DHBW Stuttgart
Duale Hochschule
Baden-Württemberg

Spatio-Temporal Predictions am Beispiel von Geschwindigkeitskontrollen

Studienarbeit

des Studiengangs Informatik/Informationstechnik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Marco von Rosenberg

Freitag, der 29. Juli 2022

Matrikelnummer, Kurs	9594981, TINF19IN
Ausbildungsfirma	Keysight Technologies, Böblingen
Betreuer	Johannes Staib
Bearbeitungszeitraum	13. September 2021 - 29. Juli 2022

Selbständigkeitserklärung

Name:	Marco von Rosenberg
Matrikelnummer:	9594981
Studiengang:	Informatik/Informationstechnik
Kurs:	TINF19IN
Titel:	Spatio-Temporal Predictions am Beispiel von Geschwindigkeitskontrollen

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Falls sowohl eine gedruckte als auch elektronische Fassung abgegeben wurde, versichere ich zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Tübingen, 03.06.2022 — Ort, Datum

M. v. Rosenberg — Unterschrift

Abstract

Spatio-temporal predictions have gained more and more popularity over the last few years. Spatio-temporal neural networks have been used for many different applications, including precipation forecasting, traffic forecasting or crime prediction. In this work, the spatio-temporal model *ConvLSTM* is applied to the prediction of speed camera locations. The model is trained and tested on a dataset covering seven years of reported speed camera locations in Germany. To be able to apply ConvLSTM, the raw locations are rasterized and a heatmap is generated for each day. The model then predicts, based on the preceeding 16 days, the heatmap of the following day. It is concluded that spatio-temporal neuronal networks are indeed applicable to the prediciton of speed camera locations, since the resulting predictions largely match the ground truth. Since the predictions match the actual ground truth significantly better than another random day of the same day of week, it can also be concluded that the data contains certain patterns by which the speed camera locations on a given day depend on those of the preceeding 16 days. Moreover it is confirmed that the model is able to learn these patterns.

Kurzfassung

Räumlich-zeitliche Vorhersagen haben in den letzten Jahren deutlich an Popularität zugenommen. Räumlich-zeitliche neuronale Netze wurden in der Vergangenheit erfolgreich auf verschiedene räumlich-zeitliche Aufgabenstellungen angewendet, wie z. B. die Vorhersage von Niederschlägen, Verkehrsaufkommen oder Verbrechen. In dieser Arbeit wird das räumlich-zeitliche Modell *ConvLSTM* auf die Vorhersage von mobilen Radarkontrollen angewendet. Das Modell wird mit einem Datensatz trainiert und evaluiert, der die in einer App gemeldeten Standorte von mobilen Radarkontrollen der letzten sieben Jahre in ganz Deutschland enthält. Um ConvLSTM auf den Datensatz anwenden zu können, werden die Standorte zunächst rasterisiert und somit wird eine Heatmap für jeden Tag erstellt. Das Modell sagt dann anhand von 16 Tagen als Eingabe die Heatmap des 17. Tages voraus. Es ist festzustellen, dass räumlich-zeitliche neuronale Netze tatsächlich auf die Vorhersage von mobilen Radarkontrollen anwendbar ist, da die Vorhersagen größtenteils mit der Grundwahrheit übereinstimmen. Da die Vorhersagen deutlich besser mit der Grundwahrheit des tatsächlichen Zieltages übereinstimmen als mit der eines zufälligen anderen Tages vom selben Wochentag, kann auch festgestellt werden, dass die Daten bestimmte Muster enthalten, durch die die Standorte von mobilen Radarkontrollen an einem bestimmten Tag von denen der letzten 16 Tage abhängen. Zudem ist bestätigt, dass das vorgestellte Modell diese Muster erlernen kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeausschnittsverzeichnis

Abkürzungen

API *Application Programming Interface*
AUPRC *Area Under the Precision Recall Curve*

CNN *convolutional neural network*

GDAL *Geospatial Data Abstraction Library*
GIMP *GNU Image Manipulation Program*
GIS *Geoinformationssystem*
GPKG *GeoPackage*
GPU *Graphical Processing Unit*
GUI *Graphical User Interface*

LSTM *Long Short Term Memory*

MSE *mean squared error*

NN *neuronales Netz*

ORM *Object Relational Mapper*

PRC *Precision Recall Curve*

ReLU *rectified linear unit*
RGB *Rot Grün Blau*
RNN *rekurrentes neuronales Netz*

SQL *Structured Query Language*

TIFF *Tag Image File Format*

1 Einführung

1.1 Problemstellung und Zielsetzung

In der App *Blitzer.de* können Benutzer mobile und teilstationäre Radarkontrollen sowie Staus, Unfälle und sonstige Gefahren melden. Diese werden anderen Benutzern angezeigt, wodurch sie vorgewarnt sind und sich entsprechend vorsichtig verhalten können. Jedoch sind die Meldungen naturgemäß zeitverzögert, da aktive Benutzer benötigt werden, um sie aktuell zu halten. Es lässt sich argumentieren, dass die Zeitverzögerung v. a. in den frühen Morgenstunden und in ländlichen Gebieten signifikant ist, da sich hier nur wenige Benutzer auf den Straßen befinden. Dies ist besonders für mobile Radarkontrollen entscheidend, da diese jeden Tag zu unterschiedlichen Zeiten und an unterschiedlichen Orten aufgebaut werden.

Andererseits ist auch denkbar, dass es Zusammenhänge zwischen vergangenen und zukünftigen Standorten von Radarkontrollen gibt. Dies ist, wie Chollet in [1] auf Seite 152 argumentiert, eine wichtige Annahme, derer man sich bei jeglichen Vorhersagen bewusst sein muss. Diese Zusammenhänge haben im vorliegenden Anwendungsfall sowohl räumliche als auch zeitliche Aspekte. Beispielsweise liegt es nahe, dass mobile Radarkontrollen zeitlich möglichst gut gestreut werden. Daher kann man annehmen, dass die Gefahr für eine mobile Radarkontrolle an einem Ort eher gering ist, wenn dort am vorherigen Tag eine solche anzutreffen war. Die Gefahr sollte jedoch stetig steigen, je länger an diesem Ort keine Radarkontrolle steht. Eine beispielhafte Annahme für einen rein räumlichen Zusammenhang ist, dass mobile Radarkontrollen in ländlichen Gebieten selten sehr dicht aufeinanderfolgen. Befindet sich also an einem bestimmten Ort eine mobile Radarkontrolle, ist es unwahrscheinlich, in direkter Umgebung eine weitere Radarkontrolle anzutreffen. Je weiter weg man sich begibt, desto höher steigt die Wahrscheinlichkeit jedoch.

Ziel der vorliegenden Arbeit ist daher, die Gefahr für mobile Radarkontrollen anhand historischer Daten und insbesondere derer der letzten Tage für den Folgetag zu prognostizieren. Wie in Abschnitt 2.6 gezeigt wird, gibt es einige vielversprechende Ansätze, wie *neuronale Netze* (NNs) verwendet werden können, um sehr ähnliche Problemstellungen anzugehen. Ähnliche Problemstellungen sind in diesem Fall Vorhersagen mit räumlichen und zeitlichen Aspekten, wie z. B. die Vorhersage von Verkehrsunfällen oder Verbrechen. Daher sollen in dieser Arbeit auch NNs verwendet werden, um die Vorhersagen anzustellen.

Die erwähnten historischen Daten bestehen aus allen gemeldeten Standorten von mobilen Radarkontrollen der vorhergegangenen Jahre. Dieser Datensatz wurde freundlicherweise von der Eifrig Media GmbH bereitgestellt. Die Eifrig Media GmbH steht hinter der Entwicklung der oben genanten App *Blitzer.de*. Der Datensatz enthält über 7,7 Millionen gemeldete mobile Radarkontrollen vom 22.05.2014 bis zum 25.10.2021. Daraus ergibt sich ein Zeitraum von 2713 Tagen und somit ca. 21.300 Meldungen pro Tag. Die Meldungen beschränken sich hierbei auf Deutschland, wobei die

Eifrig Media GmbH in [12] angibt, auch äquivalente Angebote zu *Blitzer.de* im Ausland bereitzustellen. Jeder Eintrag des Datensatzes enthält u. A. die Koordinaten sowie den ungefähren Aufbau- und Abbauzeitpunkt der Radarkontrolle. Diese Daten basieren jedoch auf den Angaben der Appbenutzer und sollten daher kritisch betrachtet werden. So entsprechen die Koordinaten einer Radarkontrolle dem Standort des Appbenutzers zum Zeitpunkt der Meldung in der App. Da die Meldungen meist bei laufender Fahrt erfolgen, kann die gemeldete Position auf einem Straßenabschnitt relativ weit vor oder nach der tatsächlichen Position der Radarkontrolle liegen. Der Abbauzeitpunkt basiert ebenfalls auf Angaben der Appbenutzer. Sobald ein Benutzer an einer eingetragenen Radarkontrolle vorbeigefahren ist, wird dieser gefragt, ob die Radarkontrolle gesehen wurde. Wenn genügend Benutzer dies verneint haben, wird die Radarkontrolle anderen Benutzern nicht mehr angezeigt. Dieser Zeitpunkt entspricht dann auch dem Abbauzeitpunkt. Da prinzipiell auch Fehlmeldungen möglich sind, ist es wichtig, die Gesamtdauer der Meldung zu betrachten. Beträgt die diese beispielsweise nur 10 Minuten, kann davon ausgegangen werden, dass es sich um eine Fehlmeldung handelt. Je länger die sie hingegen ist, desto höher ist die Wahrscheinlichkeit, dass es sich um eine valide Meldung handelt, weil sie von vielen Benutzern bestätigt wurde.

Sowohl die Ungenauigkeit der gemeldeten Position als auch die Möglichkeit für Fehlmeldungen machen es erforderlich, den Datensatz aufzubereiten. Es bietet sich an, den betrachteten Bereich in ein Raster zu unterteilen und jede Meldung am betrachteten Tag derjenigen Rasterzelle zuzuordnen, in deren Bereich die Meldung liegt. Dieses Vorgehen macht die Ungenauigkeit der Position sowie die Ungenauigkeit des Auf- und Abbauzeitpunkts weniger signifikant.

1.2 Aufbau der Arbeit

In Abschnitt 2 werden zunächst die Grundlagen des Machine Learnings erläutert. Außerdem werden einige Architekturen für verschiedene Problemstellungen vorgestellt, die für die oben definierte Problemstellung relevant sind. In Abschnitt 3 wird der verwendete Datensatz in Grundzügen analysiert und anschließend so verarbeitet, dass er als Eingabe für ein NN verwendet werden kann. Als nächstes wird in Abschnitt 4 eine geeignete Modellarchitektur ausgewählt. Diese wird implementiert und mit dem vorbereiteten Datensatz trainiert. Im Anschluss wird erörtert, wie die Ausgaben des Modells nachbearbeitet werden müssen, um eine gut interpretierbare Vorhersage zu erhalten. Anhand dieser Ergebnisse wird die Performance des Modells nun evaluiert. In Abschnitt 5 werden schlussendlich die Ergebnisse der Arbeit zusammengefasst und es wird ein Ausblick auf weitere mögliche Optimierungen gegeben.

2 Grundlagen

In diesem Abschnitt werden zunächst die Grundlagen des Machine Learning erläutert sowie zwei häufig verwendete Herangehensweisen für räumliche Daten und Zeitreihen. Dies dient einer schrittweisen Heranführung an Modelle, die für räumlich-zeitliche Vorhersagen verwendet werden können. Außerdem wird die Problemstellung der räumlich-zeitlichen Vorhersagen formalisiert.

2.1 Machine Learning und Deep Learning

Machine Learning ist ein Teilgebiet der Informatik, das sich fundamental von der klassischen Programmierung unterscheidet. Diese Unterscheidung wird von Chollet in [1] auf den Seiten 23 und 24 beschrieben. Sowohl klassische Programmierung als auch Machine Learning sollen Chollet zufolge im Rahmen von künstlicher Intelligenz verschiedenste Fragestellungen beantworten, die ansonsten von Menschen beantwortet werden. Ein Beispiel für eine solche Fragestellung könnte z. B. lauten: „Welche Ziffer ist in diesem Bild dargestellt?“. Ein anderes Beispiel wäre „Gegeben ist dieser halbe Satz, wie könnte er weiter gehen?“. Die Fragestellung der vorliegenden Arbeit wurde bereits definiert. Sie kann wie folgt in einem Satz zusammengefasst werden: „Gegeben sind die Standorte von mobilen Radarkontrollen der letzten 16 Tage, wie groß ist die Gefahr für mobile Radarkontrollen am 17. Tag an verschiedenen Stellen?“ Bei der klassischen Programmierung liegt es nach Chollet nun am Menschen, Regeln aufzustellen, nach denen die Fragestellungen beantwortet werden. Der Mensch könnte sich vergangene Daten anschauen und müsste in ihnen Muster suchen, von denen diese Regeln abgeleitet werden können. Die Regeln werden anschließend in einer Programmiersprache explizit in Form von Variablen, Verzweigungen, Schleifen und Funktionen implementiert. Für die Vorhersage von mobilen Radarkontrollen könnte ein Mensch beispielsweise aus eigener Erfahrung im Straßenverkehr gewisse Muster erkennen. Daraus könnte z. B. die Regel aufgestellt werden, dass die Wahrscheinlichkeit für eine mobile Radarkontrolle an einem Standort sehr gering ist, wenn dort am vorherigen Tag bereits eine solche aufgestellt war. Wie in Abbildung 1 zusammengefasst wird, findet die klassische Programmierung auf Basis von Eingangsdaten und explizit implementierten Regeln Antworten.

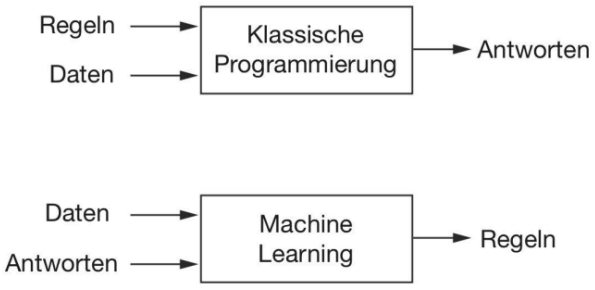

Abbildung 1: Unterschied zwischen klassischer Programmierung und Machine Learning [1, Abb. 1.2]

Beim Machine Learning ist es anders herum. Ein Machine Learning Algorithmus erhält als Eingabe

Daten und Antworten und leitet daraus Regeln ab, wie auch in Abbildung 1 zu sehen. Diese Regeln können anschließend verwendet werden, um anhand von neuen Eingabedaten neue Antworten zu erzeugen. Die Regeln müssen nun also nicht mehr explizit vom Menschen festgelegt werden. Dies bringt den Vorteil mit sich, dass möglicherweise Regeln erkannt werden, die dem Menschen (z. B. aufgrund ihrer Komplexität) verborgen blieben. Für die Vorhersage von mobilen Radarkontrollen würde ein Machine Learning Algorithmus sehr viele Beispiele von je 16 aufeinanderfolgenden Tagen als Eingangsdaten sowie den 17. Tag als Antwort erhalten. Anhand der daraus erlernten Regeln und 16 neuen Tagen als Eingabe, könnte dann der 17. Tag vorhergesagt werden. Chollet beschreibt auf Seite 152, dass hierfür jedoch zwei Voraussetzungen erfüllt sein müssen: Zum einen muss der 17. Tag anhand der 16 vorangegangenen Tage vorhersagbar sein. Es muss also einen signifikanten Zusammenhang zwischen Vergangenheit und Zukunft geben, der über zufälliges Rauschen hinaus geht. Andererseits müssen genügend Trainingsdaten vorhanden sein, um die Regeln daraus ableiten zu können. Jedes Muster, dass in Zukunft erkannt werden soll, muss also in den Trainingsdaten vorhanden sein. Zu Beginn eines Machine Learning Projekts müssen diese Voraussetzungen nach Chollet als wahr angenommen werden. Je nach Performanceergebnis können sich diese Annahmen schlussendlich als wahr oder falsch herausstellen.

Als Nächstes stellt sich die Frage, wie ein solcher Machine Learning Algorithmus funktioniert. Innerhalb des Machine Learnings gibt es verschiedene Algorithmen, die von Alzubi et al. in [13] zusammengefasst sind. Nennenswerte Algorithmen sind nach Alzubi et al. beispielsweise *Decision Tree*, *Naïve Bayes* oder *Random Forest*. In der vorliegenden Arbeit sollen jedoch neuronale Netze (NNs) verwendet werden. Das Perzeptron (auch Zelle, Neuron oder Element genannt) bildet nach [2] den Grundbaustein von NNs. Die verschiedenen Bestandteile eines Perzeptrons sind in Abbildung 2 grafisch visualisiert.

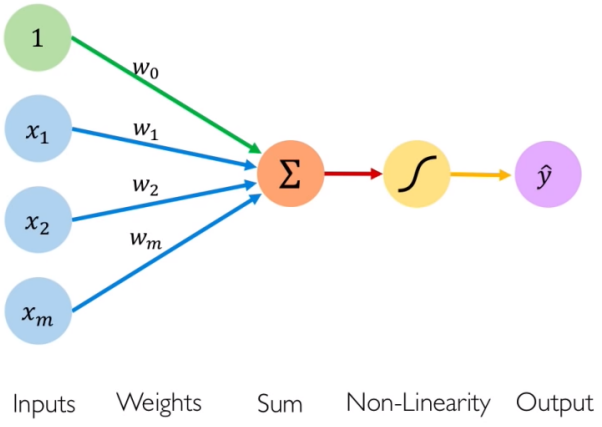

Abbildung 2: Grafische Visualisierung eines Perzeptrons als Grundbaustein von NNs [2]

Ein Perzeptron hat eine beliebige Anzahl von Eingängen (engl. *inputs*). Die Eingänge $x_1 \dots x_n$ werden mit den Gewichtungen (engl. *weights*) $w_1 \dots w_m$ multipliziert und die Ergebnisse aufsummiert. Unabhängig von den Eingängen wird dann noch ein Bias w_0 dazu addiert. Zuletzt wird eine nichtlineare Aktivierungsfunktion g auf das Ergebnis angewendet. Bekannte Aktivierungsfunktionen sind z. B. *Sigmoid* oder *ReLU (rectified linear unit)*, wobei in der Praxis meistens *ReLU* verwendet wird

[14, 15]. Die gesamte Berechnung des Ausgangs $\hat{y}$ ist in der aus [2] übernommenen Gleichung 1 zu sehen.

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right) \tag{1}$$

Ein neuronales Netz entsteht nun, wenn mehrere dieser Perzeptronen kombiniert werden. Die einfachste Möglichkeit ist ein sogenanntes Fully-connected Feedforward-Netz. Hierbei werden mehrere Schichten aus parallel angeordneten Perzeptronen gebildet. Dies ist schematisch in Abbildung 3 dargestellt.

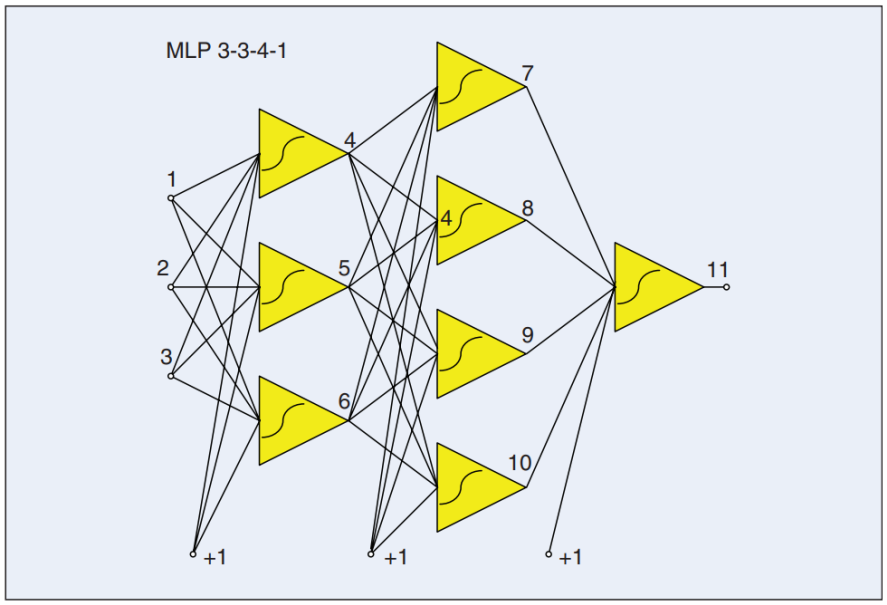

Abbildung 3: Architektur eines Fully-connected NN [3, FIGURE 1]

Wie in der Abbildung zu sehen ist, ist der Ausgang eines jeden Perzeptrons mit jedem Perzeptron der nächsten Schicht verbunden. Der häufig verwendete Begriff *Deep Learning* bezieht sich nach [1, S. 27] auf die Vorgehensweise, Perzeptronen in mehreren Schichten anzuordnen. Der Hintergedanke dabei ist nach [2], dass durch die Schichten eine Merkmalshierarchie aufgebaut werden soll. Bei der Erkennung von handgeschriebenen Ziffern könnte die erste Schicht beispielsweise Ecken und Kanten erkennen. Die zweite Schicht würde aus Ecken und Kanten zusammengesetzte Kreise und Linien erkennen. Schließlich könnte die dritte Schicht komplette Ziffern erkennen, die wiederum aus Linien und Kreisen zusammengesetzt sind.

Bisher wurde nur das eigentliche NN erklärt. Um ein NN zu trainieren, sind jedoch noch einige andere Komponenten notwendig. Das Zusammenspiel dieser Komponenten ist in Abbildung 4 dargestellt. Wie in der Abbildung erkennbar ist, wird eine Verlustfunktion benötigt. Diese erhält nach [1, S. 30] als Eingabe die Vorhersagen Y' des NNs sowie die tatsächlichen Werte Y. Die Verlustfunktion berechnet anhand dieser Eingaben einen Verlustscore (engl. *loss*), der ein Maß dafür ist, wie stark

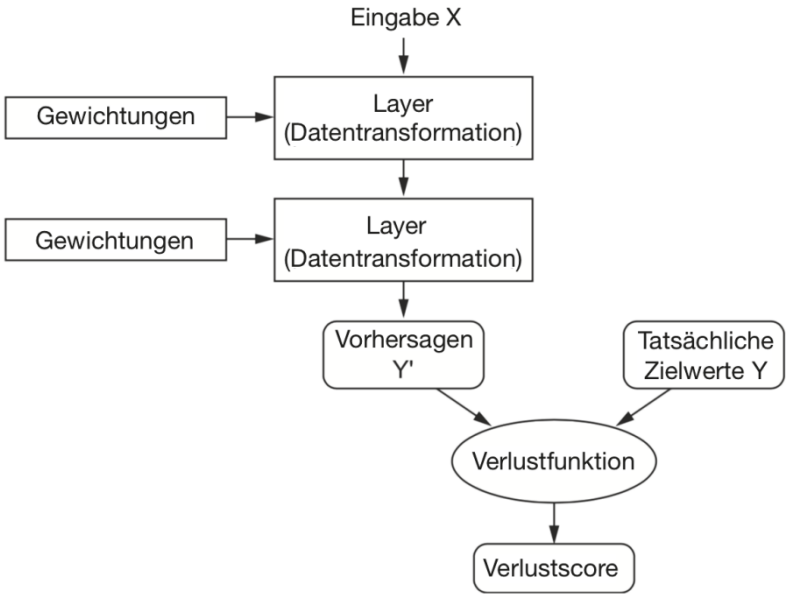

Abbildung 4: Komponenten eines Deep Learning Systems [1, Abb. 1.9]

die Vorhersagen von den tatsächlichen Werten abweichen. Die Ausgabe der Verlustfunktion ist umso größer, je stärker diese Abweichung ist. Es gibt viele verschiedene Verlustfunktionen, die für unterschiedliche Aufgabenstellungen geeignet sind. Häufig verwendet werden nach [1, S. 155] z. B. der mittlere quadratische Fehler (engl. *mean squared error* (MSE)) oder die binäre Kreuzentropie (engl. *binary crossentropy*).

Als letzte wichtige Komponente wird noch ein Optimierer benötigt. Dieser erhält als Eingabe den Verlustscore und passt anhand dessen die Gewichtungen des NNs so an, dass der Verlustscore minimiert wird. Hierfür wird ein Algorithmus namens *Backpropagation* verwendet [1, S. 30]. Von diesem allgemeinen Algorithmus gibt es viele verschiedene Implementierungen. In der vorliegenden Arbeit wird der *Adam*-Optimierer [16] verwendet. Die Vorteile dieses Optimierers sind, den Autoren zufolge, die einfache Implementierung sowie die Speicher- und Recheneffizienz.

Zusammengefasst ist Deep Learning nichts anderes als eine Funktionsminimierung. Die bedeutende Errungenschaft dabei ist, dass die große Menge an Gewichtungen (bei größeren NNs mehrere Millionen) effizient angepasst werden können, sodass der Verlustscore minimiert wird.

2.2 Long Short Term Memory

Die einzige bisher vorgestellte Architektur von NNs ist das Feedforward-Netz. Diese Netzarchitektur eignet sich gut für Klassifizierungsaufgaben. Die vorliegende Arbeit beschäftigt sich jedoch mit Standorten von mobilen Radarkontrollen über die Zeit, also mit sequenziellen Daten. Feedforward-Netze können zeitliche Zusammenhänge nicht darstellen und nicht erlernen, da sie keinen internen Zustand haben. Das bedeutet, dass die Ausgabe des NNs nur abhängig von der aktuellen Eingabe

ist, nicht aber von vorherigen Eingaben.

Abhilfe hierbei bieten *rekurrente neuronale Netze* (RNNs). Wie Chollet in [1, S. 252] erläutert, besitzen rekurrente NNs einen internen Zustand, der alle bisherigen Eingaben repräsentiert. Eine Ausgabe ist dann sowohl von der Eingabe als auch vom internen Zustand abhängig. Implementiert wird dieses Verhalten durch eine Schleife im NN. In Abbildung 5 ist diese Architektur schematisch dargestellt.

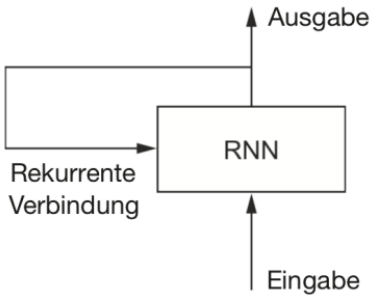

Abbildung 5: Schleife in einem RNN [1, Abb. 6.9]

Ein einfaches RNN berechnet die Ausgabe Y_t wie in Gleichung 2 [1, S. 253] gezeigt.

$$Y_t = a(W \cdot X_t + U \cdot S_t + b) \tag{2}$$

Dabei steht X_t für die Eingabe und S_t für den internen Zustand, beide zum Zeitschritt t. W und U sind Matrizen, die die trainierbaren Gewichtungen enthalten und b ist ein trainierbarer Bias-Vektor. Nach der Berechnung wird die Ausgabe zum neuen internen Zustand, man könnte also S_t in Gleichung 2 durch Y_{t-1} ersetzen.

Diese einfache Architektur unterliegt nach [1, S. 260] jedoch dem sogenannten *Problem des verschwindenden Gradienten*. Dieser Effekt sorgt dafür, dass relativ weit zurückliegende Eingaben praktisch keinen Einfluss mehr auf die Ausgabe haben. Es sind jedoch Anwendungsfälle denkbar, in denen auch weiter zurückliegende Ereignisse einen großen Einfluss auf die Gegenwart haben. Für die Vorhersage von mobilen Radarkontrollen könnte beispielsweise von Bedeutung sein, wie die Verteilung der Radarkontrollen 15 Tage in der Vergangenheit ausgesehen hat. Das liegt daran, dass eine Periodizität der Daten von z. B. 15 Tagen durchaus denkbar ist. Zur Lösung dieses Problems gibt es verschiedene alternative RNN-Architekturen. Eine davon ist *Long Short Term Memory* (LSTM). Wie der Name vermuten lässt, ermöglicht die LSTM-Architektur, sowohl Abhängigkeiten von weit zurückliegenden als auch aktuellere Eingaben zu lernen. Die LSTM-Architektur ist in Abbildung 6 dargestellt.

Eine LSTM-Zelle enthält drei sogenannte Tore (engl. *gates*). Jedes Tor ist selbst ein NN, enthält also trainierbare Gewichtungen. Die drei Tore und ihre jeweilige vorgesehene Wirkung sind nach [4]:

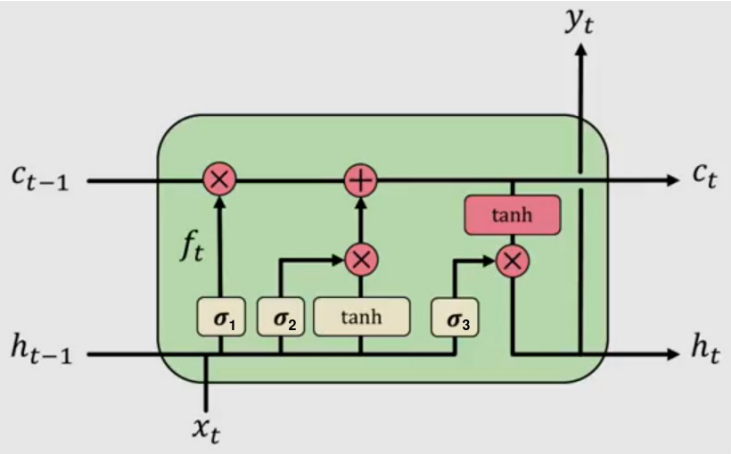

Abbildung 6: Schematische Darstellung einer LSTM-Zelle [4]

1. **Vergessens-Tor** σ_1: Dieses Tor sorgt dafür, dass irrelevante Informationen aus dem vorherigen Zustand „vergessen“ werden.
2. **Merk-Tor** σ_2: Dieses Tor fügt dem Zustand relevante neue Informationen hinzu.
3. **Ausgangs-Tor** σ_3: Dieses Tor bestimmt, welche Informationen aus dem Zustand zum Ausgang der Zelle gelangen sollen.

Amini und Soleimany stellen die vorgesehene Wirkung der Tore in [4] ohne weitere kritische Auseinandersetzung dar. Wie Chollet in [1, S. 263] argumentiert, sei diese Wirkung jedoch keinesfalls garantiert. Die tatsächliche Wirkung hinge viel mehr von den letztendlich antrainierten Gewichtungen der Tore ab. Alle drei sind sich jedoch einig, dass es nicht wichtig ist, die interne Funktionsweise einer LSTM-Zelle im Detail zu verstehen. Chollet geht noch einen Schritt weiter und argumentiert, dass dies ganz allgemein keine Aufgabe der Menschen sei. Wichtig wäre nur, sich im Klaren zu sein, welche Aufgabe eine LSTM-Zelle erfüllt: Sie ermöglicht es, anhand der Trainingsdaten sowohl langfristige als auch kurzfristige Zusammenhänge zu erlernen.

2.3 Faltungsnetze

Mit der bisher vorgestellten LSTM-Architektur können die zeitlichen Zusammenhänge bei der Vorhersage von mobilen Radarkontrollen erlernt werden. Jedoch wurde bereits erläutert, warum auch räumliche Zusammenhänge für diese Aufgabenstellung von Bedeutung sind. Prinzipiell ist es möglich, 2D-Daten mit Feedforward-Netzen zu verarbeiten. Dazu müssen die 2D-Daten verflacht werden. Für ein zweidimensionales Bild bedeutet das, dass jedes Pixel einem Eingangsneuron zugeführt wird. Hierbei geht die räumliche Struktur der Daten jedoch verloren. Das NN kann nicht wissen, welche Pixel in alle vier Richtungen benachbart waren. Mit genug Trainingsdaten können einfache Klassifizierungsaufgaben dennoch mit dieser Netzarchitektur bearbeitet werden. Chollet demonstriert beispielsweise in [1, S. 53], dass bei der Klassifizierung von handgeschriebenen Ziffern mit einem Feedforward-Netz eine Korrektklassifizierungsrate von 97,8 % erreicht werden kann. Das Problem mit Feedforward-

Netzen ist jedoch, dass sie nur globale Muster erlernen, also beispielsweise eine Ziffer als Ganzes. Ist dieselbe Ziffer nun etwas im Bild verschoben oder auf sonstige Weise verzerrt, wird sie von einem Feedforward-Netz nicht mehr erkannt. Dies kann dadurch visualisiert werden, dass nach einer Verzerrung die einzelnen Pixel an ganz anderen Eingangsneuronen anliegen.

Abhilfe hierbei verschaffen Faltungsnetze (engl. *convolutional neural network*s, CNNs). CNNs entsprechen dem Stand der Technik, wenn es um die Verarbeitung von 2D-Daten geht. Wenn das vorherige Beispiel nochmals betrachtet wird, kann man erkennen, welche Eigenschaften der Ziffern sich nicht durch eine Verzerrung ändern: Die Zusammensetzung der Ziffern aus kleineren Merkmalen. Selbst wenn eine Ziffer verschoben oder leicht verzerrt wird, werden sich (relativ zueinander) immer noch dieselben Linien an denselben Stellen kreuzen oder berühren. Die Betrachtung von 2D-Daten als Zusammensetzung von lokalen Mustern wird in Abbildung 7 verdeutlicht.

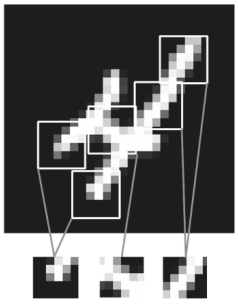

Abbildung 7: Lokale Muster wie Ränder und Linien in einer handgeschriebenen Ziffer [1, Abb. 5.1]

CNNs erkennen nach [1, S. 164] nun diese lokalen Muster. Chollet geht in [1] auf Seite 165 auf die daraus resultierenden Eigenschaften von CNNs ein. Zunächst sei die Erkennung der lokalen Muster translationsinvariant. Dies bedeute, dass die Muster an beliebigen Stellen im Bild erkannt werden können. Außerdem könne durch das Hintereinanderschalten mehrer CNN-Layer erreicht werden, dass Hierarchien von Mustern erlernt werden. Aus diesen beiden Eigenschaften folgt, dass es für ein CNN keinen Unterschied macht, ob die zu erkennenden globalen Muster (beispielsweise eine Ziffer) in der Eingabe verschoben sind.

Als Nächstes stellt sich die Frage, wie CNNs lokale Muster erkennen können. Dies wird durch die Faltungsoperation erreicht. Die Idee der Faltungsoperation ist nach [5], einen sogenannten Filter zu verwenden, der lokale Muster erkennt. Dieser zweidimensionale Filter wird mit der 2D-Eingabe gefaltet und erkennt somit, wo sich in der Eingabe ein bestimmtes Muster befindet. Mathematisch gesehen ist ein Filter eine quadratische Matrix. Die Elemente der Matrix sind die erlernbaren Gewichtungen, die je nach ihren Werten verschiedene Muster erkennen. Die Erkennung geschieht dadurch, dass der Filter an der jeweiligen Stelle komponentenweise mit der Eingabe multipliziert und die einzelnen Werte anschließend aufsummiert werden. Diese Berechnung wird in Abbildung 8 verdeutlicht.

Links oben in der Abbildung ist der Filter dargestellt. Dieser Filter erkennt schräge Linien von links oben nach rechts unten. Bei einer perfekten Übereinstimmung, wie im Beispiel der Abbildung, wird das Ergebnis der Berechnung maximal. Angenommen, das Pixel oben in der Mitte der Eingabe wäre

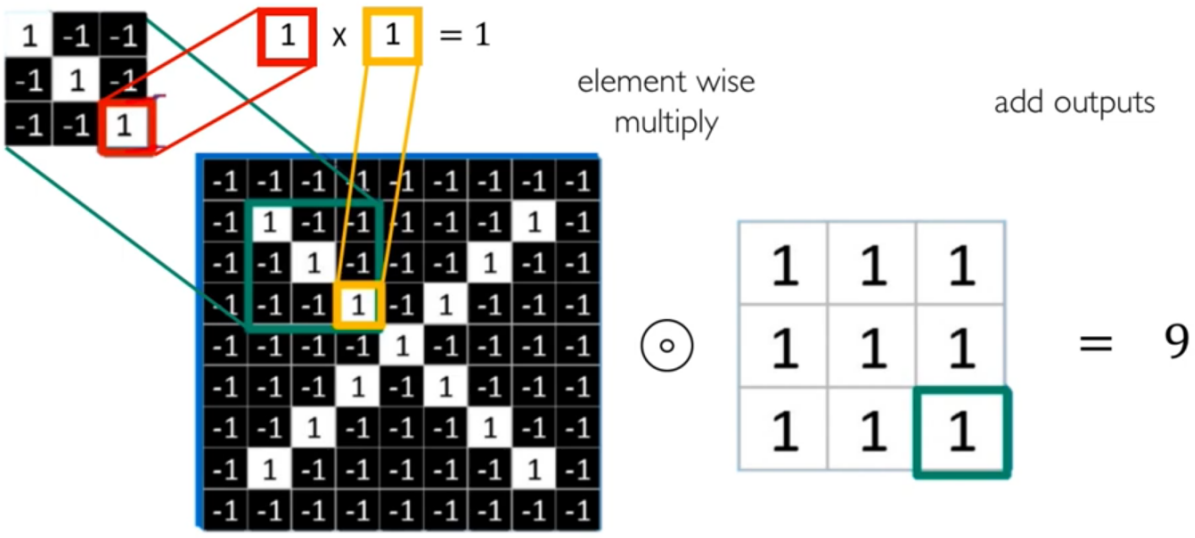

Abbildung 8: Ein Schritt der Faltung des Filters mit der Eingabe [5]

1, dann wäre das Ergebnis der Berechnung nur 7. Der Wert des Ergebnisses ist also ein Maß für die Übereinstimmung des Filters mit der Eingabe. Auf dieses Ergebnis wird in der Praxis noch eine Aktivierungsfunktion angewendet, wofür meistens *ReLU* verwendet wird [5]. Negative Pixel werden dadurch zu null. Wird der Filter nun mit einer Schrittweite von 1 in x- und y-Richtung über die gesamte Eingabe geschoben und die Berechnung jedes Mal ausgeführt, entsteht eine sogenannte Merkmalskarte (engl. *feature map*). Die Merkmalskarte gibt an, wo im Bild die Übereinstimmung mit dem durch den Filter beschriebenen Muster wie groß ist. Damit die Merkmalskarte die gleiche Höhe und Breite wie die Eingabe hat, kann die Eingabe vor Anwendung des Filters rundherum mit Nullen aufgefüllt werden [1, S. 168 f.]. Dies wird auch *Padding* genannt.

Nach der Faltungsoperation ist noch ein weiterer wichtiger Schritt notwendig. Da die Merkmalskarte zunächst die gleichen Dimensionen wie die Eingabe hat, können nach [1, S. 171] keine Merkmalshierarchien erkannt werden. Das liegt daran, dass ein Pixel der Merkmalskarte nur Informationen über einen Bereich der Eingabe enthält, der so groß ist wie der Filter. In den meisten Fällen sind dies nur 3x3 oder 5x5 Pixel. Das Ziel ist es also, eine Merkmalskarte zu erstellen, bei der ein Pixel Informationen über einen größeren Bereich der Eingabe enthält. Dies kann durch die MaxPooling-Operation erreicht werden, die auf die Merkmalskarte angewendet wird. Diese funktioniert ähnlich wie die Faltungsoperation. Auch hier gibt es einen Filter, der über die Eingabe geschoben wird. Dieser Filter wird hier jedoch als *Pool* bezeichnet. Im Gegensatz zur Faltungsoperation wird nichts multipliziert. Stattdessen entspricht die Ausgabe des Pools dem Maximalwert der Eingabe. Hat der Pool beispielsweise eine Größe von 2x2 und wird in x- und y-Richtung jeweils um die Schrittweite 2 verschoben, wird die Seitenlänge der Eingabe durch die MaxPooling-Operation halbiert. Dies wird in Abbildung 9 an einem Beispiel veranschaulicht.

Werden mehrere CNN- und MaxPooling-Layer abwechselnd hintereinandergeschaltet, lassen sich die Merkmale der Eingabe hierarchisch darstellen. Dabei wird die Höhe und Breite der Eingabe nach jedem MaxPooling-Layer halbiert. Zumm Schluss wird die Ausgabe des letzten MaxPooling-Layers

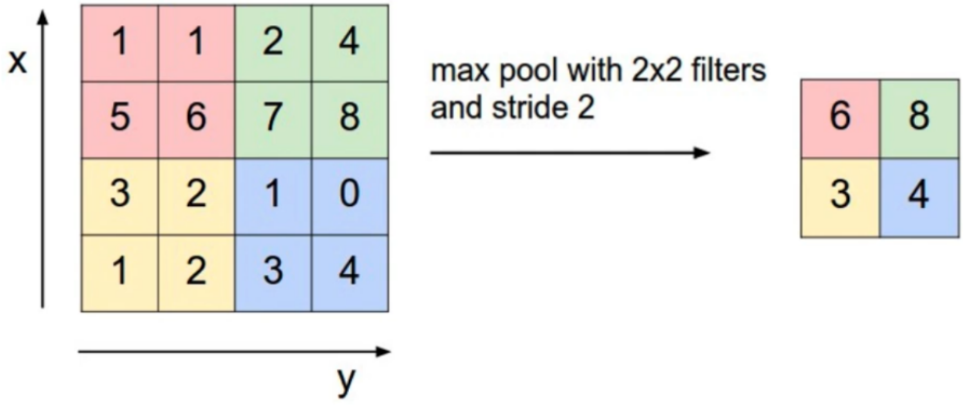

Abbildung 9: Beispiel einer MaxPooling-Operation [5]

an ein Feedforward-Netz zur Klassifizierung übergeben. Das Feedforward-Netz klassifiziert die Eingabe also nun anhand von abstrakten Merkmalen, die durch die CNN- und MaxPooling-Layer von *beliebigen Stellen* in der Eingabe extrahiert wurden.

2.4 TensorFlow

TensorFlow ist eine Python-Bibliothek für viele verschiedene Machine Learning Aufgaben. TensorFlow implementiert dafür u.A. mathematische Grundfunktionen wie Tensoroperationen oder die Gradientenberechnung. Da sehr viel Fachwissen nötig wäre, um alleine mit TensorFlow NNs zu implementieren, wird mit TensorFlow auch die Keras-Bibliothek ausgeliefert. Keras baut auf TensorFlow auf und bietet ein High-Level-Framework für neuronale Netze. Mit Keras können NNs relativ einfach definiert, trainiert und ausgewertet werden. Keras bietet sehr viele vordefinierte Layer, darunter Feedforward-, LSTM-, CNN- sowie MaxPooling-Layer. Bei Bedarf können jedoch auch auch eigene Layer definiert werden. Aus diesen vordefinierten und ggf. benutzerdefinierten Layern können beliebig komplexe Modelle zusammengesetzt werden. Außerdem kann der Ablauf des Trainings dieser Modelle komplett angepasst werden, inklusive des Optimierers und der Verlustfunktion. Da die Trainierung von NNs sehr rechenintensiv ist, bietet TensorFlow die Möglichkeit, die Berechnungen auf einer Grafikkarte auszuführen. Da Grafikkarten Berechnungen hochgradig parallelisiert ausführen, können Modelle deutlich schneller trainiert werden als auf der CPU.

2.5 Formale Problemdefinition räumlich-zeitlicher Vorhersagen

Im Themenfeld der räumlich-zeitlichen Vorhersagen (engl. *spatio-temporal predictions*) gibt es viele verschiedene Anwendungsfälle. Zu diesen Anwendungsfällen gehört z. B. die Vorhersage von Niederschlägen, Verbrechen, Verkehrsunfällen, Verkehrsaufkommen sowie der Bedarf an Taxis [7, 10, 17, 18, 19, 6]. So unterschiedlich die Themengebiete auch sind, lässt sich laut [20] dennoch eine gemeinsame, formale Problemstellung definieren. Wenn die Daten als Raster vorliegen, können sie als 4D-Tensor

$\mathbb{R}^{T\times H\times W\times C}$ beschrieben werden. Dabei ist T die Anzahl der Zeitschritte, $H \times W$ die räumliche Dimension (Höhe mal Breite) und C die Anzahl an Kanälen. Besteht der Datensatz beispielsweise aus 100 RGB-Bildern mit einer Größe von 28x28 Pixel, kann er mit dem Tensor $\mathbb{R}^{100\times 28\times 28\times 3}$ beschrieben werden. Die Problemstellung lässt sich nun als die in Gleichung 3 gezeigte Abbildung f formulieren.

$$f : [X_{t-(\alpha-1)}, \ldots, X_{t-1}, X_t] \rightarrow [X_{t+1}, X_{t+2}, \ldots, X_{t+\beta}] \tag{3}$$

Dabei repräsentiert $X_t \in \mathbb{R}^{H\times W\times C}$ ein Raster zum Zeitpunkt t. Außerdem ist α die Anzahl an Zeitschritten, anhand derer die Vorhersage getroffen wird und β die Anzahl an Zeitschritten, die vorhergesagt werden soll. Es ist auch möglich, nur einen Zeitschritt vorherzusagen. Wenn beispielsweise anhand von 16 vergangenen Tagen der 17. Tag vorhergesagt werden soll, dann ist $\alpha = 16$ und $\beta = 1$. Daraus ergibt sich $g : [X_{t-15}, X_{t-14}, \ldots, X_{t-1}, X_t] \rightarrow X_{t+1}$. Mit g ist nun die Funktion definiert, die in der vorliegenden Arbeit durch ein NN approximiert werden soll.

Obwohl eine solche allgemeine Definition der Problemstellung möglich ist, gibt es dennoch große Unterschiede, wenn die Definition auf verschiedene Datensätze angewendet wird. Das liegt vor allem daran, dass die Datensätze eine sehr unterschiedliche Sturkur haben und sehr unterschiedlichen Gesetzmäßigkeiten unterliegen. Zunächst gibt es Unterschiede in der räumlichen Struktur der Daten. Grundsätzlich können die Daten räumlich diskret oder kontinuierlich sein. Ein Beispiel für räumlich diskrete Daten sind Wetterdaten. Diese werden von Wetterstationen an bestimmten Orten aufgenommen, die sich über die Zeit nicht ändern. Beispiele für räumlich kontinuierliche Daten sind Verkehrsunfälle oder Verbrechen. Diese können an beliebigen Orten bzw. Koordinaten auftreten. Räumlich kontinuierliche und diskrete Daten können in beide Richtungen umgewandelt werden. Räumlich diskrete Daten können in räumlich kontinuierliche Daten umgewandelt werden, indem zweidimensional interpoliert wird. Hingegen können räumlich kontinuierliche Daten in räumlich diskrete Daten umgewandelt werden, indem Cluster bzw. Hotspots in den Daten identifiziert werden. Alle Datenpunkte, die zu einem Cluster gehören, bekommen dann als Koordinaten den Schwerpunkt des Clusters zugeordnet. Die Voraussetzung für dieses Vorgehen ist jedoch, dass überhaupt Cluster in den Daten erkennbar sind.

Unabhängig davon, ob die Daten ursprünglich räumlich diskret oder kontinuierlich vorliegen, lassen sie sich grundsätzlich auf zwei verschiedene Arten repräsentieren: als Raster oder als Graph. Sollen die Daten als Raster repräsentiert werden, wird ein zunächst leeres Raster aus gleichgroßen, meist quadratischen Zellen über die Daten gelegt. Anschließend werden jeder Rasterzelle diejenigen Datenpunkte zugeordnet, die innerhalb der Zelle liegen. Da jede Rasterzelle aus nur einem Zahlenwert besteht, muss eine bestimmte Metrik festgelegt werden, anhand derer dieser Zahlenwert bestimmt wird. Dies könnte beispielsweise die Anzahl der Datenpunkte innerhalb eines bestimmten Zeitraums sein. Alternativ könnte eine bestimmte Eigenschaft der Datenpunkte aufsummiert werden. Sollen die Daten hingegen als Graph repräsentiert werden, müssen sie zunächst räumlich diskret vorliegen.

Liegen die Daten räumlich kontinuierlich vor, müssen zuerst Cluster gebildet werden. Jeder diskrete Ort bzw. jedes Cluster wird dann ein Knoten des Graphs. Anschließend muss festgelegt werden, welche Verbindungen zwischen den einzelnen Knoten bestehen. Hier bietet es sich an, Verbindungen festzulegen, die in der Realität auch bestehen. Dies könnte beispielsweise das Straßennetz sein, wenn es sich um Verkehrsdaten handelt.

Nicht nur in der räumlichen Struktur der Daten kann es Unterschiede geben, sondern auch in der zeitlichen Struktur. So lassen sich Datensätze im Bezug auf die zeitliche Struktur in zwei Kategorien einteilen: Die Datenpunkte können entweder in regelmäßigen oder in unregelmäßigen Zeitabständen vorliegen. Wetterstationen beispielsweise liefern Messwerte in regelmäßigen Zeitabständen. Verkehrsunfälle geschehen hingegen in unregelmäßigen Zeitabständen. Eine weitere Eigenschaft der Daten ist die zeitliche Auflösung, also der (durchschnittliche) Zeitabstand einzelner Datenpunkte. Wetterstationen liefern Daten in sehr kurzen Zeitabständen. Daher können schon in einem relativ kurzen Zeitraum sehr viele Daten gesammelt werden. Verkehrsunfälle oder Verbrechen treten im Gegensatz dazu nur sehr sporadisch auf. Daher müssen Daten über einen sehr langen Zeitraum gesammelt werden, um auf deren Basis Vorhersagen treffen zu können. Insgesamt ist zeitliche Auflösung wichtig, da sie die erzielbare Genauigkeit und die zeitliche Auflösung der Vorhersagen beschränkt.

Die verschiedenen Möglichkeiten der Kombination von räumlicher und zeitlicher Beschaffenheit der Daten ist mit Beispielen in Tabelle 1 zusammengefasst.

	Räumlich kontinuierlich	**Räumlich diskret**
Regelmäßige Zeitabstände	Regelmäßig gesendete GPS-Daten	Wetterstationen
Unregelmäßige Zeitabstände	Verkehrsunfälle, Verbrechen, mobile Radarkontrollen	

Tabelle 1: Beispiele für Anwendungsfälle mit verschiedenen räumlichen und zeitlichen Strukturen der Daten

Zuletzt gibt es noch den Unterschied zwischen kontinuierlich gemeldeten Messwerten und sporadisch auftretenden Ereignissen. Während der Zahlenwert einer Messung zu einem bestimmten Zeitpunkt intuitiv klar ist, ist dies bei Ereignissen nicht sofort klar. Für die Behandlung von Ereignissen gibt es mehrere Möglichkeiten. Zum einen kann ein Ereignis als ein Messwert mit dem immer gleichen Zahlenwert „1“ betrachtet werden. Zum anderen kann dem Ereignis ein Zahlenwert zugeordnet werden, der das Ereignis bzw. dessen Signifikanz genauer beschreibt. Für mobile Radarkontrollen könnte dieser Zahlenwert beispielsweise die Standdauer sein.

2.6 Neuronale Netze für räumlich-zeitliche Vorhersagen

Um räumlich-zeitliche Zusammenhänge in Datensätzen durch ein NN zu erlernen, existieren verschiedene Architekturen. In [20] befindet sich ein detaillierter Vergleich vieler dieser Architekturen anhand

ihrer Ansätze und Performance. Zwei weit verbreitete Beispiele werden im folgenden zusammengefasst.

ST-ResNet wird in [6] definiert und ist eines der populärsten Modelle für räumlich-zeitliche Vorhersagen. In [20] wird dies mit der großen Anzahl an Papers begründet, die Referenzen auf [6] enthalten. Der Anwendungsfall, für den ST-ResNet konzipiert wurde, ist die Vorhersage der Bewegung von Menschenmassen (engl. *crowd flow*). Dabei ist es egal, wie sich die Menschen bewegen. So kann ST-ResNet beispielsweise nicht nur auf Fußgänger angewendet werden, sondern auch auf Leihfahrräder, Taxis oder Verkehr allgemein. Außerdem kann ST-ResNet auch auf beliebige andere rasterbasierte räumlich-zeitliche Vorhersagen angewendet werden, wie z. B. die Vorhersage von Verbrechen. Diese konkrete Adaption wird von den Autoren von ST-ResNet in [17] gezeigt. Ein Datensatz, für den ST-ResNet ursprünglich konzipiert wurde, besteht aus einem Raster über viele Zeitschritte. Pro Zeitschritt sind jeder Zelle zwei Werte zugeordnet: die Anzahl an Menschen, die die Zelle betreten (engl. *inflow*) und die Anzahl an Menschen, die die Zelle verlassen (engl. *outflow*). Dies ist ein Beispiel für einen Datensatz mit zwei Kanälen. Für die Architektur von ST-ResNet haben sich Zhang et al. genau überlegt, welche räumlichen und zeitlichen Zusammenhänge in den Daten vorhanden sein könnten. Sie stellen zunächst fest, dass der Inflow einer Zelle abhängig ist vom Outflow der benachbarten Zellen. Es können jedoch auch Zusammenhänge zu weiter entfernten Zellen bestehen, da es in größeren Städten normalerweise ein U-Bahn Netz gibt, welches weiter entfernte Zellen direkt miteinander verbindet. Bei den zeitlichen Zusammenhängen handelt es sich um verschieden große Zeitspannen. Zhang et al. führen Beispiele für Zusammenhänge innerhalb eines Tages sowie für tägliche, wöchentliche und jährliche Zusammenhänge an. Beispielsweise würden sich die Bewegungen von Menschenmassen zu Stoßzeiten jeden Tag ähnlich verhalten, wobei die Stoßzeiten im Winter aufgrund des späteren Sonnenaufgangs ebenfalls später auftreten würden. Die Bewegungen im Allgemeinen würden zusätzlich zu den räumlichen und zeitlichen Begebenheiten auch von externen Faktoren wie z. B. dem Wetter beeinflusst. Um alle diese möglichen Zusammenhänge zu berücksichtigen, haben Zhang et al. die in Abbildung 10 dargestellte Architektur entworfen.

ST-ResNet besteht im Wesentlichen aus vier Teilen, die in Abbildung 10 eingerahmt sind. Die drei rechten Teile modellieren den langfristigen Trend (*trend*), mittelfristige Periodizitäten (*period*) und kurzfristige Begebenheiten (*closeness*). Diese drei Teile haben jeweils die gleiche Struktur. Die Eingangsdaten werden zunächst einem CNN-Layer zugeführt. Dessen Ausgabe durchläuft dann mehrere sogenannte Residual Units. Diese beinhalten wiederum CNN-Layer, jedoch wird die Eingabe einer Residual Unit wieder zum Ausgang der CNN-Layer hinzuaddiert. Dadurch können sehr tiefe Netze gebildet werden, ohne dass diese dem Problem des verschwindenden Gradienten unterliegen. Der Ausgang der letzten Residual Unit wird schlussendlich nochmals durch einen CNN-Layer verarbeitet.

Der Unterschied dieser drei rechten Teile liegt in den Eingangsdaten. Der *trend*-Teil erhält als Eingangsdaten Zeitschritte im Abstand von einer Woche, die über die gesamte Zeitspanne des Datensatzes verteilt sind. Somit kann die langfristige Entwicklung der Daten erlernt werden, wie auch wöchentliche Periodizitäten. Der *period*-Teil erhält Zeitschritte im Abstand von einem Tag, die nicht sehr lange zurückliegen. Damit können tägliche Periodizitäten erlernt werden, wie die bereits er-

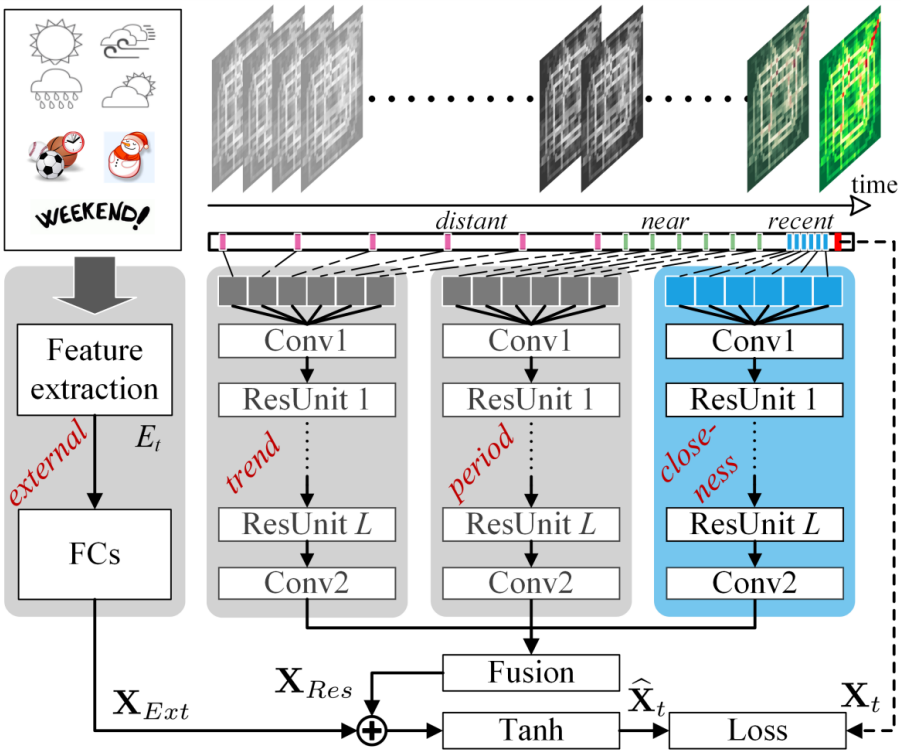

Abbildung 10: Architektur von ST-ResNet [6, Figure 3]

wähnten Stoßzeiten. Zuletzt erhält der *closeness*-Teil alle verfügbaren Zeitschritte der letzten paar Tage. Anhand dessen kann die momentane Situation erkannt werden. Diese Ausgaben dieser drei Teile werden schlussendlich gewichtet zusammengeführt. Der in Abbildung 10 ganz links dargestellte Teil verarbeitet externe Daten durch ein vollständig verbundenes Netz. Signifikante Merkmale, die als Eingabe dienen sollen, müssen hierfür manuell ermittelt werden. Die Kombination aus dem linken Teil und der zusammengeführten Ausgabe der drei rechten Teile bildet schließlich die Ausgabe von ST-ResNet.

ConvLSTM ist ebenfalls eine weit verbreitete Architektur für räumlich-zeitliche Vorhersagen. ConvLSTM wurde von Shi et al. in [7] vorgestellt. Bei ConvLSTM handelt es sich um eine Kombination aus CNN und LSTM. Durch Faltungsoperationen soll hierbei die räumliche Strukur erhalten bleiben. Außerdem sollen durch die Verwendung einer abgewandelten LSTM-Struktur die zeitlichen Zusammenhänge erfasst werden. Das ursprüngliche Ziel von ConvLSTM ist nach Shi et al. die Vorhersage einer *Sequenz* von 2D-Daten anhand einer Eingabesequenz. Dafür orientieren sie sich an der Architektur aus [21], die aus aus mehreren hintereinandergeschalteten LSTM-Zellen besteht und dadurch eine Sequenz-zu-Sequenz Vorhersage ermöglicht. Jedoch wurde sie für die Generierung von 1D-Sequenzen konzipiert und verwendet daher Matrixmultiplikationen mit Gewichtungsmatrizen. Diese werden von Shi et al. durch Faltungsoperationen ersetzt, wodurch zweidimensionale Strukturen erlernt werden können. Diese Architektur wird in Abbildung 11 verdeutlicht.

Jedes Pixel der Ausgabe H_t hängt von einem Bereich der Eingabe X_t und einem Bereich des inneren Zustands H_{t-1}, C_{t-1} ab. Dadurch, dass die Tore aus der LSTM-Architektur übernommen wurden,

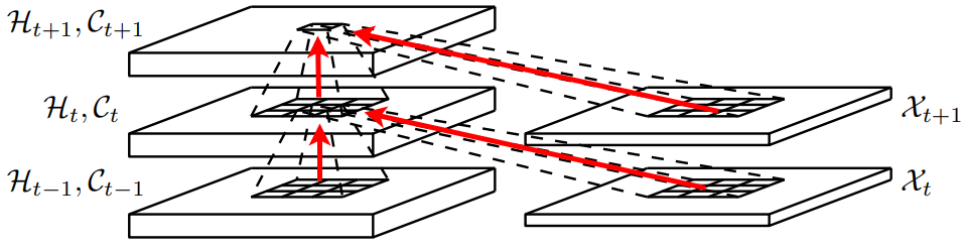

Abbildung 11: Innere Struktur von ConvLSTM aus [7] mit hinzugefügten Pfeilen zur Verdeutlichung des Informationsflusses

werden von ConvLSTM auch die räumlich-zeitlichen Zusammenhänge nach ihrer Relevanz gefiltert. Wie auch schon bei reinen LSTMs ist es nicht wichtig zu verstehen, was genau die Tore räumlich und zeitlich filtern. Es ist nur wichtig zu wissen, dass mit ConvLSTM lang- und kurzfristige Abhängigkeiten in der räumlichen und zeitlichen Dimension erlernt werden können.

Prinzipiell sind beide Modelle für den Anwendungsfall der Vorhersage von mobilen Radarkontrollen geeignet, da beide rasterbasiert arbeiten und räumliche sowie zeitliche Zusammenhänge erlernen können. Für die Auswahl eines der Modelle kommen also vor allem praktische Aspekte in Betracht. Dafür ist zunächst wichtig, dass die vorliegende Arbeit in einer begrenzten Zeit fertiggestellt werden muss. Daher sollte es so wenig wie möglich Aufwand machen, das Modell in einer Implementierung zu verwenden. Hier hat ConvLSTM den großen Vorteil, dass es bereits eine fertige Implementierung als Teil von TensorFlow bzw. Keras gibt: *tf.keras.layers.ConvLSTM2D* [22]. Im Gegensatz dazu gibt es für ST-ResNet keine Standardimplementierung, die ohne weiteres Verwendet werden kann. Es existieren zwar unabhängige, auf TensorFlow basierende Implementierungen auf GitHub [23, 24], jedoch sind diese speziell auf den in [6] beschriebenen Anwendungsfall angepasst. Daher müsste einiges an Zeit investiert werden, um die Implementierung zu verstehen und sie entsprechend an den vorliegenden Anwendungsfall anzupassen. Mit der verfügbaren Implementierung von ConvLSTM als Teil von TensorFlow geht auch einher, dass einige Ressourcen existieren, die beschreiben, wie ConvLSTM praktisch angewendet werden kann. Ein Beispiel hierfür ist das später noch verwendete Paper [10]. Aufgrund dieser praktischen Vorteile wird in der vorliegenden Arbeit mit ConvLSTM weitergearbeitet.

3 Vorverarbeitung des Datensatzes

In diesem Kapitel wird zunächst eine erste Analyse des Datensatzes durchgeführt, um diesen auf Plausibilität zu überprüfen. Anschließend wird erörtert, wie die Datenpunkte möglichst effizient in ein Raster umgewandelt werden können.

3.1 Analyse und Aufbereitung

Ziel dieses Abschnitts ist es, sich mit dem Datensatz vertraut zu machen und ihn aufzubereiten, sodass beispielsweise ungültige Einträge gelöscht werden. Außerdem ist es sinnvoll, einige Analysen durchzuführen. Dies führt zu einem besseren Verständnis des Datensatzes und legt etwaige Inkonsistenzen offen.

Der von der Eifrig Media GmbH bereitgestellte Datensatz liegt als sql-Datei vor. Für die Weiterverarbeitung muss die Datei in eine MySQL oder MariaDB Datenbank eingelesen werden. Die Datenbank kann komfortabel mit Docker erstellt werden. Eine entsprechende Docker-Compose Datei mit einer MariaDB Instanz und weiteren Komponenten, die im Verlauf dieses Abschnitts benötigt werden, befindet sich in Anhang A. Als grafische Oberfläche für den Zugriff auf die Datenbank enthält die Docker-Compose Datei außerdem eine phpMyAdmin Instanz.

Der Datensatz beinhaltet 7.787.865 Meldungen und deckt den Zeitraum vom 22.05.2014 bis 25.10.2021 ab. Dies entspricht ca. 21.300 Meldungen pro Tag. Bei Betrachtung des Datensatzes fällt zunächst auf, dass einige Meldungen Inkonsistenzen aufweisen:

- 1160 Meldungen haben als Aufbauzeitpunkt den ungültigen Wert „0000-00-00 00:00:00“.
- Bei 420 Meldungen liegt der Abbauzeitpunkt vor dem Aufbauzeitpunkt.
- Bei 9800 Meldungen beträgt die Standdauer 20 Minuten oder weniger.
- Bei 62.000 Meldungen beträgt die Standdauer mehr als 24 Stunden, bei 1960 mehr als ein Jahr.

Da der Anteil an Meldungen mit ungültigem Auf- oder Abbaudatum nur 0,02 % beträgt, bietet es sich an, diese Meldungen aus dem Datensatz zu entfernen. Auch die Meldungen mit einer sehr geringen Standdauer von 20 Minuten oder weniger sollten entfernt werden, da es sich hierbei vermutlich um Fehlmeldungen handelt. Nun stellt sich die Frage, wie mit Meldungen umgegangen werden soll, die eine sehr lange Standdauer aufweisen. Da sich im Datensatz 1962 Meldungen befinden, deren Standdauer über ein Jahr beträgt, ist davon auszugehen, dass die Standdauer einiger Meldungen im Datensatz falsch ist. Dieses Argument lässt sich bekräftigen, wenn Abbildung 12 betrachtet wird.

Es ist erkennbar, dass allein 56,1 % aller gemeldeten Radarkontrollen eine Standdauer von anderthalb,

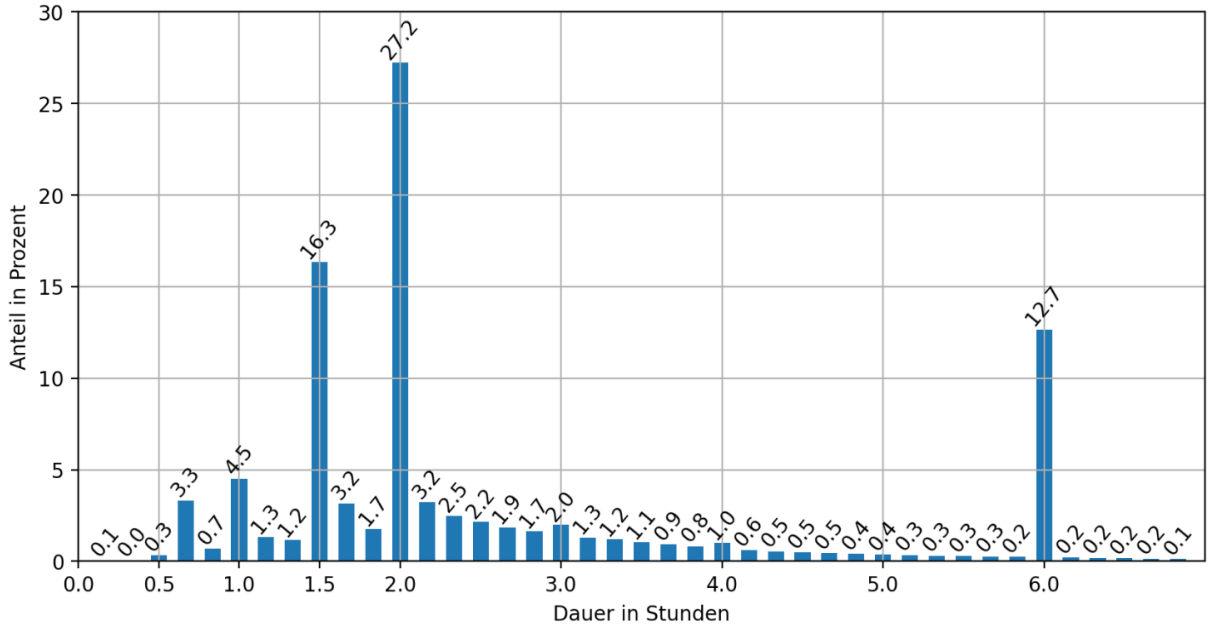

Abbildung 12: Verteilung der Standdauer mobiler Radarkontrollen

zwei oder sechs Stunden haben. Insgesamt 97,45 % haben eine Standdauer von sieben Stunden oder weniger. Damit liegt die Standdauer eines Großteils der Radarkontrollen bei wenigen Stunden. Außerdem kann nicht belegt werden, dass schon eine Standdauer von mehr als 12 Stunden in der Realität auftritt. Der Datensatz enthält jedoch ca. 91.000 Meldungen, auf die dieses Kriterium zutrifft. Dies entspricht einem Anteil von 1,1 %, was zu groß ist, um diese Meldungen komplett aus dem Datensatz zu entfernen. Die bevorzugte Alternative besteht darin, den Abbauzeitpunkt aller dieser Meldungen auf 12 Stunden nach dem Aufbauzeitpunkt zu setzen.

Eine weitere Inkonsistenz des Datensatzes fällt auf, wenn die Anzahl der Meldungen pro Tag über den gesamten verfügbaren Zeitraum betrachtet werden. Diese Metrik ist in Abbildung 13 dargestellt.

Es fällt auf, dass deutlich weniger Meldungen im Zeitraum vom 22.05.2014 bis zum 16.07.2014 vorhanden sind als in den Jahren danach. Das spricht dafür, dass der Datensatz in diesem Zeitraum unvollständig ist. Da dies das Training des NNs negativ beeinflussen würde, sollten alle Meldungen vor dem 17.07.2014 aus dem Datensatz entfernt werden. In Abbildung 13 sind noch einige weitere interessante Muster zu erkennen. Zunächst gibt es einige sehr schmale Ausschläge. Beispielsweise waren am 17.09.2014 3800 mobile Radarkontrollen aufgestellt, am 18.09.2014 dann 12.400 und am 19.09.2014 wieder nur 4140. Diese Ausschläge sind je auf einen sogenannten „Blitzermarathon“ zurückzuführen, bei denen an einem bestimmten Tag besonders viele Radarkontrollen aufgestellt werden. Ein Vergleich der Radarkontrollen auf einer Karte zwischen dem Blitzermarathon am 18.09.2014 und dem Vortag befindet sich in Anhang B, Abbildung 30.

Als Nächstes ist an Abbildung 13 noch auffällig, dass es eine jährliche Periodizität gibt. Außerdem bleibt die durchschnittliche Anzahl an Radarkontrollen pro Jahr annähernd konstant, wenn der Ausschlag im Jahr 2020 außer Acht gelassen wird. Das spricht dafür, dass der Datensatz vermutlich

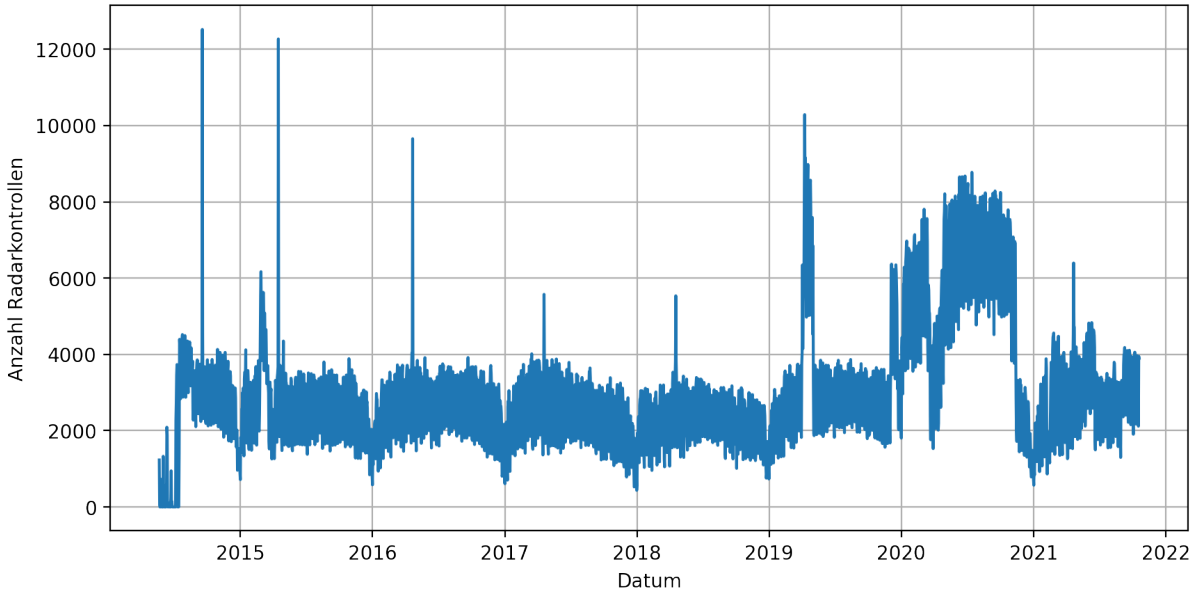

Abbildung 13: Anzahl mobiler Radarkontrollen pro Tag vom 22.05.2014 bis 20.10.2021

nicht durch eine sich verändernde Nutzerzahl verzerrt ist. Es ist auch erkennbar, dass es im Sommer mehr mobile Radarkontrollen gibt als im Winter. Dies geht auch aus Abbildung 14 hervor, in der die Anzahl mobiler Radarkontrollen pro Monat dargestellt ist.

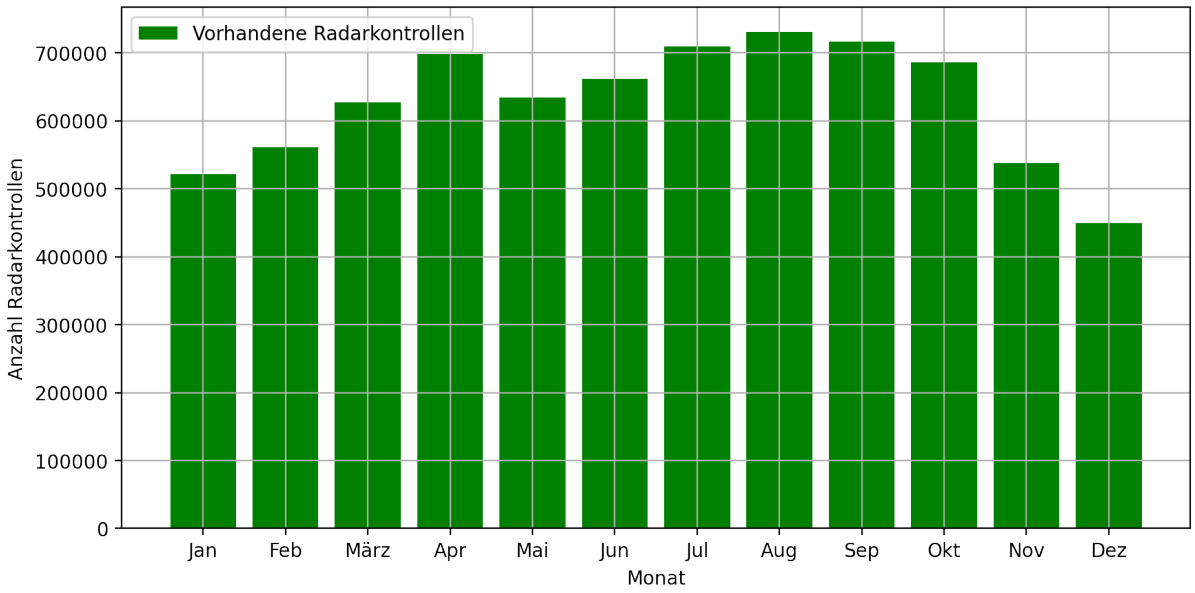

Abbildung 14: Anzahl mobiler Radarkontrollen nach Monat

Außerdem besteht eine wöchentliche Periodizität, die in Abbildung 13 aufgrund der begrenzten Auflösung nicht erkennbar ist. Daher ist in Abbildung 15 die Anzahl mobiler Radarkontrollen pro Wochentag in Rot dargestellt.

Es fällt auf, dass am Wochenende pro Tag nur ca. halb so viele Meldungen vorhanden sind wie unter

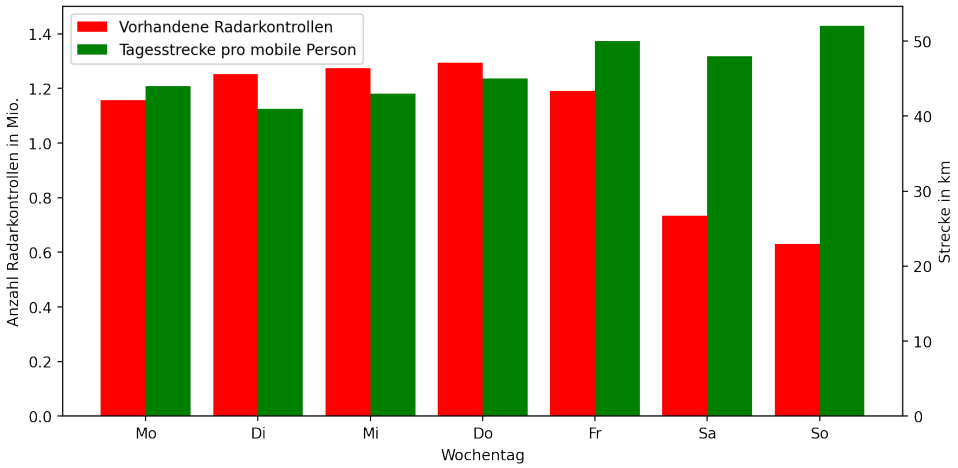

Abbildung 15: Anzahl mobiler Radarkontrollen nach Wochentag im Vergleich zur durchschnittlich gefahrenen Tagesstrecke pro mobiler Person. Die Tagesstrecke stammt aus [8, Tabelle 3].

der Woche, während die gefahrene Tagesstrecke [8, Tabelle 3] (in Grün dargestellt) ähnlich groß ist. Dadurch lässt sich das Argument bekräftigen, dass die Anzahl an Meldungen nicht signifikant vom Verkehrsaufkommen und damit der Anzahl an Appbenutzern abhängig ist.

Bisher wurde der Datensatz hauptsächlich über zeitliche Auflösungen von über einem Tag analysiert. Jedoch sind auch Statistiken interessant, die sich auf die Tageszeit beziehen. So ist in Abbildung 16 beispielsweise dargestellt, wie viele Radarkontrollen es je nach Tageszeit gibt.

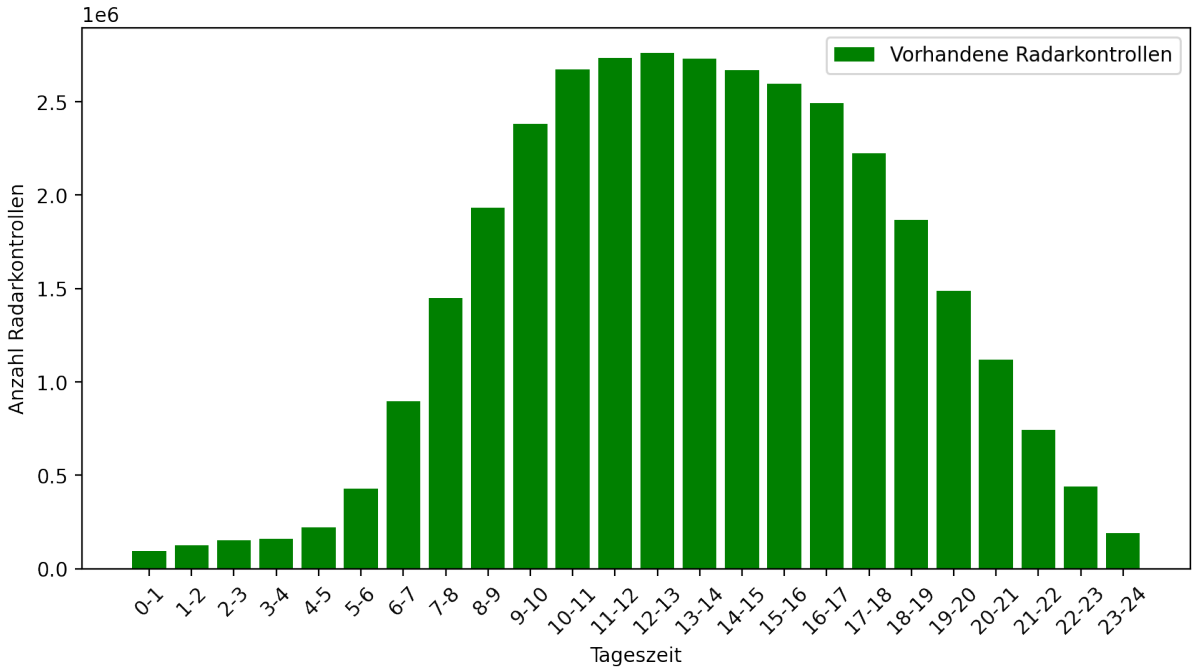

Abbildung 16: Anzahl mobiler Radarkontrollen nach Tageszeit

Es fällt zunächst auf, dass es sich nicht um eine Gleichverteilung handelt. Nachts (von 21 bis 6

Uhr) sind deutlich weniger Radarkontrollen vorhanden als tagsüber. Um die Mittagszeit erreicht die Anzahl der Radarkontrollen ihren Höhepunkt, um Mitternacht ihren Tiefpunkt. Der starke Anstieg zwischen 6 und 10 Uhr unterstreicht nochmals die Motivation dieser Arbeit. Man kann annehmen, dass in diesem Zeitraum so viele Radarkontrollen aufgestellt werden, dass die Wahrscheinlichkeit recht groß ist auf eine Radarkontrolle zu stoßen, die in der App noch nicht gemeldet ist. In Bezug auf die Gefahr neu aufgestellter Radarkontrollen ist noch anzumerken, dass aus dieser Grafik nicht erkennbar ist, ob die Radarkontrollen, die morgens aufgestellt werden, bis abends an derselben Stelle stehen bleiben oder ob sie mehrmals pro Tag an unterschiedlichen Stellen aufgebaut werden. Hierfür muss Abbildung 17 betrachtet werden.

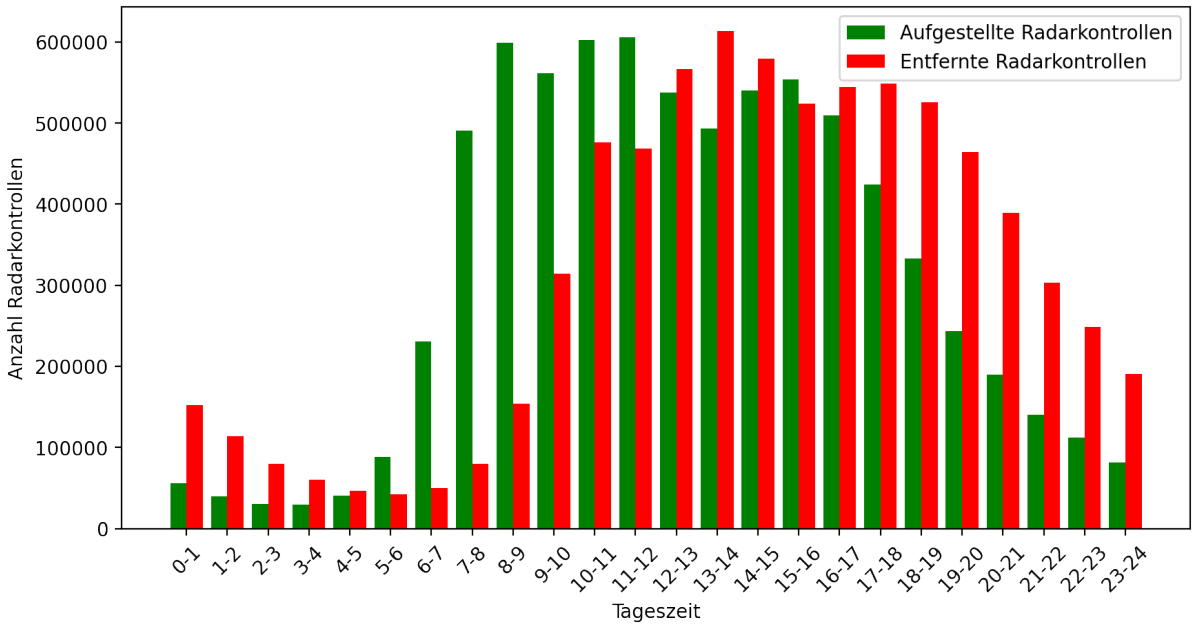

Abbildung 17: Auf- und Abbau mobiler Radarkontrollen nach Tageszeit

Wie erwartet kann man erkennen, dass zwischen 6 und 9 Uhr die Anzahl an neu gemeldeten Radarkontrollen stark ansteigt. Es ist jedoch bemerkenswert, dass bis ca. 17 Uhr weiterhin viele Radarkontrollen neu gemeldet werden. Gleichzeitig steigt bereits ab 9 Uhr die Anzahl an entfernten Meldungen stark an. Von 12 bis 17 Uhr ist dann sowohl die Zahl an neuen und entfernten Meldungen konstant hoch. Das bedeutet, dass die Radarkontrollen keineswegs den ganzen Tag an der gleichen Stelle stehen, sondern u.U. mehrmals am Tag umgestellt werden. Daher ist die Gefahr, auf noch nicht gemeldete Radarkontrollen zu stoßen nicht nur morgens hoch, sondern auch tagsüber nicht zu vernachlässigen.

3.2 Rasterisierung

Um ein rasterbasiertes Modell wie ConvLSTM oder ST-ResNet für die Vorhersage von mobilen Radarkontrollen verwenden zu können, müssen die räumlich und zeitlich kontinuierlichen Datenpunkte aus dem Datensatz zunächst rasterisiert werden. Dazu ist es sinnvoll nochmals zu betrachten, wie die Daten ursprünglich vorliegen. Jede Meldung ist eine Zeile in einer Datenbanktabelle mit den

Spalten *Längengrad*, *Breitengrad*, *Aufbauzeitpunkt* und *Abbauzeitpunkt*. Das Ziel besteht darin, die Datenpunkte so zu verarbeiten, dass pro Zeitraum T ein Raster aus $n \times n$ quadratischen Zellen mit der Seitenlänge k entsteht, bei dem jeder Zelle ein Wert x zugeordnet ist, der etwas über die gemeldeten Radarkontrollen in dieser Zelle aussagt. Die Werte T, n, k sind hierbei variable Hyperparameter und müssen möglichst sinnvoll ausgewählt werden. Außerdem muss festgelegt werden, nach welcher Regel der Wert x berechnet werden soll.

3.2.1 Auswahl der Parameter zur Rasterisierung

Bei der Auswahl des Zeitraums T muss zwischen der gewünschten zeitlichen Genauigkeit und der Anzahl an verfügbaren Datenpunkten abgewägt werden. Wird T beispielsweise zu klein gewählt, enthält der betrachtete Bereich zu wenige Meldungen, um auf deren Basis aussagekräftige Vorhersagen treffen zu können. Wird T hingegen zu groß gewählt (wie z. B. eine Woche oder ein Monat), haben die Vorhersagen weniger praktischen Nutzen. Außerdem könnten dann insgesamt zu wenige Zeitschritte vorliegen, um das Modell trainieren zu können. Nach einigen Versuchen scheinen 24 Stunden für T ein guter Wert zu sein, da somit der praktische Nutzen erhalten bleibt und es dennoch genug Meldungen pro Zeitschritt gibt.

Als Nächstes sind k und n zu wählen. Die Seitenlänge des gesamten betrachteten Bereichs berechnet sich daraus mit $n \cdot k$. Bei der Wahl von k muss dieselbe Abwägung getroffen werden wie bei der Wahl von T, jedoch geht es hier um den praktischen Nutzen der Vorhersagen je nach räumlicher Auflösung. Wird k zu groß gewählt, ist die zeitliche Auflösung zu gering, als dass die Vorhersagen nützlich wären. Wenn k hingegen zu klein gewählt wird, gibt es in zu wenigen Zellen eine Meldung, um das Modell gut trainieren zu können. Bei der Wahl von k kommt es außerdem darauf an, ob eher städtischer oder ländlicher Raum betrachtet werden soll. Für den ländlichen Raum scheint $k = 4$ km eine gute Wahl zu sein, da die Radarkontrollen dort nicht sonderlich dicht liegen. Für den städtischen Raum ist dieser Wert jedoch zu groß, da sich in einem 4 km Radius vergleichsweise deulich mehr Straßen befinden. Außerdem ist die Dichte an Radarkontrollen im städtischen Raum höher, weshalb ein geringeres k von einem oder zwei Kilometer durchaus möglich ist. Wird (beispielsweise in Baden-Württemberg) insgesamt ein größerer Raum abgedeckt, besteht die meiste Fläche aus ländlichem Gebiet. Daher wird im folgenden mit $k = 4$ km fortgefahren. Die Wahl von n beeinflusst nun den insgesamt durch das Raster abgedeckten Raum. Optimalerweise sollte dieser Wert möglichst groß sein, um möglichst viele Trainingsdaten einzuschließen. Andererseits ist n durch den während des Trainings verfügbaren Speicher begrenzt. Wird das Training auf einer Grafikkarte ausgeführt, müssen das komplette Modell und alle Trainingsdaten, die pro Batch verarbeitet werden in den Grafikspeicher passen. Da der Speicherbedarf quadratisch mit n ansteigt, ist man hier in der Praxis sehr eingeschränkt. Bei einem Grafikspeicher von 8 GB und einer Batch Size von 12 ist $n = 50$ gut möglich. Mit $k = 4$ km ergibt sich dann eine Seitenlänge des gesamten Rasters von 200 km.

Für die Berechnung von x gibt es mehrere Möglichkeiten. Zunächst könnte $x = 1$ gesetzt werden,

wenn sich in der Zelle mindestens eine Radarkontrolle befindet, ansonsten $x = 0$. Alternativ kann x auf die Anzahl der Radarkontrollen gesetzt werden, die sich im jeweiligen Zeitraum in der Zelle befinden. Dies hat jedoch den Nachteil, dass eine Radarkontrolle mit einer kurzen Standdauer gleich gewichtet wird wie eine Radarkontrolle mit einer deutlich längeren Standdauer. Daher bietet es sich an, die Standdauer aller Radarkontrollen in einer Zelle aufzusummieren. Mit dieser Möglichkeit enthält x am meisten Informationen über die Radarkontrollen in der jeweiligen Zelle. Es entsteht dadurch eine Heatmap, die die Signifikanz der Radarkontrollen pro Rasterzelle darstellt. Außerdem kann diese Repräsentation auch nach der Rasterisierung einfach in eine binäre Unterscheidung umgewandelt werden, indem pro Zelle $x := x > 0$ gesetzt wird.

3.2.2 Auswahl von Methoden und Tools zur Rasterisierung

Nun stellt sich die Frage, wie die Rasterisierung mit den erörterten Parametern möglichst effizient ausgeführt werden kann. Die einfachste Möglichkeit besteht darin, alle Meldungen pro Rasterzelle und Datum nacheinander mit einem Python-Skript von der MariaDB-Datenbank abzufragen. Pro Rasterzelle dauert diese Abfrage ca. 0,32 Sekunden. Bei 200 Zellen und 2648 Tagen würde die Rastergenerierung damit jedoch 47 Stunden dauern, was unpraktikabel ist. Eine mögliche Optimierung ist, eine Geodatenbank zu verwenden, die das Filtern nach Koordinaten effizienter ausführen kann als MariaDB. Eine solche Geodatenbank ist PostgreSQL mit der Erweiterung PostGIS. Die Abkürzung GIS in PostGIS steht für *Geoinformationssystem*. Dies ist ein Begriff für Systeme, die besonders auf die Verarbeitung von räumlichen Daten optimiert sind, wozu auch Geodatenbanken gehören. Eines der wichtigsten Features einer Geodatenbank ist ein räumlicher Index (engl. *spatial index*). Ein solcher ist laut [25, S. 2707 f.] eine Datenstruktur, die einen schnelleren Zugriff auf räumliche Daten bietet als traditionelle eindimensionale Indexe wie ein B^+-Baum. Ein bekanntes Beispiel für einen räumlichen Index ist der R-Baum, der auch von PostGIS verwendet wird [26]. Die Funktionsweise eines R-Baums ist in Abbildung 18 dargestellt.

Wie in der Abbildung zu erkennen ist, enthält jeder Knoten eines R-Baums Informationen über ein Rechteck. Die Knoten können dann nach [27] jeweils Kindknoten haben, deren Recktecke komplett innerhalb des Rechtecks des Elternknotens liegen müssen. Als Beispiel für die Suchoperation auf einem R-Baum sollen alle Rechtecke R8 bis R19 ermittelt werden, die sich mit dem in Abbildung 18 grün markierten Punkt überschneiden. Ohne den R-Baum müssten alle 12 Recktecke R8 bis R19 nacheinander mit dem Punkt verglichen werden. Mit dem R-Baum sind nur noch sieben Vergleiche notwendig. Zunächst wird ermittelt, ob der Punkt in R1 oder R2 liegt. Da der Punkt in R2 liegt, können R8 bis R14 nun schon ausgeschlossen werden. Daraufhin wird der Punkt mit R6 und R7 verglichen. Da der Punkt in R7 liegt, liefern die Vergleiche mit R17, R18 und R19, dass sich nur R17 und R18 mit dem Punkt überschneiden. Dieser Performancegewinn macht sich nach [27] vor allem bei sehr vielen Datenpunkten bemerkbar.

Da der Datensatz der mobilen Radarkontrollen ca. 7,7 Millionen Einträge enthält, ist es hier sehr

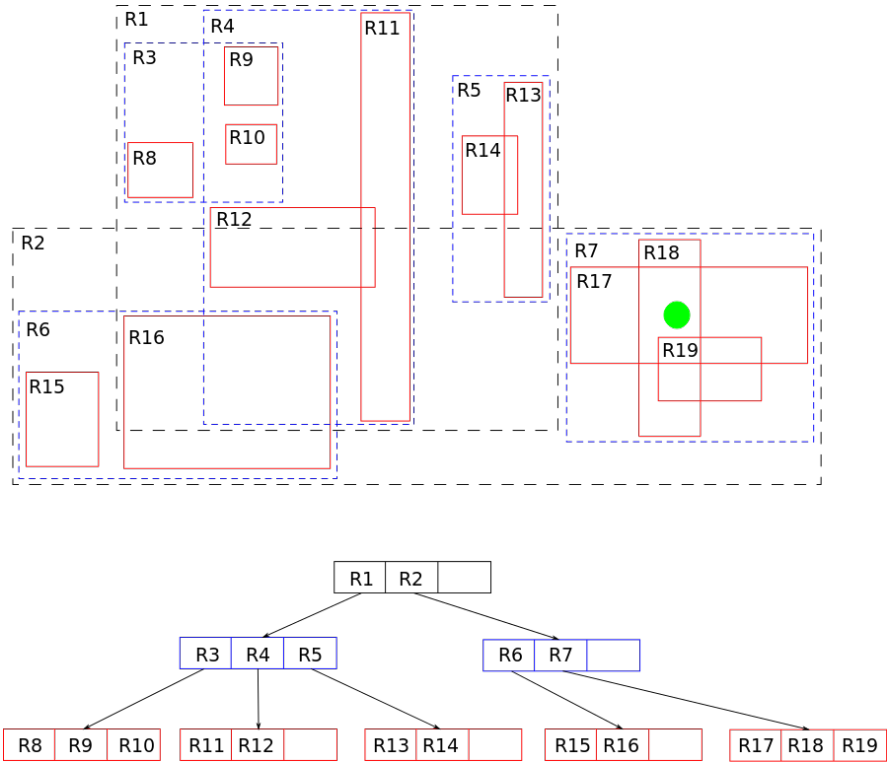

Abbildung 18: Aufbau und Funktionsweise eines R-Baums [9]

sinnvoll, eine Geodatenbank mit räumlichem Index zu verwenden. Als Geodatenbank bietet sich PostgreSQL mit der PostGIS-Erweiterung (im folgenden zusammen PostGIS genannt) an. Die Datenpunkte müssen nun von MariaDB nach PostGIS übertragen werden. Dies kann komfortabel mit einem Python-Skript unter Verwendung der Pakete SQLAlchemy und GeoAlchemy2 bewerkstelligt werden. Mit SQLAlchemy kann intuitiv auf Datenbanken zugegriffen werden, indem Tabellen der Datenbank auf Python-Klassen abgebildet sind. Dieses Konzept wird als *Object Relational Mapper* (ORM) bezeichnet [28]. Das Paket GeoAlchemy2 fügt dann noch Unterstützung für PostGIS hinzu. Nun können zwei Klassen entworfen werden, die eine Radarkontrolle in MariaDB und in PostGIS repräsentieren. Die Implementierung der beiden Klassen und des letztendlichen Transfers ist in Anhang C zu finden. Die Klasse *PostgisSpeedcam* enthält dabei einen Konstruktor, der eine MariaDB-Radarkontrolle als Parameter erhält und die Konvertierung durchführt. Die Klasse (bzw. Tabelle) *PostgisSpeedcam* enthält zusätzlich ein Attribut *duration_hours*, welches die Standdauer der Radarkontrolle in Stunden enthält. Durch dieses Attribut kann in der nachfolgenden Verarbeitung der Datenpunkte komfortabel auf die Standdauer zugegriffen werden, ohne dass die Datumsdifferenz separat berechnet werden muss. Mit der Implementierung in Anhang C, Codeausschnitt 10 können alle Tabelleneinträge in ca. einer Stunde übertragen werden. Hierbei ist wichtig anzumerken, dass der räumliche Index automatisch von PostGIS erzeugt und aktualisiert wird.

Die Verwendung von PostGIS bringt noch einen weiteren großen Vorteil mit sich. Da die Daten nun in einem im GIS-Umfeld gebräuchlichen Format vorliegen, können beliebige GIS-Tools verwendet werden, um die Daten zu analysieren. Dies hat den Vorteil, dass diese Tools viele effiziente Implementierungen der Standardalgorithmen auf geographischen Daten mitbringen. Ein Beispiel für ein solches

Tool ist QGIS. QGIS kann als Algorithmensammlung für geographische Daten angesehen werden. Außerdem beinhaltet QGIS eine GUI (*Graphical User Interface*), mit der diese Algorithmen ausgeführt werden können. Mit der GUI können zudem verschiedenste geographische Daten wie Punkte, Linien, Rechtecke, Raster etc. visualisiert werden, insbesondere auch die Ergebnisse der ausgeführten Algorithmen. Ein Beispiel für eine Visualisierung mit QGIS ist Abbildung 30 in Anhang B. Die Punkte werden über eine direkte Verbindung zu PostGIS abgerufen und dann auf einer OpenStreetMap-Karte dargestellt. Aufgrund des räumlichen Indexes von PostGIS werden die Punkte sehr schnell geladen, insbesondere wenn kleinere Bereiche betrachtet werden. QGIS ist in C++ unter Verwendung des Qt-Frameworks implementiert. Die Algorithmen können zusätzlich zur GUI über eine C++ API aufgerufen werden, für die es auch Python-Bindings gibt.

Für die Rasterisierung des Datensatzes gibt es nun zwei verschiedene Möglichkeiten. Zum einen könnte die Rasterisierung direkt in der Datenbank mit von PostGIS bereitgestellten SQL-Funktionen durchgeführt werden. Zum anderen könnten Algorithmen aus QGIS verwendet werden. Der erste Ansatz hat den Vorteil, dass mit einer Rasterisierung direkt in der Datenbank die maximal mögliche Geschwindigkeit erreichbar wäre. Allerdings ist zur Implementierung des Algorithmus in SQL ein umfangreiches Fachwissen im GIS-Umfeld erforderlich. Da GIS an sich ein großer, eigenständiger Forschungsbereich ist, überwiegt dieser Nachteil die eventuell etwas höhere Geschwindigkeit. Somit fällt die Wahl auf eine Implementierung der Rasterisierung mit Algorithmen aus QGIS. Die komplette Implementierung ist in Anhang D zu finden.

3.2.3 Implementierung der Rasterisierung

Für die Implementierung steht zunächst das Erstellen eines Rasters an, also die Berechnung der Grenzen der einzelnen Rasterzellen. Hierfür kann der in [29] beschriebene QGIS-Algorithmus „Create grid“ verwendet werden. Die wichtigsten Parameter dieses Algorithmus sind die Grenzen des gesamten Rasters (engl. *extent*) und die Seitenlänge der Rasterzellen (oben k genannt). Die Grenzen des gesamten Rasters müssen anhand von n und k abgeschätzt werden und können in der QGIS-GUI mit der Funktion „Draw on Map Canvas“ direkt auf der Karte gezeichnet werden. Der Aufruf des Algorithmus in Python ist in Anhang D, Codeausschnitt 11 in den Zeilen 17 bis 22 zu sehen. Der Rückgabewert des Algorithmus ist ein Vektorlayer, der jede Rasterzelle als Rechteck (bestehend aus vier Kanten) enthält. Dieser Vektorlayer kann nun verwendet werden, um mit dem in [30] beschriebenen QGIS-Algorithmus „Join attributes by location (summary)“ den Zahlenwert x für jede Rasterzelle zu ermitteln. Dieser Algorithmus fügt Attribute aus einem Vektorlayer zu einem zweiten Vektorlayer unter Berücksichtigung einer räumlichen Bedingung hinzu. Im vorliegenden Fall soll zu jeder Rasterzelle die Summe der Standdauer aller Radarkontrollen als Attribut hinzugefügt werden, die sich an einem bestimmten Datum mit dieser Rasterzelle überschneiden. Der komplette Aufruf des Algorithmus mit diesen Parametern ist in Anhang D, Codeausschnitt 11 in den Zeilen 24 bis 31 zu sehen. Die relevanten Parameter und Werte für dieses Ziel sind:

- **INPUT**: Der Vektorlayer, zu dem das zusätzliche Attribut hinzugefügt werden soll. Im vorliegenden Fall ist dies der im vorherigen Schritt erzeugte Layer, der die Rasterzellen enthält.
- **JOIN**: Der zweite Vektorlayer, aus dem das Attribut extrahiert werden soll. Im vorliegenden Fall besteht dieser Layer aus den Datenpunkten, die direkt aus der Datenbank abgefragt werden. Daher ist dieser Parameter in der Implementierung eine Zeichenkette, die die Datenbank-Verbindung, die Quelltabelle und den Datumsfilter beschreibt. Diese Zeichenkette ist in Anhang D, Codeausschnitt 11 in den Zeilen 5 bis 8 zu sehen. Beim eigentlichen Aufruf des Algorithmus wird in Zeile 26 noch das Datum als Filter angehängt.
- **JOIN_FIELDS**: Ein Array aus Attributen des *JOIN*-Layers, die durch eine bestimmte mathematische Funktion zusammengefasst werden sollen. Im vorliegenden Fall enthält dieses Array nur ein Element, *duration_hours*.
- **PREDICATE**: Ein Array aus räumlichen Bedingungen, die pro Rasterzelle in Verbindung mit allen Datenpunkten gelten müssen, die für diese Rasterzelle zusammengefasst werden sollen. In [30] befindet sich eine Auflistung, welcher Zahlenwert welcher räumlichen Bedingung entspricht und was diese Bedingungen genau aussagen. Im vorliegenden Fall soll nur die Bedingung *intersects* gelten, was dem Zahlenwert 0 entspricht. Es sollen pro Rasterzelle also alle Datenpunkte zusammengefasst werden, die sich mit der jeweiligen Rasterzelle überschneiden.
- **SUMMARIES**: Ein Array aus mathematischen Operationen, die für die zusammenzufassenden Datenpunkte berechnet werden sollen. Auch hier findet sich in [30] eine Auflistung, welcher Zahlenwert welcher Operation entspricht. Im vorliegenden Fall soll nur die Summe der Standdauer berechnet werden, daher ist das einzige Element des Arrays 5, was für *sum* steht.

Die Rückgabe des Algorithmus ist wiederum ein Vektorlayer, der nun pro Rasterzelle die summierte Standdauer der Radarkontrollen als Attribut enthält. Mit dem Aufruf der Funktion *store_layer()* in Zeile 33 von Anhang D, Codeausschnitt 11 wird der Layer als *GeoPackage* (GPKG-Datei) gespeichert. Eine GPKG-Datei ist nach [31] ein Container zum Speichern von geographischen Daten. Dabei ist GPKG als SQLite-Datenbank implementiert. Daher können GPKG-Dateien auch mit anderen Programmen geöffnet werden, die SQLite-Datenbanken verarbeiten können. Als Veranschaulichung des Ergebnisses des eben erläuterten Algorithmus ist in Abbildung 19 ein Ausschnitt aus der resultierenden GPKG-Datei für den 17.04.2014 dargestellt.

Man kann erkennen, dass jede Zeile der Tabelle eine Rasterzelle enthält. Die Spalten *left*, *top*, *right* und *bottom* enthalten die Begrenzung der Rasterzelle. Durch den Algorithmus „Join attributes by location (summary)“ wurde nun noch die letzte Spalte *duration_hours_sum* hinzugefügt. Wie man erkennen kann, enthält diese Spalte *NULL*, wenn sich keine Radarkontrolle in der Rasterzelle befindet.

Als Nächstes müssen die als GPKG-Datei abgespeicherten Vektordaten in Rasterdaten umgewandelt werden, damit sie später als zweidimensionale numpy-Arrays eingelesen werden können. Der Unterschied zwischen Vektor- und Rasterdaten kann am besten an GeoTIFF, einem Dateiformat für Rasterdaten, veranschaulicht werden. Während eine GPKG-Datei Vektordaten wie Punkte oder Linien speichert, speichert eine GeoTIFF-Datei Pixelwerte wie eine Bilddatei. Eine weitere Ähnlichkeit

Table: heatmap_2014-07-17

	fid	geom	id	left	top	right	bottom	duration_hours_sum
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
206	206	BLOB	206	943792.5265	6293386.3009	947792.5265	6289386.3009	NULL
207	207	BLOB	207	943792.5265	6289386.3009	947792.5265	6285386.3009	4.13
208	208	BLOB	208	943792.5265	6285386.3009	947792.5265	6281386.3009	6.15
209	209	BLOB	209	943792.5265	6281386.3009	947792.5265	6277386.3009	11.02
210	210	BLOB	210	943792.5265	6277386.3009	947792.5265	6273386.3009	1.99
211	211	BLOB	211	943792.5265	6273386.3009	947792.5265	6269386.3009	NULL
212	212	BLOB	212	943792.5265	6269386.3009	947792.5265	6265386.3009	3.02
213	213	BLOB	213	943792.5265	6265386.3009	947792.5265	6261386.3009	NULL

Abbildung 19: Einträge einer GPKG-Datei

zu Bilddateien ist, dass GeoTIFF-Dateien pro Pixel mehrere Kanäle gespeichert werden können. Im Unterschied zu gewöhnlichen Bilddateien enthält eine GeoTIFF-Datei nach [32, S. 128] zusätzlich zu den Pixelwerten noch Metadaten, die die räumlichen Eigenschaften des Rasters beschreiben. Dazu zählen z. B. Koordinaten als Georeferenz und das verwendete Koordinatenreferenzsystem. Damit kann das Raster im Koordinatensystem richtig positioniert werden, ohne dass wie bei GPKG-Dateien die Grenzen jeder einzelnen Rasterzelle gespeichert werden müssen. Diese Metadaten werden nach [33] so gespeichert, dass eine GeoTIFF-Datei auch dem TIFF-Format entspricht. Beispielsweise kann eine GeoTIFF-Datei mit der Bildverarbeitungssoftware GIMP (*GNU Image Manipulation Program*) geöffnet werden, auch wenn es dafür keine wirkliche Anwendung gibt.

Zur Umwandlung der GPKG-Dateien in GeoTIFF-Dateien kann das Programm *gdal_rasterize* verwendet werden, welches mit der *Geospatial Data Abstraction Library* (GDAL) mitgeliefert wird. Den Aufruf dieses Programms sieht man in Anhang D, Codeausschnitt 12 in den Zeilen 13 bis 17. Da *gdal_rasterize* die Höhe und Breite, sowie die Grenzen des Rasters als Parameter benötigt, werden diese in den Zeilen 6 bis 10 aus der GPKG-Datei ermittelt. Wichtig ist auch der Parameter mit der Syntax *[-a attribute_name]*, der bestimmt, welches Attribut des Vektorlayers für die Zahlenwerte des Rasters verwendet werden soll. Wie in Abbildung 19 zu sehen ist, ist dies im vorliegenden Fall *duration_hours_sum*. Ist die GeoTIFF-Datei erstellt, kann diese sehr einfach als numpy-Array eingelesen werden, was in Codeausschnitt 1, Zeile 3 bis 9 demonstriert wird. Außerdem können die Metadaten ausgelesen werden, was in Zeile 11 bis 15 zu sehen ist.

```
>>> from osgeo import gdal

>>> filename = "heatmap_2015-07-02.tif"
>>> geotiff = gdal.Open(filename)
>>> raster_data = geotiff.ReadAsArray()
>>> type(raster_data)
<class 'numpy.ndarray'>
>>> raster_data.shape
(50, 50)

>>> geotransform = geotiff.GetGeoTransform()
>>> geotransform
(927793.0, 4000.0, 0.0, 6313386.0, 0.0, -4000.0)
>>> f"{geotiff.RasterXSize}x{geotiff.RasterYSize}"
50x50
```

Codeausschnitt 1: Extrahierung der Parameter und Daten aus einer GeoTIFF-Datei

Damit ist die Implementierung der Rasterisierung des Datensatzes abgeschlossen. Die mit GDAL geladenen numpy-Arrays können nun als Input für das neuronale Netz verwendet werden. Die komplette Implementierung ist in Anhang D aufgeführt.

4 Implementierung eines Modells

Bisher wurden die Grundlagen von neuronalen Netzen behandelt und der Datensatz ist vorbereitet. Daher kann nun ein Modell definiert und implementiert werden, das die Vorhersage mobiler Radarkontrollen bewerkstelligt. In diesem Abschnitt wird das Modell außerdem trainiert und evaluiert. Zudem werden einige weitere praktische Aspekte erläutert und das Modell optimiert.

4.1 Definition des Modells

Ziel dieses Unterabschnitts ist es herauszuarbeiten, wie mehrere ConvLSTM-Layer zusammen mit anderen Layertypen für räumlich-zeitliche Vorhersagen verwendet werden können. Außerdem wird eine Verlustfunktion definiert, die für den vorliegenden Anwendungsfall geeignet ist.

ConvLSTM-Layer können unterschiedlich verwendet werden, um bestimmte Anwendungsfälle abzudecken. Die wichtigste Unterscheidung ist, ob die Ausgabe des Modells eine Sequenz oder ein einzelner 2D-Tensor sein soll. Der erste Fall wird beispielsweise für die Weiterführung einer Bildsequenz verwendet, wie in [34] demonstriert. Eine solche Architektur könnte im Kontext der vorliegenden Arbeit beispielsweise verwendet werden, um die Gefahr für mobile Radarkontrollen mehrere Tage vorauszusagen. Dies ist jedoch hier nicht notwendig. Eine alternative Architektur wird in [10] vorgestellt. Diese ist dazu in der Lage, anhand einer Sequenz von 16 oder 32 2D-Eingangstensoren den folgenden 2D-Tensor vorherzusagen. Die Architektur wurde im Kontext der Verbrechensvorhersage entwickelt. Da es sich dabei ebenfalls um sporadisch auftretende Ereignisse handelt, ist die Architektur mit hoher Wahrscheinlichkeit auch für den vorliegenden Anwendungsfall geeignet. Die Architektur ist in Abbildung 20 dargestellt.

Anhand der Abbildung kann der Datenfluss verfolgt werden. Wie ganz oben zu erkennen ist, hat die Eingabesequenz eine Größe von 16x50x50. Die Eingabe ist also eine Sequenz von 16 Frames, die jeweils 50x50 Pixel groß sind. Es ist anzumerken, dass die Eingabesequenz in [10] die Dimensionen 50x50x16 hat. Die erste und letzte Dimension ist für die vorliegende Arbeit getauscht, da die Verarbeitung so intuitiver ist. Das Resultat ist jedoch dasselbe. Die Eingabe durchläuft nun mehrmals je einen ConvLSTM- und einen MaxPooling3D-Layer. Die ConvLSTM-Layer haben nach [10] eine Kernelgröße von 3x3 Pixel. Außerdem werden Höhe und Breite der Frames durch Padding beibehalten. Auch die Anzahl an Frames bleiben durch die ConvLSTM-Layer erhalten. Um von den 16 Frames auf ein Frame als Ausgang zu kommen, befindet sich hinter jedem ConvLSTM-Layer ein MaxPooling3D-Layer. Dieser Layer berechnet je das Maximum zweier Pixel, die in zwei benachbarten Frames an derselben Position liegen. Dies entspricht einer Poolgröße von $(2, 1, 1)$. Außerdem wird eine Schrittweite von $(2, 1, 1)$ verwendet. Das Resultat ist (vereinfacht auf 3x3 Pixel) in Abbildung 21 dargestellt.

Mit der MaxPooling3D-Operation werden damit die signifikantesten Merkmale von je zwei benach-

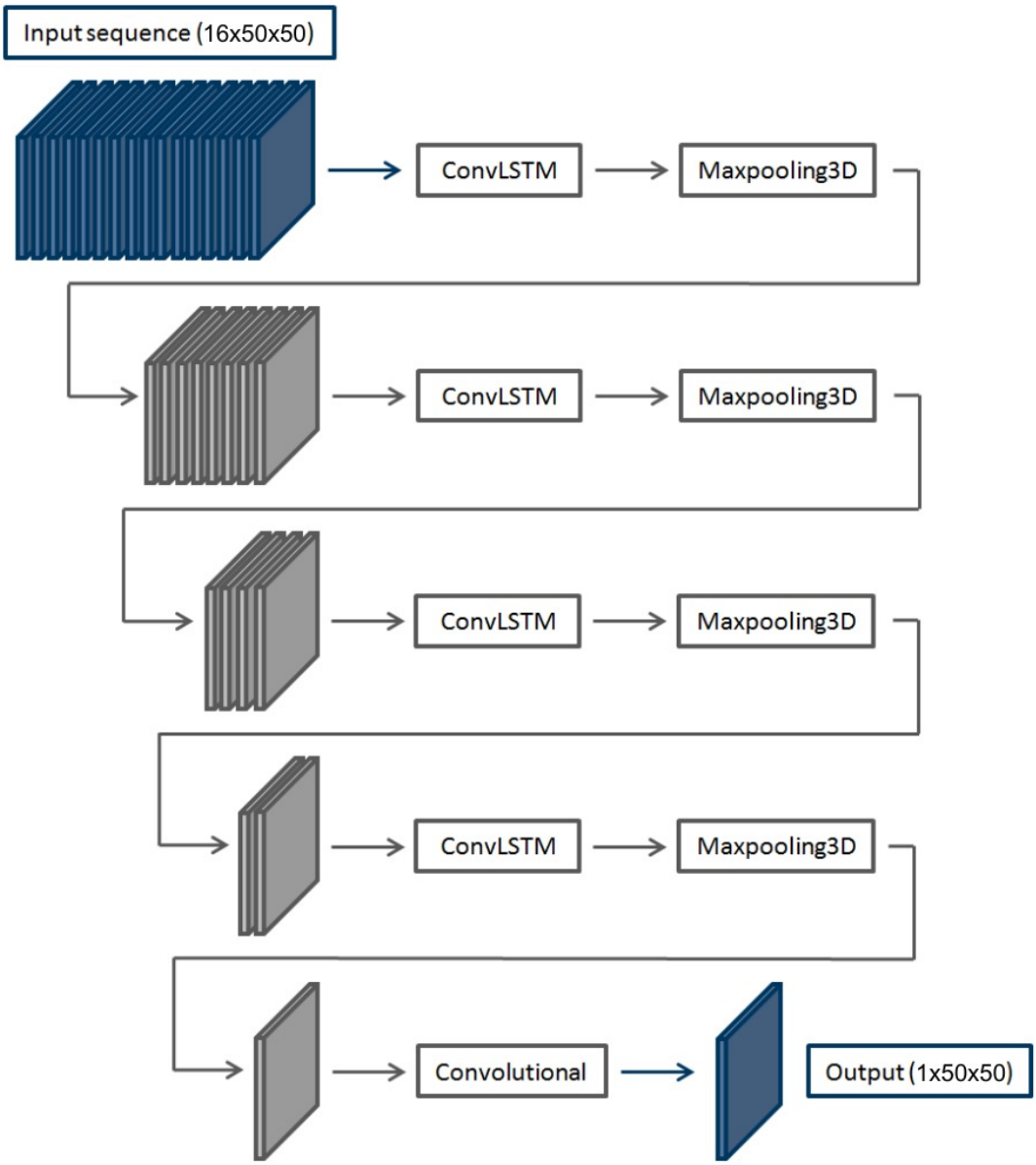

Abbildung 20: Architektur eines Modells aus mehreren ConvLSTM- und MaxPooling-Layern nach [10, Figure 2.1] mit angepassten Dimensionen

barten Frames vereint. Dadurch wird die Anzahl der Frames in der Sequenz nach jeder ConvLSTM-MaxPooling3D-Kombination halbiert. Somit ist nach vier dieser Kombinationen nur noch ein Frame übrig. Am Ende liegt nach [10] noch ein 3D-Faltungslayer mit einer Kernelgröße von $(1, 3, 3)$. Dort ist die dreidimensionale Version eines Faltungslayers nötig, da die erste und letzte Dimension, wie oben angemerkt, vertauscht ist. Da ein 3D-Faltungslayer mit einer Kernelgröße von $(3, 3, 1)$ (wie er hier verwendet werden müsste) exakt einem 2D-Faltungslayer mit einer Kernelgröße von $(3, 3)$ entspricht, wird hier letzterer verwendet.

Um das Modell trainieren zu können, muss noch eine Verlustfunktion ausgewählt werden. Dafür könnte eine gewöhnliche Verlustfunktion für die Klassifizierung wie die binäre Kreuzentropie (engl. *binary cross entropy*) verwendet werden. Das liegt daran, dass der vorliegende Anwendungsfall als

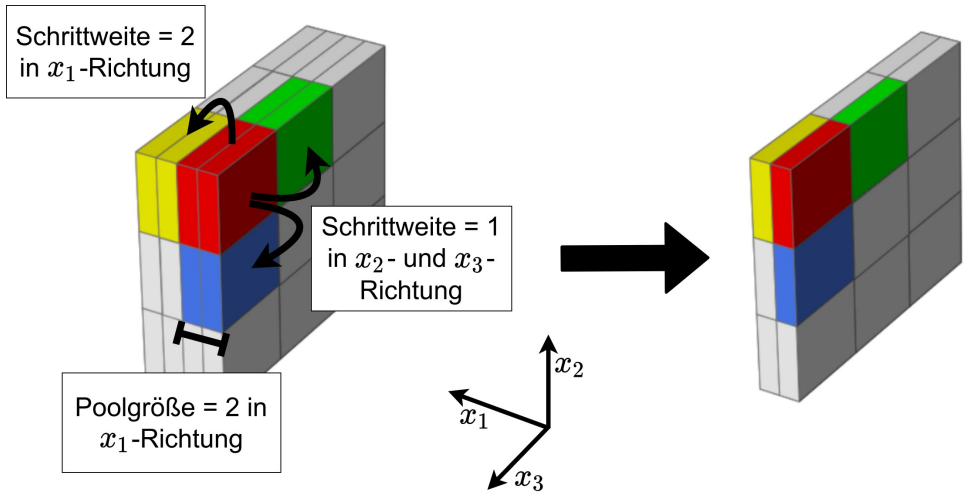

Abbildung 21: Visualisierung der MaxPooling3D-Operation mit einer Poolgröße von $(2, 1, 1)$ und einer Schrittweite von $(2, 1, 1)$ anhand von vier Eingangsframes zu je 3x3 Pixeln

Binärklassifizierung angesehen werden kann: entweder es befindet sich eine Radarkontrolle in einer Rasterzelle oder nicht. Bei einem ausgeglichenen Datensatz könnte eine solche Verlustfunktion direkt verwendet werden. Für den vorliegenden Anwendungsfall ist jedoch die Besonderheit zu beachten, dass es in den Daten deutlich mehr Rasterzellen ohne eine einzige Radarkontrolle gibt als Rasterzellen, in denen es mindestens eine gibt. Mit einer normalen Verlustfunktion werden falsch-negative Ergebnisse genauso hart bestraft wie falsch-positive. Andersherum werden richtig-positive Ergebnisse genauso begünstigt wie richtig-negative. Hier ergibt sich jedoch ein Problem. Ein bei der Trainierung sehr schnell zu findendes Minimum der Verlustfunktion liegt genau dort, wo die Vorhersage des gesamten Rasters „negativ“ ist. Somit kann beispielsweise bei einer Positivrate von 5 % im Datensatz eine Genauigkeit (engl. *accuracy*) von 95 % erzielt werden, wenn alle Rasterzellen als „negativ“ vorhergesagt werden. Allerdings entsteht damit keinerlei Erkenntnisgewinn. Damit der Optimierungsalgorithmus nicht nach kurzer Zeit in dieses schnell zu erreichende Minimum fällt, kann nach [10] eine gewichtete Verlustfunktion verwendet werden. Eine gewichtete Verlustfunktion weist den einzelnen Klassen Gewichtungen zu, die mit den Kreuzentropiewerten der jeweiligen Klasse multipliziert werden. Ist die Gewichtung größer als 1, wird der Wert der Kreuzentropie verstärkt bzw. mehr gewichtet und umgekehrt. Die in der vorliegenden Arbeit verwendete Verlustfunktion ist in Codeausschnitt 2 dargestellt.

```
1  def weighted_binary_crossentropy(weights):
2      def loss_fn(y_true, y_pred):
3          tf_y_true = tf.cast(y_true, dtype=y_pred.dtype)
4          tf_y_pred = tf.cast(y_pred, dtype=y_pred.dtype)
5
6          weights_v = tf.where(tf.equal(tf_y_true, 1), weights[1], weights[0])
7          crossentropy = K.binary_crossentropy(tf_y_true, tf_y_pred)
8          weighted_crossentropy = tf.multiply(crossentropy, weights_v)
9
10         loss = K.mean(weighted_crossentropy)
11         return loss
12
13     return loss_fn
```

Codeausschnitt 2: Implementierung einer gewichteten Kreuzentropie als Verlustfunktion für spärliche Daten

In Zeile 6 wird die Matrix *weights_v* berechnet, die die Gewichtungen jeweils an den Stellen enthalten, an denen die jeweilige Klasse in den wahren Daten vorkommen. In Zeile 7 wird die gewöhnliche binäre Kreuzentropie berechnet. In Zeile 8 werden dann die Gewichtungen durch elementweise Multiplikation zu den Vorkommnissen der jeweiligen Klasse multipliziert. Die Matrix *weighted_crossentropy* enthält nun die gewichtete Kreuzentropie für jeden Datenpunkt, also jede Rasterzelle. Der Verlustwert berechnet sich dann in Zeile 10 aus dem Durchschnitt aller dieser Werte.

Als Nächstes stellt sich noch die Frage, welche Gewichtungen verwendet werden sollten. Nach [11] bietet es sich an, die Gewichtungen anhand der Anteile der verschiedenen Klassen am gesamten Datensatz zu berechnen. Somit sind die Gewichtungen abhängig von der konkreten Beschaffenheit des Datensatzes. Konkret wird in [11] die Implementierung vorgeschlagen, die in Codeausschnitt 3 zu sehen ist.

```
1  def get_class_weights(dataset):
2      neg, pos = np.bincount(dataset.flatten())
3      total = neg + pos
4      weight_for_0 = (1.0 / neg) * (total / 2.0)
5      weight_for_1 = (1.0 / pos) * (total / 2.0)
6
7      return weight_for_0, weight_for_1
```

Codeausschnitt 3: Berechnung der Klassengewichtungen nach [11]

In Zeile 2 wird mit *np.bincount()* die Anzahl an „0“- und „1“-Werten im gesamten Datensatz berech-

net. In Zeile 4 und 5 werden dann die Gewichtungen berechnet, sodass diese reziprok proportional zum Anteil der Klassen sind. Nach die Skalierung um *total/2* bewirkt dabei, dass der Absolutwert des Verlusts nicht zu groß bzw. zu klein wird, das Verhältnis aber erhalten bleibt. Somit ist er besser interpretierbar als beispielsweise nach einer Normierung, bei der der eine Wert sehr klein wäre und der andere nahe 1. In einem zufälligen Teil des Radarkontrollen-Datensatzes sind beispielsweise 5.09 % der Rasterzellen „positiv“. Daraus ergeben sich die Gewichtungen $0,53$ für die Klasse „0“ und $9,83$ für die Klasse „1“.

Mit der Netzarchitektur und der Verlustfunktion sind nun alle Bestandteile des Modells definiert. Die Definition des Modells in TensorFlow ist in Anhang E zu sehen. In den folgenden Abschnitten kann nun mit der Trainierung des Modells begonnen werden.

4.2 Laden und vorbereiten des Datensatzes

Um den Datensatz als Trainingsdaten für das neuronale Netz zu verwenden, muss er in ein numpy-Array geladen werden. Zunächst ist das Ziel, die Frames des gesamten Datensatzes in ein numpy-Array mit der Form $(x,\ 50,\ 50)$ zu bringen, wobei x der Anzahl an Frames entspricht. Dafür kann der Code aus Codeausschnitt 1 verwendet werden, wobei über jedes Datum vom 17.07.2014 bis 25.10.2021 iteriert wird. Somit ergibt sich ein Array der Form $(2652,\ 50,\ 50)$. Die einzelnen Pixel enthalten bisher noch die summierte Standdauer der Radarkontrollen. Mit dem Wissen um die gewichtete Verlustfunktion ist jedoch klar geworden, dass es sich bei der Problemstellung eigentlich um eine Binärklassifizierung handelt. Daher muss nun ein Grenzwert ausgewählt werden, ab dem eine Rasterzelle als „positiv“ gilt. Auch hier muss zwischen der Verlässlichkeit der Meldungen und der Anzahl an positiven Datenpunkten abgewogen werden. Da es jedoch wichtiger ist, alle künftigen Radarkontrollen vorherzusagen als alle Rasterzellen korrekt vorherzusagen, in denen es keine Radarkontrolle gibt, sollte der Grenzwert relativ niedrig gewählt werden. Im Bezug auf die Verteilung der Standdauer in Abbildung 12 scheint eine halbe Stunde ein guter Grenzwert zu sein. Damit wird ein Großteil der Meldungen eingeschlossen, wobei die wenig verlässlichen Meldungen unter einer halben Stunde vernachlässigt werden. Die Klassenzuordnung je nach Überschreitung des Grenzwertes kann mit numpy einfach implementiert werden als *dataset = np.where(dataset > 0.5, 1, 0)*.

Als Nächstes muss der Datensatz in Sequenzen von je 17 Frames aufgeteilt werden, von denen die ersten 16 als Eingabesequenz dienen und das letzte als Zielframe. Da die Vorhersage von beliebigen Wochentagen möglich sein soll und möglichst viele Trainingsdaten benötigt werden, sollten auch die Sequenzen ausgenutzt werden, die sich überschneiden. Somit soll beispielsweise aus dem Array $[1,\ 2,\ 3,\ 4]$ bei einer Sequenzlänge von 2 nicht nur das Array $[[1,\ 2], [3,\ 4]]$ entstehen, sondern das Array $[[1,\ 2],\ [2,\ 3],\ [3,\ 4]]$. Wird der Datensatz auf diese Weise in Sequenzen unterteilt, entsteht ein Array der Form $(2636,\ 17,\ 50,\ 50)$. Das Array enthält nun also 2636 Sequenzen aus je 17 Frames mit einer Größe von 50x50 Pixel. Als Nächstes müssen die Sequenzen in zufällige Trainings- und Validierungsdaten unterteilt werden. Dazu werden die Sequenzen zufällig gemischt und anschließend

werden 90 % der Frames dem Trainingsdatensatz zugewiesen und 10 % den Validierungsdatensatz. Die Sequenzen müssen im letzten Schritt noch je in die Ein- und Zielausgabe des NNs unterteilt werden. Dies wird in Codeausschnitt 4 von der Funktion *create_xy_frames()* implementiert.

```
1 def create_xy_frames(data):
2     x = data[:, 0:-2, :, :]
3     y = data[:, -1, :, :]
4     y = np.expand_dims(y, axis=1)
5     return x, y
6 
7 x_train, y_train = create_xy_frames(train_dataset)
8 x_val, y_val = create_xy_frames(val_dataset)
```

Codeausschnitt 4: Implementierung der Funktion *create_xy_frames()* zum Unterteilen der Sequenzen in Eingabe und Zielausgabe

In Zeile 2 wird zunächst die Eingabesequenz extrahiert. Diese besteht aus dem ersten bis vorletzten Frame, somit hat der Eingabedatensatz *x_train* schließlich die Form $(2372,\ 16,\ 50,\ 50)$. Die Zielausgabe hingegen wird in Zeile 3 erstellt und besteht nur aus dem letzten Frame jeder Sequenz. Durch die Auswahl nur eines Elements einer Dimension geht diese Dimension jedoch komplett verloren. Da das NN die Dimension jedoch erwartet, muss sie in Zeile 4 wieder hinzugefügt werden, sie hat dann die Länge 1. Damit hat die Zielausgabe *y_train* die Form $(2372,\ 1,\ 50,\ 50)$. Nun sind die Trainingsdaten bereit und das Modell kann damit trainiert werden.

4.3 Trainierung des Modells

Für die Trainierung werden zunächst noch zwei Callbacks definiert, um den Trainingsprozess zu verbessern. Callbacks sind Funktionen, die nach jeder Epoche des Trainings aufgerufen werden. Diese erhalten das Modell bei jedem Aufruf in seinem momentanen Zustand. Daraufhin können Callbacks das Modell beispielsweise auf Testdaten anwenden, die Parameter des Trainings verändern oder das Training beenden. Ein hilfreicher Callback ist *EarlyStopping*. Dieser überwacht die Performance des Modells mithilfe einer beliebigen Metrik und beendet das Training, wenn sich die Metrik nicht mehr signifikant verbessert [35]. Somit kann überflüssige Trainingszeit gespart und Überanpassung vermieden werden. Die Definition des Callbacks ist in Codeausschnitt 5 in Zeile 1 und 2 zu sehen.

```
1  early_stopping = keras.callbacks.EarlyStopping(monitor="val_loss",
2      patience=10, restore_best_weights=True)
3  reduce_lr = keras.callbacks.ReduceLROnPlateau(monitor="val_loss", patience=5)
4
5  epochs = 32
6  batch_size = 12
7  history = model.fit(
8      x_train, y_train,
9      batch_size=batch_size, epochs=epochs,
10     validation_data=(x_val, y_val),
11     callbacks=[early_stopping, reduce_lr]
12 )
```

Codeausschnitt 5: Definition der Callbacks zur Verbesserung des Trainings und Trainierung des Modells

Die überwachte Metrik ist der Verlustwert der Validierung. Der Parameter *patience* bestimmt, wie viele Epochen ohne eine Verbesserung der überwachten Metrik gewartet werden sollen, bis das Training beendet wird. Die in Zeile 2 aktivierte Option *restore_best_weights* besagt, dass das Modell am Ende des Trainings in den Zustand gebracht werden soll, bei dem die überwachte Metrik den besten Wert hatte. Dies kann beispielsweise sinnvoll sein, wenn sich der Verlustwert der Validierung durch Überanpassung gegen Ende wieder verschlechtert. Allerdings muss diese Option mit Vorsicht verwendet werden, da sie auch eine Überanpassung an die Validierungsdaten begünstigt.

Ein weiterer hilfreicher Callback ist *ReduceLROnPlateau*. Dieser Callback verringert die Lernrate, wenn sich eine bestimmte Metrik nicht weiter verbessert [36]. Dies hat dann zur Folge, dass die Parameter des Modells noch etwas weiter optimiert werden können. In Codeausschnitt 5 wird dieser Callback in Zeile 3 definiert. Auch hier wird der Verlustwert der Validierung überwacht. Ähnlich wie bei *EarlyStopping* bestimmt der Parameter *patience* hier, nach wie vielen Epochen ohne eine Verbesserung der überwachten Metrik die Lernrate verringert wird. Dieser Wert muss kleiner sein als der von *EarlyStopping*, damit der Callback einen Effekt hat. In Zeile 11 werden die Callbacks schließlich an *model.fit()* übergeben, wodurch sie bei der Trainierung angewendet werden.

Als Nächstes müssen noch die maximale Anzahl an Trainingsepochen und die Stapelgröße (engl. *batch size*) ausgewählt werden. Die maximale Anzahl an Trainingsepochen kann großzügig gewählt werden, da das Training letztendlich vom *EarlyStopping*-Callback beendet werden soll. Wird das Training davor durch die maximale Anzahl an Trainingsepochen beendet, bleibt eventuelles Potenzial ungenutzt. Für das hier verwendete Modell ist eine maximale Anzahl an Trainingsepochen von 32 angemessen. Die Stapelgröße hingegen ist ein kritischer Parameter. Je größer die Stapelgröße ist, desto schneller verläuft das Training, da der gesamte Stapel parallel verarbeitet wird. Jedoch muss der gesamte Stapel in den Arbeitsspeicher passen. Besonders unter Verwendung einer GPU ist der

Arbeitsspeicher jedoch stark limitiert, da das Modell mit dem Stapel komplett in den Speicher der GPU passen muss. Die Stapelgröße sollte also so gewählt werden, dass der Speicher bestmöglich ausgefüllt wird. Da die Trainingsdaten im vorliegenden Anwendungsfall größer sind als bei einfachen Klassifizierungsaufgaben, kann die Stapelgröße nicht sehr groß gewählt werden. Bei einer Stapelgröße von 12 reicht der 8 GB große Speicher der verwendeten Grafikkarte *Nvidia GeForce GTX 1080* gerade aus.

Nun, da die notwendigen Hyperparameter definiert sind, kann das Modell trainiert werden. Dazu wird *model.fit()* aufgerufen, wie in Codeausschnitt 5 in Zeile 7 bis 12 gezeigt. Diese Methode gibt die Entwicklung verschiedener Metriken im Verlauf des Trainings zurück. Auf der oben erwähnten Grafikkarte dauert das Training pro Epoche ca. 80 Sekunden. Nach 26 Epochen beendet der *EarlyStopping*-Callback das Training, insgesamt dauert das Training also ca. 35 Minuten. Der Verlauf des Verlustwerts ist in Abbildung 22 dargestellt.

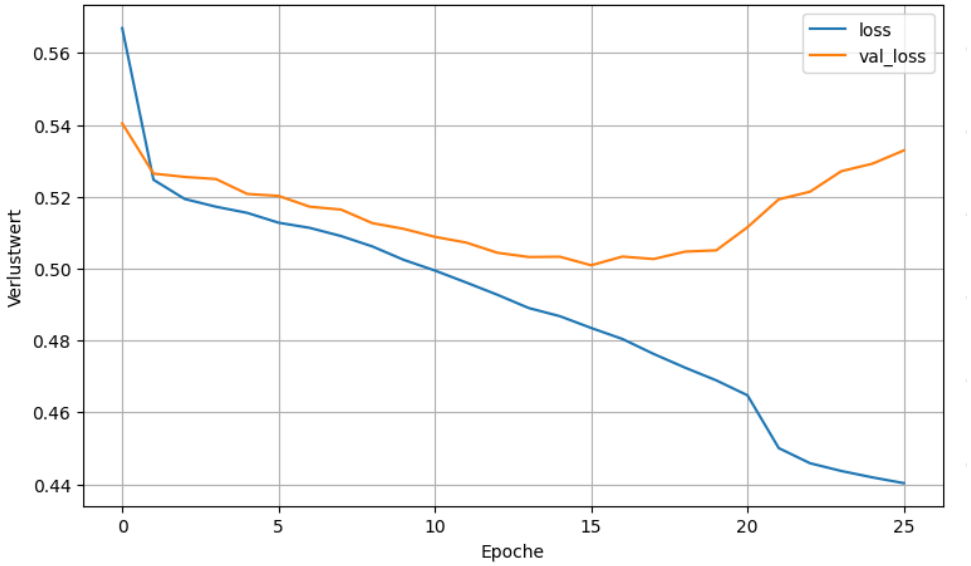

Abbildung 22: Entwicklung des Verlustwerts über die Epochen des Trainings

In der Abbildung ist zunächst erkennbar, dass der Verlustwert über die Epochen hinweg optimiert werden kann. Ab der 15. Epoche beginnt der Verlustwert der Validierung jedoch anzusteigen, während der Verlustwert des Trainings weiter sinkt. Dieser Effekt spricht für eine Überanpassung. Dies bedeutet, dass das Modell ab einer bestimmten Epoche beginnt, die Trainingsdaten „auswendig“ zu lernen. Dadurch handelt das Modell bei der Validierung jedoch nicht mehr nach den globalen Mustern, wodurch die Vorhersagen anhand der Validierungsdaten schlechter werden. Der *EarlyStopping*-Callback stellt jedoch den besten Zustand des Modells wieder her, der in diesem Fall bei Epoche 15 bestand.

4.4 Nachbearbeitung der Vorhersagen

Mit dem trainierten Modell können nun Vorhersagen erzeugt werden. In Codeausschnitt 6 ist dieser Prozess anhand eines zufälligen Beispiels aus den Validierungsdaten gezeigt.

```
1 def predict_random_example(model, val_dataset):
2     example = val_dataset[np.random.choice(range(len(val_dataset)), size=1)[0]]
3     x = np.expand_dims(example[0:-1, ...], axis=0)
4     y_true = np.squeeze(example[-1, ...])
5     y_pred = np.squeeze(model.predict(x))
6 
7     return y_true, y_pred
```

Codeausschnitt 6: Erzeugung einer Vorhersage anhand der Validierungsdaten

Dafür wird in Zeile 1 zunächst eine zufällige Sequenz von 17 Frames aus den Validierungsdaten ausgewählt. In Zeile 2 und 3 wird die Sequenz in die Eingabe- und Zieldaten unterteilt, die dann jeweils in die richtige Form gebracht werden. Die eigentliche Vorhersage findet dann in Zeile 5 statt, wo sie durch *model.predict(x)* ausgeführt wird. Die Arrays *y_true* und *y_pred* enthalten nun die wahren und die vorhergesagten Ergebnisse und haben die Form $(50,\ 50)$. In Abbildung 23 sind zwei Beispiele für Vorhersagen grafisch dargestellt.

Links im Bild sieht man die realen Standorte der mobilen Radarkontrollen. Rechts hingegen ist die Ausgabe des NNs dargestellt. Je dunkler die Pixel, desto geringer ist der Zahlenwert und umgekehrt. Zunächst kann man leicht erkennen, dass nicht alle Pixel schwarz sind, sondern dass ein weiter Bereich an Zahlenwerten abgedeckt wird. Das spricht dafür, dass die gewichtete Verlustfunktion ihren Zweck erfüllt. Als Nächstes ist erkennbar, dass das Modell scheinbar teilweise das Straßennetz erlernt hat. In den Vorhersagen sind große Straßen als Linien und große Städte als Cluster deutlich erkennbar. Dies ist durchaus nachvollziehbar, da die Radarkontrollendichte hier besonders hoch ist. Dadurch stellt sich jedoch die Frage, ob die Vorhersage des Modells überhaupt maßgeblich von der Eingabe abhängig ist. Dafür spricht jedenfalls der Unterschied zwischen dem oberen und unteren Beispiel. Im oberen Beispiel sind in der Realität deutlich weniger Radarkontrollen vorhanden als im unteren Beispiel. Das könnte beispielsweise dadurch zustande kommen, dass das obere Beispiel von einem Samstag oder Sonntag stammt. In der bereits diskutierten Abbildung 15 ist schließlich deutlich zu erkennen, dass es am Wochenende deutlich weniger Radarkontrollen gibt als unter der Woche. Auch die Vorhersage spiegelt diesen Sachverhalt wieder. Während die Vorhersage im oberen Beispiel viele dunkle Pixel enthält, ist die Vorhersage im unteren Beispiel insgesamt deutlich heller, das Modell geht also insgesamt von einer höheren Wahrscheinlichkeit für mobile Radarkontrollen aus.

Zuletzt ist noch eine offensichtliche Tatsache anzumerken, die aber dennoch wichtig ist, und zwar die Beschaffenheit der Ausgabe des Modells. Jedes Pixel wird als Zahlenwert zwischen 0 und 1

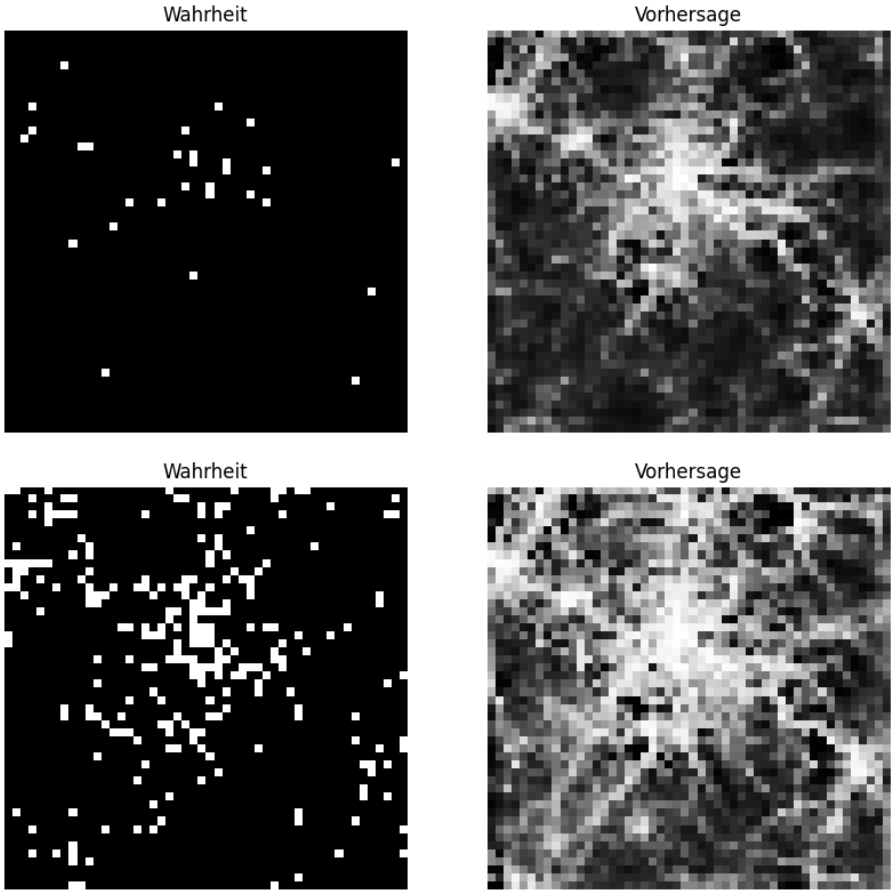

Abbildung 23: Beispiele für Vorhersagen des trainierten Modells (rechts) im Vergleich zur Realität (links)

ausgegeben. Das liegt an der Aktivierungsfunktion *Sigmoid* des letzten Layers, die die Zahlengerade auf den Bereich zwischen 0 und 1 abbildet. Der Bereich zwischen 0 und 1 legt nahe, dass die Ausgabe des Modells als Wahrscheinlichkeitskarte angesehen werden kann. Nun stellt sich die Frage, ob die Ausgabe so belassen oder ob sie noch nachverarbeitet werden sollte. Hierzu gibt es zwei relevante Aspekte: die praktische Nutzbarkeit und die Performanceevaluierung. Zur Performanceevaluierung muss beachtet werden, dass die Problemstellung bisher als Binärklassifizierung angesehen wird. Daher muss die Ausgabe zur Evaluierung wie auch die Eingabe eine Binärklassifizierung sein. Nur so kann bestimmt werden, wie viele Rasterzellen richtig klassifiziert werden. Im Bezug auch die praktische Nutzbarkeit ist es durchaus möglich, das Ergebnis als Wahrscheinlichkeit zu belassen. Jedoch hat diese Vorgehensweise den Nachteil, dass eine Wahrscheinlichkeit unter Umständen schwer einzuschätzen sein kann, besonders in der Hektik des Straßenverkehrs. Daher bietet es sich an, die ausgegebenen Wahrscheinlichkeiten in mehrere Gefahrenstufen zu unterteilen, die einfach verständlich sind. Diese Gefahrenstufen könnten beispielsweise *niedrig*, *mittel*, *hoch* und *sehr hoch* heißen.

Unabhängig davon, ob die Wahrscheinlichkeiten als Binärklassifizierung in zwei Stufen aufgeteilt werden soll oder als Gefahrenangabe in mehrere, stellt sich die Frage, wo die Grenzen gesetzt werden sollten. Eine gute und einfach zu interpretierende Möglichkeit ist die Entscheidung nach Perzentil. In

diesem Fall beziehen sich Perzentile auf die Verteilung der Rasterzellenwerte. Die 90. Perzentilgrenze liegt beispielsweise so, dass 90 % aller Rasterzellen einen Wert unter dieser Grenze haben. Bei der Wahl der Perzentilgrenze hat im Fall einer Binärklassifizierung eine große Auswirkung auf das Ergebnis. Dies soll in Abbildung 24 verdeutlicht werden.

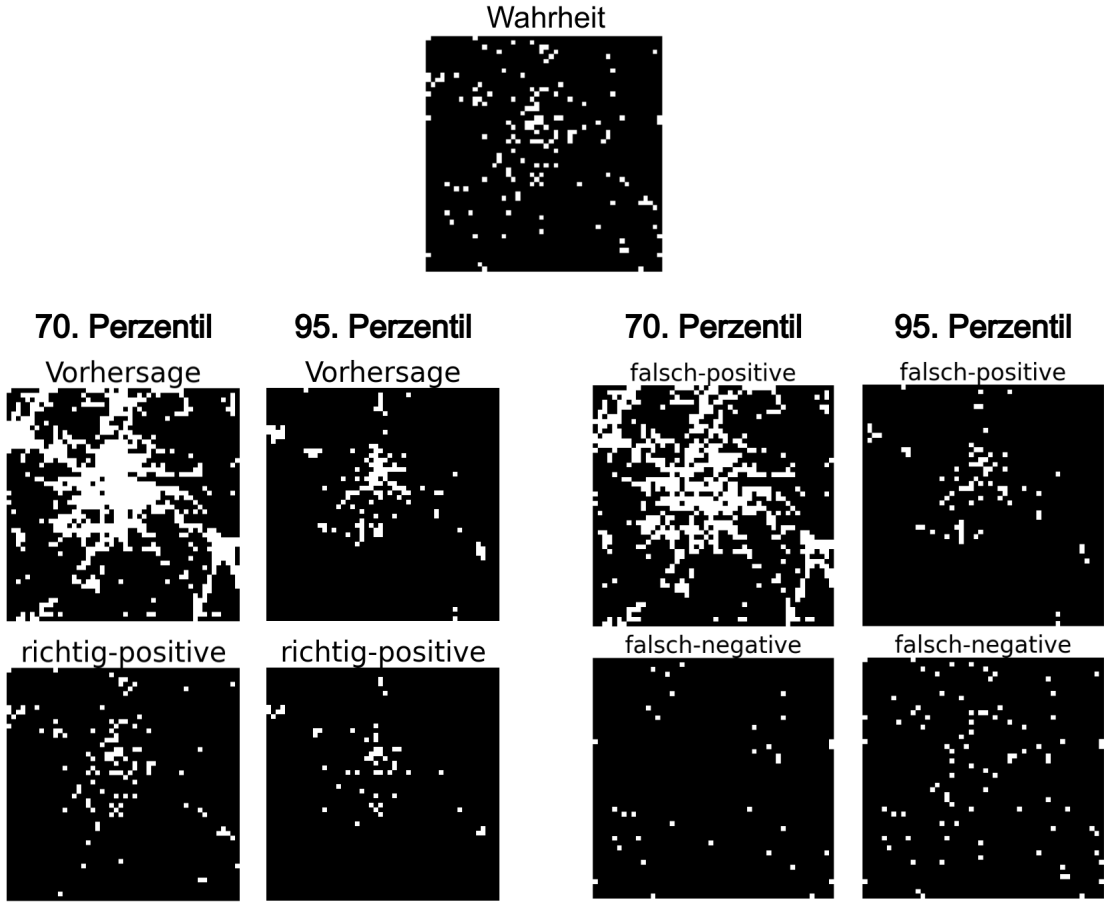

Abbildung 24: Auswirkung der Perzentilgrenze auf die Vorhersage im Fall einer Binärklassifizierung

In der Abbildung ist als Beispiel eine Vorhersage bei zwei verschiedenen Perzentilgrenzen zu sehen. Dabei zeigt das oberste Bild die Grundwahrheit. Unten ist die Vorhersage je für die 70. und 95. Perzentilgrenze dargestellt. Außerdem ist die Vorhersage jeweils in die richtig-positiv, falsch-positiv und falsch-negativ vorhergesagten Rasterzellen aufgeschlüsselt. Im Einklang mit der Definition der Perzentilgrenze ist beim Vergleich der Vorhersagen offensichtlich, dass mit der 70. Perzentilgrenze deutlich mehr Rasterzellen als „positiv“ eingestuft werden als mit der 95. Perzentilgrenze. Dies hat zur Folge, dass mit der 70. Perzentilgrenze mehr der tatsächlich positiven Rasterzellen korrekt vorhergesagt werden. Jedoch resultieren daraus auch deutlich mehr Falsch-Positive. Stellt man sich dieses Szenario in der Praxis vor, würden Benutzer relativ häufit gewarnt werden, obwohl keine akute Gefahr für Radarkontrollen besteht. Es lässt sich jedoch argumentieren, dass es in der Praxis wichtiger ist, dass vor so vielen tatsächlich positiven Rasterzellen gewarnt werden sollte wie möglich, auch wenn dadurch Fehlwarnungen häufiger werden. In Abbildung 24 sind bisher nur Beispiele für zwei konkrete Perzentilgrenzen dargestellt. Für eine genauere Analyse sind jedoch Werte über alle Perzentilgrenzen interessant. Diese sind als Graphen für dasselbe Beispiel in Abbildung 25 dargestellt.

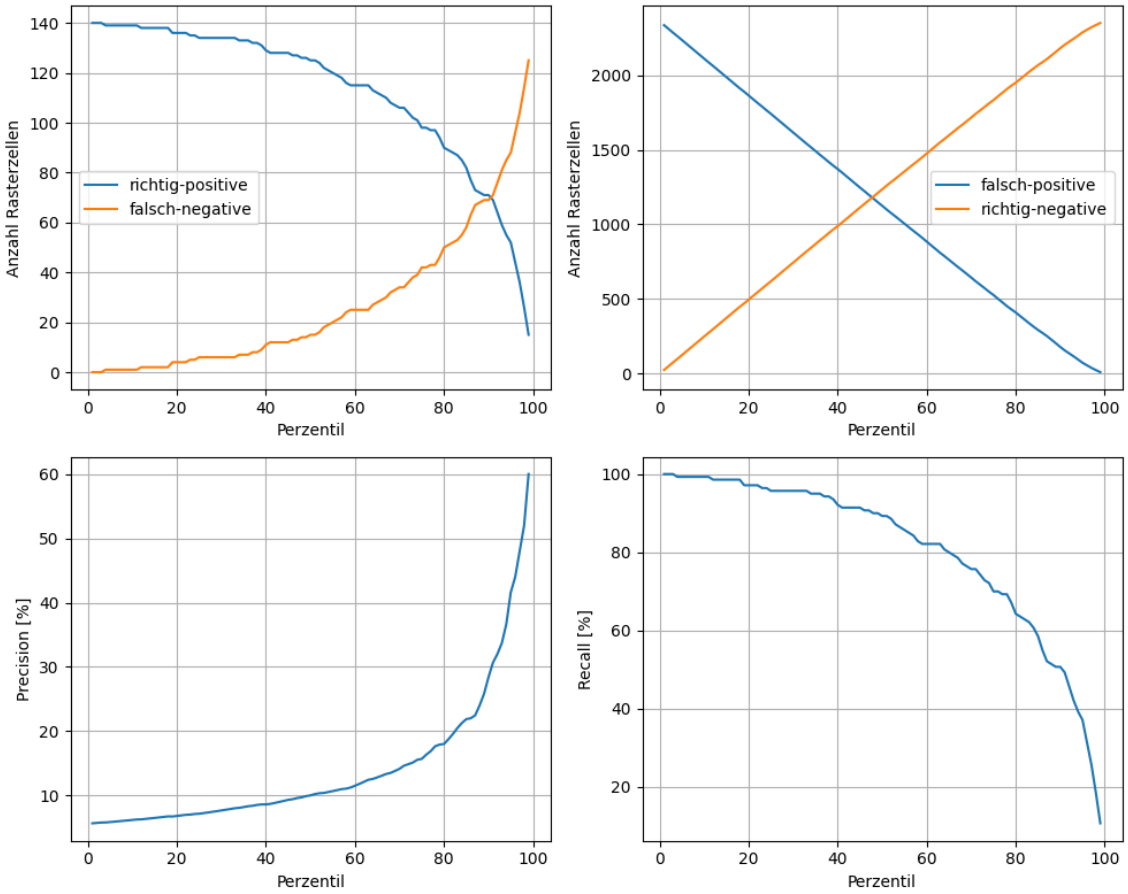

Abbildung 25: Verschiedene Metriken einer Beispielvorhersage über Perzentilgrenzen von 1 % bis 99 %

In den oberen beiden Graphen sind die Kennzahlen, die oben schon für zwei Perzentilgrenzen analysiert wurden, über alle Perzentilgrenzen aufgetragen. Für die Betrachtung ist wichtig zu wissen, dass es insgesamt 2500 Rasterzellen gibt, von denen in diesem Beispiel in der Realität 140 „positiv“ sind. Dies wird auch deutlich, wenn man die linken Enden der Graphen betrachtet. Bei der nullten Perzentilgrenze werden per Definition alle Zellen als „positiv“ markiert, weshalb dort auch die Anzahl der Richtig-Positiven maximal ist. Jedoch ist dort auch die Anzahl der Falsch-Positiven maximal. Aufgrund der unbalancierten Verteilung der positiven Rasterzellen verhält sich die Richtig-positiven- und Falsch-negativen-Rate über die Perzentilgrenzen hinweg exponentiell, während sich die Falsch-positiven- und Richtig-negativen-Rate annähernd linear verhält. Zur intuitiveren Bewertung der Perzentilgrenzen sind die absoluten Zahlen wenig aussagekräftig, zumal sie sich bei verschiedenen Beispielen unterschiedlich verhalten. Daher gibt es verschiedene Metriken, die die absoluten Werte zueinander ins Verhältnis setzen. Diese werden in [11] beschrieben. Die einfachste solche Metrik ist die Genauigkeit, die den Anteil der richtig klassifizierten Elemente darstellt, es gilt also

$$\text{Genauigkeit} = \frac{\text{Korrekt klassifizierte Elemente}}{\text{Gesamtanzahl der Elemente}}.$$

In Abschnitt 4.1 wurde jedoch bereits erörtert, warum die Genauigkeit für unbalancierte Daten nicht aussagekräftig ist. Deulich hilfreicher sind die Metriken *Precision* und *Recall*. Precision ist nach [11]

der Anteil der positiv vorhergesagten Elemente, die auch in der Realität „positiv“ sind, es gilt also

$$\text{Precision} = \frac{\text{Richtig-Positive}}{\text{Richtig-Positive} + \text{Falsch-Positive}}.$$

Recall gibt hingegen an, welcher Anteil der tatsächlich positiven Elemente auch als „positiv“ erkannt werden. Hier gilt somit

$$\text{Recall} = \frac{\text{Richtig-Positive}}{\text{Richtig-Positive} + \text{Falsch-Negative}}.$$

Um diese Metriken genauer zu verstehen, ist es hilfreich zu betrachten, wann sie 100 % betragen. Damit die Precision 100 % beträgt, darf es keine Falsch-Positiven geben. Somit ist die Precision ein Maß dafür, wie präzise *nur* die tatsächlich Positiven erkannt werden. Wichtig ist hierbei anzumerken, dass die Precision nichts darüber aussagt, wie viele der tatsächlich Positiven erkannt werden. Die Precision könnte also 100 % betragen, obwohl nur wenige Prozent der tatsächlich Positiven erkannt werden. Wichtig ist nur, dass es keine Falsch-Positiven gibt. Genau andersherum ist es nun beim Recall. Der Recall beträgt genau dann 100 %, wenn es keine Falsch-Negativen gibt. Es müssen also alle tatsächlich positiven Elemente erkannt werden. Dabei ist es jedoch unerheblich, wie viele in der Realität negativen Elemente zusätzlich noch als „positiv“ erkannt werden.

Insgesamt ist also klar, dass bei der Wahl der Perzentilgrenze zwischen Precision und Recall abgewogen werden muss. In Abbildung 25 sind Precision und Recall in den unteren beiden Graphen für das konkrete Beispiel dargestellt. Anhand des Precision-Graphs ist zu erkennen, dass selbst mit einer Perzentilgrenze von 99 % keine Precision über 60 % möglich ist. Es gibt also immer eine gewisse, nicht zu vernachlässigende Anzahl an Falsch-Positiven. Außerdem ist erkennbar, dass die Precision-Kurve bei Perzentilgrenzen gegen 99 % sehr stark ansteigt. Die Rasterzellen, die einen Wert in diesem Bereich haben, sind also mit großer Wahrscheinlichkeit tatsächlich „positiv“. Im Bezug auf den Anwendungsfall der Vorhersage von mobilen Radarkontrollen hat die Recall-Kurve eine noch größere Bedeutung, da möglichst viele tatsächlich vorhandene Radarkontrollen in der Vorhersage enthalten sein sollten. Jedoch bleibt es eine Abwägung zwischen Precision und Recall. Daher bietet es sich an, beide Metriken in einem Graph in Relation zueinander zu setzen. Diesen Graph nennt man *Precision Recall Curve* (PRC). Die PRC der bisher verwendeten Beispielvorhersage ist in Abbildung 26 links dargestellt.

An diesem Graphen kann abgelesen werden, welche Precision bei welchem Recall erreichbar ist und umgekehrt. Soll beispielsweise ein Recall von 70 % erreicht werden, entspricht dies einer Precision von ca. 15 %. In Worten werden also 70 % aller Rasterzellen korrekt vorhergesagt, die tatsächlich eine Radarkontrolle beinhalten, jedoch enthalten nur 15 % aller als „positiv“ vorhergesagten Rasterzellen tatsächlich eine Radarkontrolle. Wenn die PRC für den gesamten Validierungsdatensatz erstellt wird, eignet sie sich auch dazu, die Performance verschiedener Modelle zu vergleichen. Falls ein Modell besser performt als ein anderes, liegt die PRC dieses Modells eher weiter oben rechts. Es ergeben sich also insgesamt größere Precision-Recall-Paare.

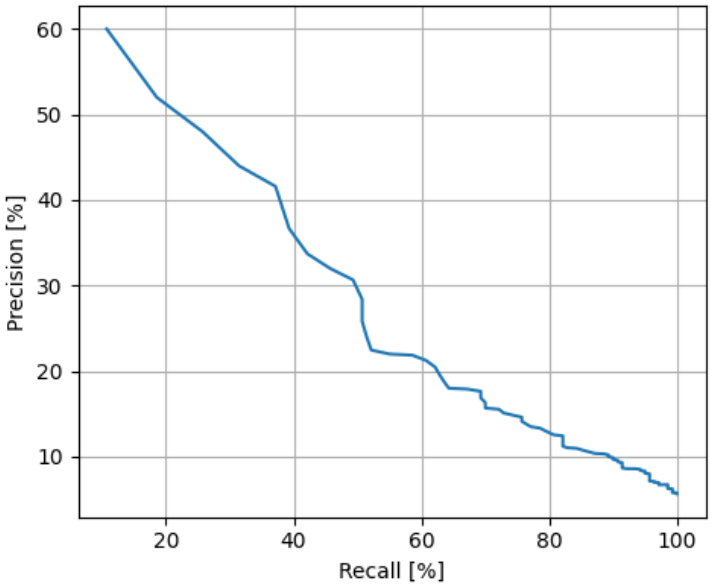

Abbildung 26: *Precision Recall Curve* (PRC) einer Beispielvorhersage

Die PRC kann nun gemeinsam mit den einzelnen Graphen von Precision und Recall verwendet werden, um geeignete Perzentilgrenzen für eine Gefahreneinstufung auszuwählen. Für die Einteilung in die vier Gefahrenstufen *niedrig*, *mittel*, *hoch* und *sehr hoch* müssen drei Perzentilgrenzen ausgewählt werden. Für die Grenze, ab der die Gefahrenstufe *mittel* gelten soll, scheint das 60. Perzentil ein guter Wert zu sein. Dies entspricht einem Recall von ca. 82 % und einer Precision von ca. 12 %. Der Recall-Wert bedeutet, dass nur 18 % der tatsächlichen Radarkontrollen (fälschlicherweise) auf die Gefahrenstufe *niedrig* fallen. Die Precision ist jedoch nicht sonderlich hoch. Es befinden sich folglich sehr viele Falsch-Positive in der Gefahrenstufe *mittel*. Die nächste Gefahrenstufe (für die Gefahrenstufe *hoch*) kann beim 85. Perzentil angelegt werden. Hier beträgt der Recall ca. 60 % und die Precision ca. 23 %. Die Precision hat sich demnach im Vergleich zur Gefahrenstufe *mittel* fast verdoppelt, während der Recall nur um ein Viertel geringer geworden ist. Bei der Gefahrenstufe *sehr hoch* sollte sich das Modell sehr sicher sein, dass sich in einer solchen Rasterzelle eine Radarkontrolle befindet. Daher sollte eine Perzentilgrenze nahe 100 % gewählt werden, wie beispielsweise die 98. Perzentilgrenze. Hier beträgt der Recall zwar nur noch ca. 10 %, die Precision jedoch ca. 55 %. Statistisch befinden sich demnach in ca. der Hälfte aller Rasterzellen der Gefahrenstufe *sehr hoch* tatsächlich eine Radarkontrolle.

Werden die Gefahrenstufen mit unterschiedlichen Farben markiert, ergibt sich für das bisher verwendete Beispiel die in Abbildung 27 dargestellte Gefahrenkarte.

Außerdem sind in der Abbildung die tatsächlichen Radarkontrollen mit einem Kreuz markiert. Wird einem Autofahrer bei jedem Rasterzellenwechsel die neue Gefahrenstufe mitgeteilt, kann dieser sie leichter einschätzen und sich entsprechend verhalten.

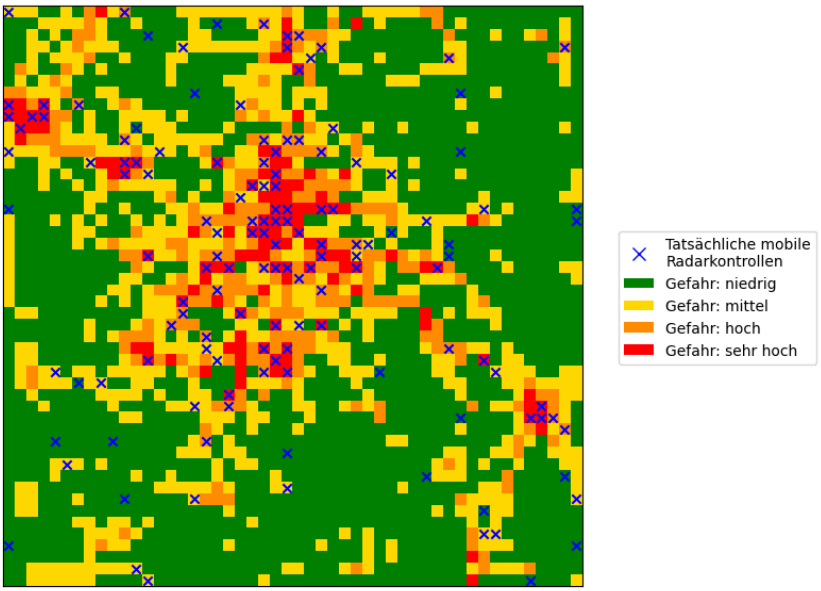

Abbildung 27: Resultierende nachbearbeitete Vorhersage mit vier Gefahrenstufen im Vergleich zur Grundwahrheit

4.5 Evaluierung des Modells

In diesem Abschnitt werden zunächst Metriken festgelegt, mit denen die Performance des Modells unter verschiedenen Bedingungen verglichen werden kann. Anschließend wird die Performance des Modells anhand der Validierungsdaten evaluiert, indem untersucht wird, inwiefern die Vorhersagen des Modells von den konkreten Eingabeframes abhängen.

Zunächst gilt es Metriken zu definieren, anhand derer das Modell unter verschiedenen Bedingungen verglichen werden kann. Da Precision und Recall von der Perzentilgrenze abhängig sind, eignen sie sich nicht direkt. Was sich hingegen eignet, ist die Precision-Recall-Kurve, da sie Precision und Recall über alle Perzentilgrenzen hinweg darstellt. Zwei PRCs können verglichen werden, indem ermittelt wird, welche Kurve über der anderen liegt. Jedoch könnte bei verschiedenen Recall-Werten eine unterschiedliche Kurve die höhere Precision haben. Eine Möglichkeit, eine komplette PRC in einem einfach vergleichbaren Zahlenwert zusammenzufassen, besteht darin, die Fläche unter der Kurve zu berechnen. Dieser Wert wird *Area Under the Precision Recall Curve* (AUPRC) genannt. Mit der AUPRC kann die Performance eines Modells mit einer Zahl beschrieben werden und zwei Modelle können somit einfach verglichen werden.

Anhand der PRC und der AUPRC soll nun zunächst das in den vorherigen Abschnitten implementierte Modell evaluiert werden. Dabei soll auch eine der Kernfragen der vorliegenden Arbeit beantwortet werden. Diese Kernfrage ist, ob sich die Standorte von mobilen Radarkontrollen anhand von historischen Daten und insbesondere derer der letzten 16 Tage vorhersagen lassen. Besonders interessant ist hierbei, inwieweit die Vorhersagen konkret von den 16 Eingabeframes abhängen. Wenn die Ausgabe praktisch unabhängig von den Eingabeframes wäre, würde eine rein statistische Auswertung des Da-

tensatzes für die Vorhersagen genügen, wie z. B. die Identifizierung von Hotspots. Um das Ausmaß der Abhängigkeit zu ermitteln, bietet es sich an, die Zielframes zufällig zu vertauschen. Somit kann überprüft werden, ob die Vorhersagen des Modells mit einem zufällig ausgewählten Zielframe genau so gut übereinstimmen wie mit dem Zielframe des tatsächlichen nächsten Tages. Wenn dem so ist, sind die Vorhersagen weitestgehend unabhängig von den konkreten Eingabeframes. Erzielt das Modell jedoch bessere Ergebnisse mit den wahren Zielframes, ist eine Abhängigkeit bestätigt. Bevor die Überprüfung durchgeführt werden kann, sollte jedoch die Wahl des Validierungsdatensatzes angepasst werden. In Abschnitt 4.2 wurde bisher definiert, dass sich die Sequenzen des Datensatzes beliebig überschneiden können, um eine größere Menge an Trainingsdaten zu erhalten. Die erzeugten Sequenzen wurden dann zufällig dem Trainings- und Validierungsdatensatz zugewiesen. Dies hat zur Folge, dass sich die Sequenzen des Trainings- und Validierungsdatensatzes u. U. nur um zwei Frames unterscheiden - das Erste und das Letzte. Dies hat auch zur Folge, dass die Zielframes des Validierungsdatensatzes im Trainingsdatensatz vorhanden sein können. Somit hätte das Modell die Zielframes der Validierung schon gesehen, was die Evaluierung der Abhängigkeit verfälschen würde. Um sicherzustellen, dass dies nicht der Fall ist, werden die nach dem Anfangsdatum sortierten Sequenzen zunächst in Gruppen von je 34 Sequenzen unterteilt. Da dieser Wert der doppelten Sequenzlänge inklusive der Zielframes entspricht, überschneiden sich die Gruppen nicht. Als Nächstes werden die Sequenzgruppen gemischt und zufällig in Trainings- und Validierungsdaten unterteilt. Zuletzt werden die Trainings- und Validierungsdaten nochmals in sich gemischt. Mit diesen korrigierten Datensätzen muss das Modell nun erneut trainiert werden und die Evaluierung mit gemischten Zielframes kann durchgeführt werden. Die daraus resultierenden PRCs sind in Abbildung 28 zu sehen. Außerdem sind die AUPRC-Werte der Kurven in der Legende aufgeführt.

Hierbei ist anzumerken, dass die PRCs den Durchschnitt des gesamten Validierungsdatensatzes darstellen und nicht wie in Abbildung 26 nur eine einzelne Vorhersage. Dadurch verlaufen die Kurven weicher und gleichmäßiger. Es ist deutlich zu erkennen, dass die mit den zufälligen Zielframes erzeugte PRC (orange) am niedrigsten liegt und auch den geringsten AUPRC-Wert von 0,236 aufweist. Die PRC mit den korrekten Zielframes liegt hingegen höher. Dadurch erfährt auch der AUPRC-Wert einen Zuwachs von ca. 15 % und liegt somit bei 0,271.

Im vorherigen Abschnitt wurde bereits aufgezeigt, dass das Modell den Unterschied zwischen Wochenende und unter der Woche gut erlernen kann. Dieser Unterschied ist jedoch sehr einfach zu erkennen, da es am Wochenende durchschnittlich nur halb so viele Radarkontrollen gibt wie unter der Woche. Daher ist es für die Evaluierung interessant, diesen Effekt bei der Analyse auszuschließen. Dies kann realisiert werden, indem nur die Samstage und Sonntage untereinander gemischt werden, wie auch die Tage Montag bis Freitag. Es kann auch noch einen Schritt weiter gegangen werden, indem nur gleiche Wochentage gemischt werden. Hierdurch werden jedoch u. U. wochentagabhängige Muster unterdrückt, die eine durchaus valide Leistung des Modells darstellen. Andererseits kann mit diesem Vorgehen untersucht werden, ob überhaupt wochentagabhängige Muster in den Daten existieren. Daher sind in Abbildung 28 beide Ansätze dargestellt. Es ist erkennbar, dass die Kurven wie erwartet etwas über der Kurve der komplett zufälligen Zielframes liegt. Daraus kann abgeleitet werden, dass das Modell tatsächlich den Unterschied zwischen Wochenende und unter der Woche

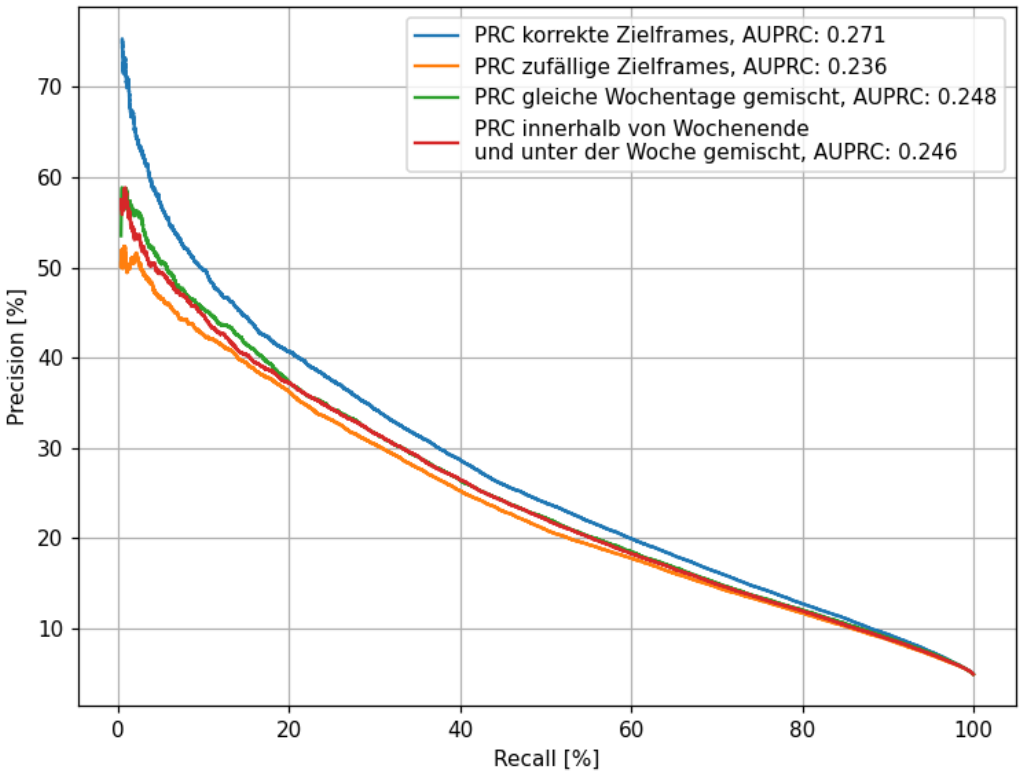

Abbildung 28: PRCs und AUPRC-Werte mit wahren und gemischten Zielframes

gelernt hat. Jedoch liegen die Kurven immer noch deutlich unter der Kurve der korrekten Zielframes. Daraus folgt, dass es im Datensatz noch weitere Muster gibt, die über den Unterschied zwischen Wochenende und unter der Woche hinaus gehen und dass die gewählte Modellarchitektur fähig ist, diese zu erlernen. Bei den Kurven ist besonders der linke Rand interessant, da dies der Bereich ist, in dem sich das Modell mit den Vorhersagen sehr sicher ist. Daher ist dieser Bereich in Abbildung 29 vergrößert dargestellt.

Der dargestellte Bereich entspricht einer Perzentilgrenze von etwa 96 % am rechten Rand bis 99,5 % am linken Rand. Hier bestätigt sich die Vermutung, dass es wochentagabhängige Muster in den Daten gibt, da die grüne Kurve, für die nur gleiche Wochentage gemischt wurden, etwas über der roten Kurve liegt. Die Signifikanz dieses Unterschieds ist jedoch diskutabel. Zusammenfassend bestätigt die durchgeführte Evaluierung, dass es verschiedene Muster in den Daten gibt und dass das Modell dazu in der Lage ist, diese Muster zu erlernen.

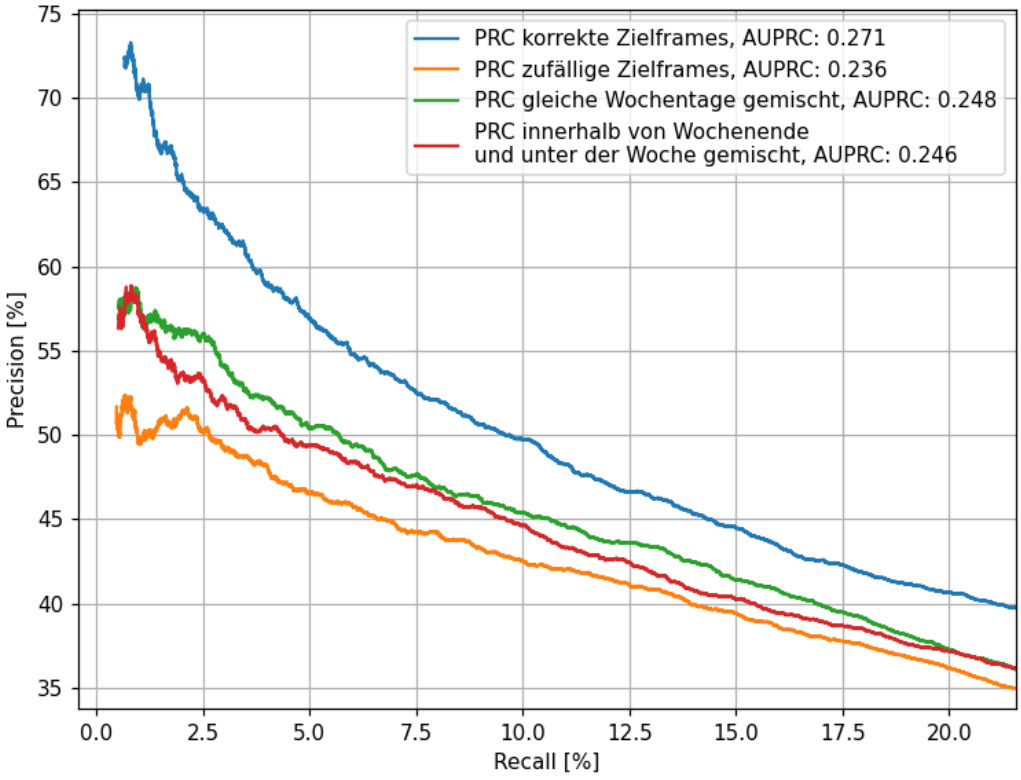

Abbildung 29: PRCs im Detail bei geringen Recall- und hohen Precision-Werten

5 Ergebnis

Im ersten Schritt wurden die Grundlagen des Deep Learnings und einiger wichtiger Modelle erarbeitet. Dies stellte kein großes Problem dar, da dieses Thema in den letzten Jahren sehr populär geworden ist. Daher existiert eine Vielzahl an Ressourcen, die dieses komplexe Thema sehr anschaulich erklären. Dieses Grundwissen wurde anschließend verwendet, um räumlich-zeitliche Vorhersagen zu definieren und geeignete Modelle zu finden. Hierbei ist anzumerken, dass einige elementare Unterschiede von verschiedenen Anwendungsfällen räumlich-zeitlicher Vorhersagen herausgearbeitet wurden, anhand derer die Anwendungsfälle Eingeordnet werden können. Dieses Vorgehen konnte nicht in existierender Literatur gefunden werden, daher ist davon auszugehen, dass eine Einordnung anhand der Beschaffenheit der Datensätze in dieser Arbeit erstmals vorgenommen wurde. Mit ST-ResNet und ConvLSTM konnten zwei mögliche Modelle für räumlich-zeitliche Vorhersagen gefunden werden. Aufgrund von praktischen Vorteilen wurde ConvLSTM als das geeignetere Modell ausgewählt.

Durch die Analyse des Datensatzes konnten einige Inkonsistenzen festgestellt und erfolgreich behoben werden. Um eine geeignete Implementierung der Rasterisierung zu finden wurde einiges an Zeit benötigt, jedoch ist die letztendlich gefundene Lösung sehr flexibel. Für die Implementierung eines NNs für den vorliegenden Anwendungsfall konnte die Architektur aus [10] übernommen werden, inkl. der Idee für eine gewichtete Verlustfunktion. Dadurch konnte hier etwas Zeit eingespart werden. Nach dem Training des Modells wurde erörtert, wie die rohen Ausgabewerte des Modells nachbearbeitet werden müssen. Dafür wurde die Auswirkung der Perzentilgrenze untersucht. Mit der Einteilung in vier Gefahrenstufen wurde eine Möglichkeit gefunden, möglichst einfach verständliche Vorhersagen zu erzeugen.

Zuletzt wurde die Performance des Modells evaluiert. Dieser Schritt ist sehr wichtig, da er die Sinnhaftigkeit einer Anwendung von Deep Learning auf die vorliegende Problemstellung bestätigen oder wiederlegen soll. Durch zufälliges Vertauschen der Zielframes konnte bestätigt werden, dass die Vorhersagen tatsächlich von den konkreten Eingabeframes abhängen. Es wurde mit bestem Wissen und Gewissen dafür gesorgt, dass das Modell die wahren Zielframes zuvor noch nie gesehen hat. Daher lässt sich schlussfolgen, dass sich die anfängliche Vermutung bestätigt, dass es gewisse Muster in den Daten gibt, anhand derer sich die Standorte von mobilen Radarkontrollen vorhersagen lassen. Es besteht also teilweise ein direkter Zusammenhang zwischen den Standorten der letzten 16 Tage und denen des Folgetages. Des weiteren lässt sich aus diesem Ergebnis ableiten, dass Deep Learning und insbesondere die ConvLSTM-Architektur auf die vorliegende Problemstellung anwendbar ist, was nicht von Anfang an klar war. Mit dem gewählten Vorgehen ist die Vorhersage der Gefahr für mobile Radarkontrollen somit grundsätzlich möglich. Allerdings ist die Vorhersage fehlerbehaftet, da es sowohl einige Falsch-Negative als auch Falsch-Positive gibt. Dies war jedoch von Anfang an zu erwarten, da schon der Datensatz nicht perfekt ist. Außerdem kann nicht davon ausgegangen werden, dass ein komplett kausaler Zusammenhang zwischen den Standorten der letzten 16 Tage und denen des Folgetages besteht. Daher werden die Vorhersagen immer fehlerbehaftet sein.

6 Ausblick

Es gibt einige Ansätze, wie die Genauigkeit der Vorhersagen auch unter den oben erwähnten Umständen weiter erhöht werden könnte. Zunächst kann man bei einer räumlichen Betrachtung des Datensatzes schnell erkennen, dass es für die Standorte von mobilen Radarkontrollen Hotspots gibt, die über die Zeit konstant bleiben, wie z. B. an bestimmten Kreuzungen. Daher würde es zum vorliegenden, räumlich kontinuierlichen Datensatz besser passen, diesen als Graph zu betrachten, bei dem Hotspots Knoten und das Straßennetz Kanten entsprechen. Für graphbasierte Daten gibt es einige vielversprechende Modellarchitekturen, über die in [20] ein Überblick gegeben wird. Beispiele hierfür sind die Modelle STGCN oder DCRNN. Auch der rasterbasierte Ansatz kann noch weiter verfeinert werden. In [18], werden beispielsweise im Rahmen der Vorhersage von Verkehrsunfällen einige vielversprechende Vorgehensweisen vorgestellt, die über das Vorgehen der vorliegenden Arbeit deutlich hinausgehen. Zunächst versuchen die Autoren, Unterschiede zwischen Stadt und Land möglichst gut zu berücksichtigen. Um ihnen gerecht zu werden, verwenden sie einen Sliding-Window Ansatz, bei dem ein 32x32 Pixel großes Fenster über verschiedene Bereiche der Eingabe gelegt wird. Somit überwiegt bei jeder einzelnen Position nicht immer der Verlustwert der städtischen Gebiete, was sonst der Fall wäre. Der zweite große Unterschied zur in der vorliegenden Arbeit verwendeten Implementierung ist die Einbindung von deutlich mehr Merkmalen. Zusätzlich zu den Verkehrsunfällen der letzten Tage werden noch ca. 30 weitere Merkmalskarten pro Tag verwendet. Diese enthalten u. A. ob sich in einer Zelle überhaupt eine Straße befindet, die Anzahl der Verkehrsknoten, das Verkehrsaufkommen, Wetterdaten, ein RGB-Satellitenbild und den Zustand der Straßen. Solche zusätzlichen Merkmalskarten könnten auch für die vorliegende Problemstellung verwendet werden. Außerdem wird die Ausgabe des Modells am Ende mit einer Matrix multipliziert, die alle Rasterzellen auf null setzt, in denen sich keine Straße befindet.

Bei Betrachtung aller dieser Optimierungen wird klar, dass mit der vorliegenden Arbeit nur eine Grundlage geschaffen wurde. Daher ist es bemerkenswert, dass schon die vorliegende Implementierung einigermaßen gute Vorhersagen erzeugen kann. Es bleibt interessant, wie sehr die Vorhersagen durch zusätzliche Optimierungen noch verbessert werden könnten.

Literatur

[1] F. Chollet, *Deep Learning mit Python und Keras*. mitp, 2018.

[2] A. Amini and A. Soleimany, "Lecture 1: Intro to deep learning," in *6.S191 Introduction to Deep Learning*, Cambridge MA, January IAP 2020, Massachusetts Institute of Technology: MIT OpenCourseWare. [Online]. Available: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-s191-introduction-to-deep-learning-january-iap-2020/

[3] B. Wilamowski, "Neural network architectures and learning algorithms," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 56–63, 2009.

[4] A. Amini and A. Soleimany, "Lecture 2: Recurrent neural networks," in *6.S191 Introduction to Deep Learning*, Cambridge MA, January IAP 2020, Massachusetts Institute of Technology: MIT OpenCourseWare. [Online]. Available: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-s191-introduction-to-deep-learning-january-iap-2020/

[5] A. Amini and A. Soleimany, "Lecture 3: Convolutional neural networks," in *6.S191 Introduction to Deep Learning*, Cambridge MA, January IAP 2020, Massachusetts Institute of Technology: MIT OpenCourseWare. [Online]. Available: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-s191-introduction-to-deep-learning-january-iap-2020/

[6] J. Zhang, Y. Zheng, and D. Qi, "Deep spatio-temporal residual networks for citywide crowd flows prediction."

[7] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-k. Wong, and W.-c. Woo, "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting."

[8] infas Institut für angewandte Sozialwissenschaft, "Mobilität in Deutschland - MiD. Ergebnisbericht," Februar 2019, Studie von infas, DLR, IVT und infas 360 im Auftrag des Bundesministers für Verkehr und digitale Infrastruktur (FE-Nr. 70.904/15). Bonn, Berlin. [Online]. Available: http://www.mobilitaet-in-deutschland.de/pdf/MiD2017_Ergebnisbericht.pdf [Accessed: 26-Mar-2022].

[9] "File:R-tree.svg." [Online]. Available: https://commons.wikimedia.org/wiki/File:R-tree.svg [Accessed: 03-Apr-2022].

[10] N. Holm and E. Plynning, "Spatio-temporal prediction of residential burglaries using convolutional LSTM neural networks," 2018, Dissertation. [Online]. Available: https://kth.diva-portal.org/smash/get/diva2:1215575/FULLTEXT01.pdf

[11] "Classification on imbalanced data." [Online]. Available: https://www.tensorflow.org/tutorials/structured_data/imbalanced_data [Accessed: 06-Apr-2022].

[12] Eifrig Media GmbH (Hrsg.), "Über uns - Blitzer.de," 2021. [Online]. Available: https://www.blitzer.de/ueber-uns/ [Accessed: 25-Feb-2022].

[13] J. Alzubi, A. Nayyar, and A. Kumar, "Machine learning from theory to algorithms: An overview," *Journal of Physics: Conference Series*, vol. 1142, p. 012012, 2018.

[14] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *International*

Journal of Engineering Applied Sciences and Technology, vol. 04, no. 12, pp. 310–316, 2020.

[15] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning." [Online]. Available: http://arxiv.org/pdf/1811.03378v1

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 12/22/2014, published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. [Online]. Available: http://arxiv.org/pdf/1412.6980v9

[17] B. Wang, D. Zhang, D. Zhang, P. J. Brantingham, and A. L. Bertozzi, "Deep learning for real time crime forecasting."

[18] Z. Yuan, X. Zhou, and T. Yang, "Hetero-ConvLSTM," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Y. Guo and F. Farooq, Eds. New York, NY, USA: ACM, 07192018, pp. 984–992.

[19] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting."

[20] R. Jiang, D. Yin, Z. Wang, Y. Wang, J. Deng, H. Liu, Z. Cai, J. Deng, X. Song, and R. Shibasaki, "Dl-traff: Survey and benchmark of deep learning models for urban traffic prediction," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, G. Demartini, G. Zuccon, J. S. Culpepper, Z. Huang, and H. Tong, Eds. New York, NY, USA: ACM, 10262021, pp. 4515–4525.

[21] A. Graves, "Generating sequences with recurrent neural networks." [Online]. Available: http://arxiv.org/pdf/1308.0850v5

[22] "tf.keras.layers.ConvLSTM2D." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/ConvLSTM2D [Accessed: 07-Mar-2022].

[23] "GitHub - DL-Traff-Grid/ST_ResNet.py at master." [Online]. Available: https://github.com/deepkashiwa20/DL-Traff-Grid/blob/master/workTaxiNYC/predflowio/ST_ResNet.py [Accessed: 07-Mar-2022].

[24] "GitHub - snehasinghania/STResNet: A TensorFlow Implementation of Deep Spatio-Temporal Residual Networks (ST-ResNet)." [Online]. Available: https://github.com/snehasinghania/STResNet [Accessed: 07-Mar-2022].

[25] Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras, "Spatial indexing techniques," in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, pp. 2702–2707.

[26] "Introduction to PostGIS: Spatial Indexing." [Online]. Available: https://postgis.net/workshops/postgis-intro/indexing.html [Accessed: 03-Apr-2022].

[27] A. Guttman, "R-trees," *ACM SIGMOD Record*, vol. 14, no. 2, pp. 47–57, jun 1984. [Online]. Available: https://doi.org/10.1145/971697.602266

[28] "Key Features of SQLAlchemy." [Online]. Available: https://www.sqlalchemy.org/features.html [Accessed: 03-Apr-2022].

[29] "Create grid." [Online]. Available: https://docs.qgis.org/3.22/en/docs/user_manual/processing_algs/qgis/vectorcreation.html#create-grid [Accessed: 04-Apr-2022].

[30] "Join attributes by location (summary)." [Online]. Available: https://docs.qgis.org/3.22/en/docs/user_manual/processing_algs/qgis/vectorgeneral.html#

join-attributes-by-location-summary [Accessed: 04-Apr-2022].

[31] Open Geospatial Consortium, "GeoPackage Encoding Standard." [Online]. Available: https://www.ogc.org/standards/geopackage [Accessed: 04-Apr-2022].

[32] G. Farkas, *Practical GIS: Learn novice to advanced topics such as QGIS, spatial data analysis, and more*. Birmingham, UK: Packt Publishing, 2017.

[33] Open Geospatial Consortium, "OGC GeoTIFF standard." [Online]. Available: https://www.opengis.net/doc/IS/GeoTIFF/1.1 [Accessed: 04-Apr-2022].

[34] Joshi, Amogh, "Next-Frame Video Prediction with Convolutional LSTMs." [Online]. Available: https://keras.io/examples/vision/conv_lstm/ [Accessed: 05-Apr-2022].

[35] "EarlyStopping." [Online]. Available: https://keras.io/api/callbacks/early_stopping/ [Accessed: 07-Apr-2022].

[36] "ReduceLROnPlateau." [Online]. Available: https://keras.io/api/callbacks/reduce_lr_on_plateau/ [Accessed: 07-Apr-2022].

Anhang

A Docker-Compose Datei zum Erstellen der Datenbanken

```yaml
version: "3.7"
services:
    mariadb:
        image: mariadb:latest
        volumes:
            - ./database:/var/lib/mysql
        environment:
            - MYSQL_ROOT_PASSWORD=speedcam
            - MYSQL_DATABASE=speedcam_mining
        ports:
            - 3306:3306
    phpmyadmin:
        image: phpmyadmin/phpmyadmin:latest
        ports:
            - 8000:80
        environment:
            - PMA_HOST=mariadb
            - UPLOAD_LIMIT=1G
        depends_on:
            - mariadb
    geodb:
        image: kartoza/postgis:latest
        volumes:
            - ./geodb-data:/var/lib/postgresql
        environment:
            - POSTGRES_DB=speedcam_archive
            - POSTGRES_USER=speedcam
            - POSTGRES_PASS=speedcam
        ports:
            - 5432:5432
```

Codeausschnitt 7: Docker-Compose Datei mit MariaDB, PhpMyAdmin und PostGIS

B Mobile Radarkontrollen während eines Blitzermarathons

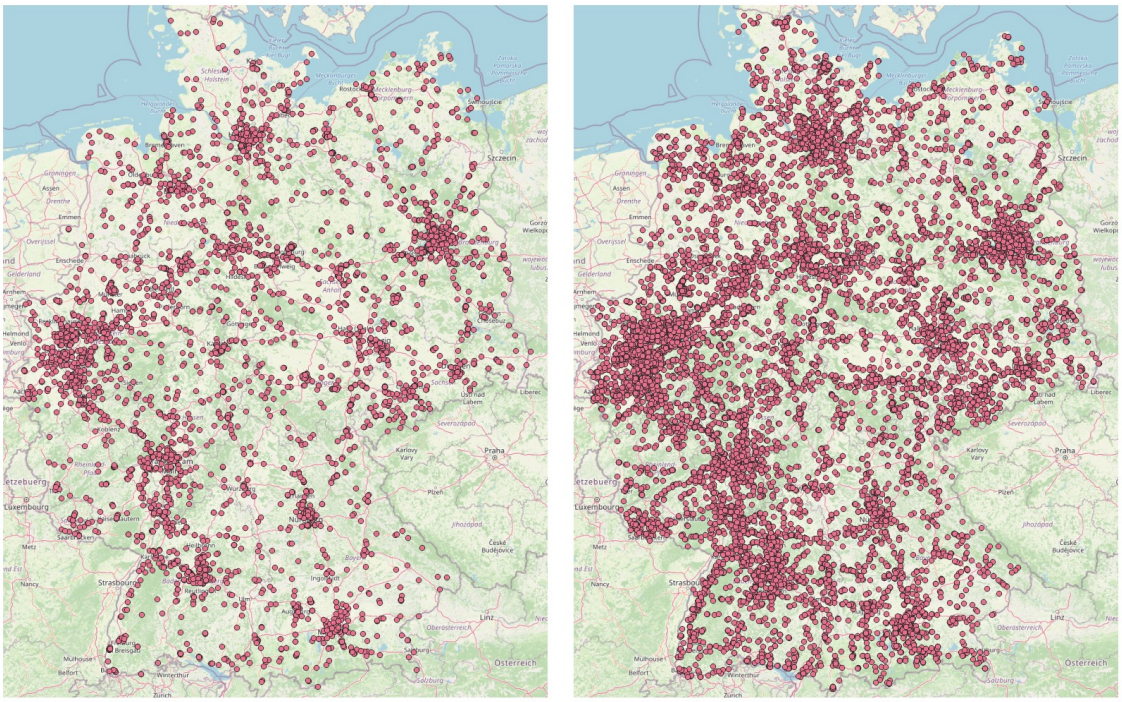

Abbildung 30: Mobile Radarkontrollen in Deutschland während des Blitzermarathons am 18.09.2014 im Vergleich zum Vortag. Es ist erkennbar, dass am Tag des Blitzermarathons deutlich mehr mobile Radarkontrollen vorhanden waren als am Vortag.

C Implementierung des Transfers von MariaDB nach PostGIS

```python
from sqlalchemy.orm import declarative_base
from sqlalchemy import Column, String, BigInteger, DateTime, Numeric

# See https://docs.sqlalchemy.org/en/13/changelog/migration_12.html#change-4102
class LiberalBoolean(TypeDecorator):
    impl = Boolean
    def process_bind_param(self, value, dialect):
        if value is not None:
            value = bool(int(value))
        return value

Base = declarative_base()

class ArchivedShownSpeedcam(Base):
    __database__ = 'speedcam_archive'
    __tablename__ = 'output_180d_dump'

    id = Column(BigInteger, primary_key=True, nullable=False, autoincrement=True)
    autobahn = Column(LiberalBoolean, nullable=False, default=False)
    cvmax = Column(String(3), nullable=False, default='')
    vmax = Column(String(3), nullable=False, default='')
    lastuse_date = Column(DateTime, nullable=False, default='0000-00-00 00:00:00')
    laenge = Column(Numeric(9, 6), nullable=False, default=0.0)
    breite = Column(Numeric(9, 6), nullable=False, default=0.0)
    direction = Column(String(3), nullable=False, default='')
    created_date = Column(DateTime, nullable=False, default='0000-00-00 00:00:00')
```

Codeausschnitt 8: SQLAlchemy-Modell einer Radarkontrolle in MariaDB

```python
import datetime
from sqlalchemy import Column, BigInteger, DateTime, Numeric
from sqlalchemy.orm import declarative_base
from sqlalchemy.ext.hybrid import hybrid_method
from geoalchemy2 import Geometry
from common.archived_shown_speedcam import ArchivedShownSpeedcam

Base = declarative_base()
SRID = 4326  # This is the coordinate system that GPS uses

class PostgisSpeedcam(Base):
    __tablename__ = 'speedcam'

    id = Column(BigInteger, primary_key=True)
    position = Column(Geometry('POINT', srid=SRID))
    created_date = Column(DateTime, nullable=False, default='0000-00-00 00:00:00')
    lastuse_date = Column(DateTime, nullable=False, default='0000-00-00 00:00:00')
    duration_hours = Column(Numeric(4, 2), nullable=False, default=0.0)

    def __init__(self, sc: ArchivedShownSpeedcam):
        self.id = sc.id
        # Sometimes laenge and breite are swapped for some reason.
        # Laenge should be between 5.9 and 15.0 for Germany.
        # So to give some buffer, we swap them if laenge is too large.
        if sc.laenge < 20:
            self.position = sc.position = f'SRID={SRID};POINT({sc.laenge} {sc.breite})'
        else:
            self.position = sc.position = f'SRID={SRID};POINT({sc.breite} {sc.laenge})'

        self.created_date = sc.created_date
        self.lastuse_date = sc.lastuse_date
        self.duration_hours = (sc.lastuse_date - sc.created_date).total_seconds() / 3600

        # If duration is longer than 12 hours, clip it to 12 hours from created_date
        if self.duration_hours > 12:
            self.lastuse_date = sc.created_date + datetime.timedelta(hours=12)
            self.duration_hours = 12
```

Codeausschnitt 9: SQLAlchemy-Modell einer Radarkontrolle in PostGIS

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from common.archived_shown_speedcam import ArchivedShownSpeedcam

mariadb_engine = create_engine(
    'mysql://root:speedcam@localhost/speedcam_archive')
MariaDBSession = sessionmaker(bind=mariadb_engine)
postgis_engine = create_engine(
    'postgresql://speedcam:speedcam@localhost/speedcam_archive')
PostGisSession = sessionmaker(bind=postgis_engine)
mariadb_session = MariaDBSession()
postgis_session = PostGisSession()

# Drop all leftover data and recreate table
Base.metadata.drop_all(postgis_session.bind)
Base.metadata.create_all(postgis_session.bind)

speedcam_count = mariadb_session.query(ArchivedShownSpeedcam).count()

# Transfer all speedcams in chunks of 10000
for i in range(0, speedcam_count, 10000):
    speedcams = mariadb_session.query(ArchivedShownSpeedcam) \
        .limit(10000) \
        .offset(i) \
        .all()

    postgis_session.add_all([PostgisSpeedcam(sc) for sc in speedcams])
    postgis_session.commit()
    print(f'Transferred {i} out of {speedcam_count} speedcams')

mariadb_session.close()
postgis_session.close()
```

Codeausschnitt 10: Transfer der Radarkontrollen von MariaDB nach PostGIS

D Implementierung der Rasterisierung mit QGIS-Algorithmen

```
import processing
from qgis.core import *

def generate_heatmap(gpkg_path, date, extent, grid_cell_size=4000):
    db_uri = 'postgres://dbname=\'speedcam_archive\' host=localhost ' + \
        'port=5432 user=\'speedcam\' password=\'speedcam\' sslmode=allow ' + \
        'key=\'id\' srid=4326 type=Point checkPrimaryKeyUnicity=\'1\' ' + \
        'table="public"."speedcam" (position) sql=created_date::date = '

    if os.path.isfile(gpkg_path):
        heatmap = QgsVectorLayer(gpkg_path, '', 'ogr')
        return heatmap

    date_str = date.strftime("'%Y-%m-%d'")
    print(f"Processing date {date_str}", flush=True)

    grid_layer = processing.run('qgis:creategrid', {
        'EXTENT' : extent,
        'HSPACING' : grid_cell_size, 'VSPACING' : grid_cell_size,
        'CRS' : QgsCoordinateReferenceSystem('EPSG:3857'),
        'HOVERLAY' : 0, 'VOVERLAY' : 0,
        'OUTPUT' : 'TEMPORARY_OUTPUT', 'TYPE' : 2})['OUTPUT']

    heatmap = processing.run('qgis:joinbylocationsummary', {
        'INPUT' : grid_layer,
        'JOIN' : db_uri + date_str,
        'JOIN_FIELDS' : ['duration_hours'],
        'PREDICATE' : [0],
        'SUMMARIES' : [5],
        'OUTPUT' : 'TEMPORARY_OUTPUT',
        'DISCARD_NONMATCHING' : False})["OUTPUT"]

    store_layer(heatmap, gpkg_path)
    return heatmap
```

Codeausschnitt 11: Rasterisierung der Datenpunkte an einem bestimmten Datum

```python
import subprocess
from qgis.core import *
from processing.core.Processing import Processing

def gpkg_to_tif(gpkg_path, tif_path, heatmap, grid_cell_size=1000):
    extent = heatmap.extent()
    extent = [round(extent.xMinimum()), round(extent.xMaximum()),
        round(extent.yMinimum()), round(extent.yMaximum())]
    raster_width = round((extent[1] - extent[0])/grid_cell_size)
    raster_height = round((extent[3] - extent[2])/grid_cell_size)

    subprocess.call(['gdal_rasterize', '-a', 'duration_hours_sum', '-ts',
        str(raster_width), str(raster_height), '-a_nodata', '0.0', '-te',
        str(extent[0]), str(extent[2]), str(extent[1]), str(extent[3]),
        '-ot', 'Float16', '-of', 'GTiff', gpkg_path, tif_path],
        stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

def process_dates(dates):
    QgsApplication.setPrefixPath('/usr', True)
    qgs = QgsApplication([], False)
    qgs.initQgis()
    Processing.initialize()

    grid_cell_size = 4000
    # 200x200 km centered in Baden-Württemberg
    extent = '927792.5265,1127792.5265,6113386.3009,6313386.3009 [EPSG:3857]'
    heatmaps_path = os.path.join(os.path.dirname(__file__), 'heatmaps')

    for date in dates:
        base_path = os.path.join(heatmaps_path,
            'heatmap_' + date.strftime('%Y-%m-%d'))
        gpkg_path = base_path + '.gpkg'
        tif_path = base_path + '.tif'

        heatmap = generate_heatmap(gpkg_path, date, extent, grid_cell_size)
        gpkg_to_tif(gpkg_path, tif_path, heatmap, grid_cell_size=grid_cell_size)
        os.remove(gpkg_path)
```

Codeausschnitt 12: Umwandlung einer GPKG-Datei in eine GeoTIFF-Datei mit *gdal_rasterize* und high-level Ablauf der Rasterisierung

```python
import datetime
from joblib import Parallel, delayed

def main():
    start_date = datetime.date(2014, 7, 17)
    end_date = datetime.date(2021, 10, 25)
    dates = [start_date + datetime.timedelta(days=x)
        for x in range(0, (end_date - start_date).days)]

    # Make batches of 50 dates and include the rest in the last batch.
    batch_size = 50
    batches = [dates[i:i + batch_size]
        for i in range(0, len(dates), batch_size)]

    rest_of_dates = [date for date in dates
        if date not in np.array(batches, dtype=object).flatten()]
    batches.append(rest_of_dates)

    # Use parallel processes to speed up the process
    Parallel(n_jobs=24)(delayed(process_dates)(batch)
        for batch in batches)
```

Codeausschnitt 13: Parallele Rasterisierung des Datensatzes

E Definition des ConvLSTM-Modells in TensorFlow

```
def get_model(input, weights):
    model = keras.models.Sequential([
        input,
        layers.ConvLSTM2D(
            filters=32, kernel_size=(3, 3),
            padding="same", return_sequences=True, activation="relu",
        ),
        layers.MaxPooling3D(pool_size=(2, 1, 1), strides=(2, 1, 1)),
        layers.ConvLSTM2D(
            filters=32, kernel_size=(3, 3),
            padding="same", return_sequences=True, activation="relu",
        ),
        layers.MaxPooling3D(pool_size=(2, 1, 1), strides=(2, 1, 1)),
        layers.ConvLSTM2D(
            filters=32, kernel_size=(3, 3),
            padding="same", return_sequences=True, activation="relu",
        ),
        layers.MaxPooling3D(pool_size=(2, 1, 1), strides=(2, 1, 1)),
        layers.ConvLSTM2D(
            filters=32, kernel_size=(3, 3),
            padding="same", return_sequences=True, activation="relu",
        ),
        layers.MaxPooling3D(pool_size=(2, 1, 1), strides=(2, 1, 1)),
        layers.Conv2D(
            filters=1, kernel_size=(3, 3), activation="sigmoid", padding="same"
        )
    ])

    loss_fn = weighted_binary_crossentropy(weights)

    model.compile(
        loss=loss_fn, optimizer=keras.optimizers.Adam(),
        metrics=METRICS
    )

    return model, loss_fn
```

Codeausschnitt 14: Definition des ConvLSTM-Modells in TensorFlow