services, die vom Layout unabhängig sind. Diese Anwendungsfälle können sehr gut mit XML-basierten Listen- und Formularlayouts bedient werden. Diese Überlegungen und die Tatsache, dass die offiziellen Dokumentation in diesem Punkt reichhaltig und ausführlicher sind, haben uns dazu bewogen uns für die XML-Variante zu entscheiden.

5.3 Android-Komponenten und -Konzepte

In diesem Kapitel werden verschiedene Komponenten und Konzepte von Android behandelt, die für die Implementierung der App von Bedeutung sind.

5.3.1 Activities

Activities gehören zu den Kernkomponenten von Android Apps. Sie bilden das „Fenster“, in welches die UI von der Anwendung gezeichnet wird. Wie sich aus der Namensgebung

[3] Android Developers Guides: `https://developer.android.com/guide/`
[4] Android Developers Samples: `https://developer.android.com/samples/`

bereits vermuten lässt, hat eine solche Activity in der Regel eine bestimmte Aufgabe oder Tätigkeit. Eine App besteht für gewöhnlich aus verschiedenen Activities, zwischen denen je nach Anwendungsfall gewechselt werden kann.

Das Konzept hinter den Activities sieht vor, dass jede Activity für sich alleine als abgeschlossene Modul steht. Die Idee ist, dass eine Activity einen wiederverwendbaren und austauschbaren Block innerhalb einer App bildet. Um die Unabhängigkeit von anderen Activities zu garantieren, dürfen sie weder auf Methoden oder Instanzvariablen einer anderen Activity zugreifen, noch die eigenen Daten für eine fremde Activity zur Verfügung stellen. Um mit anderen Activities zu kommunizieren werden in den meisten Fällen Intents verwendet (siehe Abschnitt 5.3.5).

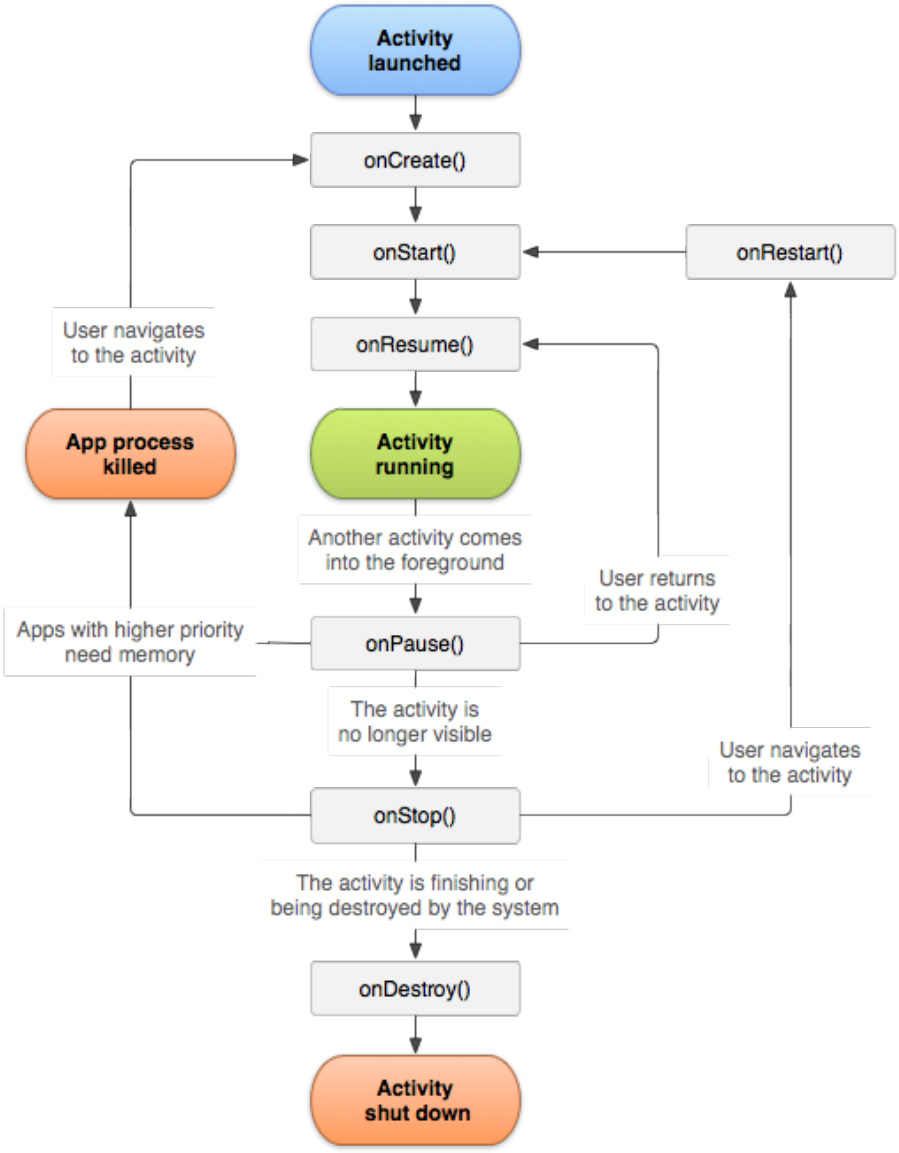
Activities werden nicht durch das Aufrufen einer `main()` Methode gestartet. Vielmehr enthalten sie eine Reihe von Callback-Methoden, die im Lauf des Lebenszyklus einer Activity aufgerufen werden (siehe Abbildung 5.2). Für gewöhnlich wird in der `onCreate()` Methode die UI erzeugt, indem beispielsweise ein XML-Layout geladen und mit Funktionalität versehen wird (siehe Abschnitt 5.2.2).

5.3.2 Single-Thread-Modell

Wenn eine Android App auf dem Endgerät aufgerufen wird, dann wird im Hintergrund ein Linux-Prozess gestartet, der einen Ausführungsthread für die gesamte Anwendung enthält. Alle Komponenten werden standardmäßig in diesem Thread ausgeführt. Unter anderem auch alle nativen UI-Komponenten von Android. Sieht ein Anwendungsfall der App vor, eine umfangreichere Aufgabe auszuführen, dann kann dies zur Blockade der gesamten Anwendung führen. Aus Sicht des Nutzers reagiert die UI nicht mehr und nach einiger Zeit teilt auch das Betriebssystem mit, dass die Anwendung nicht mehr reagiert. [Google LLC, 2018f]

Ein blockierender Anwendungsfall könnte zum Beispiel das Senden und Empfangen von größeren Datenmengen, das Abspielen von Musik oder, wie in dieser App, die Synchronisation mit dem REST-Server und die Erkennung bzw. Verarbeitung von Geofences (siehe auch Kapitel 5.3.7) sein.

Abbildung 5.2: Vereinfachte Darstellung des Activity-Lebenszyklus



Quelle: [Google LLC, 2018h]

Um eine Blockade der App zu verhindern, sollten aufwendigere Aufgabe in eigene Threads ausgelagert werden. Die Android-Architektur bietet hierzu verschiedene Möglichkeiten, von denen einige in den folgenden Kapiteln näher beschrieben werden. Dabei gibt es zwei Regeln, die jederzeit eingehalten werden müssen, um das Single-Thread-Modell von Android nicht zu verletzen: [Google LLC, 2018f]

1. Blockiere nicht den UI-Thread.
2. Greife nicht auf die UI-Komponenten zu, wenn du dich außerhalb des UI-Threads (bzw. des Hauptthreads der App) befindest.

5.3.3 Threads

Um Aufgaben der Anwendung in eigenen, vom UI-Thread gelösten Threads laufen zu lassen, verwendet Android unter anderem die vererbbare Java-Klasse „Thread“. Sie enthält eine zu implementierende Methode namens „run“, die die Aufgabe des jeweiligen Threads zur Laufzeit festlegt. Wird ein solches Thread-Objekt von einer UI-Komponente erzeugt und gestartet, dann erfolgt die Abarbeitung im Hintergrund und kann den UI-Thread selbst nicht blockieren. Der Einsatz von Threads ist hauptsächlich für Aufgaben vorgesehen, die sequentiell abgearbeitet werden können. Nach der Beendigung der Aufgabe wird der Thread beendet und die Ressourcen werden wieder freigegeben. [Google LLC, 2018f] [Google LLC, 2018b]

Handler Wie bereits in Kapitel 5.3.2 beschrieben, dürfen eigenständige Threads nicht auf den UI-Thread zugreifen, also keine Manipulationen an der UI vornehmen. Um trotzdem aus einem solchen Thread heraus eine Änderung der UI zu erwirken, muss ein Handler implementiert werden, den beide Komponenten kennen. Der Handler arbeitet als Vermittler zwischen dem externen Thread und dem UI-Thread, indem er eine Message-Queue implementiert, die vom externen Thread befüllt und von den Komponenten der UI konsumiert wird. Android stellt hierfür die Klasse „HandlerThread“ zur Verfügung. [Ali, 2017]

AsyncTaks Mit der Helferklasse „AsyncTask“ bietet Android eine einfache Möglichkeit zur Verwendung ausgelagerter Threads und Kommunikation mit der UI, ohne explizit mit selbst implementierten Threads und Handlern arbeiten zu müssen. Die Klasse besteht dabei aus drei Hauptkomponenten bzw. -methoden, die den Lebenszyklus des Tasks beschreiben. [Google LLC, 2018a]

- **doInBackground(Params...)**
  Hier wird festgelegt, welche Aufgabe der AsyncTask durchführen soll.
- **onProgressUpdate(Params...)**
  Diese Methode bietet die Möglichkeit, während der Ausführung Benachrichtigungen über den Fortschritt an die UI weiterzureichen.
- **onPostExecute(Result)**
  Nach Beendigung des Tasks wird diese Code ausgeführt.

OnProgressUpdate() und onPostExecute() werden dabei im Kontext der Komponente aus dem Hauptthread ausgeführt, die den Task gestartet hat. Dadurch wird Androids Single-Thread-Modell nicht verletzt. Wird der AsyncTask als innere Klasse direkt in einer UI-Komponente deklariert, so können die beiden Methoden direkt auf die UI-Elemente zugreifen. Wird der Task in einer eigenständigen Klasse implementiert, um z.B. wiederverwendbar zu sein, dann können die zu bearbeitenden UI-Elemente über den Konstruktor an den AsyncTask durchgereicht werden. [Google LLC, 2018a]

Die Verwendung dieser Helferklasse ist nur für kurze Hintergrundoperationen empfohlen, die nicht länger als wenige Sekunden dauern sollen. [Google LLC, 2018a]

5.3.4 Services

Mit den Diensten (engl. Services) bietet Android die Möglichkeit, langlebige Aufgaben im Hintergrund auszuführen. Sie sind dafür designed, auch dann im Hintergrund weiterlaufen zu können, wenn der Anwender gerade nicht mit der App interagiert, oder sie sogar bereits geschlossen hat. Im Wesentlichen unterscheidet man drei Arten von Services: [Google LLC, 2018f]

- **Foreground**
  Sie führen Operationen aus, die für den Anwender bemerkbar sind. Dabei müssen sie ihre Aktivität über eine Benachrichtigung in der Android-Statuszeile mitteilen. Foreground-Services laufen auch dann weiter, wenn die Anwendung nicht mehr im Vordergrund läuft. [Google LLC, 2018f]

- **Background**
  Diese Services führen Aufgaben aus, die den Nutzer nicht direkt betreffen bzw. deren Ausführung er nicht zur Kenntnis nimmt. Sie laufen ebenfalls weiter, wenn die App sich nicht im Fokus befindet, allerdings müssen sie keinen Hinweis in der Statuszeile von Android publizieren. Mit Einführung der Android Version 26 „Oreo“ gibt es neue Richtlinien zum Betreiben von Background-Services. Da die vorliegende App keine Background-Services verwendet, werden die Details nicht näher ausgeführt. Sie können in der Android-Dokumentation nachgelesen werden. [5] [Google LLC, 2018f]

- **Bound**
  Man spricht von einem Bound-Service, wenn sich eine oder mehrere Komponente(n) der Anwendung an ihn binden. Ein Bound-Service stellt ein Client-Server-Interface bereit, welche es den gebundenen Komponenten erlaubt, mit dem Service zu interagieren, Anfragen zu senden und Ergebnisse zu empfangen. Nur so lange Komponenten an ihn gebunden sind, bleibt ein Bound-Service aktiv. Sobald die letzte Verbindung aufgelöst wird, wird der Service beendet. [Google LLC, 2018f]

5.3.5 Kommunikation mit Intents

Mit der Verwendung von Intents können andere Komponenten dazu veranlasst werden, eine Aktion auszuführen. Intents sind Nachrichten-Objekte, die bei Android eingesetzt werden, um eine Activity (vgl. Kapitel 5.3.1) bzw. einen Service (vgl. Kapitel 5.3.4) zu starten oder um einen Broadcast zu senden, wenn die Nachricht an mehrere Empfänger gleichzeitig gehen soll. Dabei können dem Intent auch Daten (Extras) angehängt werden, die der Empfänger als Übergabeparameter entgegen nimmt und entsprechend weiterverarbeiten kann. Intents sind nicht auf den lokalen Einsatz innerhalb einer App beschränkt, sondern

[5] Android Oreo - Background Execution Limits `https://developer.android.com/about/versions/oreo/background`

können auch anwendungsübergreifend eingesetzt werden, um Aktionen in fremden oder bekannten Apps auszulösen. Man unterscheidet zwei Arten von Intents: [Google LLC, 2018d]

- **Explizite Intents**, bei denen deklariert wird, wer den Intent verarbeiten soll. Sie werden meist für das Starten von Activities und Services eingesetzt. [Google LLC, 2018d]
- Ist der Empfänger aus Sicht des Senders unbekannt, so sendet er einen **impliziten Intent**, der nur deklariert, welche Aktion (engl. Action) ausgeführt werden soll. Der typische Anwendungsfall hier wären Broadcasts. Um einen impliziten Intent verarbeiten zu können, muss die Empfänger-Komponente einen Broadcast-Receiver implementieren, der die systemweit gesendeten Actions filtert und nur diejenigen entgegennimmt bzw. weiterverarbeitet, die für die Komponente relevant sind. [Google LLC, 2018d]

5.3.6 Positionsbestimmung

Rund um die Themen Orte und Positionsbestimmung lassen sich zwei unterschiedliche Ansätze identifizieren. Zum einen die `Android Framework Location APIs` und zum anderen die `Location And Context APIs` der Google Play Dienste.

Im offiziellen Android Developers Guide wird mehrfach und ausdrücklich darauf hingewiesen, dass Google die Verwendung der `Location And Context APIs` empfiehlt und dazu rät, von den `Android Framework Location APIs` auf die Google Play Dienste zu migrieren. Als Vorteile werden unter anderem eine einfachere Schnittstelle, eine bessere Leistungs- und Batterieverbrauchsoptimierung und die aktive Weiterentwicklung der `Location And Context APIs` genannt. Zusätzlich bieten die `Location And Context APIs` ein höheres Level an Abstraktion bei der Auswahl der zu verwendenden Technik. Bei der Anwendung muss nur die gewünschte Qualität der Positionsbestimmung angegeben werden und um die dafür einzusetzende Technik (z.b. GPS, WiFi oder Informationen der aktuellen Funkzelle) kümmert sich die API. [Google LLC, 2018e]

Aus den dargestellten Gründen wurden für die Implementierung dieser App die `Location And Context APIs` der Google Play Dienste ausgewählt.

5.3.7 Geofencing

Neben der Möglichkeit die aktuelle Position eines Benutzers zu bestimmen, ist es für die Funktionalität dieser App auch entscheidend zu erkennen, wenn sich der Anwender einem bestimmten Punkt von Interesse (engl. Point of Interest (POI)) nähert. POIs sind dabei je nach Anwendungsfall Kameras, Ausfahrten oder Benachrichtigungspunkte für eine Kamera.

Man könnte den eigenen Standpunkt mit dem Standpunkt aller POIs vergleichen und aus den Längen- und Breitenangaben die Distanz zwischen beiden errechnen. Allerdings ist diese Distanzberechnung alles andere als trivial, müssen doch neben Längen- und Breitengrad auch die Erdkrümmung und die Abstände zwischen den jeweiligen Längen- und Breitengraden miteinbezogen werden. Selbst wenn man für die Berechnung der Distanz eine einfache Formel heranziehen könnte, so müsste man die Berechnung bei jeder Positionsänderung mit allen POIs wiederholen, was einem linearen Aufwand $O(n)$ entspricht.

Eine Alternative bieten die bereits zur Positionsbestimmung verwendeten `Location And Context APIs` mit dem Konzept der Geofences. Ein solcher Geofence wird durch die Angabe eines Zentrums, definiert durch Längen- und Breitenangaben, und eines Radius angelegt. Man kann ihn sich als Kreis mit dem angegeben Radius um das Zentrum vorstellen. Die APIs bieten die Möglichkeit Ereignisse (engl. Events) zu senden, wenn der Anwender diesen imaginären Kreis betritt (enter), wenn er ihn verlässt (exit) und wenn er sich eine bestimmte Zeit darin aufgehalten hat (dwell). Diese Ereignisse können von der Anwendung dann entsprechend weiterverarbeitet werden. [Google LLC, 2018c]

5.4 Algorithmus zum Lernen einer Route

Der Use Case „Route lernen“ wird in Kapitel 5.1.1 bereits beschreiben. Dieses Kapitel soll sich damit beschäftigen, wie der Use Case mit einem Algorithmus umgesetzt werden kann.

5.4.1 Problem der Richtungserkennung

Da beim Abfahren einer Route die Kameras sowohl in Fahrtrichtung, als auch gegen die Fahrtrichtung erkannt werden, bestand die Herausforderung beim Algorithmus zum Lernen einer Route darin, nur die Kameras zu identifizieren welche für die Fahrtrichtung des Anwenders relevant sind.

5.4.2 Lösungsansätze

Richtungsvektor Die erste Idee sieht vor, bei jeder Kamera einen Richtungsvektor zu hinterlegen, der angibt, für welche Fahrtrichtung diese Kamera zuständig ist. Aus den Standorten, die von der Positionsbestimmung (vgl. Kapitel 5.3.6) geliefert werden, soll ein Richtungsvektor ermittelt werden, der bestimmt, in welche Richtung sich der Anwender aktuell bewegt. Wenn die beiden Richtungsvektoren innerhalb einer gewissen Toleranz in die gleiche Richtung zeigen, dann handelt es sich um eine relevante Kamera, die es aufzuzeichnen gilt.

Vorhergehende Ausfahrt Eine Kamera besitzt immer mindestens eine Ausfahrt, die in Fahrtrichtung vor der Kamera liegt. Wenn die Kamera nach einem Autobahndreieck platziert ist, dann können es sogar zwei, und bei einem Autobahnkreuz drei vorhergehende Ausfahrten sein. Der zweite Lösungsansatz sieht vor, beim Abfahren der Route alle passierten Ausfahrten zu speichern. Beim Erkennen einer Kamera auf der Route wird geprüft, ob wir eine ihrer möglichen Vorgängerausfahrten passiert haben. Ist dies der Fall, so handelt es sich um eine Kamera in Fahrtrichtung. Haben wir keine der in Frage kommenden Ausfahrten passiert, dann ignorieren wir diese Kamera.

Vergleich der Lösungsansätze Das Berechnen der Richtungsvektoren würde einen zusätzlichen Aufwand bedeuten, der bei jeder neuen Position wiederholt werden muss. Zusätzlich müsste die Berechnung mit statistischen Methoden erfolgen, da bei alleiniger Betrachtung des aktuellen und des letzten Standpunktes die Gefahr besteht, einen fehlerhaften Richtungsvektor zu erhalten (z.B. durch ungenaue GPS-Daten). Das Erfassen der Ausfahrten bestünde aus dem Hinzufügen zusätzlicher Geofences und der einmaligen

Erfassung einer passierten Ausfahrt. Die Gefahr, eine Ausfahrt zu verpassen, ist dabei genau so hoch oder niedrig wie die Gefahr, eine Kamera zu verpassen. Sofern dieser Fehler auftritt, kann er vom Anwender reguliert werden, indem der Geofence-Radius und das Polling-Intervall für die Standortabfrage angepasst werden (siehe Use Case „Einstellungen verwalten“).

Wahl des Lösungsansatzes Aus unserer Sicht sind beim Lösungsansatz mit den vorhergehenden Ausfahrten sowohl der Berechnungsaufwand, als auch die Fehleranfälligkeit geringer. Daher haben wir uns dafür entschieden, den Lernalgorithmus mit diesem Ansatz zu implementieren. Zusätzlich kann das Erfassen der vorhergehenden Ausfahrten dazu genutzt werden, um den Benachrichtigungspunkt über einen Stau an einer Kamera auf die jeweils vorherige Ausfahrt zu setzen. Die so gefunden relevanten Kameras werden in einer Map gespeichert, die als Key die UUID der Kamera und als Value die UUID der vorhergegangenen Ausfahrt enthält.

5.4.3 Randfälle

Beim Entwerfen und Diskutieren der beiden Lösungsansätze sind uns einige Randfälle aufgefallen, die beim Erlernen der Route zu betrachten sind. Im Folgenden werden sie kurz dargelegt und untersucht, wie sie mit dem gewählten Lösungsansatz erfasst werden können.

Start bei einer Ausfahrt mit Kameras Bei diesem Randfall erfassen wir zwei Kameras in unterschiedliche Richtungen und haben noch keine vorhergehende Ausfahrt erfasst, die zu einer der beiden Kameras passen würde. Um dieses Problem zu lösen, führen wir zunächst eine boolesche Variable, die angibt, ob bereits ein Match (eine Kamera mit vorher passierter Ausfahrt) gefunden wurde. So lange dies nicht der Fall ist, fügen wir alle gefunden Kameras zu den relevanten Kameras hinzu und tragen für die vorhergehende Ausfahrt `null` ein. Zusätzlich führen wir eine Liste der ungematchten Kameras. Bei jeder erkannten Ausfahrt iterieren wir über die Liste der ungematchten Kameras. Für jede ungematchte Kamera betrachten wir ihre möglichen Alternativen. Ist die aktuelle Ausfahrt in dieser Auswahl enthalten, dann handelt es sich um eine Kamera, die entgegen unserer

Fahrtrichtung ausgerichtet ist und wir entfernen sie aus den relevanten Kameras. Durch dieses Vorgehen bleiben lediglich die ungematchten Kameras zu Beginn der Route übrig, die sich in Fahrtrichtung befinden.

Ende bei einer Ausfahrt mit Kameras Die kürzeste realistische Route auf einer Autobahn führt von einer Auffahrt- bis zur nächsten Abfahrtmöglichkeit. Wir befinden uns am Beginn der Route und haben daher noch keine gematchte Kamera. Beim Erreichen der Abfahrt mit den Kameras ist die Reihenfolge, in der die Geofences ausgelöst werden, von Bedeutung:

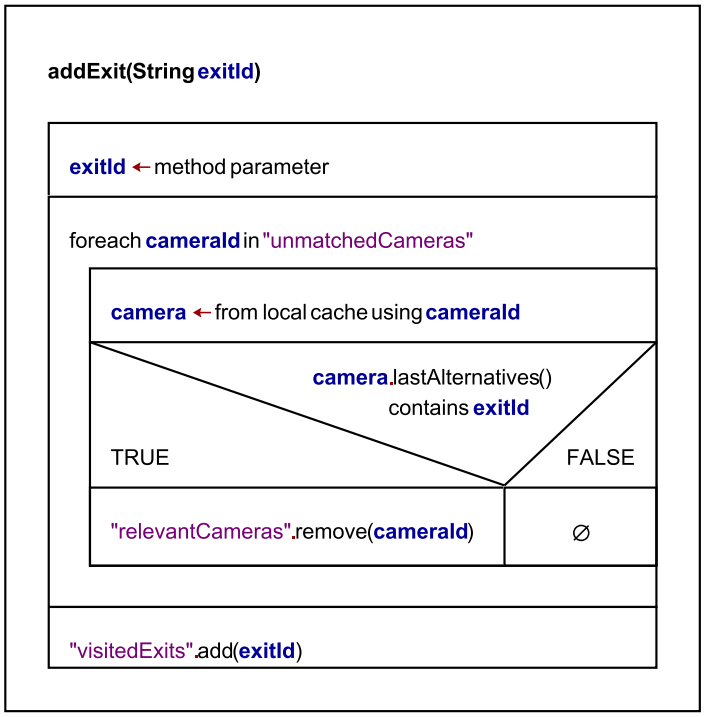
- **Die relevante Kamera wird vor der irrelevanten erkannt** Die relevante Kamera wird gematcht und aufgrund des erfolgreichen Matches wird die irrelevante Kamera nicht mehr hinzugefügt. Dieser Fall ist unproblematisch.
- **Die irrelevante Kamera wird zuerst erkannt** Hier wird es problematisch. Da wir zu diesem Zeitpunkt noch kein erfolgreiches Match gefunden haben, wird die irrelevante Kamera zu den relevanten Kameras hinzugefügt. Wir fahren an dieser Ausfahrt ab und erreichen somit nie die nächste Autobahnausfahrt, die dafür sorgen würde, dass die irrelevante Kamera wieder aus der Liste entfernt wird.

Wir lösen dieses Problem, indem wir dafür sorgen, dass jeder Kamera ihre Schwester, also ihr Pendant in der Gegenrichtung, kennt. Bei einem erfolgreichen Match einer Kamera prüfen wir nun zusätzlich, ob die Schwester-Kamera bereits in den relevanten Kameras enthalten ist. Sofern dies der Fall ist wird die Schwester entfernt.

5.4.4 Ablaufdiagramme der Kernkomponenten

Anhand des gewählten Lösungsansatzes (vgl. Kapitel 5.4.2) und den behandelten Randfällen (vgl. Kapitel 5.4.3) ergeben sich Struktogramme für das Hinzufügen einer Ausfahrt (siehe Abbildung 5.3) und das Hinzufügen einer Kamera (siehe Abbildung 5.4). So wie sie hier beschrieben werden, werden beide Abläufe als Methoden im Quellcode des Algorithmus zum Lernen einer Route implementiert.

Abbildung 5.3: Struktogramm für das Hinzufügen einer Ausfahrt



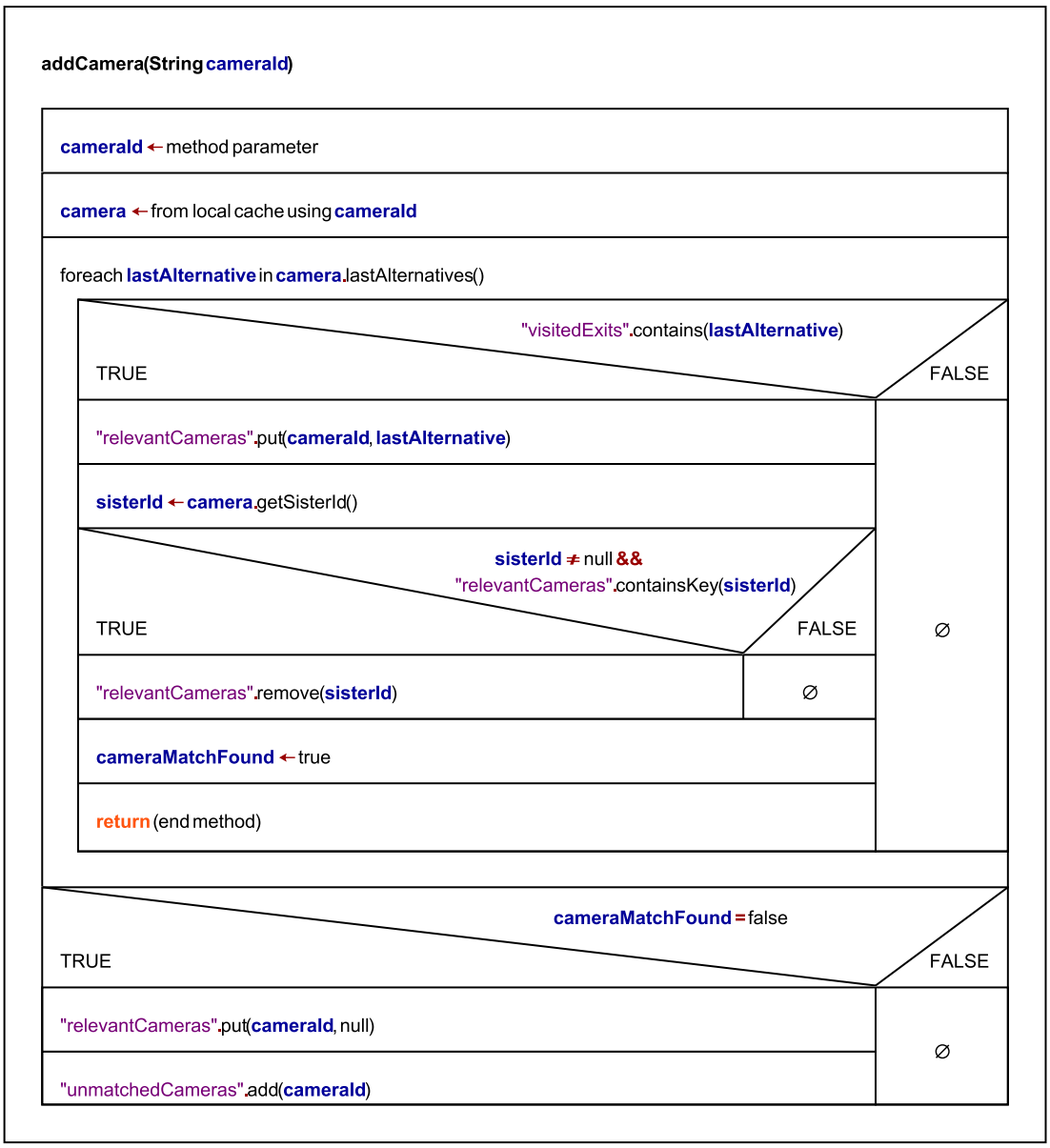
Quelle: Eigene Darstellung

5.4.5 Ablauf des Lernvorgangs

Start Zu Beginn wird der Anwender mit einem Dialog darauf hingewiesen, dass er im Begriff ist, den Lernmodus zu aktivieren. Dabei muss er sich entscheiden, was mit Kameras geschehen soll, die bereits zur Beobachtung angelegt wurden. Die bestehenden Kameras können entfernt oder durch die neu gefundenen Kameras ergänzt werden. Außerdem hat der Anwender die Möglichkeit den Dialog abzubrechen ohne den Lernmodus zu starten.

Durchführung Die `Location And Context APIs` werden initialisiert und für alle bekannten Kameras und Ausfahrten werden Geofences angelegt (siehe auch Kapitel 5.3.7). Beim Eintritt in einen Geofence werden die jeweiligen Methoden zum Hinzufügen eine Kamera oder einer Ausfahrt (siehe auch Kapitel 5.4.4) aufgerufen.

Abbildung 5.4: Struktogramm für das Hinzufügen einer Kamera



Quelle: Eigene Darstellung

Abschluss Nach Beendigung des Lernvorgangs werden alle relevanten Kameras geprüft. Hat der Anwender sich zum Löschen der bereits existierenden Kameras entschieden, dann werden alle relevanten Kameras in zu beobachtende Kameras überführt. Andernfalls werden nur relevante Kameras überführt, die sich noch nicht in der Liste zu beobachtender Kameras befinden. Als Benachrichtigungspunkt über einen Stau an der jeweiligen Kamera wird die zuvor passierte Ausfahrt eingetragen. Bei Kameras, für die keine vorhergehende Ausfahrt aufgezeichnet wurde, wird ihr eigener Standpunkt zum Benachrichtigungspunkt. Dem Anwender wird anschließend eine Zusammenfassung angezeigt, die darüber Aufschluss gibt, welche Kameras hinzugefügt wurden und was für Benachrichtigungspunkte jeweils gewählt wurden. Außerdem wird er darüber in Kenntnis gesetzt, dass er die Benachrichtigungspunkte manuell anpassen kann.

5.5 Algorithmus zur Ausgabe einer Benachrichtigung

Die Implementierung dieses Algorithmus stellt die Kernkomponente des Use Case „Stauerkennung nutzen“ dar, wie er in Kapitel 5.1.1 beschrieben wurde.

5.5.1 Problem der Richtungsfindung

Ähnlich wie beim Lernen der Route haben wir auch bei der Stauerkennung das Problem der Richtungsfindung. Wenn der Anwender sich einer Kamera bzw. deren Benachrichtigungspunkt aus der „falschen“ Fahrtrichtung nähert, dann soll er keine Benachrichtigung zu dieser Kamera erhalten.

5.5.2 Lösungsansatz

Wir orientieren uns an dem bereits beim Lernen der Route gewählten Ansatz und verwenden keine Richtungsvektoren. Stattdessen greifen wir auch hier auf zusätzliche Geofences zurück. Zusätzlich zu den Benachrichtigungspunkten erfassen wir jede Kamera an der wir vorbeifahren. Beim Erreichen eines Benachrichtigungspunktes wird geprüft, ob die Kamera, für die er eingerichtet wurde, bereits passiert wurde. Wenn das der Fall ist, dann handelt

es sich um einen Benachrichtigungspunkt bzw. eine Kamera, die in die entgegengesetzte Fahrtrichtung zeigt. Wir senden keine Nachricht an den Anwender.

5.5.3 Randfälle

Nahe beieinander liegend Wenn die Standpunkte der Kamera und des Benachrichtigungspunktes identisch sind oder sehr nahe beieinander liegen, ist die Reihenfolge, in der die Geofence-Events ausgelöst werden, entscheidend.

- **Der Benachrichtigungspunkt wird vor der Kamera erkannt**
  Dieser Fall verursacht kein Problem. Die Sprachnachricht wird ausgegeben, da wir die Kamera noch nicht passiert haben.
- **Die Kamera wird zuerst erkannt**
  Dieser Fall ist problematisch. Da wir die Kamera des Benachrichtigungspunktes bereits passiert haben, wird keine Nachricht erzeugt.

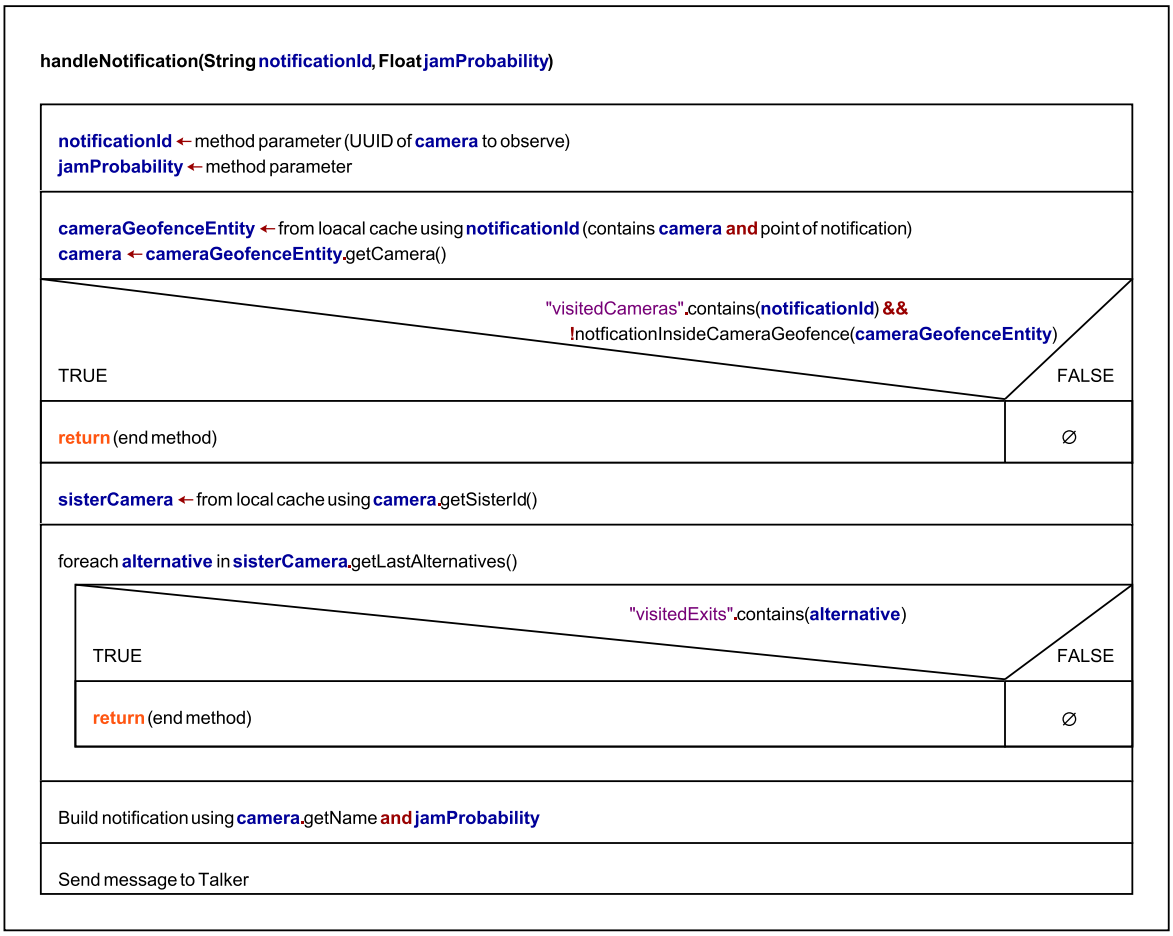
Wir erweitern unsere Bedingung für die Ausgabe eine Nachricht an den Anwender. Wenn der Benachrichtigungspunkt innerhalb des Geofence der Kamera liegt, dann wird ebenfalls eine Sprachnachricht ausgegeben.

Nahe beieinander liegend - aus falscher Richtung Aus der Lösung für den ersten Randfall entsteht ein zweiter. Wenn wir uns nun der beschriebenen Konstellation nähern, wird die Benachrichtigung ausgegeben, selbst wenn die Kamera eigentlich in die falsche Richtung zeigt. Dieser Fall kann ausgeschlossen werden, indem wir während der Stauerkennung auch die besuchten Ausfahrten speichern. Wir betrachten zu Beginn unseres Algorithmus zuerst die Schwester-Kamera der Kamera, die wir eigentlich beobachten. Haben wir eine der Vorgängerausfahrten der Schwesterkamera bereits passiert, dann nähern wir uns dem Benachrichtigungspunkt aus der falschen Richtung.

5.5.4 Struktogramm

Die Implementierung des Algorithmus zur Ausgabe einer Benachrichtigung ist in Abbildung 5.5 als Struktogramm schematisch dargestellt.

Abbildung 5.5: Struktogramm für die Ausgabe einer Benachrichtigung



Quelle: Eigene Darstellung

5.6 Implementierung

Der folgende Abschnitt geht darauf ein, wie wir die Android App designed und implementiert haben. Die beschriebenen Komponenten finden sich in der visuellen Darstellung der App-Architektur wieder (siehe Abbildung 5.6).

5.6.1 Activities

Overview Diese Activity besteht aus zwei Komponenten. In der oberen Hälfte befindet sich ein Button-Layout, welches es ermöglicht, die Stauerkennung und das Lernen einer Route zu beginnen oder zu stoppen. Außerdem kann von hier aus zu den Activites `Map` und `CameraList` navigiert werden. Eine Liste der Kameras, die der Nutzer bereits beobachtet, füllt die untere Hälfte der Activity. Zu den einzelnen Kameras wird die aktuelle Stauwahrscheinlichkeit angezeigt. Beim Antippen eines Listeneintrags erscheint das letzte Bild der jeweiligen Kamera. Durch einen Swipe-Refresh (Wischgeste nach unten) können die aktuellen Daten von der REST-Schnittstelle des Backends bezogen werden. Das Bearbeiten einer Kamera ist durch ein langes Tippen auf den Listeneintrag möglich und leitet einen auf die `Add/Edit Camera` Activity.

CameraList Hier wird dem Anwender eine Liste aller Kameras angezeigt, die er potentiell beobachten lassen kann. Aus dieser Liste heraus kann er eine Kamera zur Beobachtung hinzufügen oder sie entsprechend von der Beobachtung entfernen. Das Hinzufügen führt den Anwender auf die `Add/Edit Camera` Activity, wohingegen das Entfernen keiner weiteren Interaktion bedarf. Die Liste der Kameras lässt sich ebenfalls durch einen Swipe-Refresh mit dem Backend abgleichen.

Add/Edit Camera Dieser Dialog wird angezeigt, wenn eine zu beobachtende Kamera angelegt oder bearbeitet werden soll. Der Nutzer muss den Ort auswählen, an dem er über einen möglichen Stau an dieser Kamera informiert werden möchte. Diesen Benachrichtigungsort kann er über eine Karte auswählen. Als Initialisierungspunkte für die Karte kann zwischen den möglichen vorhergehenden Ausfahrten, dem Standort der Kamera selbst und dem zuletzt gewählten Ort, sofern es diesen gibt, entschieden werden. Für die

Abbildung 5.6: Architektur der Android App

Quelle: Eigene Darstellung

Kartenauswahl wird die `PlacePicker`-Komponente[6] der `Location And Context APIs` von Google verwendet.

Map Auch von der `Map` Activity aus können die Stauerkennung oder das Lernen einer Route gestartet oder beendet werden. Hierzu stehen wie bei der `Overview` Activity zwei Buttons im oberen Bereich zur Verfügung. Den Hauptteil diese Activity nimmt allerdings ein Kartenausschnitt ein. Wenn der `Geofence` Service aktiv ist, werden in die Karte die entsprechenden Geofences eingezeichnet. Je nach Anwendungsfall handelt es sich dabei um Kameras, Ausfahrten und/oder Benachrichtigungspunkte. Außerdem wird der aktuelle Standpunkt regelmäßig vom `Geofence` Service mit einem Intent übermittelt und in der Karte dargestellt. Zusätzlich wird die Karte auf den neuen Standort zentriert.

Settings Im Einstellungsdialog kann die Größe des Geofence-Radius in Metern und das Polling-Intervall für die Standortabfrage in Millisekunden manuell festgelegt werden. Beide Einstellungen sind mit sinnvollen Werten vorbelegt. Der Minimalwert für das Polling-Intervall liegt bei 500 Millisekunden, da zu häufige Abfragen einerseits den Stromverbrauch erhöhen und andererseits ab einem gewissen Punkt keinen Mehrwert mehr bringen.

5.6.2 Services

Geofence Der `Geofence` Service ist für das Abfragen des aktuellen Standortes und die Registrierung bzw. Überwachung der Geofences verantwortlich. Er kann sowohl im Lern- als auch im Stauerkennungsmodus gestartet werden. Entsprechend registriert er die benötigten Geofences und initialisiert den `LearningHandler` bzw. den `TrafficHandler`. Sobald ein mobiles Endgerät in einen Geofence eintritt, sendet der `Geofence` Service einen Intent an den `GeofenceRegistration` Service. Dieser Intent enthält Informationen über die Art des Geofences (Kamera, Ausfahrt oder Benachrichtigungspunkt) und die jeweilige UUID der Komponente.

Auch wenn die App sich nicht im Vordergrund befindet oder beendet wird, läuft der Service im Hintergrund weiter. Eine Benachrichtigung in der Android-Statusleiste zeigt

[6] PlacePicker: `https://developers.google.com/places/android-api/placepicker`

seine Aktivität an. Über eine Schaltfläche innerhalb der Benachrichtigung kann der Anwender den Service jederzeit beenden.

GeofenceRegistration Dieser Service lauscht auf die Intents mit neuen Geofencing-Events, verarbeitet diese und leitet sie dann entsprechend an den `LearningHandler` oder den `TrafficHandler` weiter. Handelt es sich dabei um ein Traffic-Event (Stauerkennung) ruft der `GeofenceRegistration` Service noch die Stauwahrscheinlichkeit für die entsprechende Kamera über den `REST-Client` ab und leitet sie ebenfalls weiter.

5.6.3 AsyncTasks

TrafficInformationLoader Der `TrafficInformationLoader` kann von anderen Komponenten als eigenständiger Thread im Hintergrund gestartet werden. Er ruft über den `REST-Client` die Stauwahrscheinlichkeiten und die letzten Bilder für alle Kameras ab, die derzeit beobachtet werden. Unter Verwendung des `FileManager`s legt er die erhaltenen Daten im lokalen Speicher ab. Bei Erfolg sendet er einen Intent an die aktive Activity, damit diese die Daten gegebenenfalls erneut aus dem Speicher laden kann.

CameraLoader Dieser Task verhält sich analog zum `TrafficInformationLoader` mit den Unterschied, dass er die Lister der potentiell beobachtbaren Kameras mit dem Backend-Server abgleicht.

5.6.4 Helferklassen

REST-Client Diese Klasse übernimmt die Kommunikation mit der REST-Schnittstelle des Backends. Er stellt für jeden Endpoint eine statische Methode zur Verfügung, die von anderen Komponenten der Android-App abgefragt werden kann.

LearningHandler Der `LearningHandler` nimmt während des Lernvorgangs die entsprechenden Geofence-Events als Intents vom `GeofenceRegistration` Service entgegen. Diese

verarbeitet sie entsprechend des in Kapitel 5.4 beschriebenen Algorithmus zum Lernen einer Route.

TrafficHandler Im Laufe der Stauerkennung nimmt diese Klasse ebenfalls Geofence-Events vom `GeofenceRegistration` Service entgegen. Die Verarbeitung wird in Kapitel 5.5 näher beschrieben. Zur Ausgabe eine Sprachnachricht an den Anwender wird der `Talker` verwendet.

Talker Zur Ausgabe von Sprachnachrichten bietet der `Talker` eine statische Methode an. Dabei nimmt er einen String entgegen und wandelt ihn mittels Androids TextToSpeech[7] in eine Audio-Ausgabe um.

SettingsManager Für den Zugriff auf die Werte von Einstellungsparametern hält der `SettingsManager` Methoden bereit, die von allen Komponenten der App genutzt werden können.

FileManager Die Aufgabe des `FileManagers` besteht im Ablegen und Bereitstellen von Daten im bzw. aus dem lokalen Speicher. Über seine Methoden können andere Komponenten der App auf die Daten zugreifen, ohne sich mit der Komplexität der Dateiverwaltung zu befassen.

[7] TextToSpeech: `https://developer.android.com/reference/android/speech/tts/TextToSpeech`

6 Fazit und Ausblick

6.1 Ausblick

6.1.1 Neuronales Netz

Witterungen Das neuronale Netz liefert bereits passable Ergebnisse bei guter Witterung. Fragwürdig ist allerdings, inwieweit das neuronale Netz bei veränderten Sichtbedingungen funktioniert. Diese können durch vielfache verschiedene Witterungen entstehen, so zum Beispiel Nebel, welcher das Bild verschwommen und grau erscheinen lässt. Dasselbe gilt für dichtes Schneetreiben oder Regenschauer, welche das Bild ebenfalls unscharf werden lassen. Weiterhin wird das Bild von Schnee im Winter verändert, denn liegen bleibender Schnee färbt die Fahrbahn weiß. Da wir keinerlei Testdaten mit solchen Sichtbedingungen hatten ist bisher unklar, ob das neuronale Netz mit solchen Veränderungen umgehen kann oder dann falsche Ergebnisse liefert.

Mehr Daten Für eine Verbesserung der Erkennungsergebnisse wären mehr Trainingsdaten sehr hilfreich, besonders Bilder bei anderen Licht- und Witterungsverhältnissen.

Auch die systematische Evaluierung krankt an zu wenigen Validierungsdaten. Zusammen mit mehr Trainingsdaten würde es sich wahrscheinlich lohnen, tiefere neuronale Netze mit mehr Layern zu trainieren. Dafür wäre dann allerdings mehr Hardware erforderlich.

6.1.2 Webservice

Statistiken Für den Webservice bieten sich verschiedene Erweiterungsmöglichkeiten. Eine davon wäre die systematische Analyse von alten Daten. Daraus lassen sich Statistiken über die Stauwahrscheinlichkeiten der einzelnen Ausfahrten beziehungsweise Kameras in Abhängigkeiten der zeitlichen Entwicklung erstellen. Für den Endnutzer könnten damit Vorhersagen über die Stauentwicklung im Laufe des Tages oder der Woche möglich sein. Dadurch könnten längere Reisen eventuell besser geplant werden, wenn bekannt ist, dass die Stauwahrscheinlichkeit an gewissen Tagen und zu gewissen Zeiten geringer ist. Diese Informationen könnten als Diagramme über die Android App oder alternativ über eine dedizierte Website abgerufen werden.

Web-Version Eine Website wäre generell denkbar, auch um die aktuellen Stauwerte ohne Android App zur Verfügung zu stellen. Aktuell gibt es zwar eine solche Seite[1], allerdings ist diese nicht für die Öffentlichkeit bestimmt, sondern nur zu Debug-Zwecken entwickelt worden. Nach einer gründlichen Überarbeitung wäre dies allerdings ein mögliches Feature.

Weitere Kameras Eine weiteres Feature für die Zukunft wäre die Erweiterung der Kameraliste. Die Möglichkeiten hierfür sind bereits gegeben, da die gesamte Anwendung nur auf den angelegten Datenbankeinträgen für die Ausfahrten und Kameras arbeitet. Das genau Datenmodell ist in Kapitel 4.1 detailliert beschrieben. Zusätzliche Ausfahrten und Kameras können einfach über einen REST Endpoint hinzugefügt werden. Diese neuen Kameras werden dann beim nächsten Abrufen der Kamerabilder mitverarbeitet und damit den Endgeräten zur Verfügung gestellt. Das Erweitern der Kamera-Liste ist also bereits im laufenden Betrieb möglich.

Stau-Länge Eine vierte Erweiterung wäre der Versuch, Informationen über die Länge eines Staus zu erhalten. Dies wäre möglich, falls ein Stau über aufeinanderfolgende Kameras erkannt wird. Alternativ könnte hier auch die Verwendung von externen Quellen angestrebt werden, um dem Benutzer möglichst detaillierte Informationen zur Verfügung zu stellen.

[1] `https://ihatestau.tinf15b4.de/ihatestau/info/html`

6.1.3 Android App

Mehrere Routen Die Android App kann ebenfalls um verschiedene Features erweitert werden. Das wohl wichtigste ist es, mehrere Routen zu ermöglichen. Aktuell kann nur eine einzige Route konfiguriert werden, sodass die App beispielsweise für eine Hin- und Rückfahrt ungeeignet ist. Diese Einschränkung könnte einfach behoben werden, indem das App-interne Datenmodell leicht abgeändert wird. Die aktuell gespeicherte Liste von konfigurierten Geofencing Punkten müsste in ein eigenes Objekt gekapselt werden um die Route zu repräsentieren. Beim Starten des Geofencings könnte dann eine der gespeicherten Routen ausgewählt werden.

Prinzipiell wäre auch möglich, die Route bei der Aktivierung der Stauerkennung automatisch auszuwählen, basierend auf der aktuellen Position des Benutzers. Falls eine eindeutige Entscheidung möglich ist, könnte die Route verwendet werden, deren Startauffahrt möglichst nahe an der Position des Benutzers liegt. Auch eine Analyse der Historie des Benutzers wäre denkbar. Fälle wie der Arbeitsweg wären damit einfach abzudecken, da meistens jeden morgen und jeden Abend etwa um die selbe Uhrzeit die selbe Route abgefahren wird.

Smart Activation Eine nützliche Weiterentwicklung wäre das automatische Starten des Geofencings. Eine Möglichkeit wäre hier (soweit vorhanden) die Erkennung der Bluetooth-Verbindung des Autos. Das hat allerdings den Nachteil, dass sich die App aktiviert, sobald der Benutzer das Auto betritt. Auch wenn eine andere Route gefahren wird, auf der die Geofences niemals erreicht werden können, würde die App im Hintergrund Daten senden und empfangen und damit unnötigerweise den Akku des Smartphones belasten.

Eine weitere Möglichkeit wäre das Erkennen, wenn die erste Autobahn-Auffahrt der Route erreicht wird. Da die App allerdings noch nicht aktiv ist, steht hierfür kein GPS zur Verfügung. Denkbar wäre das Annähern der Position über die Funkzelle, in welcher sich das Smartphone aktuell angemeldet hat. Hierfür müsste allerdings irgendwie die Funkzelle der Auffahrt vom Benutzer selbst eingetragen werden, da diese je nach Anbieter natürlich abweichen können. Da das Smartphone während der Autofahrt allerdings nicht verwendet werden darf, könnte sich dieser Punkt als schwer erweisen. Eventuell könnte die Funkzelle gespeichert werden, sobald die Auffahrt bei aktivem Geofencing zum ersten mal

erreicht wird. Die Zuverlässigkeit dieses Verfahrens ist allerdings sehr beschränkt, da sich Funkzellen überschneiden und das Smartphone sich jederzeit eine andere Funkzelle suchen kann. Das Abspeichern mehrerer Funkzellen könnte dieses Problem beheben, allerdings steigt damit die Ungenauigkeit.

Per Benutzereinstellung könnte man auch ermöglichen, dass die App dauerhaft GPS Daten senden und dann anhand des GPS Signals entschiedenen werden kann, ob die erste Auffahrt erreicht wurde. Um Akku zu sparen, könnte man hier die Abtastrate im Vergleich zum Geofencing deutlich verkleinern. Um ein etwaiges Überspringen der ersten Auffahrt zu vermeiden, müsste das Geofencing Gebiet allerdings entsprechend groß sein. Auch bei abgeschwächter Abtastrate würde dies die Akku-Laufzeit allerdings erheblich beeinträchtigen, sodass diese Lösung höchstens auf ausdrücklichen Wunsch des Benutzers aktiv werden sollte.

Radius Aktuell ist der Radius des Geofencings vom Benutzer selbst eingestellt. Hier könnte man die Erweiterung schaffen, den Radius automatisch anhand der aktuellen Geschwindigkeit festzulegen. Bei 100km/h auf der Autobahn würde ein Radius von 1km ausreichen, um rechtzeitig zu reagieren und die Ausfahrt noch zu erreichen. Verdoppelt man die Geschwindigkeit, halbiert sich die Zeit, welche dem Benutzer bleibt um auf die Ansage der App zu reagieren. Eine Möglichkeit wäre es den Radius zu verdoppeln, um diesen Nachteil auszugleichen.

Abtastrate Ebenfalls möglich wäre das Einstellen der Abtastrate basierend auf der aktuellen Geschwindigkeit. Je geringer die Geschwindigkeit, desto seltener ist eine Abtastung des GPS nötig um keine Geofences zu überspringen. Sofern möglich, wäre dieses Verfahren auch für die Einstellung der Ansage-Lautstärke denkbar. Bei höheren Geschwindigkeiten herrscht normalerweise eine höhere Umgebungslautstärke im Fahrzeug, sodass das Erhöhen der Lautstärke nötig sein könnte.

Info bei App-Start Als weiteres zusätzliches Feature wäre es denkbar, die erste Autobahn-Auffahrt direkt beim App-Start anzusagen, sodass der Benutzer frühzeitig entscheiden kann, ob er auf die Autobahn auffahren will oder nicht. Generell bietet die App einige sehr kleine Features oder Unschönheiten, die schnell umgesetzt werden könnten. Das

Hinzufügen von Kameras könnte beispielsweise intuitiver erfolgen. Nach der Einstellung des Benachrichtgungsorts muss ein weiterer Button „Kamera hinzufügen“ gedrückt werden. Wird dies vergessen und die Übersichtsseite der App aufgerufen, ist die Kamera nicht eingefügt worden. Die unter Umständen zeitaufwändige Einstellung des Ortes muss wiederholt werden.

iOS Eine andere, große Erweiterung wäre das Entwickeln einer App für Apples Betriebssystem iOS. Hierbei gibt es verschiedene Frameworks, die das Schreiben der App vom Zielsystem abstrahieren, sodass aus dem selben Quellcode Apps für iOS und Android erzeugt werden können. Dabei entstehen allerdings erstens enorme Aufwände für das Portieren des Quellcodes in ein solches Framework wie z.B. Xamarin, und zweitens hohe Kosten aufgrund der benötigten Apple Hard- und Software, inklusive etwaiger Lizenzen.

Abbildungsverzeichnis

Literaturverzeichnis

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.

[Ali, 2017] Ali, J. (2017). Understanding android core: Looper, handler, and handlerthread. `https://blog.mindorks.com/android-core-looper-handler-and-handlerthread-bd54d69fe91a` zuletzt abgerufen am 10. Mai 2018.

[Bottou and Bousquet, 2008] Bottou, L. and Bousquet, O. (2008). The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168.

[Cauchy, 1847] Cauchy, A. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538.

[Dazer, 2012] Dazer, M. (2012). Restful apis - eine Übersicht. `http://snet.tu-berlin.de/fileadmin/fg220/courses/WS1112/snet-project/restful-apis_dazer.pdf` zuletzt abgerufen am 11. Mai 2018.

[Demuth et al., 2014] Demuth, H. B., Beale, M. H., De Jess, O., and Hagan, M. T. (2014). *Neural network design*. Martin Hagan.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Google LLC, 2018a] Google LLC (2018a). Android api reference: Asynctask. https://developer.android.com/reference/android/os/AsyncTask zuletzt abgerufen am 10. Mai 2018.

[Google LLC, 2018b] Google LLC (2018b). Android api reference: Thread. https://developer.android.com/reference/java/lang/Thread zuletzt abgerufen am 10. Mai 2018.

[Google LLC, 2018c] Google LLC (2018c). Android developers guides: Create and monitor geofences. https://developer.android.com/training/location/geofencing zuletzt abgerufen am 10. Mai 2018.

[Google LLC, 2018d] Google LLC (2018d). Android developers guides: Intents and intent filters. https://developer.android.com/guide/components/intents-filters zuletzt abgerufen am 10. Mai 2018.

[Google LLC, 2018e] Google LLC (2018e). Android developers guides: Migrate to location and context apis. https://developer.android.com/guide/topics/location/migration zuletzt abgerufen am 10. Mai 2018.

[Google LLC, 2018f] Google LLC (2018f). Android developers guides: Processes and threads overview. https://developer.android.com/guide/components/processes-and-threads zuletzt abgerufen am 10. Mai 2018.

[Google LLC, 2018g] Google LLC (2018g). Android developers guides: Reduce the apk size. https://developer.android.com/topic/performance/reduce-apk-size zuletzt abgerufen am 10. Mai 2018.

[Google LLC, 2018h] Google LLC (2018h). Android developers guides: The acitivity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle zuletzt abgerufen am 12. Mai 2018.

[Google LLC, 2018i] Google LLC (2018i). Android studio user guide: Use java 8 language features. `https://developer.android.com/studio/write/java8-support` zuletzt abgerufen am 12. Mai 2018.

[Jähne, 2013] Jähne, B. (2013). *Digitale bildverarbeitung*. Springer-Verlag.

[Javalin, 2018] Javalin (2018). Javalin homepage. `https://javalin.io/` zuletzt abgerufen am 11. Mai 2018.

[JavalinDoc, 2018] JavalinDoc (2018). Javalin documentation. `https://javalin.io/documentation` zuletzt abgerufen am 11. Mai 2018.

[Jersey, 2018] Jersey (2018). Jersey homepage. `https://jersey.github.io/` zuletzt abgerufen am 11. Mai 2018.

[JerseyDoc, 2018] JerseyDoc (2018). Jersey documentation. `https://jersey.github.io/documentation/latest/index.html` zuletzt abgerufen am 11. Mai 2018.

[Karpathy, 2018] Karpathy, A. (2018). Cc231n convolutional neural networks for visual recognition. Stanford Lecture, `http://cs231n.github.io/` zuletzt abgerufen am 30. April 2018.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

[Lewis and Fowler, 2014] Lewis, J. and Fowler, M. (2014). Microservices. `https://martinfowler.com/articles/microservices.html` zuletzt abgerufen am 10. Mai 2018.

[Manning et al., 2008] Manning, C. D., Raghavan, P., Schütze, H., et al. (2008). *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge.

[Martin, 2009] Martin, R. C. (2009). *Clean Code*. Prentice Hall.

[Mitchell et al., 1997] Mitchell, T. M. et al. (1997). Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877.

[Newman, 2015] Newman, S. (2015). *Microservices: Konzeption und Design*. mitp Verlag.

[Pan and Yang, 2010] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

[Razavian et al., 2014] Razavian, A. S., Azizpour, H., Sullivan, J., and Carlsson, S. (2014). Cnn features off-the-shelf: an astounding baseline for recognition. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, pages 512–519. IEEE.

[Russakovsky et al., 2015] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252.

[Smyth, 2017] Smyth, N. (2017). *Android Studio 3.0 Development Essentials – Android 8 Edition*. Payload Media, Inc.

[Springenberg et al., 2014] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

[Szegedy et al., 2015] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2015). Rethinking the inception architecture for computer vision. corr abs/1512.00567 (2015).

[TensorFlow, 2018] TensorFlow (2018). Tensorflow documentation: tf.layers.dropout. `https://www.tensorflow.org/api_docs/python/tf/layers/dropout`, zuletzt abgerufen am 12. Mai 2018.

[Zeiler and Fergus, 2014] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.