TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Bauingenieurwesen Institut für Bauinformatik

PROJEKTARBEIT

BIM-basierte Schadensidentifikation mittels Machine Learning

(BIM-based damage identification using Machine Learning)

eingereicht von
cand. Ing. Pablo Alonso Baeza Guarda
geb. am 07.09.1995 in Osorno, Chile
Matrikel-Nummer: 4113307

Betreuer/in:

- Prof. Dr.-Ing. habil. Karsten Menzel / Prof. Dr.-Ing. Raimar J. Scherer
- Dipl.-Ing. Al-Hakam Hamdan

Dresden, den 24. März 2022

(Aufgabenstellung (1x im ORIGINAL + 1 oder 2 gebundene Ausgaben mit Kopien)

SELBSTSTÄNDIGKEITSERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich reiche sie erstmals als Prüfungsleistung ein. Mir ist bekannt, dass ein Betrugsversuch mit der Note „nicht ausreichend“ (5,0) geahndet wird und im Wiederholungsfall zum Ausschluss von der Erbringung weiterer Prüfungsleistungen führen kann.

Name(n): Baeza Guarda

Vorname(n): Pablo Alonso

Matrikelnummer: 4113307

Dresden, den 24. März. 2022

Unterschrift cand. Ing. Pablo A. Baeza Guarda

I INHALTSVERZEICHNIS

II ABBILDUNGSVERZEICHNIS

III TABELLENVERZEICHNIS

IV QUELLTEXTVERZEICHNIS

1 EINLEITUNG

Die Identifikation von Schäden sowie die Klassifikation dieser basierend auf ausgewählten Kriterien ist ein zentraler Bestandteil von wiederkehrenden Bauwerksprüfungen wie es z.B. bei Ingenieurbauwerken wie Brücken, Stützwänden oder Tunneln nach DIN 1076 vorgeschrieben ist. Zur Aufnahme der Schäden gibt es digitale Datenumgebungen beispielsweise SIB-Bauwerke, die allerdings manuell von Inspekteuren basierend auf zuvor angefertigten Berichten bedient werden müssen. Die manuelle Eingabe von Schadensinformationen ist mitunter sehr zeitaufwendig, insbesondere dann, wenn Schadensbilder mit bereits früher aufgenommenen Schäden abgeglichen und in Relation zueinander gesetzt werden müssen. Zwar existieren automatisierte Aufnahmeverfahren mit Drohnen oder Tunnelscannern, diese können allerdings nur Anomalien detektieren, die daraufhin von einem Experten analysiert und klassifiziert werden müssen, um einen tatsächlichen Schaden zu identifizieren.

Oberflächenschäden an Gebäuden sind am einfachsten zu erkennen. Der Prozess der Anerkennung und Dokumentation ist jedoch zeitaufwändig und trivial. Durch den Einsatz von Computer Vision und Algorithmen des maschinellen Lernens können beide Prozesse automatisiert werden, wodurch sich ihre Dauer und Kosten verringern.

Prüfungen nach DIN 1076:

- Hauptprüfung
- Einfache Prüfung
- Prüfung auf besonderen Einlass (Sonderprüfung)
- Prüfung nach besonderen Vorschriften
- Besichtigung
- Laufende Beobachtung

Tabelle 2: Zyklen der Bauwerksprüfung und Bauwerksüberwachung nach DIN 1076

Prüfungsart [1]	Prüfung vor Abnahme der Leistung	Anzahl der Prüfungen bis zur Verjährung der Mängelansprüche					Anzahl der Prüfungen bis zum Ende der Nutzungsdauer						
						Prüfung vor Ablauf der Verjährungsfrist für Mängelansprüche							
	Baujahr	1	2	3	4	5	6	7	8	9	10	11	weiterhin
LB [2]		2x	2x	2x	2x	2x	2x	2x	2x	2x	2x	2x	2x
B		1x	1x		1x		1x	1x		1x	1x		1x [3]
E				●					●				Alle 6 Jahre
H [4]	●					●						●	Alle 6 Jahre
S		Auf Anordnung oder nach größeren Unwettern, Hochwasser, Verkehrsunfällen oder sonstigen den Bestand der Bauwerke beeinflussenden Ereignissen											

[1] LB = Laufende Beobachtung, B = Besichtigung, E = Einfache Prüfung, H = Hauptprüfung, S = Sonderprüfung
[2] Beobachtung laufend im Rahmen der Streckenkontrolle und zusätzlich 2x/Jahr Beobachtung aller Bauteile von der Verkehrsebene/Geländeebene aus
[3] außer in den Jahren, in denen eine Haupt- oder einfache Prüfung durchgeführt wird
[4] für Holzbrücken gelten abweichende Regelungen gemäß RI-EBW-PRÜF [3]

Abb. 1 Zyklen der Bauwerksprüfung
Quelle: (Bundesministerium für Verkehr, 2013)

Ein Auszug aus der Bauwerksprüfung nach DIN 1076 (2013) besagt:

„Die wesentlichen Untersuchungsleistungen bei der Bauwerksprüfung sind in der DIN 1076 baustoff- und bauteilbezogen aufgeführt. Generell sind folgende Leistungen auszuführen“:

- Einrichten der Arbeitsstelle (z.B. Aufbau der Verkehrssicherung, Inbetriebnahme der Zugangstechnik und Beleuchtung, Vorkehrungen zur Gewährleistung des Arbeitsschutzes)
- Einweisung, Koordinierung und Kontrolle aller an der Bauwerksprüfung Beteiligten
- Durchführen der Prüfung
- Protokollieren der festgestellten Mängel und Schäden, Anfertigen von Skizzen und Fotos

Um Rückschlüsse auf das Sicherheitsniveau hinsichtlich Standsicherheit, Verkehrssicherheit und Dauerhaftigkeit ziehen zu können, schreibt die DIN 1076 für die Hauptprüfung

die handnahe Untersuchung aller Bauwerksteile vor (z.B. durch Sichtprüfung und Abklopfen). Für die Durchführung der Prüfung ist ein geeignetes Prüfverfahren auszuwählen und die dafür erforderlichen Zusatzgeräte vorzuhalten und einzusetzen (z.B. Rückprallhammer, Schichtdickenmessgerät).

Kann durch den Bauwerksprüfer die Schadensursache oder das Schadensausmaß nicht ausreichend ermittelt werden, so ist in der Regel eine weitere Untersuchung nach dem Leitfaden „Objektbezogene Schadensanalyse (OSA)“ durchzuführen.

1.1. PROBLEMSTELLUNG DER ARBEIT

Im Rahmen dieser Diplomarbeit soll untersucht werden inwiefern unter Verwendung von Methoden des Machine Learning bereits bestehende Schäden aus aufgenommenen Schadensbildern automatisiert hergeleitet werden können. Außerdem soll zur Unterstützung einer automatisierten Aufnahme untersucht werden, inwiefern mittels Machine Learning ein Schaden hinsichtlich vordefinierter Kriterien (z.B. netzartiges Muster oder parallele Anordnung von Rissen) klassifiziert werden kann. Hierfür soll ein Lösungskonzept entwickelt werden, dass in einen BIM-unterstützten Inspektionsprozess implementiert werden soll. Das entwickelte Verfahren soll mit Hilfe eines Software-Prototypen an Testszenarien validiert und die Ergebnisse evaluiert werden.

1.2. ZIELE UND ABGRENZUNG DER ARBEIT

Während der Ausarbeitung sollen folgende Punkte bearbeitet werden:

1. Überblick und Bewertung des Standes der Forschung und Technik im Bereich des Machine Learning mit Fokus auf mögliche Methoden zur Erkennung von Schäden.
2. Konzeptionierung eines BIM-unterstützten Workflows zur Implementierung eines Systems für die Erkennung und Identifikation von Schäden während einer Bauwerksinspektion.
3. Entwicklung eines Software-Prototypen zur Erfassung bereits bekannter Schadensbilder und einer ersten Vor-Klassifikation dieser, nach ausgewählten Merkmalen mittels Machine Learning.
4. Validierung des Software-Prototypen an ausgewählten Testszenarien.
5. Evaluation der Methodik, des Prototyps und der Ergebnisse aus den Testszenarien im Hinblick auf weitere Anwendungsmöglichkeiten.

1.3. AUFBAU DER ARBEIT

Diese Arbeit wird sich in zwei Teile gliedern: Erstens einem theoretischen und zweitens einem praktischen Teil. Im theoretischen Teil werden die grundlegenden Konzepte erläutert, die für die spätere Umsetzung im praktischen Teil notwendig sind. Der theoretische Teil der Arbeit konzentriert sich auf die Grundlagen des Machine Learning, die Methode des überwachten Lernens und die Funktionsweise von neuronalen Netzen. Abschließend werden einige Forschungsarbeiten erwähnt, in denen maschinelles Lernen für die automatische Erkennung verschiedener Arten von Schäden auf Bauwerken eingesetzt wurde.

Anschließend wird ein Workflow für den Prozess der Schadenserkennung während der Bauinspektion vorgeschlagen, der in drei Phasen unterteilt ist: Die Erfassung von Schadensbildern während der Inspektion, die Identifizierung solcher Schäden mit Hilfe eines maschinellen Lernmodells und die anschließende Aufzeichnung der Schadensinformationen in Tabellenform für die spätere Eingabe in eine Schadensdatenbank.

Schließlich wird ein praktischer Test der Funktionsweise eines neuronalen Netzes anhand eines Prototyps zur Erkennung von Rissen in Beton durchgeführt. Darüber hinaus wird ein Prototyp für die Erfassung der erkannten Schäden durch ein neuronales Netzmodell vorgeschlagen, das auf den Grundlagen der Photogrammetrie und der projektiven Geometrie basiert, um die genauen Koordinaten des Schadens innerhalb eines BIM-Modells zu lokalisieren.

2 STATE OF THE ART

2.1. KÜNSTLICHER INTELLIGENZ

„Künstlicher Intelligenz (KI)“ ist ein Teilgebiet der Informatik, die das Imitieren der menschlichen Intelligenz untersucht. Man kann zwischen Starke KI und Schwache KI unterscheiden:

Eine Starke Künstliche Intelligenz bezeichnet die Fähigkeit eines Computers, neue Information zu lernen, mit dem vorher gelernten Wissen zu vergleichen und zu analysieren. Von einer starken KI sind das Assoziieren von multidisziplinärem Wissen und das Herausfinden von kreativen Ideen zu erwarten. Allerdings existiert solch eine Maschine in der Gegenwart (noch) nicht.

Was derzeit doch existiert, sind Maschinen, die aus Daten lernen können und mit diesem Wissen dann wie ein Mensch bestimmte Aufgaben erfüllen können. Jedoch sind sie auf keinen Fall denkfähig; die Maschine kann dank ihres Trainings ein Ergebnis ausgeben, aber sie kann dieses Ergebnis nicht selbstständig interpretieren und daraus neues Wissen erzeugen. Dieses Konzept wird als Schwache Künstliche Intelligenz bezeichnet und in vielfältigen Disziplinen benutzt (Naturwissenschaften, Ingenieurwissenschaften, Ökonomie, Medizin, etc.).

2.2. KONZEPT DES MACHINE LEARNING

Der traditionelle Ansatz der Programmierung besagt, dass ein Computer, erstens Eingabedaten und zweitens einen Quellcode mit Anweisungen benötigt, um ein bestimmtes Ergebnis zu erzielen. Wie gut das Programm funktioniert, hängt davon ab, wie gut diese Anweisungen definiert sind. Dieser Ansatz ist jedoch nicht unproblematisch, wenn es kein klares Muster gibt, wie diese Anweisungen zu handhaben sind. Ein Beispiel hierfür ist die Spam-Erkennung: Es ist sehr schwierig, einen Quellcode zu schreiben, der jede Spam-Nachricht erkennt, da sich Spam selbst im Laufe der Zeit verändert, was bedeutet, dass der Code ständig aktualisiert werden muss. Um dieses Problem zu lösen, wurde das Konzept des maschinellen Lernens entwickelt. Anstelle fester Anweisungen sammelt ein maschinelles Lernmodell eine Reihe von Daten und schließt daraus auf ein allgemeines Verhalten der Daten. Wenn also neue Daten eingegeben werden, kann der Computer eine Vorhersage über eine mögliche Antwort machen.

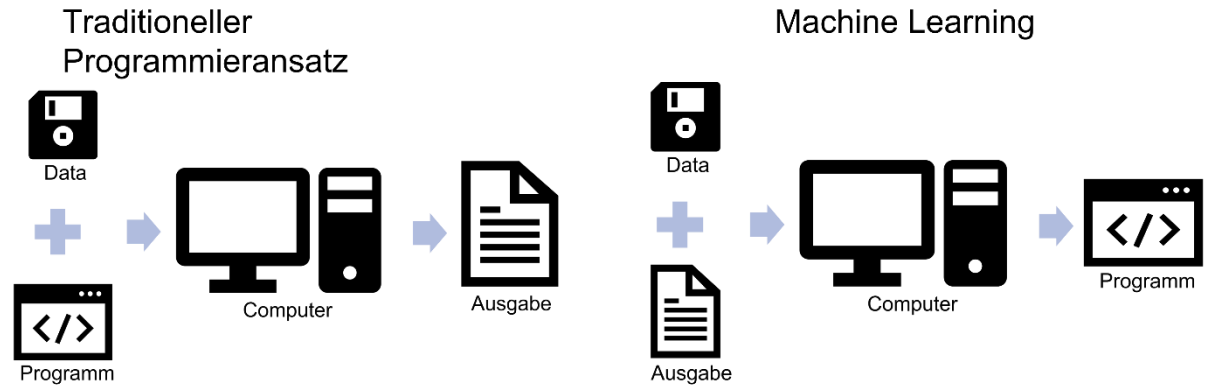


Abb. 2 Traditioneller Programmieransatz vs. Machine Learning

Ein sehr wichtiger zu beachtender Punkt ist die Art des Lernens, die bei der Verwendung von Machine Learning eingesetzt wird. Hier ist das maßgebliche Kriterium der Grad der menschlichen Beteiligung während des Lernvorgangs. Man kann zwei Kategorien definieren: *Unüberwachtes*- und *Überwachtes Lernen.*

Beim Überwachten Lernsystemen ist das Ergebnis zu einer bestimmten Aufgabe bekannt, jedoch ist eine allgemeine Regel für die Automatisierung solcher Aufgabe nicht offensichtlich. Es wird dann einen beschrifteten Datensatz für die Erstellung eines Vorhersagemodells verwendet. Je nach Art der *Labels* (deu. *Beschriftungen*) im Datensatz, werden überwachten Lernsystemen in *Klassifikationssystemen*, wobei die Labels der Daten eine Klasse entsprechen, und *Regressionssystemen*, in denen die Beschriftungen dichte Wertmengen sind, untergeordnet. Ein gutes Beispiel ist die Verwendung von *Deep Neural Networks* zur Erkennung von Rissen (Cha, Choi, & Büyüköztürk, Deep Learning-Based Crack Damage Detection Using Convolutional Neural Networks, 2017). Die Zuordnung eines Bildes als „Riss"- Positiv oder -Negativ stellt eine Klassifikationsaufgabe, während die Lokalisierung eines Risses innerhalb eines Bildes eine Regressionsaufgabe entspricht (Bestimmung von Koordinaten).

Es gibt andere Fälle, in denen die vorhandene Daten unbeschriftet sind und das Interesse nicht in der Erstellung eines Vorhersagemodells liegt, sondern in der Untersuchung eines auf den ersten Blick unbekannten Verhaltens der Daten. Für diese Problematik sind Unüberwachte Lernsysteme sehr vorteilhaft. Zwei bemerkenswerte Algorithmen innerhalb dieser Ansatz sind *Clustering*, das zur Gruppierung von unbeschrifteten Daten nach gemeinsamen Merkmalen verwendet wird, und *Dimensionality Reduction* (deu. *Dimensionalitätsreduktion*), die die Komprimierung von Datenattributen ermöglicht, um die Erkennung von Clustern in den Daten zu erleichtern. (Raschka & Mirjalili, 2019, S. 7-8)

Es gibt eine dritte Kategorie von Machine Learning, nämlich das *Bestärkendes Lernen*. Bei diesem Lernkonzept geht es darum, ein System zu entwickeln, das mit seiner Umgebung

interagiert. Dadurch erhält das System eine Belohnung von der Umwelt, die von der Qualität ihrer Leistung abhängt und entweder positiv oder negativ sein kann. Solche Systeme werden oft so programmiert, dass sie immer die positive Belohnung maximieren und dadurch ihre Leistungsfähigkeit verbessern. Bekannt sind Spiel-KIs wie AlphaGo (Go) und Stockfish (Schach), die durch die Nutzung von Bestärkendes Lernen ihre Fähigkeiten dermaßen verbessert haben, dass sie die besten menschlichen Spieler der Welt besiegen können.

Im Rahmen dieser Arbeit wird die Methode des überwachten Lernens angewandt.

2.2.1. KONZEPT VON ÜBERWACHTEN LERNEN

Die Daten, die zum Trainieren eines Lernalgorithmus verwendet werden, umfassen den *Trainingsdatensatz*. Jeder Instanz innerhalb des Datensatz entspricht einem *Trainingsbeispiel*, das eine oder mehrere *features* (deu. *Merkmale*) besitzen kann. Auf diese Weise kann ein Trainingsdatensatz in Matrixform ausgedrückt werden.

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & \cdots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,m} \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{2.1}$$

Jede Reihe der Matrix $\mathbf{X}$ entspricht einem Trainingsbeispiel des Datensatz und jede Spalte, einem bestimmten Merkmal der Trainingsdaten. Der Vektor $\mathbf{y}$ enthält die von jedem Trainingsbeispiel zugehörigen Labels y_1 bis y_n. Ein Trainingsdatensatz hat eine Anzahl n von Trainingsbeispielen, Labels und m unabhängigen Features.

Ein einfacher Lernalgorithmus kann wie in *(2.2)* dargestellt werden:

$$h: \mathbf{X} \to \mathbf{y}, \qquad h(\mathbf{X}) = \mathbf{X^T W} + \mathbf{b} \tag{2.2}$$

Hier ist $h(\mathbf{X})$ die Hypothese des Lernalgorithmus. Die Matrix $\mathbf{W}$ und der Vektor $\mathbf{b}$ sind die Parameter des Lernalgorithmus, die am Anfang des Trainings unbekannt sind. Es ist dann gewollt, eine bestimmte Parameterkonfiguration zu finden, so dass die Hypothese des Lernalgorithmus $h(\mathbf{X})$ erfüllt wird (Wenn $h(\mathbf{X}) = \mathbf{y}$, dann heißt das, dass die Vorhersagen des Lernalgorithmus zu den Labels gleich sind).

Um solche Konfigurationen zu finden, werden *Optimierungsalgorithmen* benutzt. Diese Algorithmen sind dafür zuständig, die Konfiguration der Parameter so zu ändern, damit der Lernalgorithmus eine möglichst hohe Effektivität beim Vorhersagen erreicht. Dies wird durch die Berechnung des globalen Minimums einer Verlustfunktion (wie zum Beispiel das *Mean Square Error*, die im *(2.3)* dargestellt wird).

$$MSE(\mathbf{X}) = \frac{1}{n}\sum_{i=1}^{n}(h(\mathbf{X}) - \mathbf{y})^2 \qquad (2.3)$$

Mögliche Optimierungsalgorithmen in Machine Learning sind:

- *Normal Equation*
- *Gradient Descent*
- *Backpropagation*

2.3. KÜNSTLICHE NEURONEN

Das künstliche Neuron ist der Grundbaustein eines künstlichen neuronalen Netzes und hat die Funktion, eine Reihe von Eingabedaten zu empfangen, diese Daten zu verarbeiten und schließlich einen einzigen Ausgabewert zurückzugeben. Während ein einfaches neuronales Netz aus einem einzigen Neuron bestehen kann, gibt es normalerweise Tausende von Neuronen in nur einer Schicht des Netzes. Die meisten Neuronen in einem neuronalen Netz erhalten Eingaben von anderen Neuronen, und die von ihnen erzeugte Ausgabe wird zur Eingabe für die nächste neuronale Schicht.

Sowohl die Verarbeitung der Eingabedaten, als auch die Art der Ausgabe des Neurons lässt sich durch zwei mathematische Funktionen bestimmen: eine Lineare Funktion und eine *Aktivierungsfunktion*: Die *Gewichte* eines Neurons sind Parameter, die sich aus dem Trainieren eines neuronalen Netzes ergeben und bilden zusammen mit den Eingangswerten eine lineare Funktion. Auf das Ergebnis dieser Operation wird eine Funktion angewendet, die den Aktivierungsgrad des Neurons bestimmt, die sogenannte Aktivierungsfunktion. Dabei handelt es sich in der Regel um eine Stufenfunktion mit einem oder mehreren Schwellenwerten. Wenn das Ergebnis der gewichteten Summe einen Schwellenwert überschreitet, wird das Neuron aktiviert und sendet ein positives Signal an die mit ihm verbundenen Neuronen.

Die Aktivierungsfunktion hängt von der Art des verwendeten künstlichen Neurons ab, und diese wiederum von der Art des gewünschten Ausgangswertes. Im Falle einer Klassifizierung, bei der es nur zwei Klassen gibt, wäre eine mögliche Aktivierungsfunktion die Treppenfunktion, da sie nur zwei mögliche Ergebnisse bietet.

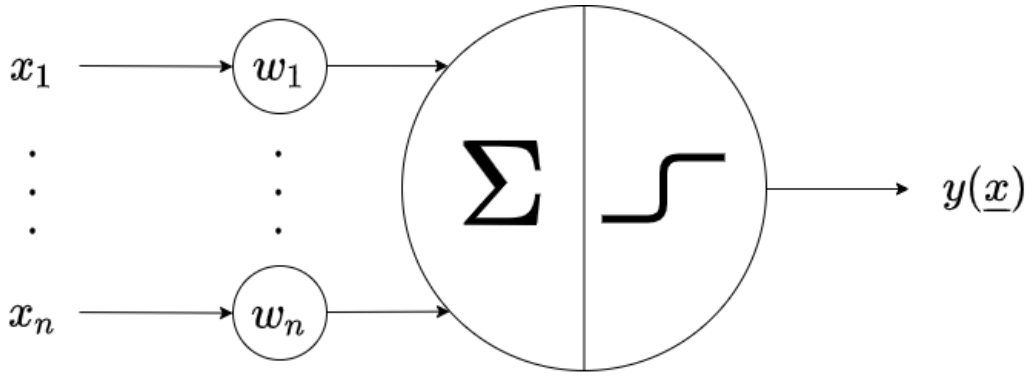


Abb. 3 Struktur eines Künstlichen Neuron
Quelle: Eigene Darstellung, adaptiert von (Géron, 2019, S. 284)

Die folgende Funktion beschreibt die Ausgabe eines einzelnen künstlichen Neurons:

$$y(\underline{x}) = \phi(\underline{x}^T \cdot \underline{w}) \tag{2.4}$$

Es gibt verschiedene Arten von Aktivierungsfunktionen, die je nach verwendetem Modell eingesetzt werden. Einige von ihnen sind in der folgenden Tabelle aufgeführt.

Tabelle 1 Beispiele für Aktivierungsfunktionen in Machine Learning

Aktivierungsfunktion	Gleichung	Beispiel
Linear	$\phi(z) = z$	Adaline, linear regression
Unit step (Heaviside function)	$\phi(z) = \begin{cases} 0, & z < 0 \\ 0.5, & z = 0 \\ 1, & z > 0 \end{cases}$	Perceptron variant
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, multilayer NN
Hyperbolic tangent (tanh)	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multilayer NN, RNNs
ReLU	$\phi(z) = \begin{cases} 0, & z < 0 \\ z, & z > 0 \end{cases}$	Multilayer NN, CNNs

2.4. NEURONALES NETZ

2.4.1. STRUKTUR EINES NETZES

Neuronale Netze sind in *Layers* (deu. Schichten) unterteilt. Jede dieser Schichten enthält eine Reihe künstlicher Neuronen mit gemeinsamen Merkmalen, wie z.B. der Anzahl der Parameter pro Neuron und der gleichen Aktivierungsfunktion. Der erste Layer eines Netzes entspricht die „Input-Layer" (deu. „Eingabeschicht"), in der jeder Knoten einen Eingabewert darstellt, und der letzte Layer ist die „Output-Layer" (deu. „Ausgabeschicht"), wo endgültig die Vorhesage des Netzes ausgegeben wird. Alle Layers zwischen der Input- und Output-Layer werden als *Hidden-Layers* bezeichnet.

Die Art und Weise, wie die Schichten eines Neurons miteinander verbunden sind, ist wichtig für die Berechnung der Ausgabe des neuronalen Netzes. In Modellen mit einer geringen Anzahl von Schichten ist es üblich, dass alle Neuronen in einer Schicht mit allen Neuronen in der vorhergehenden Schicht verbunden sind. Solche Neuronenschichten werden als *fully connected layers* (deu. vollvermaschte Schichten) bezeichnet (Frochte, 2018, S. 174). Jede dieser Verbindungen stellt also einen Parameter des neuronalen Netzes dar und je mehr Verbindungen vorhanden sind, desto größer wird der Rechenaufwand des neuronalen Netzes sein. Wenn jedoch die Anzahl der Hidden Layers innerhalb des Netzes beträchtlich ansteigt, wird es rentabel, die Verbindungen zwischen den Neuronen zu reduzieren, um die Anzahl der Parameter zu verringern und die Rechengeschwindigkeit des Netzes zu erhöhen.

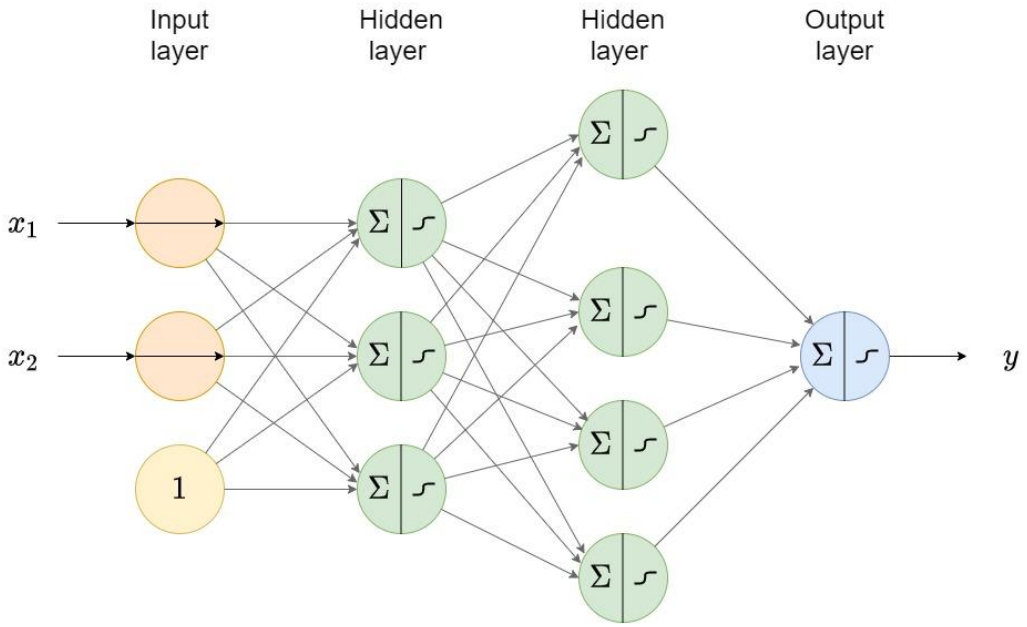


Abb. 4 Beispiel für ein künstliches neuronales Netz mit zwei Hidden Layers. Die Eingabeschicht besteht aus zwei Eingabeneuronen, die die Werte x_1 und x_2 erhalten, und einem Bias-Neuron mit einem festen Wert von 1. Das Output des Netzes ist ein einzelner Wert y
Quelle: Eigene Aufnahme

2.4.2. FUNKTIONSWEISE EINES NEURONALEN NETZES

Der Mechanismus, wie ein neuronalen Netz Eingangsdaten verarbeitet und somit Vorhersagen macht, kann als eine Reihenfolge von Matrixoperationen beschrieben werden, indem die Netzparameter (Gewichten) und die Aktivierungsfunktion jeder Schicht eine Rolle spielen.

Die Ausgabe einer beliebigen Schicht $\mathbf{O}^{(i)}$ eines neuronalen Netzes ist:

$$\mathbf{O}^{(i)} = \phi\left(\mathbf{W}^{(i)}\mathbf{O}^{(i-1)} + \mathbf{b}^{(i)}\right) \tag{2.5}$$

Die Matrix $\mathbf{O}^{(i-1)}$ enthält den Ausgabewerten der vorherigen Schicht und entspricht die Eingangsmatrix der Schicht i. Die Matrix $\mathbf{W}^{(i)}$ und der Vektor $\mathbf{b}^{(i)}$ enthalten die lineare Netzparameter in der Schicht i. Schließlich wird das Ergebnis der Matrixoperation durch die Aktivierungsfunktion $\phi()$ transformiert.

2.5. TRAINING DES NETZES

Das Training neuronaler Netze basiert auf dem Konzept des überwachten Lernens, das schon im Kapitel 2.2.1 vorgestellt wurde, und auf der Anwendung von zwei besonderen Algorithmen: *Gradient Descent* (deu. *Gradientenverfahren*) und *Backpropagation* (deu. *Fehlerrückführung*).

Gradient Descent

Die allgemeine Idee dieses Algorithmus besteht darin, die Parameter iterativ zu verändern, um die Verlustfunktion eines Lernalgorithmus zu minimieren (Géron, 2019, S. 118). In jeder Iteration dieses Verfahrens, das als *Learning Step* (deu. Lernschritt) bezeichnet wird, sind für jeden Parameter des Lernalgorithmus die partiellen Ableitungen von der Verlustfunktion zu berechnen. Alle partiellen Ableitungen bildet der *Gradient* der Verlustfunktion.

$$\nabla_w MSE(\mathbf{W}) = \begin{pmatrix} \frac{\partial}{\partial w_1} MSE(\mathbf{W}) \\ \vdots \\ \frac{\partial}{\partial w_n} MSE(\mathbf{W}) \end{pmatrix} \tag{2.6}$$

Da das Ziel des Algorithmus darin besteht, das globale Minimum der Verlustfunktion zu erreichen, ist es notwendig, einen zu dem Gradienten proportionalen Vektor von den Trainingsparametern abzuziehen. Die Festlegung der *Learning Rate* η (deu. Lernrate) ermöglicht, die Länge jedes Iterationsschrittes zu steuern.

$$\mathbf{W}^{(i+1)} = \mathbf{W}^{(i)} - \eta \cdot \nabla_w MSE\left(\mathbf{W}^{(i)}\right) \tag{2.7}$$

Es ist zu beachten, dass die Lernrate die Geschwindigkeit beeinflusst, mit der der Algorithmus zu einem Minimum konvergiert. Falls sie zu klein ist, dann erhöht sich die Anzahl der erforderlichen Iterationen. Falls sie zu groß ist, entsteht das Risiko der Divergenz. (Géron, 2019, S. 118-119)

Ein weiteres wichtiges Kriterium ist die Form der Verlustfunktion. Mit zunehmender Anzahl von Parametern wird die Funktion komplexer und es entstehen lokale Minima. Dies bedeutet, dass es mehr als einen Konvergenzpunkt gibt. Aus diesem Grund ist es wichtig, verschiedene Ausgangswerte für die Parameter und die Lernrate auszuprobieren.

Backpropagation

Kurz gesagt, *Backpropagation* ist ein an die Verwendung in neuronalen Netzen angepasster *Gradient Descent*-Algorithmus. Der Unterschied zwischen den beiden besteht darin, dass der *Backpropagation*-Algorithmus die Abhängigkeit der Parameter einer Neuronenschicht von der vorangegangenen Neuronenschicht berücksichtigt.

Backpropagation läuft wie folgend ab:

1. Alle Netzparameter werden mit Zufallswerten initialisiert.
2. Der Algorithmus unterteilt alle Trainingsbeispiele in *Minibatches* (in der Implementierung es *batch-size* genannt). Jedes Minibatch wird durch das Netz durchgeführt und ihre jeweiligen Outputs werden gespeichert (*forward pass*). Dieser Schritt ist identisch mit dem Prozess, wenn das Netz Vorhersagen macht.
3. Es wird dann der Verlust für das ganze Netz berechnet (z.B durch *mean square error (2.3*)).
4. Durch Anwendung von *Gradient Descent* wird berechnet, wie viel jede Verbindung des Netzes zum Netzverlust beigetragen hat (Fehlergradienten). Dies erfolgt zuerst in der Output-Layer, dann in den Hidden-Layer davor und so weiter, bis die Input-Layer erreicht wird (*reverse pass*).
5. Alle Verbindungen werden unter Verwendung der berechneten Fehlergradienten optimiert.

Jede Iteration dieses Durchlaufs wird als *Epoch* bezeichnet. Zusammen mit anderen Hyperparametern (*Learning Rate, Batch-Size*) wird der Optimierungsalgorithmus mehrmals reguliert und durchgeführt, bis das neuronale Netz zu einem Minimum konvergiert und somit eine genügende Leistungsfähigkeit erreicht.

2.6. CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (abk. *CNN*, deu. „Faltungsneuronale Netze“) sind eine Art von künstlichen neuronalen Netzen, die hauptsächlich auf die Forschungsarbeit von Yan LeCun (1989) zurückgehen. Eine Eigenschaft, die sie von anderen neuronalen Netzmodellen abhebt, ist die Art der Verbindung zwischen ihren neuronalen Schichten. *CNNs* besitzen Neuronen mit spezialisierten Rezeptoren, die Informationen aus einer begrenzten Gruppe von Neuronen in der vorherigen Schicht sammeln. Dieses Merkmal beruht auf mehreren Untersuchungen des Gehirns bei David H. Hubel und Torsten Wiesel ((Hubel, 1959), (Hubel & Wiesel, 1959) und (Hubel & Wiesel, 1968)), die zeigen, dass Neuronen in der Sehrinde kleine rezeptive Felder haben, die nur auf visuelle Reize reagieren, die sich in bestimmten Regionen des Gesichtsfeldes befinden. In Übereinstimmung mit dem Konzept, auf dem sie beruhen, können *CNNs* Computer Vision-Aufgaben sehr effizient erfüllen.

2.6.1. KONZEPT DES COMPUTER VISIONS

Das Betrachten eines Bildes und die automatische Erkennung seines Inhalts ist für Menschen ein trivialer Vorgang, für einen Computer jedoch relativ komplex. Der Grund dafür ist, dass die Maschine kein Bild als solches sieht, sondern eine Reihe von Zahlen, die die Intensität jedes Pixels im Bild darstellen. Bei Graustufenbildern haben diese Werte einen Bereich zwischen 0,0 und 255,0, wobei der erste Wert für Schwarz und der zweite Wert für Weiß steht. Bei digitalen Bildern mit einem Farbmodell (wie z.B. RGB) wird die Intensität jedes Pixels aus drei Farbkanäle definiert, in diesem Fall Rot, Grün und Blau, und durch die Kombination von allen Farbkanäle erfolgt das ganze Farbbild.

Ein Neuronales Netz, dessen Trainingsdaten aus Bildern besteht, muss in der Lage sein, die Pixelwerte auf den Bildern als Eingangswerte aufnehmen zu können. Das heißt, dass das Input-Layer aus der gleichen Anzahl von Neuronen bestehen muss, wie es Pixel und Farbkanäle gibt.

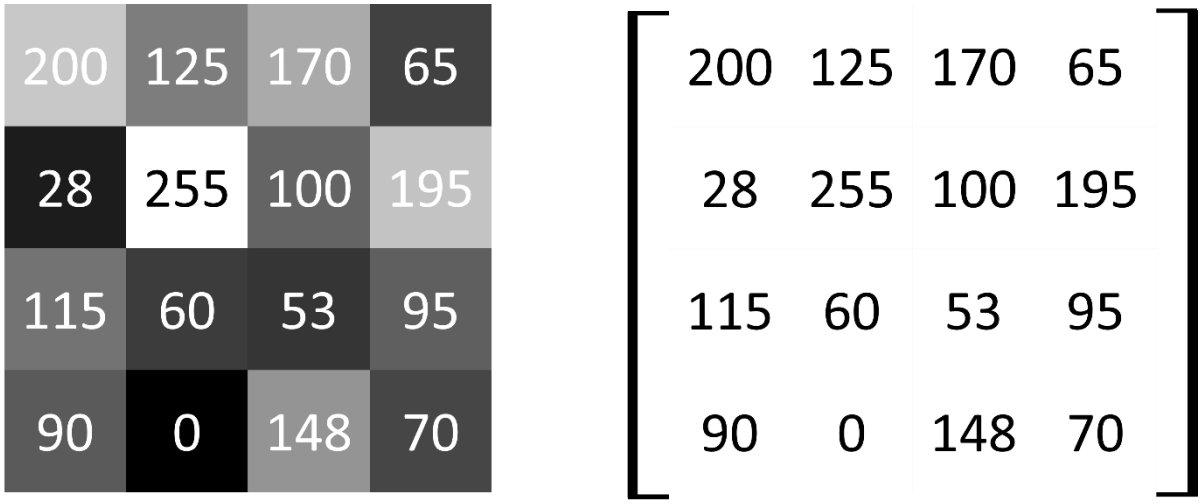


Abb. 5 Ein Graustufenbild kann als 2 x 2 Matrix dargestellt werden. Jeder Wert steht für ein Pixel des Bildes
Quelle: Eigene Darstellung

2.6.2. KONZEPT EINES *CNNS-MODELLS*

Alle CNNs zeichnen sich durch die Verwendung von zwei speziellen Arten von neuronalen Schichten aus: *Convolutional Layers* und *Pooling Layers.*

Convolutional Layers unterscheiden sich von *fully-connected-Layers* dadurch, dass ihre Neuronen nur mit bestimmten *lokalen Feldern* der Input-Layer verbunden sind. Die Verbindungsparameter bei diesen Layers werden als *Filters,* und die mathematische Operation zur Berechung der Layeroutputs, *Convolution* (deu. Faltung) bezeichnet. Analog zu gewöhnlichen Künstlichen Neuronen wenden die Neuronen eines *Convolutional Layer* auf dem Ergebnis einer *Convolution* eine Aktivierungfunktion an, wobei die ReLU-Funktion (*Rectifier Linear Unit*) aufgrund ihrer Vorteile bei der Implementierung am häufigsten benutzt wird (Cha, Choi, & Büyüköztürk, 2017, S. 365). Der Output eines *Convolutional Layers* ist ein Bündel von *feature maps,* wobei jede *feature map* durch einen anderen Filter erzeugt wird.

Bei der Implementierung von *Convolutional Layers* müssen die folgenden Parameter angegeben werden:

- Anzahl der Filter (Entspricht der Anzahl der *feature maps*)
- Kernelgröße (Entpricht die Dimensionen eines lokalen Felds)
- *Stride* (deu. Schritt, ist analog zu dem Abstand zwischen benachbarten lokalen Feldern)
- Aktivierungsfunktion (In der Regel ReLU)
- *Padding* (Es kann von Typ SAME oder VALID sein (Géron, 2019, S. 454-455))
- Form des Inputs (Entpricht der Länge, der Breite und der Anzahl von Farbkanälen auf einem Bild)

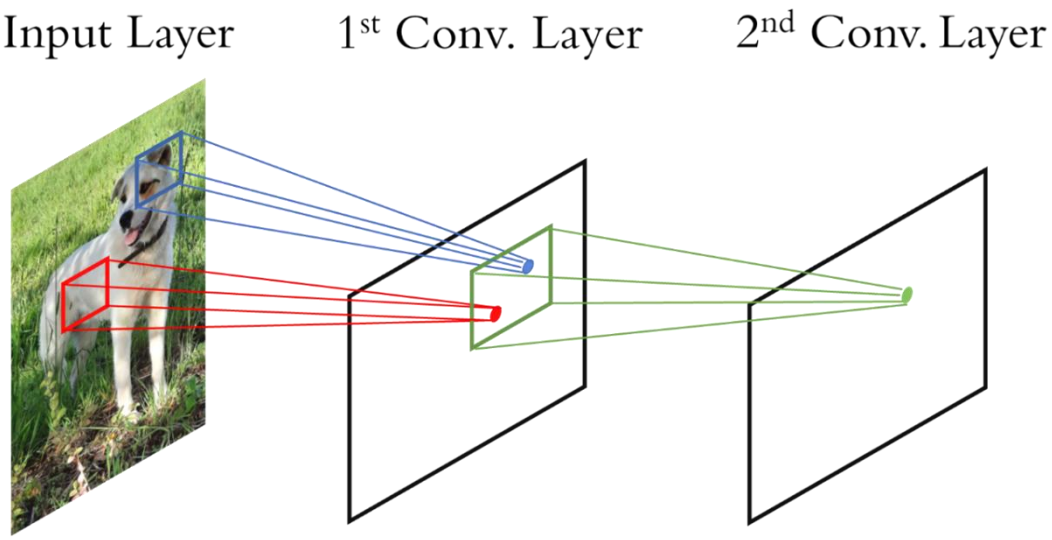


Abb. 7 Die roten und blauen Neuronen innerhalb des ersten Convolutional-Layers sind nur mit ihren lokalen Rezeptoren im Bild verbunden. Ebenso erhält das grüne Neuron in dem zweiten Convolutional-Layer nur Informationen von den Neuronen des ersten Convolutional-Layers, die sich in seinem lokalen Rezeptor befinden.
Quelle: Eigene Darstellung, adaptier von (Géron, 2019, S. 448)

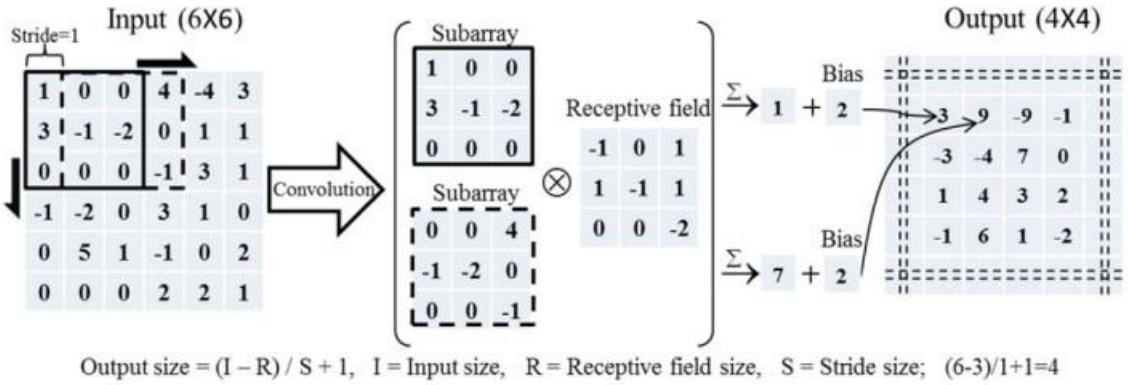


Abb. 6 Beispiel eines Convolution-Vorgangs
Quelle: (Cha, Choi, & Büyüköztürk, 2017, S. 364)

Pooling-Layers haben die Funktion, die Datenmenge ihres Inputs zu reduzieren. Die Gründe für dieses Verfahren sind die Verringerung der Rechenlast, des Speicherbedarfs und der Anzahl der Netzwerkparameter, sowie die Einführung von Invarianz gegenüber kleinen Translationen (Géron, 2019, S. 457-458). Das Ergebnis solcher Layers ist ein Bild mit einer, im Vergleich zum Original, verringerten Auflösung.

Im Wesentlichen besteht der *Pooling*-Vorgang darin, dass ein Neuron die Werte seines lokalen Feldes sammelt, aber anstatt eine *Convolution* zu berechnen, wie es eine *Convolutional Layer* durchführen würde, nimmt es einfach einen Wert nach einem simplen Prinzip

an. Zum Beispiel holt ein *Max-Pooling-Layer* den größten Wert innerhalb des lokalen Empfängers ab. Andererseits berechnet ein *Average-Pooling-Layer* den Durchschnitt der Werte, die in das Neuron eingehen.

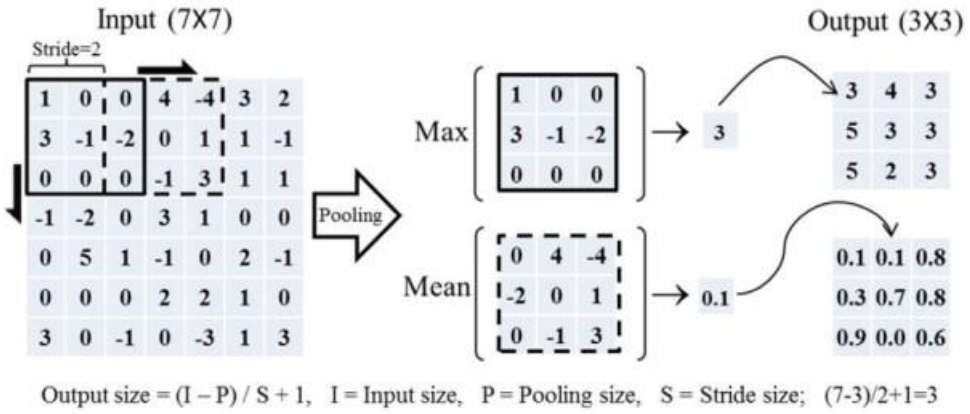


Abb. 8 Beispiel eines Pooling-Vorgangs
Quelle: (Cha, Choi, & Büyüköztürk, 2017, S. 365)

2.6.3. NETZWERKARCHITEKTUR EINES CNNS-MODELLS

Seit ihrer Entstehung wurden Modelle für faltige neuronale Netze mit dem Ziel entwickelt, immer komplexere Aufgaben mit höherer Genauigkeit zu lösen. Ein weiterer Faktor ist die Verarbeitungsgeschwindigkeit des neuronalen Netzes, die mit der Anzahl der Parameter des Netzes zusammenhängt.

Die traditionelle Architektur eines *CNN-Modells* besteht aus einer Reihe von verschachtelten *Convolutional*- und *Pooling*-Layers, gefolgt von zwei bis drei *fully-connected-Layers*. Es besteht die Tendenz, dass das Bild kleiner wird, je weiter es durch das Netz läuft. Allerdings wächst die Anzahl der Merkmalskarten mit jeder *Convolution*. Die Eigenschaften des Input-Layers hängen von den Dimensionen der Trainingsbilder ab (Bildgröße, Anzahl der Farbkanäle) und die Output-Layer von der Art der Aufgabe des *CNNs* (Anzahl der Output-Neuronen, passende Aktivierungsfunktion für Klassifikations- oder Regressionsaufgabe).

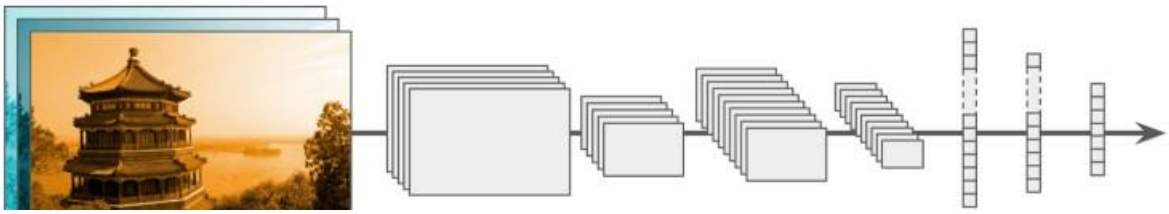


Abb. 9 Beispiel einer traditionellen CNN-Architektur
Quelle: (Géron, 2019, S. 461)

Einer der Pioniere und vielleicht die bekannteste CNN-Architektur ist LeNet-5 (Géron, 2019, S. 463). Es wurde von Yann LeCun et al. (1989) entwickelt und wird häufig für die Erkennung handgeschriebener Ziffern verwendet. Der für das Training verwendete Datensatz, MNIST genannt, enthält Graustufenbilder mit 28x28 Pixeln, die mit Zero-Padding auf 32x32 Pixel angepasst werden. Das Netz besteht aus drei *Convolution*- und zwei *Average-Pooling-Layers* gefolgt von einem *fully-connected-Layer*, wobei die *Hyperbolic-Tangent*-Aktivierungsfunktion angewandt wird. Das Output-Layer ist ein *fully-connected-Layer* mit der RBF-Aktivierungsfunktion (Radial Basis Function). (LeCun, Bottou, Bengio, & Haffner, 1998)

Krizhevsky et al. (2012) schlugen das *AlexNet*-Modell auf der Grundlage von *LeNet-5* vor, dass jedoch erhebliche Änderungen aufweisen. Sie fügten dem Modell weitere *Convolutional-Layers* hinzu und tauschten die *Average-Pooling-Layers* gegen *Max-Pooling-Layers* aus. Außerdem optimierten sie das Netzwerk durch die Verwendung von *ReLU* als globale Aktivierungsfunktion und *Softmax* als Aktivierungsfunktion der Ausgabeschicht. Der Trainingsdatensatz besteht aus RGB-Bildern mit 224 x 224 Pixeln. Dieses Netz wurde bereits mehrfach für die Entwicklung von Betonschadensdetektoren eingesetzt (Cha, Choi, & Büyüköztürk, 2017) (Dorafshan, Thomas, & Maguire, 2018).

Die *Inception*-Architektur wurde von Szegedy et al. (2015) entwickelt. Dieses Modell zeichnet sich durch seine im Vergleich zu anderen Architekturen der damaligen Zeit große Anzahl von *Layers* aus. Außerdem wurden damit *Inception-Modules* eingeführt, die eine effizientere Nutzung der Parameter als in anderen Netzen ermöglichen (Géron, 2019, S. 467).

He et al. (2016) entwickelten das Konzept des *Residual Learning* (deu. residuales Lernen), wobei *Skip-Connections* (deu. Verbindung überspringen) auf ein neuronales Netz mit einer großen Anzahl von Schichten angewendet werden. Das vorgeschlagene neuronale Netz, genannt *ResNet*, erreicht mit diesem Konzept insgesamt 152 Schichten.

2.7. AUFGABEN EINES CNN MODELLS

2.7.1. KLASSIFIKATION UND LOKALISIERUNG

Klassifizierung und Lokalisierung ist die Aufgabe, bei der ein Netz einen *Bounding Box* (deu*. Begrenzungsrahmen*) um Elemente innerhalb eines Bildes vorhersagt. Ein typisches Beispiel ist die Gesichtserkennungsfunktion von Facebook. Auf einem Foto mit mehreren Personen steht der Name jeder Person für die Klassenbezeichnung. Wenn man ein Foto veröffentlicht, hat man die Möglichkeit, ein Rechteck über jedes Gesicht auf dem Bild zu ziehen. Dadurch werden Informationen über ein *Class-Label* (Klassifikationsaufgabe) und ihre Koordinaten im Bild (Lokalisierung in Form einer Regressionsaufgabe) erzeugt. Ein neuronales Netz, das für die Klassifikation und Lokalisierung trainiert wird, benötigt als Trainingsdaten Bilder mit ihren jeweiligen *Bounding Boxes*. Dies ist ein zeitaufwändiger manueller Prozess, der bei der Planung eines ML-Projekts immer berücksichtigt werden sollte. (Géron, 2019, S. 483-484)

2.7.2. OBJECT DETECTION

Object Detection (deu. Objekterkennung) umfasst eine Klassifizierungs- und Lokalisierungsaufgabe für mehrere Objekte in einem einzigen Bild. Dies wurde zunächst durch das Gleiten eines CNN-Modells über ein Bild erreicht (*Sliding-Window-Methode*), wobei Methoden wie das *Non-Max-Suppression* zum Einsatz kamen (Géron, 2019, S. 485-486). Long et al. (2015) führten das Konzept der *Fully Convolutional Networks* (abk. *FCN*, deu. vollständig gefalteten Netzwerke) ein, wobei die *fully-connected* Output-Layers eines *CNNs* durch *Convolutional Layers* ersetzt werden. Ein zuvor trainiertes Netz kann in ein FCN umgewandelt werden, ohne dass der Wert seiner Netzparameter verloren geht, und sein Output entspricht der *Non-Max-Suppression* Methode (Géron, 2019, S. 488).

Einige bemerkenswerte Modelle zur Objekterkennung sind *YOLO* (Redmon, Divvala, Girshick, & Farhadi, 2016), *SSD* (Liu, et al., 2016) und *Faster R-CNN* (Ren, He, Girshick, & Sun, 2015)*.*

2.7.3. SEMANTIC SEGMENTATION

Bei *Semantic Segmentation* (deu. semantische Segmentierung) wird jeder Pixel eines Bildes im Hinblick auf das Objekt klassifiziert, zu dem es gehört. Im Gegensatz zur *Object Detection* wo jedes Objekt lokalisiert und in einem *Bounding Box* eingerahmt wird, werden bei *Semantic Segmentation* präzise Konturen zwischen verschiedenen Objektklassen ermittelt. Es ist jedoch nicht möglich, Kanten zwischen zwei nebeneinander liegenden Objekten, die zur selben Klasse gehören, hervorzuheben. Seine Umsetzung basiert, wie bei der Objekterkennung, auf dem von Long et al. (2015) vorgeschlagenen FCN-Modell. Es gibt eine ähnliche Aufgabe, die als *Instance Segmentation* (deu. Instanzensegmentierung)

bezeichnet wird und darauf abzielt, zwischen Objekten der gleichen Klasse zu unterscheiden. (Bolya, Zhou, Xiao, & Lee, 2019)

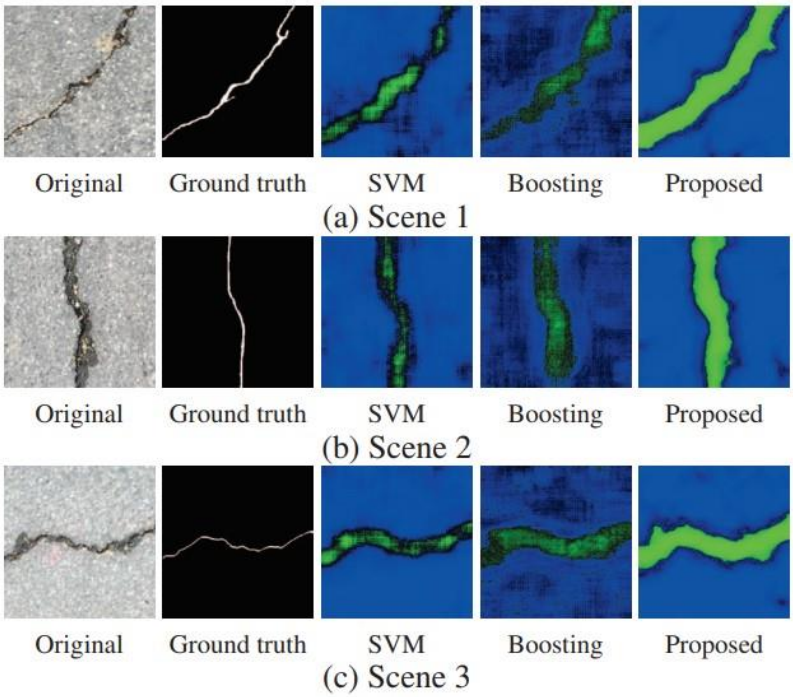


Abb. 10 Beispiel für eine semantische Segmentierung zur Erkennung von Rissen in Straßen. Quelle: (Zhang, Yang, Zhang, & Zhu, 2016)

Abb. 11 Beispiel für Image Segmentation

2.8. BESTEHENDE ANSÄTZE DER SCHADENSIDENTIFIKATION MIT MACHINE LEARNING-VERFAHREN

2.8.1. ERKENNUNG VON SCHÄDEN IN BETON

Es hat sich gezeigt, dass bei der automatischen Erkennung von Rissen in Beton, *CNNs* eine höhere Genauigkeit erreichen als herkömmliche *Edge-Detection*- Algorithmen (deu. Kantenerkennung). Cha et al. (2017) verglichen ihr vorgeschlagenes CNN-Modell mit den *Canny-Edge-Detection*- und *Soble-Edge-Detection*-Methoden. Das neuronale Netz erreichte in den Trainings- und Validierungsdatensätzen eine Genauigkeit von 98,22 %. Die *Canny*- und die *Sobel*-Methode lieferten dagegen keine relevanten Informationen zur Erkennung von Schäden. Dorafshan et al. (2018) kamen zu demselben Ergebnis, nachdem sie sechs verschiedene *Edge-Detection*-Methoden getestet hatten. Darüber hinaus schlagen sie eine hybride Methode vor, die die Vorteile von *Edge Detection* und *CNNs* kombiniert. Beckman et al. (2019) haben ein System zur volumetrischen Erkennung und Quantifizierung von Betonschäden entwickelt, das auf *Faster-CNN* basiert.

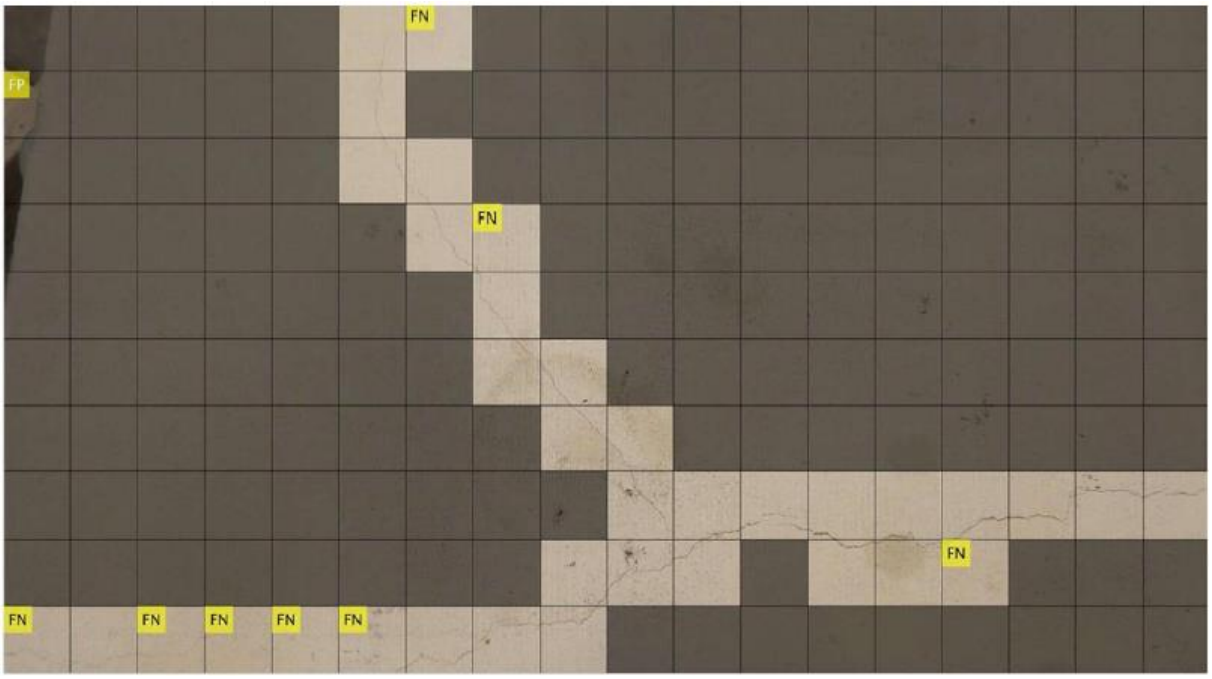


Abb. 12 Beispiel für einen Algorithmus zur Erkennung von Rissen in Beton unter Verwendung der Sliding Window-Methode
Quelle: (Dorafshan, Thomas, & Maguire, 2018)

2.8.2. ERKENNUNG VON SCHÄDEN IN STAHL

Atha und Jahanshahi (2018) untersuchten die Wirksamkeit verschiedener *CNN*-Modelle zur Erkennung von Korrosion in Stahl. Nash, Drummond und Birbilis (2018) schlussfolgerten, dass Datensätze mit einer großen Anzahl schlecht beschrifteter Bilder für *Semantic Segmentation* in Korrosionsbilder effektiver sind als kleine, detailliert beschriftete Datensätze. Katsamenis et al. (2020) kritisierten die Verwendung von *bounding-oriented Object Detection* (Anwendung von *Bounding Boxes*) in bisherige Studien zur Erkennung von Stahlkorrosion und untersuchten die Leistung von *Semantic Segmentation-oriented*- CNN-Modelle (*FCN*, *Mask R-CNN* und *U-Net*). Sie argumentieren, dass die Verwendung von Bounding Boxes für die Erkennung von Korrosion in Stahl für die Strukturanalyse nicht geeignet ist, da sie nicht die notwendigen Informationen für die Bewertung des Schadensgrades liefert (area, aspect ratio, maximum distance). Rahman, Wu und Kalfarisi (2021) wendeten das ursprünglich von Chen et al. (2018) entwickelte *ASPP (Atrous Spatial Pyramid Pooling)* für eine auf semantischer Segmentierung basierende Korrosionserkennung an.

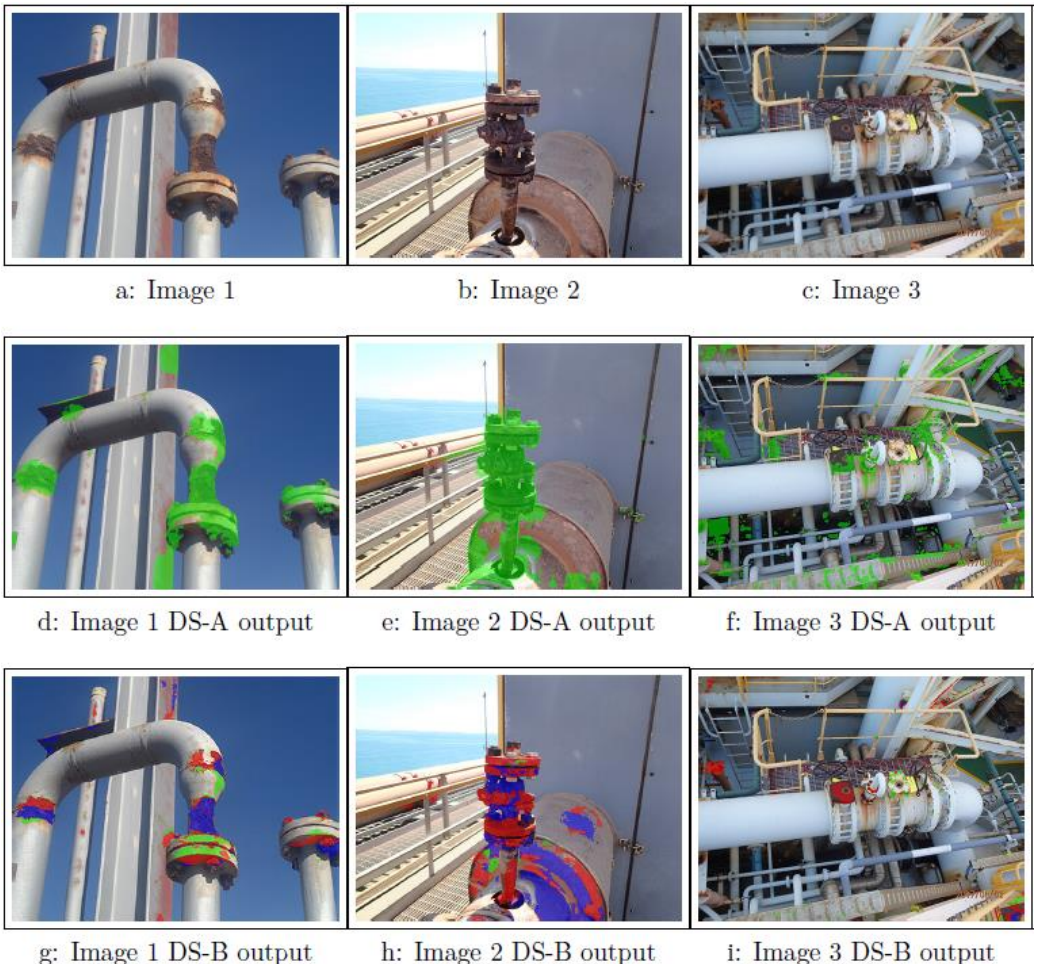


Abb. 13 Ergebnisse eines auf semantischer Segmentierung basierenden Korrosionsdetektors
Quelle: (Nash, Drummond, & Birbilis, 2018)

2.8.3. AUTOMATISCHE SCHADENSAUFNAHME BEI DER BAUINSPEKTION

Zhang et al. (2016) präsentierten eine auf *Semantic Segmentation* basierende Schadenserkennungsalgorithmus für die Inspektion von Straßenaufbrüchen. Die in ihrem neuronalen Netz verwendeten Trainingsdaten bestehen ausschließlich aus Bildern, die mit einem preisgünstigen Smartphone aufgenommen wurden.

Kim et al. (2018 (6)) schlagen eine Methode zur Anwendung von ML-basierten Schadenserkennungstechniken bei der Brückeninspektion mit Hilfe eines UAVs vor.

Lee et al. (2019) sammelten mit Hilfe von UAVs Bilder von Bolzen-, Nieten- und Stiftverbindungen einer Stahlfachwerkbrücke, um einen auf *Instance Segmentation* basierenden Korrosionsdetektor zu entwickeln.

2.8.4. MULTIKLASSIFIKATION VON SCHÄDEN IN BAUWERKEN

Cha et al. (2018) stellen eine *Faster R-CNN*-basierte Methode zur Erkennung von fünf Arten von Strukturschäden vor: Betonrisse, Stahl-Delamination, Bolzenkorrosion, mittlere und hohe Stahlkorrosion. Li et al. (2019) schlugen eine Methode zur automatischen Erkennung und Klassifizierung von Kanalisationsdefekten aus nicht balancierten CCTV-Inspektionsdatensätzen (*closed circuit television inspection*) mithilfe eines CNN vor. Sie verwendeten hierarchische Softmax anstelle der traditionellen Softmax, sodass die Fehler auf verschiedenen Ebenen klassifiziert werden. Dies ermöglicht die Einführung von Unterkategorien in der Klassifikation. Kim et al. (2019) schlagen vor, dass die Verwendung von *CCR* (*crack candidate region*) bei der Klassifizierung mit *CNN* die Leistung des Modells erhöht, da das Erscheinen von rissähnlichen Mustern im Bild berücksichtigt wird.

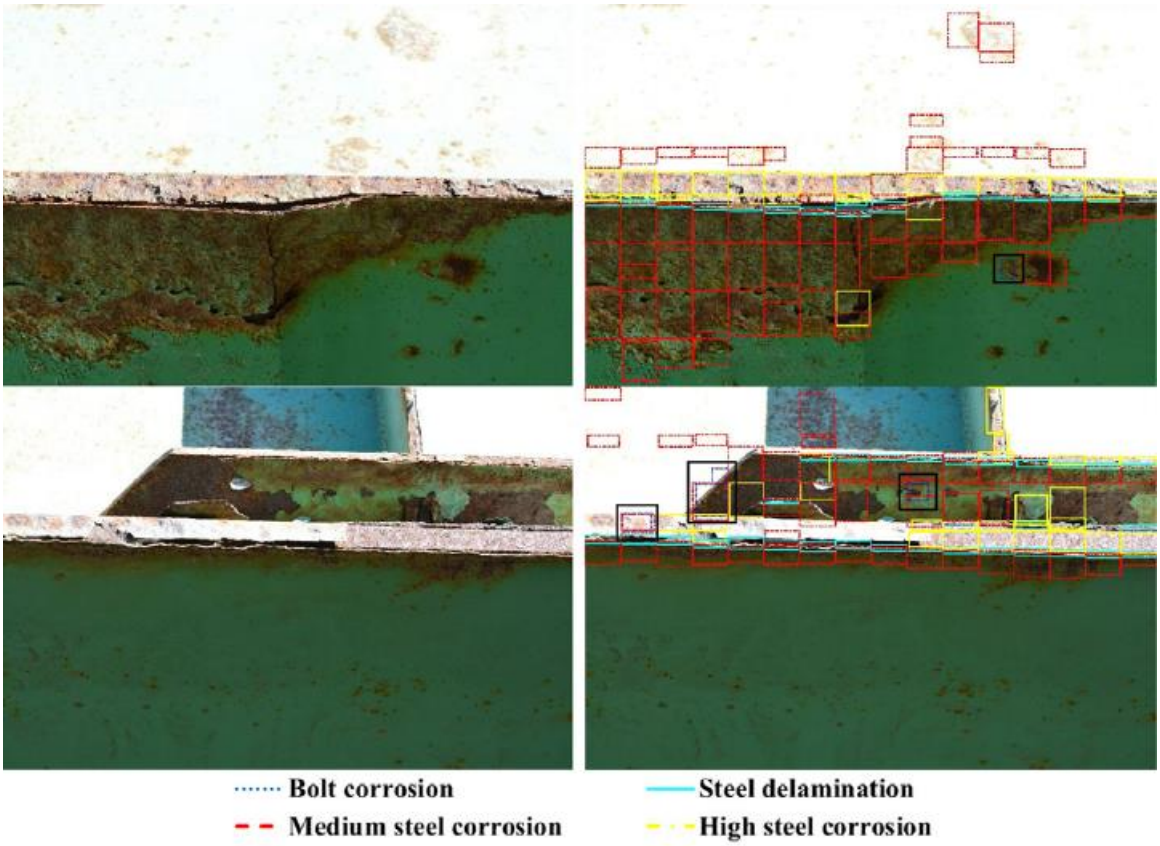


Abb. 14 Beispiel für einen Multi-Klassifikator-Schadensalgorithmus
Quelle: (Cha, Choi, Suh, Mahmoudkhani, & Büyüköztürk, 2018)

2.8.5. AUTOMATISCHE SCHÄDENÜBERTRAGUNG IN BIM-MODELLEN

Isailović et al. (2020) schlagen einen semiautomatischen Ansatz zur Erstellung von *as-is-BIM*-Modellen (Ist-Zustand) aus *as-designed*-BIM-Modellen und 3D-*point-cloud*-Daten von Brückenstrukturschäden vor. Zunächst konvertieren sie das BIM-Modell in einem 3D-Dreiecknetz und erfassen sie dann manuell das 3D-*point-cloud*. Schließlich werden die eingefügten Schäden geometrisch und semantisch angereichert, um das *as-is-BIM*-Modell zu vervollständigen (Isailović, Stojanovic, Trapp, Richter, & Hajdin, 2020). Artus et al. (2021) schlagen einen IFC-basierten Rahmen für die Erzeugung und Positionierung von Abplatzungsgeometrien in BIM-Modellen vor. Es folgt eine automatische Schadenserkennung auf der Grundlage von *Multiview-classification* (Der Trainingsdatensatz besteht aus Cube-map-Bildern anstelle von flachen Bildern) und CNNs.

2.8.6. ANALYSE DER BIM-BASIERTEN SCHADENSIDENTIFIKATION MITTELS MACHINE LEARNING

Was sind die Merkmale bestehender Studien zum maschinellen Lernen und Computer Vision zur Schadenserkennung?

- Es lassen sich zwei Ansätze unterscheiden: Objekterkennung auf der Grundlage der Implementierung von Bounding Boxes, und der *Semantic Segmentation*-Ansatz.
- Neuere Studien konzentrieren sich auf sichtbare Schäden an Beton und Stahl.
- Im Beton sind Schäden in Form von Rissen und Abplatzungen zu beobachten.
- Korrosionsschäden werden an Stahl in verschiedenen Intensitätsstufen beobachtet.

Aktuelle Studien über den Einsatz von Computer Vision zur Schadenserkennung konzentrieren sich auf Oberflächenschäden in Beton und Stahl. Untersucht wird insbesondere die Erkennung von Rissen (Dorafshan, Thomas, & Maguire, 2018) und Abplatzungen (Beckman, Polyzois, & Cha, 2019) in Betonstrukturen sowie die Erkennung von Rissen in Straßen (Zhang, Yang, Zhang, & Zhu, 2016).

Bei Stahlstrukturen wurde die automatische Erkennung und Klassifizierung von Stahlkorrosion in verschiedenen Stadien (Nash, Drummond, & Birbilis, 2018) sowie die Lokalisierung von Delamination untersucht (Cha, Choi, Suh, Mahmoudkhani, & Büyüköztürk, 2018).

Während sich die meisten Studien auf die Erkennung einer einzelnen Schadensart konzentrieren, gibt es auch Versuche, ML-Modelle zu entwickeln, die mehrere Schadensarten klassifizieren können (Cha, Choi, Suh, Mahmoudkhani, & Büyüköztürk, 2018). Neben der Klassifizierung geht es auch darum, Merkmale zu extrahieren, die eine Bewertung der festgestellten Schäden ermöglichen.

Was sind die Vor- und Nachteile beider Ansätze?

- Mit Hilfe der semantischen Segmentierung können mehr Informationen über Schäden extrahiert werden als mit dem Bounding-Box-Ansatz.
- Der Bounding Box-Ansatz benötigt weniger Zeit für die Kennzeichnung von Trainingsbildern.
- Die Berechnungsgeschwindigkeit eines auf Bounding Boxes basierenden Detektors ist nahezu augenblicklich.

Für den alleinigen Zweck, den Ort des Schadens zu erkennen, zeigt der *Bounding-Box*-Ansatz gute Ergebnisse (Cha, Choi, Suh, Mahmoudkhani, & Büyüköztürk, 2018) (Kim, Ahn, Shin, & Sim, 2019). Dieser Ansatz ist jedoch nicht geeignet, wenn Sie Informationen über die Geometrie des Schadens erhalten möchten (Katsamenis, Protopapadakis, Doulamis, Doulamis, & Voulodimos, 2020) (Rahman, Wu, & Kalfarisi, 2021).

3 ENTWICKLUNG EINES SOFTWARE-PROTOTYPS FÜR EINE BIM-BASIERTE SCHADENSIDENTIFIKATION

3.1. ALLGEMEIN

Die Erfassung von Schäden in einer Schadensdatenbank ist zeitaufwändig und ungenau. Ein anschauliches Beispiel ist das Programm SIB-BAUWERKE, das derzeit zur Schadensdokumentation im Rahmen der Bauwerksprüfung nach DIN 1076 eingesetzt wird. Die Eintragung von neuen Schäden erfolgt manuell durch eine textliche Schadensbeschreibung, die aus Angaben zu dem beschädigten Bauteil, der Art des Schadens und der grobe Schadensmenge (z.B „kleine“, „mittlere“ oder „große“) besteht. Eine präzisere Angabe der Schadensdimensionen ist als sechsstellige Zahl mit zwei Nachkommazahlen möglich. Der Ort des Schadens wird durch zwei Informationen angegeben: ein Bild des beschädigten Bauelements und eine textliche Beschreibung der Position des Schadens im Bild. Schließlich ist die Schadensbewertung beigefügt, die auf dem gewählten Schadensbeispiel aus dem Katalog im Anhang der RI-EBF-PRÜF basiert. (Pohl, 2020)

Ziel dieses Kapitels ist die Konzeption eines Software-Prototyps, der in der Lage ist, Bauschäden aus Bildern zu erkennen, die genaue Position der gefundenen Schäden in einem dreidimensionalen Koordinatensystem zu bestimmen und die daraus resultierenden Informationen schließlich automatisch in eine Schadensdatenbank zu übertragen.

3.2. GRUNDKONZEPT EINES WORKFLOWS

In den vorangegangenen Kapiteln wurden Beispiele vorgestellt, bei denen verschiedene Arten von Schäden mit Hilfe von *Convolutional Neural Networks* erkannt wurden. Das Output des neuronalen Netzes hängt von dem für die Schadenserkennung verwendeten Ansatzes ab. Bei *Boundary-Based Object Detection* besteht die Ausgabe aus einer Reihe von rechteckigen Labels, die die Position und die Klasse des lokalisierten Schadens angeben, während bei *Image Segmentation* farbige Abschnitte erhalten werden, die zusammen ein neues Bild bilden, wobei die Farbe jedes Abschnitts die Klasse des Schadens darstellt. Beide Methoden beschränken sich auf die Lokalisierung von Pixelgruppen, in denen Schäden gefunden werden könnten. Mit diesen Informationen allein lässt sich jedoch der tatsächliche Ort des Schadens in der realen Welt nicht ermitteln. Es wird eine Methode benötigt, um den Pixeln eines zweidimensionalen Bildes Koordinaten im realen Raum zuzuordnen.

Welche sind die Ziele des Software-Prototyps?

- Implementierung von ML für die automatische Erkennung von Schäden aus den Inspektionsbildern (Klassifikation und Lokalisierung der Schäden)
- Transformation der Schadensortkoordinaten in den Bildern in ein globales Koordinatensystem des BIM-Modells (2D zu 3D).
- Automatische Eintragung der erhaltenen Schadensinformationen in eine Schadensdatenbank.

Da das derzeitige System vorsieht, dass der Gebäudeinspektor alle Schadensinformationen manuell in die Schadensdatenbank eingibt, besteht eine Möglichkeit zur direkten Beschleunigung des Prozesses darin, die Dateneingabe zu automatisieren. Die Verwendung von neuronalen Netzen bei der Schadenserkennung ermöglicht die Erfassung von Informationen sowohl über die Art des Schadens als auch über seine Lage im Bild. Wenn ein neuronales Netzmodell in die Schadenserfassung integriert wird, könnte es die Dauer der Eingabe von Informationen in die Schadensdatenbank erheblich verkürzen. Falls ein BIM-Modell des untersuchten Bauwerks existiert, wäre es möglich, jeden identifizierten Schaden mit Hilfe einer relationalen Datenbank einem Bauelement des BIM-Modells zuzuordnen. Die Arbeit des Inspektors würde sich auf die Erfassung der Schadensinformationen und die anschließende Bewertung des entstandenen Schadens beschränken.

Was wäre ein Grundkonzept für einen Workflow?

- Der Gebäudeinspektor macht Fotos von sichtbaren Schäden an einem Bauwerk.
- Die Schäden auf die Bilder werden mittels ML-Algorithmen automatisch erkannt.
- Die erkannten Schäden können dann in einer Schadensdatenbank eingetragen werden.
- Jede Zeile der Tabelle entspricht einem identifizierten Schaden. Die Spalten der Tabelle geben die Referenzdaten an (Schadensart, zugeordnete Bauelement, Dimensionen, Beschreibung, etc.).
- Die Schadensdatenbank kann mit BIM-basierter Software abgerufen werden.

Welche Überlegungen sollten bei der Gestaltung eines Schadenserkennungsmerkmals berücksichtigt werden?

- Es soll die verschiedenen Arten von Schäden an Strukturen betrachten (Betonrisse und -abplatzungen, Stahlkorrosion, etc.)
- Im Rahmen der Gebäudeinspektion muss es in der Lage sein, festgestellte Schäden mit den bei früheren Inspektionen gesammelten Daten zu vergleichen.
- Die Methode zur Sammlung von Trainingsdaten sollte schnell sein, um die Bauinspektion zu beschleunigen.
- Die Ausrüstung für die Bilderfassung sollte so weit wie möglich aus kostengünstigen, handelsüblichen Geräten (z. B. Smartphones) bestehen.
- Der Einsatz anderer Instrumente für die Bauinspektion kann integriert werden (z. B. UAVs für die Brückeninspektion), ist aber nicht zwingend erforderlich.

Welche Komponenten sind für die Realisierung des Workflows erforderlich?

- Ein System zur Schadenserkennung, das ein Netzwerkmodell und Trainingsdaten enthält.

Zusätzlich muss diese Software die Ergebnisse so darstellen, dass sie in ein BIM-Modell integriert werden können. Weitere Punkte, die zu berücksichtigen sind, sind die zu verwendenden elektronischen Geräte, wie z. B. Digitalkameras, die Methodik für die Bilderfassung und die Dokumentation der nach der Inspektion gewonnenen Informationen.

Die Objekterkennungsmethode beschreibt jeden erkannten Schaden durch fünf Parameter: Die erste ist eine Klassenbezeichnung, die die Art des Schadens angibt, während die anderen vier die Abmessungen und die Lage eines *Bounding Box* darstellen (Breite, Höhe und Koordinaten des Mittelpunkts).

Der Gesamtprozess der Bilderfassung, der Identifizierung durch das neuronale Netz und der Eingabe in eine Schadensdatenbank wird in einer *Ereignisgesteuerte Prozesskette* (abk. *EPK*) vorgestellt. Auch die Elemente, die die einzelnen Prozessfunktionen beeinflussen, werden anhand von Modellen detailliert erläutert.

3.3. SCHADENSERFASSUNG

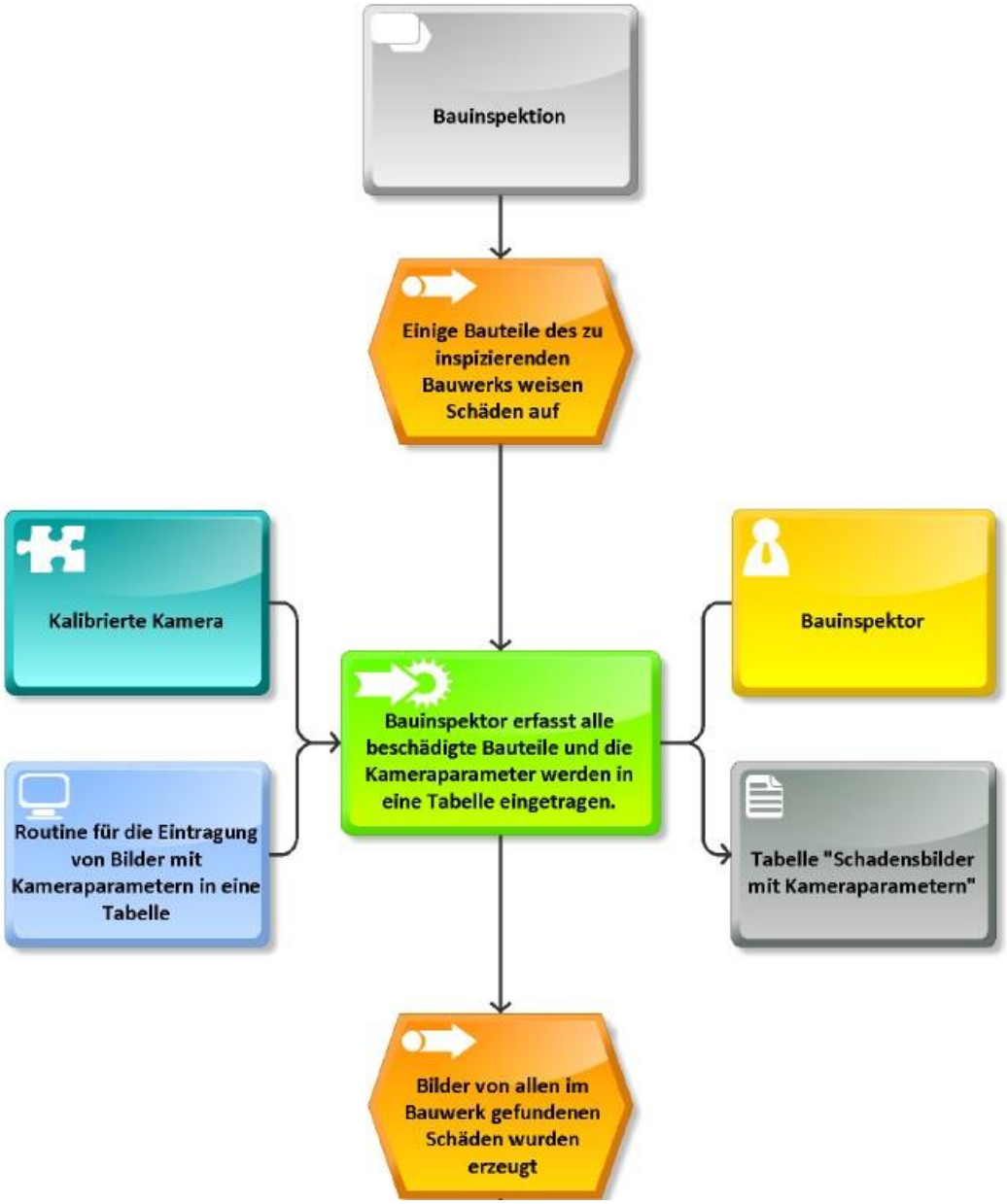


Abb. 15 eEPK-Diagramm zur Schadenserfassung

Bei der Inspektion eines Gebäudes achtet der Bauinspektor auf das Vorhandensein von sichtbaren Schäden. Alle festgestellten Schäden müssen mit einer *kalibrierten Kamera* fotografiert werden. Dieser Schritt ist sehr wichtig, da eine kalibrierte Kamera zusammen mit dem Bild Metainformationen über die dreidimensionale Position der Kamera zum Zeitpunkt der Aufnahme speichert. Dies ermöglicht eine Zuordnung der in der

zweidimensionalen Ebene des Fotos identifizierten Schäden zu einem dreidimensionalen Koordinatensystem, das mit dem BIM-Modell verknüpft werden kann.

Die für die Kalibrierung erforderlichen Parameter, die so genannten Kammerparameter, werden im Folgenden vorgestellt, gefolgt von den Kalibrierungsmethoden zur Ermittlung dieser Parameter.

3.3.1. KAMERAPARAMETER UND DIRECT LINEAR TRANSFORMATION (DLT)

Im Bereich der Computer Vision werden *Kameramodelle* verwendet, um eine Beziehung zwischen realen Objekten und ihren fotografischen Projektionen herzustellen. Ein Beispiel ist das *Pinhole camera model* (deu. *Lochkameramodell*), das eine Eins-zu-Eins-Abbildung zwischen Punkten auf Objekten der 3D-Welt und dem Bild ermöglicht. Zu diesem Zweck muss eine Beziehung zwischen den am Kameramodell beteiligten Koordinatensystemen hergestellt werden.

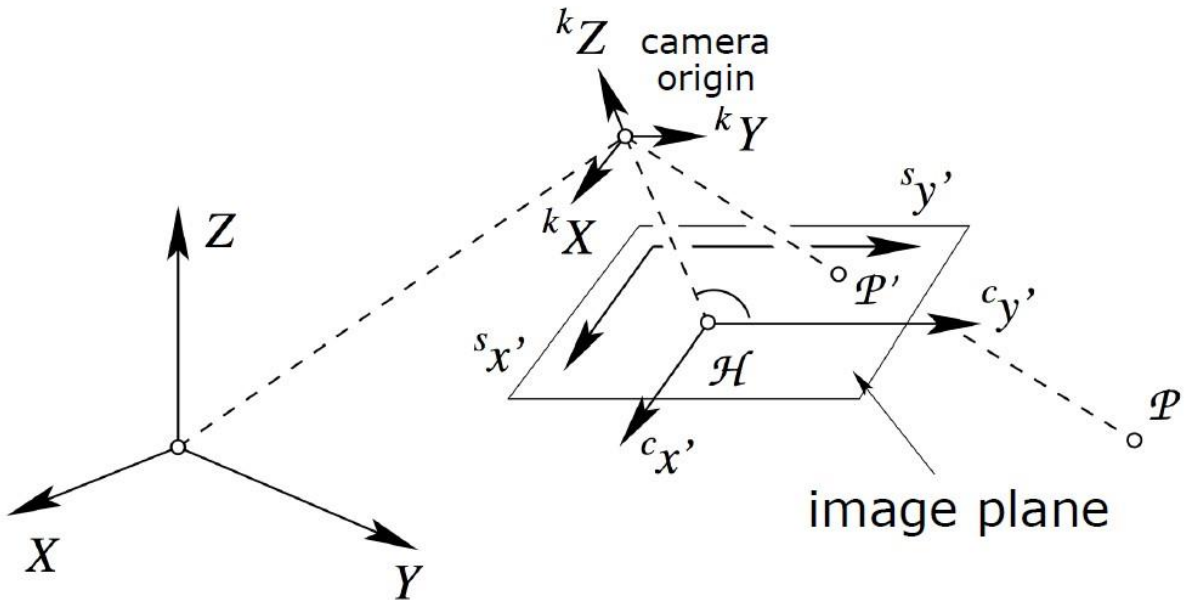


Abb. 16 Beteiligte Koordinatensysteme an einem Kameramodell
Quelle: (Förstner & Wrobel, 2016, S. 470)

Es gibt insgesamt vier Koordinatensysteme, die für die Kameramodelle zu berücksichtigen sind:

1. Weltkoordinatensystem mit Hauptachsen
2. Kamerakoordinatensystem mit Hauptachsen ${}^{k}X,{}^{k}Y,{}^{k}Z$
3. Bildebenen-Koordinatensystem mit Hauptachsen ${}^{c}x',{}^{c}y'$
4. Sensor-Koordinatensystem mit Hauptachsen ${}^{s}x',{}^{s}y'$

Es müssen also insgesamt drei Koordinatentransformationen durchgeführt werden. Unter Berücksichtigung der nicht linearen Abweichungen kann eine vierte Transformation hinzugefügt werden.

1. Weltkoordinatensystem →Kamerakoordinatensystem: Euklidische Transformation
2. Kamerakoordinatensystem → Bildebenen-Koordinatensystem: Zentralprojektion
3. Bildebenen-Koordinatensystem → Sensor-Koordinatensystem: Affine Abbildung
4. Sensor-Koordinatensystem → Sensor-Koordinatensystem: nichtlineare Transformation

Die Koordinatenabbildung eines Punktes in der realen Welt auf einen Punkt in einer Bildebene mit kann mathematisch durch die folgende Formel ausgedrückt werden.

$$\mathbf{x} = \mathbf{PX} \tag{3.1}$$

Dabei steht $\mathbf{x}$ für die Pixelkoordinaten eines Punktes in der Bildebene und $\mathbf{X}$ für die Koordinaten desselben Punktes im dreidimensionalen Raum. Die Matrix $\mathbf{P}$ wird als Parametermatrix bezeichnet und ergibt sich aus der Koordinatentransformation zwischen dem dreidimensionalen realen Raumkoordinatensystem und dem zweidimensionalen Koordinatensystem des Kamerasensors.

Alle Parameter, aus denen sich die Parametermatrix $\mathbf{P}$ zusammensetzt, werden als *Kameraparameter* definiert. Sie können als intrinsische und extrinsische Parameter klassifiziert werden. Die *intrinsischen Parameter* beschreiben die Abbildung der Szene vor der Kamera auf die Pixel des endgültigen Bildes und bilden die Kalibrierungsmatrix $\mathbf{K}$. Die *Extrinsische Parameter* bezeichnen die dreidimensionalen Koordinaten der Kamera (Ausgedrückt durch die Matrix $[\mathbf{I}|-\mathbf{X_0}]$,, wobei $\mathbf{I}$ die 3x3 Identitätsmatrix und $\mathbf{X_0}$ die Position des fotografierten Objekts entsprechen) und ihre Rotationen zu den entsprechenden Hauptachsen (Rotationsmatrix $\mathbf{R}$).

$$\mathbf{P} = \mathbf{K} \cdot \mathbf{R} \cdot [\mathbf{I}|-\mathbf{X_0}] \tag{3.2}$$

$$\mathbf{K} = \begin{bmatrix} c & cs & x_H \\ s & c(1+m) & y_H \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, [\mathbf{I}| - \mathbf{X_0}] = \begin{bmatrix} 1 & 0 & 0 & -X_O \\ 0 & 1 & 0 & -Y_O \\ 0 & 0 & 1 & -Z_O \end{bmatrix} \tag{3.3}$$

Die intrinsischen Parameter innerhalb der Kalibrierungsmatrix $\mathbf{K}$ sind wie folgt definiert:

- Kamerakonstante (oder Brennweite) c:
- Bildscherung s
- Skalierungsdifferenz der x- und y-Bildkoordinaten m
- Koordinaten des Hauptpunktes im Sensorkoordinatensystem x_H, y_H

Die Abbildung der Punkte vom Weltkoordinatensystem auf das Koordinatensystem des Kamerasensors, die durch die Formeln (3.1) und (3.2) gegeben ist, wird als *affines Kameramodell* bezeichnet. Dieses Modell berücksichtigt alle intrinsischen und extrinsischen Eigenschaften einer Kamera, modelliert aber keine nichtlineare Verzerrung in den Bildern. Dennoch erlaubt dieses Modell eine erste Annäherung an das Problem der Koordinatentransformation zwischen der realen Welt und dem Bild. Darüber hinaus ist es möglich, die Bildverzerrung mit Hilfe von Computer-Vision-Methoden zu behandeln und dann das Kameramodell für eine genauere Koordinatenzuordnung zu Kalibrierung der Kamera

Unter geometrischer Kamerakalibrierung versteht man die Schätzung der Intrinsischen Kameraparameter von Objektiv und Bildsensor einer Bild- oder Videokamera. Mit anderen Worten, eine kalibrierte Kamera ist eine, bei der die durch die Kalibrierungsmatrix gegebenen intrinsischen Parameter bekannt sind. Hierzu wird die Methode der *Direct Linear Transformation* (deu. *direkte lineare Transformation*, abk. DLT) angewandt. Bei dieser Methode werden alle Kameraparameter in einem affinen Kameramodell, sowohl intrinsisch als auch extrinsisch, anhand von 6 Kontrollpunkten mit bekannten Koordinaten bestimmt.

Die in homogenen Koordinaten ausgedrückte Formel (3.1) sieht wie folgt aus:

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{bmatrix} \tag{3.4}$$

Der erste Schritt bei der Entwicklung des DLT besteht darin, die Formel (3.1) in ein System linearer Gleichungen umzuwandeln, wobei die zu lösenden Werte die Parameter der Parametermatrix $\mathbf{P}$ sind. Dies führt zu dem folgenden äquivalenten Ausdruck:

$$\mathbf{M} \cdot \mathbf{p} = \mathbf{0} \tag{3.5}$$

Der Vektor $\mathbf{p}$ entspricht der Parametermatrix $\mathbf{P}$, die in einen Vektor mit 12 Komponenten umgeschrieben wird, während die Matrix $\mathbf{M}$ die Faktoren enthält, mit denen jedes der Elemente in $\mathbf{p}$ multipliziert wird. Da der DLT 12 Unbekannte hat, was der Anzahl der Elemente des Vektors p entspricht, sind insgesamt 12 Gleichungen zur Lösung des Gleichungssystems erforderlich. Da jeder bekannte Punkt zwei Koordinaten hat, sind sechs Kontrollpunkte zu berücksichtigen.

$$\mathbf{M} = \begin{bmatrix} \mathbf{a}_{x_1}^T \\ \mathbf{a}_{y_1}^T \\ \mathbf{a}_{x_2}^T \\ \mathbf{a}_{y_2}^T \\ \mathbf{a}_{x_3}^T \\ \mathbf{a}_{y_3}^T \\ \mathbf{a}_{x_4}^T \\ \mathbf{a}_{y_4}^T \\ \mathbf{a}_{x_5}^T \\ \mathbf{a}_{y_5}^T \\ \mathbf{a}_{x_6}^T \\ \mathbf{a}_{y_6}^T \end{bmatrix}, \quad \mathbf{a}_{x_i} = \begin{bmatrix} -X_i \\ -Y_i \\ -Z_i \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ x_i X_i \\ x_i Y_i \\ x_i Z_i \\ x_i \end{bmatrix}, \quad \mathbf{a}_{y_i} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -X_i \\ -Y_i \\ -Z_i \\ -1 \\ y_i X_i \\ y_i Y_i \\ y_i Z_i \\ y_i \end{bmatrix}, \quad \mathbf{p} = \begin{bmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \end{bmatrix} \tag{3.6}$$

Durch Lösen des Gleichungssystems in (3.5) erhält man die numerischen Werte für den Vektor $\boldsymbol{p}$ und folglich für die Parametermatrix $\mathbf{P}$**.** Weiterhin wird sie gemäß Formel (3.2) in ihre Komponenten zerlegt, d. h. in die Kalibrierungsmatrix $\mathbf{K}$, die Rotationsmatrix $\mathbf{R}$ und den Positionsvektor $\mathbf{X_0}$.

3.3.2. PROJECTIVE 3-POINT ALGORITHM (P3P-ALGORITHM)

Unter der Annahme, dass sich die interne Ausrichtung der Kamera während der Inspektion nicht ändert, müssten die externen Parameter nur dann neu berechnet werden, wenn die Kamera ihre Position ändert. Dies liegt daran, dass der Inspektor sich bewegen muss, um alle Schäden an einem Bauwerk zu erfassen. Anstatt bei jeder Aufnahme neu zu kalibrieren, ist es praktischer, eine Methode zu verwenden, bei der nur die äußere

Ausrichtung der Kamera berechnet wird. Zu diesem Zweck eignet sich das *Projective 3-Point Algorithm* (deu Projektives 3-Punkt-Algorithmus) bei Grunert (1841).

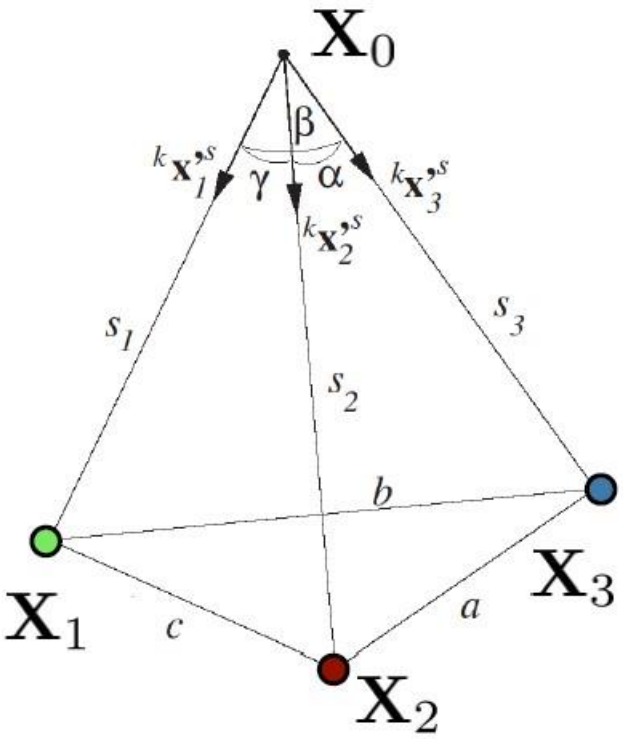


Abb. 17 Darstellung vom Projective 3-Point Algorithm
Quelle: (Stachniss, 2015)

Das Verfahren dieses Algorithmus besteht aus zwei Schritten: Zunächst werden die Abstände zwischen dem Projektionszentrum der Kamera und drei bekannten Punkten bestimmt. Die Koordinaten der Punkte des fotografierten Objekts im Kamerakoordinatensystem ergeben sich aus der folgenden Formel:

$$s_i {}^k\mathbf{x}_i^s = \mathbf{R}(\mathbf{X_i} - \mathbf{X_0}), \qquad \mathrm{i} = 1{,}2{,}3 \tag{3.7}$$

Die Differenz zwischen den Koordinaten des Punktes $\mathbf{X}_i$ und der Kamera im Weltkoordinatensystem $\mathbf{X}_O$, multipliziert mit der Rotationsmatrix $\mathbf{R}$, ergibt den Abstand in Vektorform innerhalb des Kamerakoordinatensystems. Dieser Vektor entspricht der Multiplikation zwischen dem normierten Richtungsvektor ${}^k\mathbf{x}_i^s$ und dem Abstand zwischen der Kamera und dem Objekt s_i.

Aus der Kalibrierungsmatrix $\mathbf{K}$ und den Koordinaten der Punkte im Sensorkoordinatensystem $\mathbf{x}$ wird der normierte Richtungsvektor berechnet. Es ist zu beachten, dass das Glied $-sign(c)$ vorangestellt wird, damit der Richtungsvektor in die richtige Richtung zeigt.

$$^{k}\mathbf{x}_{i}^{s} = -sign(c)\frac{\mathbf{x}_{i}}{|\mathbf{x}_{i}|}(\mathbf{K}^{-1}\mathbf{x}_{i}) \tag{3.8}$$

Die Winkel zwischen Richtungsvektoren können direkt durch Trigonometrie bestimmt werden. Daher werden die Distanzen s_i mit Hilfe des Kosinussatzes berechnet. Es besteht jedoch das Problem, dass die Lösung der resultierenden Gleichungen für die Abstände ein Polynom vierten Grades ergibt, was vier mögliche Ergebnisse für die Abstände bedeutet. Durch den Vergleich mit einer Näherungslösung (z.B. durch GPS) oder durch die Einbeziehung eines vierten Kontrollpunktes ist es möglich, die richtige Lösung und damit die Berechnung der Rotationsmatrix $\mathbf{R}$ und der Kameraposition $\mathbf{X}_O$ zu ermitteln.

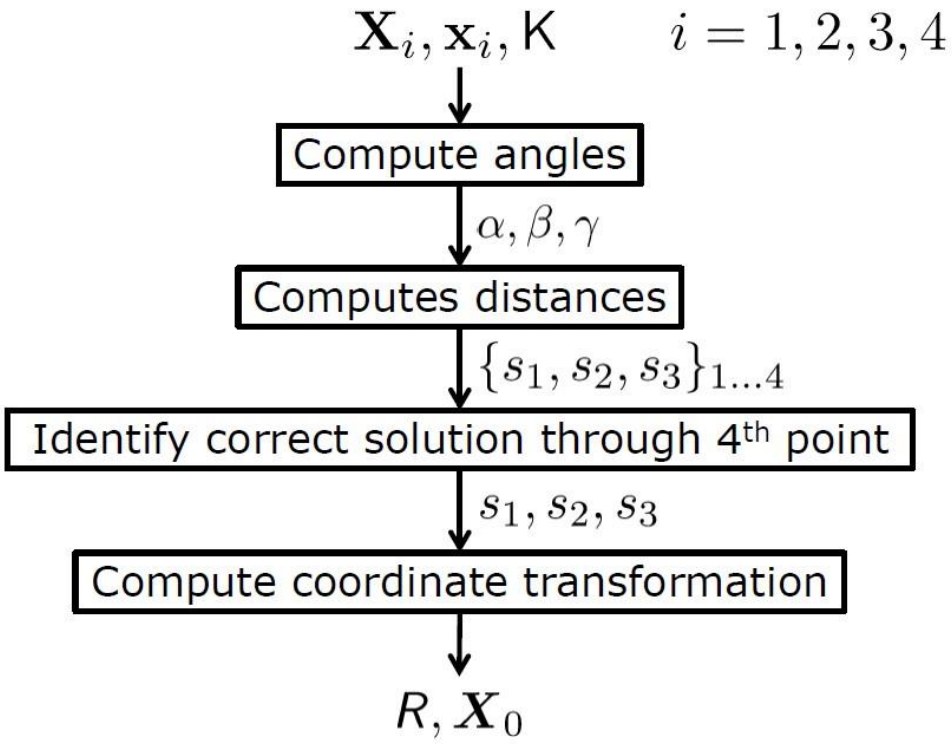


Abb. 18 Herangehensweise für die Berechnung des P3P-Algorithmus
Quelle: (Stachniss, 2015)

3.3.3. ERFASSUNG DER KAMERAPARAMETER

Um die bei der Inspektion festgestellten Schäden zu lokalisieren, ist die Kenntnis der Kameraparameter bei der Aufnahme der Schäden erforderlich. Aus diesem Grund ist es

unbedingt erforderlich, die Kamera zu kalibrieren, bevor Schadensbilder aufgenommen werden. Für den Kalibrierungsprozess mithilfe der DLT-Methode werden 6 bekannte Punkte benötigt (siehe Kapitel 0). Ein möglicher Ansatz besteht daher darin, die Kalibrierung der Kamera vor der Inspektion durchzuführen. Eine weitere Möglichkeit ist die Verwendung eines BIM-Modells zur Lokalisierung von sechs Kontrollpunkten im Bauwerk. Unter der Annahme, dass sich die intrinsischen Parameter der Kamera nicht ändern, ist nur die Aktualisierung der extrinsischen Parameter vor jeder Aufnahme erforderlich. Eine Methode aus dem Bereich der Photogrammetrie und analog zur DLT-Methode ist der P3P-Algorithmus, der in Kapitel 3.3.2 vorgestellt wird.

Zu Beginn der Schadenserfassung muss eine Tabelle erstellt werden, in der jedes einzelne Foto erfasst wird. In dem Moment, in dem ein Bild aufgenommen wird, wird eine neue ID in der Tabelle eingetragen. Die Kameraparameter werden zu dieser ID hinzugefügt. Sobald die Schadenserfassung abgeschlossen ist, wird die Tabelle unter dem Namen der entsprechenden Inspektion gespeichert.

Tabelle 2: Tabelle „Schadensbilder mit Kameraparameter", bestehend aus einer Bild-ID und den intrinsischen und extrinsischen Parametern zum Zeitpunkt der Aufnahme des Bildes

Img_path											
0001.jpg	3.54	1.54	1.71	5.765657	2.100887	2.982911	1,0	0	0	0	0
0002.jpg	8.25	8.46	1.73	0.053648	3.116938	0.323905	1,0	0	0	0	0
0003.jpg	14.95	5.45	1.72	4.461004	2.756478	3.057744	1,0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...

Anmerkung: Aufgrund mangelnder praktischer Erfahrung ist es mir nicht möglich, den Kalibrierungsprozess während einer Inspektion mit Sicherheit zu beschreiben. In den Quellen heißt es, dass sich die intrinsischen Parameter einer Kamera nicht ändern, solange sich die innere Ausrichtung der Kamera nicht ändert.

3.4. IMPLEMENTIERUNG DES NEURONALEN NETZWERKMODELLS

Zur Implementierung des maschinellen Lernens in den Workflow ist ein trainiertes neuronales Netzmodell erforderlich. Für das Training neuronaler Netze gelten wiederum zwei Voraussetzungen: Zunächst wird ein Bilddatensatz benötigt, der die zu klassifizierenden Schadensarten enthält. Zweitens werden die Labels benötigt, die den vom Netz zu identifizierenden Bauschäden entsprechen. Sobald diese Anforderungen erfüllt sind, wird das neuronale Netzmodell trainiert, wobei die Bewertungsmetriken während des Trainings berücksichtigt werden, um die Qualität des Modells einzuschätzen.

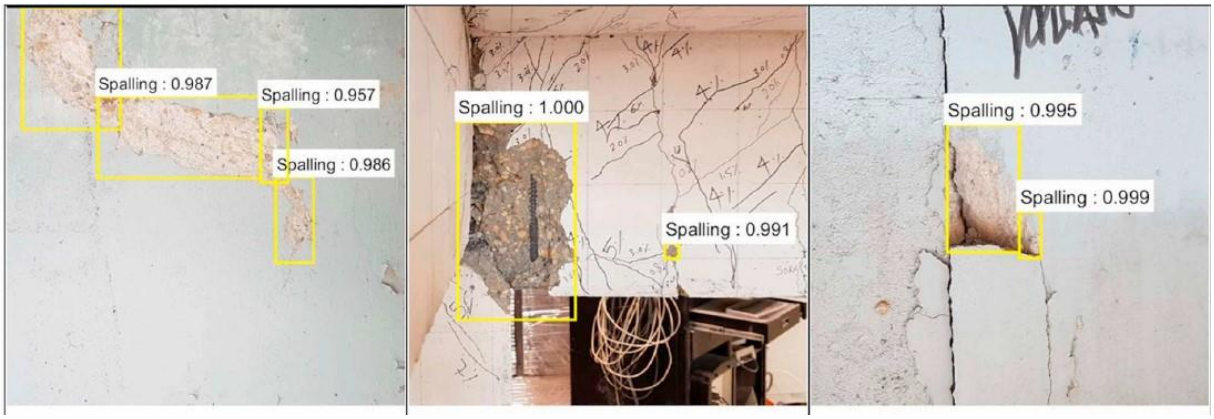


Abb. 19 Objekterkennungsmodell für die Erkennung von Betonabplatzungen
Quelle: (Beckman, Polyzois, & Cha, 2019)

Das trainierte Modell ist dann in der Lage, Vorhersagen über die Art und den Ort der Schäden auf den Bildern der Bauinspektion zu treffen. Im Falle der Objekterkennung besteht jede dieser Vorhersagen aus einem Label, das dem vorhergesagten Schadenstyp entspricht, und einem *Bounding Box*, der den Schaden einrahmt und zu vier Koordinaten im Bild zusammengefasst wird. Wenn man jedes Bild durch das neuronale Netz laufen lässt, erhält man insgesamt fünf Werte für jeden Schaden, der sich in einem Bild befindet. Die Registrierung der vom neuronalen Netz gefundenen Schäden führt zu einer Tabelle mit sieben Spalten, deren Felder die ID des gefundenen Schadens, die ID des Bildes, auf dem der Schaden gefunden wurde, und die fünf Werte, die der vom neuronalen Netz getroffenen Vorhersage entsprechen, enthalten.

Tabelle 3 Mögliche Darstellung eine Tabelle zur Erfassung der lokalisierten Schäden einer Bauinspektion in Bildkoordinaten

Schaden-ID	**Img_path**	**Klasse**	***X_min***	***X_max***	***Y_min***	***Y_max***
0001_0001	0001.jpg	Riss	118	1685	20	1006
0002_0001	0002.jpg	Abplatzung	0	100	0	664
0002_0002	0002.jpg	Abplatzung	1	849	3	565
0003_0001	0003.jpg	Mittl.Korr.	104	560	20	120
...	...	...	...	...	...	...

3.5. UMWANDLUNG VON 2D IN 3D KOORDINATEN

Damit die bisher gewonnenen Schadensinformationen mit einem BIM-Modell gekoppelt werden können, ist es notwendig, die Koordinaten des Schadens im Koordinatensystem des BIM-Modells zu erhalten. Die Umwandlung von Pixelkoordinaten in dreidimensionale

Koordinaten des vom neuronalen Netz lokalisierten Schadens ist mit Hilfe eines Kameramodells möglich. Dazu werden die bei der Bilderfassung ermittelten Kameraparameter und die vom neuronalen Netz lokalisierten Schadenskoordinaten benötigt.

Die Kameramodell-Formel in (3.1) beschreibt die Position eines Punktes in einem Bild in Abhängigkeit von seiner Position im dreidimensionalen Raum. Um die dreidimensionalen Koordinaten eines Punktes aus den Koordinaten im Bild zu erhalten, muss man die Formel wie folgt umstellen:

$$\lambda \mathbf{x} = \mathbf{P}\mathbf{X} = \mathbf{K}\mathbf{R}[\mathbf{I}| - \mathbf{X_0}]\mathbf{X} \tag{3.9}$$

$$\lambda \mathbf{x} = [\mathbf{K}\mathbf{R}\mathbf{I} - \mathbf{K}\mathbf{R}\mathbf{X_0}] \begin{bmatrix} \mathbf{X} \\ \mathbf{1} \end{bmatrix} \tag{3.10}$$

$$\lambda \mathbf{x} = \mathbf{K}\mathbf{R}\mathbf{X} - \mathbf{K}\mathbf{R}\mathbf{X_0} \tag{3.11}$$

Wenn die Funktion nach $\mathbf{X}$ umgestellt wird, dann folgt es:

$$\mathbf{X} = (\mathbf{K}\mathbf{R})^{-1}\mathbf{K}\mathbf{R}\mathbf{X_0} + \lambda(\mathbf{K}\mathbf{R})^{-1}\mathbf{x} \tag{3.12}$$

$$\mathbf{X} = \mathbf{X_0} + \lambda(\mathbf{K}\mathbf{R})^{-\mathbf{1}}\mathbf{x} \tag{3.13}$$

Die Funktion (3.13) zeigt die Inversion der Kameramodellbeziehung. Der Ortsvektor $\mathbf{X_0}$ gibt die Koordinaten der Kameraposition relativ zum globalen Koordinatensystem an, während das Glied $\lambda(\mathbf{K}\mathbf{R})^{-\mathbf{1}}$ die Richtung des Strahls zwischen dem Kameranullpunkt $\mathbf{X_0}$ und dem Punkt $\mathbf{X}$ angibt. Das Parameter λ ist ein Skalar, der die Distanz zwischen die Kamera und den Punkt entspricht. Es ist nicht in den Kameraparametern enthalten und kann nicht durch die Inversion der Kameramodellfunktion ermittelt werden. Es ist jedoch möglich, den Skalarparameter λ zu erhalten, wenn die Koordinaten des beschädigten Bauelements bekannt sind.

3.5.1. BESTIMMUNG DER DREIDIMENSIONALEN PUNKTKOORDINATEN MITTELS VEKTORRECHNUNG

In der Vektorrechnung kann eine Gerade durch die Punkt-Richtung-Form dargestellt werden. Eine parametrische Darstellung wird wie folgt ausgedrückt:

$$g: \quad \vec{r}(\lambda) = \vec{r}_1 + \lambda\vec{a}, \qquad \lambda \in \mathbb{R};\ \vec{a} \neq \vec{0} \tag{3.14}$$

Die Gerade g wird durch einen Ortsvektor $\vec{r}_1$ und einen Richtungsvektor $\vec{a}$ dargestellt. Der Vektor $\vec{r}(\lambda)$ entspricht alle Punkte auf die Gerade g. Aus dem Vergleich der parametrischen Darstellung der Gerade mit der Funktion (3.13) ergibt sich, dass:

$$\vec{r}_1 = \mathbf{X_0}, \qquad \vec{a} = (\mathbf{KR})^{-1}\mathbf{x} \tag{3.15}$$

Sowohl der Ortsvektor $\vec{r}_1 = \mathbf{X_0}$ als auch der Richtungsvektor $\vec{a} = (\mathbf{KR})^{-1}\mathbf{x}$ sind nach der Lösung von Gleichung (3.13) bekannt. Es ist jedoch nicht möglich, den Punkt $\mathbf{X}$ direkt aus der Geradengleichung zu ermitteln, da der spezifische Wert von λ für diesen Punkt benötigt wird. Eine Lösung ist die Feststellung, dass der Schaden auf der Oberfläche des fotografierten Bauteils liegt. Die lokalisierten Punkte entsprechen dann den Schnittpunkten zwischen der von der Kamera projizierten Linie und der Ebene, die der Oberfläche des Bauelements entspricht.

Der Schnittpunkt einer Gerade mit der Ebene wird wie im Folgenden berechnet werden:

$$\vec{r}_S = \vec{r}_1 + \frac{\vec{n} \cdot (\vec{r}_0 - \vec{r}_1)}{\vec{n} \cdot \vec{a}}\vec{a} \tag{3.16}$$

Der Vektor $\vec{r}_S$ ist der Ortsvektor des Schnittpunktes, $\vec{r}_0$ ist der Ortsvektor eines Punktes in der Ebene und $\vec{n}$ ist der Normalenvektor, der relativ zu $\vec{r}_0$ berechnet wird. Ein Vergleich der Gleichungen (3.13) und (3.16) zeigt, dass:

$$\vec{r}_S = \mathbf{X}, \qquad \lambda = \frac{\vec{n} \cdot (\vec{r}_0 - \vec{r}_1)}{\vec{n} \cdot \vec{a}} \tag{3.17}$$

Durch Eingabe der Variablen, die der Funktion (3.16) entsprechen, erhält man eine neue Funktion (3.18), die die Koordinaten eines Punktes in der Ebene des beschädigten

Bauelements in Abhängigkeit von seiner Position im Bild und den Parameter der Kamera und der ebenen Schadensfläche bestimmt.

$$\mathbf{X} = \mathbf{X_0} + \frac{\mathbf{n_e} \cdot (\mathbf{X_E} - \mathbf{X_0})}{\mathbf{n_e} \cdot (\mathbf{KR})^{-1}\mathbf{x}} (\mathbf{KR})^{-1}\mathbf{x} \qquad (3.18)$$

Es ist zu beachten, dass die Koordinatenabbildungsfunktion in (3.18) nur gültig ist, wenn die folgenden Einschränkungen berücksichtigt werden:

1. Das verwendete Kameramodell ist das einer perspektivischen Kamera, das 6 extrinsische Parameter und 5 intrinsische Parameter berücksichtigt. Dieses Modell berücksichtigt jedoch keine Bildverzerrungen. Daher muss für die Gültigkeit der Koordinatenzuordnung davon ausgegangen werden, dass es keine Verzerrungen gibt.
2. Die Schadensfläche muss eine ebene Fläche sein. Darüber hinaus müssen drei Punkte innerhalb der Ebene bekannt sein, um den Normalenvektor der Ebene zu berechnen.
3. Die in die Koordinatenabbildungsfunktion eingegebenen Punkte entsprechen den Koordinaten der vom neuronalen Netzmodell getroffenen Vorhersagen. Daher hängt die Genauigkeit der Abbildungsfunktion von der Effizienz des neuronalen Netzes ab.

Tabelle 4: Zusammenfassung der Komponenten der Koordinatenabbildungsfunktion

Variable	**Beschreibung**	
Schadenskoordinaten im dreidimensionalen Raum $\mathbf{X}$	Vektor mit 3 Koordinaten im globalen Koordinatensystem	$\mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$
Kameraposition $\mathbf{X_0}$	Vektor mit 3 Koordinaten im globalen Koordinatensystem	$\mathbf{X_0} = \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix}$
Rotationsmatrix $\mathbf{R}$	Drei Hauptdrehwinkel α, β, γ Drei Hauptdrehmatrizen $\mathbf{R_z}(\gamma), \mathbf{R_y}(\beta), \mathbf{R_x}(\alpha)$	$\mathbf{R_z}(\gamma) = \begin{bmatrix} \cos\gamma & -\sin\gamma & 1 \\ \sin\gamma & \cos\gamma & 1 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{R_y}(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$

	Rotationsmatrix ergibt sich aus der Matrixmultiplikation aller Hauptdrehmatrizen	$\mathbf{R_x}(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$ $\mathbf{R} = \mathbf{R_z}(\gamma) \cdot \mathbf{R_y}(\beta) \cdot \mathbf{R_x}(\alpha)$
Kalibrierungsmatrix $\mathbf{K}$	Intrinsische Kameraparameter: • Kamerakonstante c • Scherparameter s • Maßstabparameter m • Hauptpunktkoordinaten x_H, y_H	$\mathbf{K} = \begin{bmatrix} c & \text{cs} & x_H \\ s & c(1+m) & y_H \\ 0 & 0 & 1 \end{bmatrix}$
Schadenskoordinaten im Bild $\mathbf{x}$	Vektor mit 3 Koordinaten im lokalen Pixelkoordinaten $x, y, z, \quad z = 1$	$\mathbf{x} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$
Ortsvektor der Bauteilebene $\mathbf{X_E}$	Vektor mit 3 Koordinaten im Globalen Koordinatensystem	$\mathbf{X_E} = \begin{bmatrix} X_E \\ Y_E \\ Z_E \end{bmatrix}$
Normalenvektor der Bauteilebene $\mathbf{n_e}$	Vektor mit 3 Koordinaten im Globalen Koordinatensystem	$\mathbf{n_e} = \begin{bmatrix} n_{e,x} \\ n_{e,y} \\ n_{e,z} \end{bmatrix}$

4 PROTOTYPISCHE UMSETZUNG

Alle im Folgenden vorgestellten Anwendungen und Programme wurden auf einem Computer mit den folgenden Spezifikationen getestet:

Tabelle 5 Computerspezifikationen

Prozessor	Intel® Core™ i5-9300H CPU @ 2.40 GHz
Installierter RAM	16.0 GB
Grafikkarte	Nvidia Geforce GTX 1050 (Laptop)
Betriebssystem	Windows 11 Home

In Ermangelung eines Testbauwerks wird die Erfassung von Schadensbildern nicht in den Rahmen des Anwendungsprototyps einbezogen. Alle Bilder, die für das Training des neuronalen Netzes verwendet wurden, stammen aus dem von Çağlar Fırat Özgenel (2019) zur Verfügung gestellten Datensatz. Der für die Implementierung eines Crack-Detektors verwendete Code wurde von Priya Dwivedi (2019) geschrieben und von mir leicht modifiziert, damit er auf meinem Computer besser funktioniert.

Das Konzept einer Schadensdatenbank wird mit Hilfe eines Datenbankprototyps umgesetzt, die aus einer Reihe von Tabellen und Python-Dateien besteht, welche die Umwandlung und Übertragung von Informationen zwischen den Tabellen ermöglicht. Zur Demonstration des projektiven Punktzuordnungsprinzips werden 10 Testbilder mit Schäden verwendet. Trainieren eines Neuronalen Netzes für die Erkennung von Schäden.

4.1. IMPLEMENTIERUNG EINES RISSDETEKTORS IN BETONOBERFLÄCHEN

Das implementierte neuronale Netzmodell besteht aus einem Klassifizierungsmodell, das kleine viereckige Bildausschnitte analysiert und sie hinsichtlich des Vorhandenseins von Rissen als positiv oder negativ klassifiziert. Die Risserkennung erfolgt dann durch die Unterteilung eines hochauflösenden Bildes in kleine Felder, die dann durch das Modell klassifiziert werden können.

Der von Özgenel (2019) erstellte Datensatz enthält 40000 Bilder von 277 auf 277 Pixel für die Erkennung von Rissen in Betonoberflächen. Sie sind in 20000 Bilder mit Rissen und 20000 Bilder ohne Risse unterteilt. Aufgrund der begrenzten Rechenleistung werden nur 3000 Bilder pro Klasse verwendet.

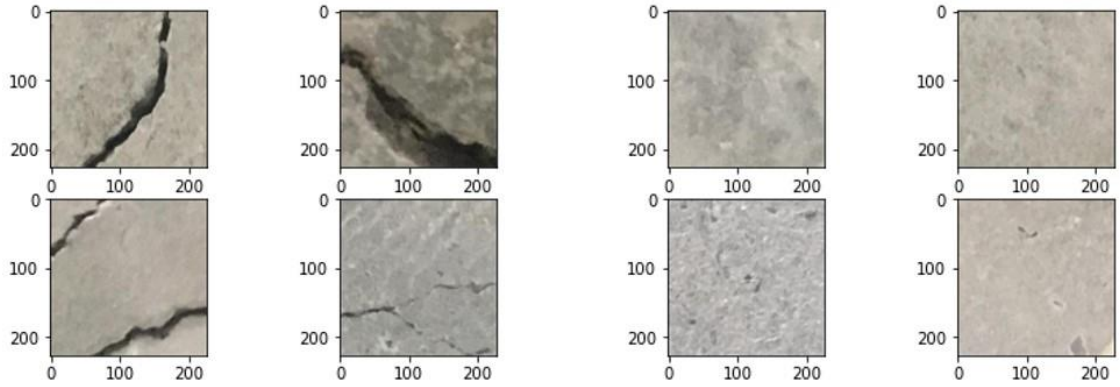


Abb. 20 Links: Beispiel für Trainingsbilder mit Rissen; Rechts: Beispiel für Trainingsbilder ohne Risse
Quelle: Eigene Darstellung

4.1.1. VORBEREITUNG DER TRAININGSDATEN

Der erste Schritt besteht darin, die Datensätze für das Training und die Validierung des neuronalen Netzes zu erstellen. Zunächst werden alle Bilder in ein Trainingsverzeichnis verschoben. Bilder mit Rissen werden in einem Unterverzeichnis der positiven Bilder gespeichert, während Bilder ohne Risse in einem Unterverzeichnis der negativen Bilder gespeichert werden. Dies sind die Labels, die das neuronale Netzmodell für die Klassifizierungsaufgabe verwenden wird.

Ein Teil der Trainingsbilder wird dann in den Validierungsdatensatz verschoben. Für großen Datensätze wird in der Regel ein Prozentsatz für die Aufteilung der Bilder in die beiden Datensätze verwendet, der zwischen 10 % und 20 % liegt. Ein alternatives Verfahren ist die zufällige Auswahl eines Prozentsatzes der Bilder für die Bewertung mit einer Wahrscheinlichkeit von 15 % bis 20 % (Dwivedi, 2019).

Quelltext 1 Erstellung des Trainingsdatensatzes
Quelle: (Dwivedi, 2019)

```
base_dir = cwd
files = os.listdir(base_dir)

def create_training_data(folder_name):
    train_dir = f"{base_dir}/train/{folder_name}"
    for f in files:
        search_object = re.search(folder_name, f)
        if search_object:
            shutil.move(f'{base_dir}/{folder_name}', train_dir)

create_training_data('Positive')
create_training_data('Negative')
```

Quelltext 2 Zufällige Auswahl von Bildern für den Validierungsdatensatz
Quelle: (Dwivedi, 2019)

```
os.makedirs('val/Positive')
os.makedirs('val/Negative')
positive_train = base_dir + "/train/Positive/"
positive_val = base_dir + "/val/Positive/"
negative_train = base_dir + "/train/Negative/"
negative_val = base_dir + "/val/Negative/"

positive_files = os.listdir(positive_train)
negative_files = os.listdir(negative_train)
for f in positive_files:
    if random.random() > 0.80:
        shutil.move(f'{positive_train}/{f}', positive_val)
for f in negative_files:
    if random.random() > 0.80:
        shutil.move(f'{negative_train}/{f}', negative_val)
```

4.1.2. LADEN EINES VORTRAINIERTEN MODELLS

Das für dieses Beispiel gewählte Basismodell ist das ResNet-50-Architekturmodell, das mit dem ImageNet-Datensatz trainiert wurde. Es basiert auf dem Prinzip des residualen Lernens (He, Zhang, Ren, & Sun, 2016) und beinhaltet insgesamt 50 Schichten und 23 Millionen Parameter. Am Basismodell werden zwei Änderungen vorgenommen: Zunächst werden die Parameterwerte des Basismodells eingefroren. Dann werden zwei neue, vollständig verbundene Schichten hinzugefügt. Die erste mit 128 Neuronen und ReLU-Aktivierungsfunktion, und die zweite Ausgangsschicht mit nur 2 Neuronen. Mit diesen Änderungen werden die Werte der Netzparameter während des Trainings nicht verändert, mit Ausnahme der Parameter in den letzten gerade hinzugefügten neuronalen Schichten. Damit reduziert sich die Gesamtzahl der trainierbaren Parameter und damit auch die Anzahl der Berechnungen. Die Ausgangsschicht mit 2 Neuronen entspricht der Anzahl der Klassen, mit denen das Modell trainiert wird (positiv oder negativ).

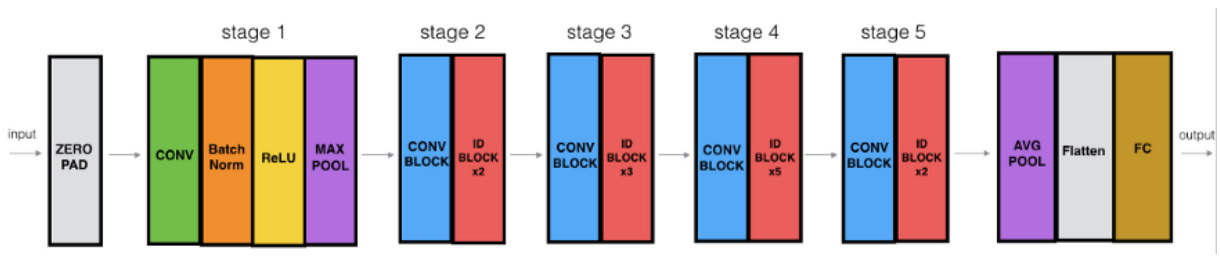


Abb. 21 Struktur des neuronalen Netzes des Typen ResNet-50
Quelle: (Dwivedi, 2019)

```
==========================================================================================
Total params: 23,770,562
Trainable params: 262,530
Non-trainable params: 23,508,032
Total mult-adds (G): 4.68
==========================================================================================
Input size (MB): 0.59
Forward/backward pass size (MB): 183.01
Params size (MB): 90.68
Estimated Total Size (MB): 274.28
==========================================================================================
```

Abb. 22 Anzahl der trainierbaren Parameter und Gesamtparameter

4.1.3. TRAINING UND ERGEBNISSE

Das Training des neuronalen Netzes erfolgt unter Berücksichtigung der folgenden Faktoren:

- Die Kostenfunktion ist die Kreuzentropie für die Multiklassen-Klassifikation.
- Der Optimierer ist Adam (Adaptive Moment Estimation), eine Erweiterung des stochastischen Gradientenabstiegs, bei dem die Lernrate eine abnehmende Funktion (Decay) in Abhängigkeit von einem Lernraten-Zeitplan ist.
- Die Lernrate nimmt in diesem Fall alle drei Epochen um den Faktor 0,1 ab.

Quelltext 3 Einstellung des Optimierers und der Kostenfunktion
Quelle: (Dwivedi, 2019)

```
# Define Optimizer and Loss Function
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet50.parameters())
# Decay LR by a factor of 0.1 every 3 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)
```

Für jede der sechs Epochen werden die Werte der Kostenfunktion und die Genauigkeit des Modells sowohl für den Trainings- als auch für den Validierungsdatensatz ermittelt. Die Ergebnisse sind wie folgt:

Tabelle 6 Verlust und Genauigkeit des trainierten Netzwerkmodells auf dem Trainings- und Validierungsdatensatz pro Epoche

Epoche	Training Set		Validation Set	
	Loss [-]	Accuracy [%]	Loss [-]	Accuracy [%]
1	0,5194	73,78	0,3015	94,59
2	0,2809	90,09	0,2028	92,60
3	0,2455	91,51	0,1802	94,19
4	0,2353	91,63	0,1721	94,82
5	0,2262	92,03	0,1713	94,67
6	0,2121	92,75	0,1687	94,98

Obwohl mit jeder Trainingsepoche nur die Parameter der letzten beiden Schichten des neuronalen Netzes geändert wurden, ist eine hohe Genauigkeit der Vorhersagen des Modells für die Trainingsbilder zu beobachten. Nach der ersten Trainingsepoche wird bereits eine Genauigkeit von 73,78 % erreicht, und zwischen der ersten und der dritten Epoche ist ein Anstieg der Genauigkeit von 17,73 % zu verzeichnen. Die endgültige Genauigkeit beträgt 92,75 %, was bedeutet, dass etwa 7 von 100 Trainingsbildern falsch klassifiziert werden. Was den Validierungsdatensatz betrifft, so liegt die Genauigkeit zwischen 92 % und 94 %. Ein hoher Genauigkeitswert in der Validierungsgruppe deutet darauf hin, dass das Modell nicht unter einer Überanpassung leidet. Die maximale Genauigkeit beträgt etwa 95 %. Das gleiche Netzmodell, das von Dwivedi (2019) mit allen Bildern des Trainingsdatensatzes trainiert wurde, ergibt einen Höchstwert von 98 %, was zeigt, dass die Effektivität des neuronalen Netzes von der Anzahl der Trainingsdaten abhängt.

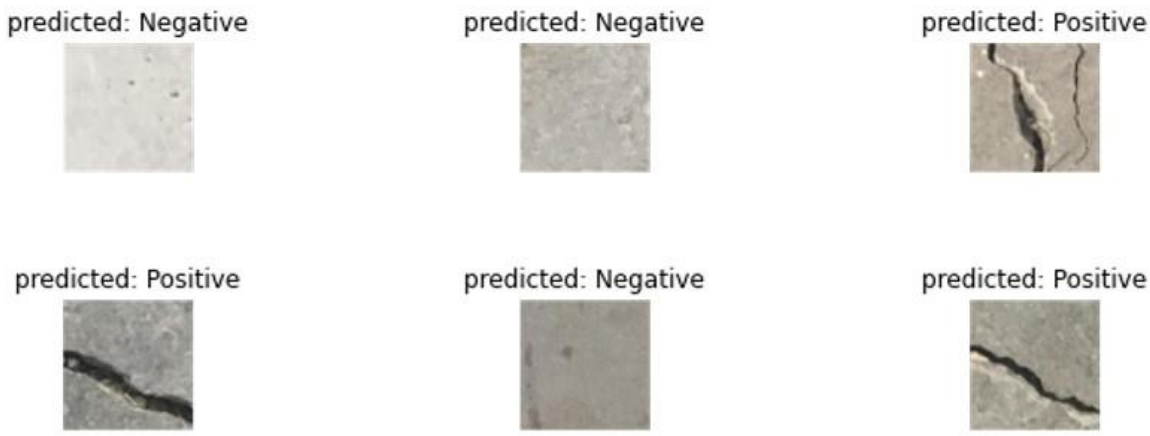


Abb. 23 Beispielhafte Vorhersagen des neuronalen Netzes für Trainingsdaten

4.1.4. ÜBERPRÜFUNG DER RISSERKENNUNG VON OBERFLÄCHEN

Das trainierte Modell ist in der Lage, Bilder von 277 x 277 Pixeln zu klassifizieren. Die bei einer Inspektion aufgenommenen Schadensbilder können jedoch unterschiedlich groß sein und sind vorzugsweise von hoher Auflösung. Anschließend muss das Eingabebild in quadratische Felder unterteilt werden, die vom Klassifikator ausgewertet werden können. Die Risserkennung besteht dann aus einer Funktion, die vom Klassifikator auf das Bild angewendet wird.

Quelltext 4 Funktion zur Klassifizierung eines quadratischen Kästchens
Quelle: (Dwivedi, 2019)

```
1  def predict(model, test_image, print_class = False):
2
3      transform = chosen_transforms['val']
```

```
    test_image_tensor = transform(test_image)

    if torch.cuda.is_available():
        test_image_tensor = test_image_tensor.view(1, 3, 227, 227).cuda()
    else:
        test_image_tensor = test_image_tensor.view(1, 3, 227, 227)

    with torch.no_grad():
        model.eval()
        # Model outputs log probabilities
        out = model(test_image_tensor)
        ps = torch.exp(out)
        topk, topclass = ps.topk(1, dim=1)
        class_name = idx_to_class[topclass.cpu().numpy()[0][0]]
        if print_class:
            print("Output class :  ", class_name)
    return class_name
```

Quelltext 5 Funktion zur Vorhersage von Rissen in einem Schadensbild beliebiger Größe

```
def predict_on_crops(input_image, height=227, width=227, save_crops = False):
    im = cv2.imread(input_image)
    imgheight, imgwidth, channels = im.shape
    k=0
    output_image = np.zeros_like(im)
    for i in range(0,imgheight,height):
        for j in range(0,imgwidth,width):
            a = im[i:i+height, j:j+width]
            ## discard image cropss that are not full size
            predicted_class = predict(base_model,Image.fromarray(a))
            ## save image
            file, ext = os.path.splitext(input_image)
            image_name = file.split('/')[-1]
            folder_name = 'out_' + image_name
            ## Put predicted class on the image
            if predicted_class == 'Positive':
                color = (0,0, 255)
            else:
                color = (0, 255, 0)
            cv2.putText(a, predicted_class, (50,50), cv2.FONT_HERSHEY_SIMPLEX , 0.7, color,
1, cv2.LINE_AA)
            b = np.zeros_like(a, dtype=np.uint8)
            b[:] = color
            add_img = cv2.addWeighted(a, 0.9, b, 0.1, 0)
            ## Save crops
            if save_crops:
                if not os.path.exists(os.path.join('real_images', folder_name)):
                    os.makedirs(os.path.join('real_images', folder_name))
                filename = os.path.join('real_images', folder_name,'img_{}.png'.format(k))
                cv2.imwrite(filename, add_img)
            output_image[i:i+height, j:j+width,:] = add_img
```

```
            k+=1
    ## Save output image
    cv2.imwrite(os.path.join('real_images','predictions', folder_name+ '.jpg'), output_im-
age)
    return output_image
```

Die Erkennungsfunktion wurde an drei Bildern mit unterschiedlichen Abmessungen getestet. Das Ergebnis besteht aus drei neuen Bildern, die in klassifizierte quadratische Flecken unterteilt sind, wobei rote Flecken das Vorhandensein eines Risses und grüne Flecken das Fehlen von Rissen anzeigen.

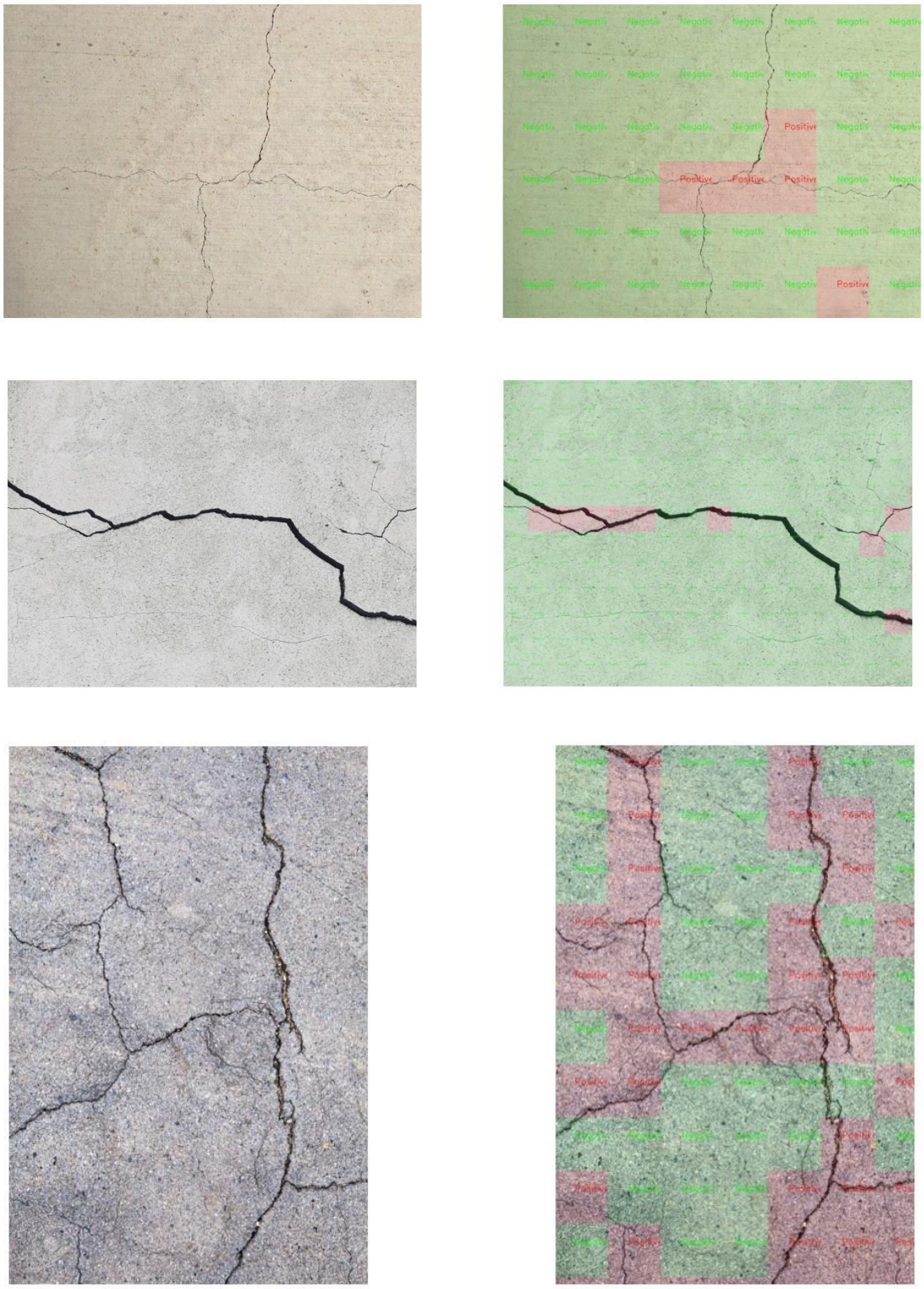


Abb. 24 Bilder zur Überprüfung der Risserkennungsfunktion: Bild 1 (oben) und Bild 2 (Mitte) sind beispielhafte Schadensbilder auf einer Betonoberfläche. Bild 3 (unten) zeigt Risse auf eine Landstraße, jedoch funktioniert der Rissdetektor auch auf dieser Oberfläche.

Die Schwachstellen des Detektors sind deutlich erkennbar. Nur ein kleiner Teil der Risse in Bild 1 wird als positiv eingestuft, während einer der Flecken ein falsches Positiv ist. In Bild 2, das etwa dreimal so groß ist wie Bild 1, ist ein ähnliches Verhalten wie in Bild 1 zu beobachten. Die Erkennungsgenauigkeit in Bild 3 ist erheblich besser, obwohl einige falsch klassifizierte Bereiche sichtbar sind. Eine höhere Anzahl von Trainingsbildern erhöht die Genauigkeit des Klassifikators und würde sich folglich auf die Genauigkeit der Risserkennung auswirken.

4.2. IMPLEMENTIERUNG EINES PROTOTYPS FÜR EINE BIM-BASIERTE SCHADENSDATENBANK

In Kapitel 3 wird eine Methode vorgeschlagen, um die 3D-Koordinaten von Punkten in einem globalen Koordinatensystem aus den Koordinaten der Projektion dieser Punkte in Bildern zu erhalten. Mit diesem Prinzip lassen sich die Koordinaten von lokalisierten Schäden durch maschinelles Lernen ableiten. Vor diesem Hintergrund wird ein Prototyp einer Schadensdatenbank vorgeschlagen, in der die durch maschinelles Lernen lokalisierten Schäden zusammen mit ihren Koordinaten im Koordinatensystem eines BIM-Modells dokumentiert werden.

Die für dieses Konzept erforderlichen Informationen sind:

- Eine Tabelle mit den Informationen zu den Gebäudeelementen und ihren Koordinaten. Diese Informationen sind für die Berechnung der Schadensfläche in Form eines Plans erforderlich.
- Tabelle mit den für jedes Foto ermittelten Kameraparametern. Sie ergeben sich aus der Kamerakalibrierung und dem P3P-Algorithmus.
- Eine Tabelle mit den durch das neuronale Netz ermittelten lokalen Schäden.

Es wird ein Verfahren zur Aufzeichnung von Informationen in einer Datenbanktabelle vorgeschlagen: Zunächst werden alle Informationen in einer einzigen Tabelle zusammengefasst, die den im lokalen Koordinatensystem der Bilder befindlichen Schadensinformationen entspricht. Im zweiten Schritt wird die in Kapitel (3.5) beschriebene Koordinatenzuordnungsfunktion implementiert, um eine neue Tabelle mit den lokalisierten Schäden zu erstellen, die nun mit ihren dreidimensionalen Koordinaten im BIM-Modell enthalten sind.

Jedes Foto ist mit einer Reihe von Kameraparametern verknüpft. Die vom neuronalen Netz gefundenen Schäden können wiederum mit dem Bild verknüpft werden, auf dem sie sich befinden. Beim Zusammenführen der beiden Tabellen muss jeder der lokalisierten Schäden durch eine Schadens-ID repräsentiert werden, die der Primärschlüssel der Tabelle sein wird. Die Bild-ID wird zu einem Fremdschlüssel, der die Schadenstabelle mit der Kameraparameter-Tabelle verbindet.

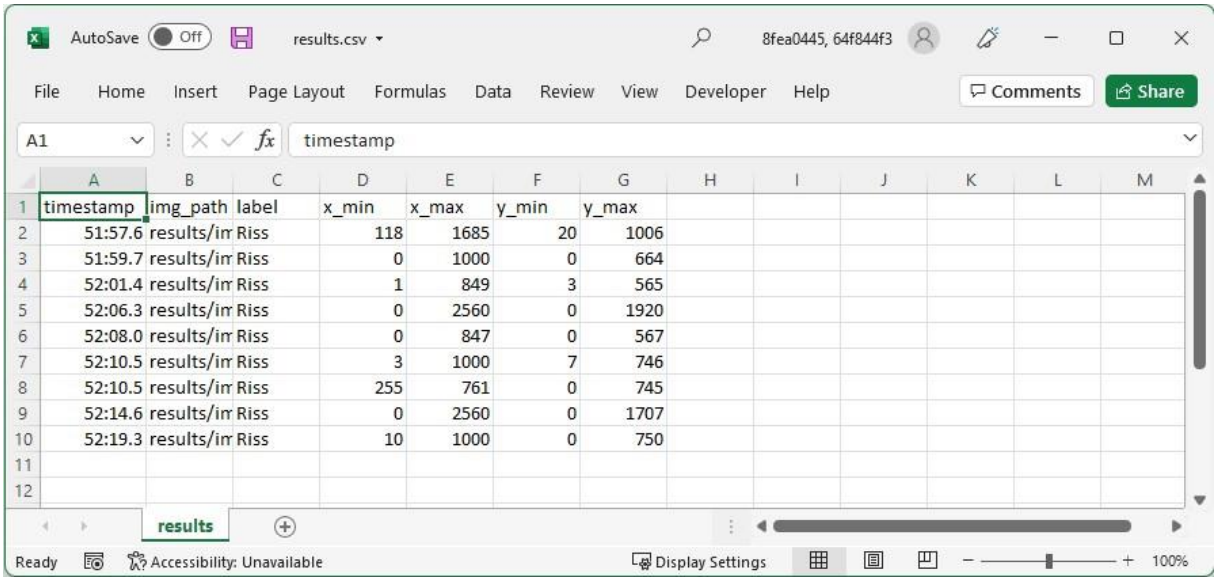
Da die Aufgabenstellung weder die Erstellung eines BIM-Modells noch die Verwendung eines bereits vorhandenen BIM-Modells vorschreibt, werden Koordinaten auf der Grundlage eines grundlegenden Beispiels für ein BIM-Modell verwendet. Aus demselben Grund wird davon ausgegangen, dass die Kameraparameter bekannt sind, und es werden Beispielwerte für sie verwendet. Für die Koordinaten der lokalisierten Schäden wurde ein Objekterkennungsmodell mit Begrenzungsrahmen verwendet, da die Koordinaten der einzelnen Begrenzungsrahmen leicht zu ermitteln sind. Die Trainingsdaten sind die gleichen wie im Kapitel 4.1.

Das Hauptziel dieses Abschnitts besteht darin, die Informationen zu sammeln, die für die Umsetzung der Koordinatenzuordnungsfunktion erforderlich sind. Zu diesem Zweck wurde ein Beispiel mit den folgenden Bedingungen vorbereitet:

- Es wird von einem einheitlichen Kameramodell ausgegangen, dass die extrinsischen Kameraparameter für jedes Foto berücksichtigt. Bei den intrinsischen Parametern wird für den Kamerakoeffizienten der Wert 1 angenommen, während die anderen Parameter den Wert 0 haben.
- Jeder lokalisierte Schaden wird durch einen durch vier Punkte definierten Begrenzungskasten beschrieben. Die Koordinaten der Punkte ergeben sich aus der Zusammensetzung der Koordinaten x_{min}, x_{max}, y_{min}, y_{max}.
    - $P_1(x_{min}, y_{min})$
    - $P_2(x_{max}, y_{min})$
    - $P_3(x_{min}, y_{max})$
    - $P_4(x_{max}, y_{max})$

4.2.1. ZUSAMMENFÜHRUNG DER TABELLEN

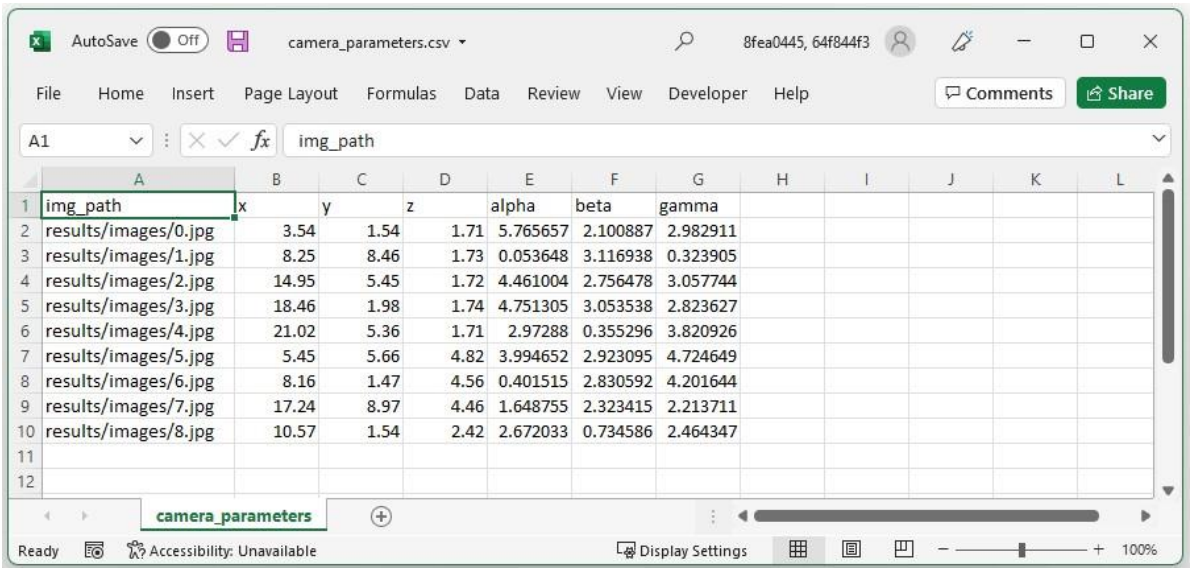
Die Überprüfung der Testbilder durch das neuronale Netz ergab folgende Ergebnisse:



	timestamp	img_path	label	x_min	x_max	y_min	y_max
2	51:57.6	results/im	Riss	118	1685	20	1006
3	51:59.7	results/im	Riss	0	1000	0	664
4	52:01.4	results/im	Riss	1	849	3	565
5	52:06.3	results/im	Riss	0	2560	0	1920
6	52:08.0	results/im	Riss	0	847	0	567
7	52:10.5	results/im	Riss	3	1000	7	746
8	52:10.5	results/im	Riss	255	761	0	745
9	52:14.6	results/im	Riss	0	2560	0	1707
10	52:19.3	results/im	Riss	10	1000	0	750

Abb. 25 Ausgabe eines Object-Detection-Modell als Tabelle: Zeitstempel, Bildpfad, vorhergesagte Label durch Klassifikationaufgabe, Bounding-Box-Koordinaten aus der Lokalisierungsaufgabe

Für die Kameraparameter stehen die folgenden Werte zur Verfügung:



	img_path	x	y	z	alpha	beta	gamma
2	results/images/0.jpg	3.54	1.54	1.71	5.765657	2.100887	2.982911
3	results/images/1.jpg	8.25	8.46	1.73	0.053648	3.116938	0.323905
4	results/images/2.jpg	14.95	5.45	1.72	4.461004	2.756478	3.057744
5	results/images/3.jpg	18.46	1.98	1.74	4.751305	3.053538	2.823627
6	results/images/4.jpg	21.02	5.36	1.71	2.97288	0.355296	3.820926
7	results/images/5.jpg	5.45	5.66	4.82	3.994652	2.923095	4.724649
8	results/images/6.jpg	8.16	1.47	4.56	0.401515	2.830592	4.201644
9	results/images/7.jpg	17.24	8.97	4.46	1.648755	2.323415	2.213711
10	results/images/8.jpg	10.57	1.54	2.42	2.672033	0.734586	2.464347

Abb. 26 Tabelle mit Bildinformationen: Bildpfad, X-, Y- und Z-Koordinaten der Kameraposition und Rotationswinkeln Alpha, Beta und Gamma

Beide Tabellen werden lokal als Dateien im CSV-Format gespeichert. Um beide Tabellen zusammenzuführen, ist Code erforderlich, der beide Tabellen liest und den Inhalt in Listen speichert.

Quelltext 6 Erstellung eines neuen csv-Dokuments und Importierung der Bild- und Schadensinformationstabellen

```
# open the file in the write mode
f = open('damage_in_images.csv', 'w')

# create the csv writer
writer = csv.writer(f)

# write a row to the csv file
writer.writerow(['img_path','la-
bel','x_min','x_max','y_min','y_max','x_cam','y_cam','z_cam','alpha','beta','gamma'])

camera_params = []
with open('camera_parameters/camera_parameters.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        camera_params.append(row)

dd_results = []
with open('results/results.csv','r') as file:
    reader = csv.reader(file)
    for row in reader:
        dd_results.append(row)
```

Dann wird eine neue Liste mit allen Informationen erstellt. Schließlich wird eine neue Tabelle erstellt, in die der Inhalt der Liste mit den vollständigen Schadensinformationen geschrieben wird.

Quelltext 7 Erstellung einer neuen Liste von Schäden in Bildern und Export als csv-Dokument

```
damage_in_images = [['img_path','la-
bel','x_min','x_max','y_min','y_max','x_cam','y_cam','z_cam','alpha','beta','gamma']]
for dd_row in range(1,len(dd_results)):
    for cp_row in range(1,len(camera_params)):
        if dd_results[dd_row][1] == camera_params[cp_row][0]:
            #print(dd_results[dd_row][1])
            line = [dd_results[dd_row][1],dd_results[dd_row][2],dd_re-
sults[dd_row][3],dd_results[dd_row][4],dd_results[dd_row][5],dd_results[dd_row][6],cam-
era_params[cp_row][1],camera_params[cp_row][2],camera_params[cp_row][3],cam-
era_params[cp_row][4],camera_params[cp_row][5],camera_params[cp_row][6]]
            damage_in_images.append(line)

# open the file in the write mode
f = open('damage_in_images.csv', 'w', newline='')
# create the csv writer
writer = csv.writer(f)
writer.writerows(damage_in_images)
```

```
# write a row to the csv file
#for row in damage_in_images:
#    writer.writerow(row)
# close the file
f.close()
```

4.2.2. IMPLEMENTIERUNG DER MAPPING-FUNKTION

Zunächst werden die Schadensinformationen aus der zusammengeführten Tabelle importiert. Die ersten beiden Spalten, die dem Bildpfad und der Schadensart entsprechen, werden in einer separaten Liste gespeichert. Die übrigen Spalten werden in einem Array zur Verwendung in der Funktion gespeichert.

Quelltext 8 Codefunktion zum Abrufen von Schadensinformationen

```
# Import damage information
# Step 1: import the CSV File with the extrinsics
def get_damage_information():
    data = pd.read_csv(r'damage_in_images.csv')
    dinfo_df = pd.DataFrame(data, col-
umns=['x_min','x_max','y_min','y_max','x_cam','y_cam','z_cam','alpha','beta','gamma'])

    # Step 2: transform dataframe to numpy array
    dinfo_np = dinfo_df.to_numpy()
    return dinfo_np

def get_damage_classes():
    data = pd.read_csv(r'damage_in_images.csv')
    dclasses_df = pd.DataFrame(data, columns=['img_path','label'])

    return dclasses_df
```

Eine Funktion ist erforderlich, um die Rotationsmatrix zu erstellen, die später verwendet wird.

Quelltext 9 Funktion für die Konstruktion der Rotationsmatrix nach drei Hauptrotationen

```
# Build the rotation matrix
# Extrinsic rotation whose (improper) Euler angles are alpha, beta and gamma about axes x,
y and z (https://en.wikipedia.org/wiki/Rotation_matrix#General_rotations)
def build_rotationmatrix(a, b, c):
    z_rotation = np.array([[np.cos(c), -np.sin(c), 0],[np.sin(c), np.cos(c), 0],[0, 0, 1]],
np.float32)
    y_rotation = np.array([[np.cos(b), 0, np.sin(b)],[0, 1, 0],[-np.sin(b), 0, np.cos(b)]],
np.float32)
    x_rotation = np.array([[1, 0, 0],[0, np.cos(a), -np.sin(a)],[0, np.sin(a), np.cos(a)]],
np.float32)
    rotation_matrix = z_rotation * y_rotation * x_rotation
    return rotation_matrix
```

Anschließend wird die Mapping-Funktion implementiert:

```
# Mapping function
# Attributes: Camera Positionmatrix X0, Rotationmatrix R, Calibrationmatrix K, Point coordinates x
def point_mapping():
    # Get the calibration matrix FOR AN IDEAL CAMERA (c=1, other parameters = 0)
    calibration_matrix = get_calibration_matrix(1, 0, 0, 0, 0)
    damage_info_np = get_damage_information()

    damage_3DCoordinates = []
    for x in damage_info_np:
        # Rotation Matrix, Center of Damage Box
        rotation_matrix = build_rotationmatrix(x[7],x[8],x[9])
        box_centerpoint = [(x[0]+x[1])/2,(x[2]+x[3])/2,1]
        #print(box_centerpoint)
        #print(rotation_matrix)

        # Matrix multiplication and Compute inverse matrix
        matmul = np.matmul(calibration_matrix,rotation_matrix)
        inv_matmul = np.linalg.inv(matmul)

        # Mapping function
        newbox_centerpoint = np.matmul(inv_matmul,box_centerpoint)

        # Reshape coordinate vector
        #row = np.array(newbox_centerpoint).reshape(1,3)

        # Append to point list
        damage_3DCoordinates.append(newbox_centerpoint)

    # Reshape matrix to list
    damage_3DCoordinates = np.array(damage_3DCoordinates).reshape(9,3).tolist()

    return damage_3DCoordinates
```

Schließlich wird alles zusammengefügt, um eine neue Tabelle zu erstellen, in der die Schäden in dreidimensionalen Koordinaten angegeben sind. Die erstellte Liste enthält die Koordinaten $\Delta X, \Delta Y, \Delta Z$ entsprechend den Koordinaten des Richtungsvektors des von der Kamera projizierten Strahls.

$$\Delta\mathbf{X} = (\mathbf{KR})^{-1}\mathbf{x} = \begin{bmatrix}\Delta X\\ \Delta Y\\ \Delta Z\end{bmatrix} \qquad (4.1)$$

Quelltext 10 Quellcode zur Erzeugung einer Schadenstabelle mit 3D-Koordinaten als csv-Datei

```
# Output 3D Information in a new csv File
def outputCSVFile():
    # Import classes and 3D Coordinate values
    classes_df = get_damage_classes()      # Dataframe
    camera_np = get_damage_information()   # Numpy Array
    delta_np = point_mapping()             # Numpy Array

    # Transform "classes" Dataframe into a list
    classes_list = classes_df.values.tolist()

    # Get Camera position
    data = pd.read_csv(r'damage_in_images.csv')
    df = pd.DataFrame(data, columns=['x_cam','y_cam','z_cam'])
    camera_np = df.to_numpy()

    damage_in_world = [['img_path', 'label', 'X_cam', 'Y_cam', 'Z_cam', 'X_delta', 'Y_del-
ta', 'Z_delta']]
    # damage_in_world = [['img_path', 'label', 'X_cam', 'Y_cam', 'Z_cam']]
    for i in range(len(classes_list)):
        #print(classes_list[i])
        for j in range(len(camera_np)):
            for k in range(len(delta_np)):
                if i==j and i==k and j==k:
                    print(delta_np[k][0])
                    row = [classes_list[i][0],classes_list[i][1], camera_np[j][0], cam-
era_np[j][1], camera_np[j][2], delta_np[k][0], delta_np[k][1], delta_np[k][2]]
                    damage_in_world.append(row)
                else:
                    continue
```

Die erzeugte Tabelle ist das Ergebnis der Anwendung der Mapping-Funktion und enthält die Positions- und Richtungsvektoren der Linie, die den von der Kamera auf den Punkt im Bild projizierten Strahl darstellt. Es fehlt die Berechnung des skalaren Parameters λ, der anhand des Schnittpunkts zwischen dem von der Kamera projizierten Strahl und der Ebene der Schadensoberfläche bestimmt wird. Aufgrund des Fehlens eines BIM-Modells wurde dieser letzte Schritt in diesem Beispiel jedoch nicht umgesetzt.

5 SCHLUSSFOLGERUNGEN

5.1. ZUSAMMENFASSUNG

- Im theoretischen Teil wurde ein Überblick über maschinelles Lernen, überwachtes Lernen und neuronale Netze gegeben. Der Schwerpunkt lag dabei *auf Convolutional Neural* Networks und ihrer Verwendung bei der Objekterkennung. Schließlich wurden Forschungsarbeiten zur Erkennung verschiedener Arten von Schäden mit Hilfe neuronaler Netze erwähnt, z. B. Risse und Abplatzungen in Beton und Korrosion von Baustahl.
- Im praktischen Teil wurde ein Arbeitsablauf für ein Schadenserkennungssystem vorgeschlagen, der die Erfassung von Schadensbildern während einer Bauinspektion, die Implementierung von maschinellem Lernen für die Klassifizierung und Lokalisierung von Schäden in den Bildern und eine 2D-nach-3D-Koordinatentransformationsfunktion für die Lokalisierung von Schäden in einem BIM-Modell umfasst. Dieser Prozess wird mit Hilfe eines eEPK-Diagramms beschrieben.
- Es wurde ein maschinelles Lernmodell auf der Grundlage eines neuronalen Faltungsnetzwerks vom Typ Resnet-50 trainiert, das Bildausschnitte von 277 x 277 Pixeln als positiv oder negativ klassifiziert, je nachdem, ob ein Riss innerhalb des Ausschnitts vorhanden ist oder nicht. Nach 6 Trainingsepochen und unter Verwendung von insgesamt 6000 Trainingsbildern erreichte die Genauigkeit des Klassifikatormodells auf den Validierungsdaten einen Höchstwert von 94,98 %.
- Die Risserkennung in der Bildgebung funktioniert unregelmäßig. Bei den falsch eingestuften Abschnitten handelt es sich größtenteils um falsch negative Ergebnisse. Dies ist jedoch ein zu erwartendes Ergebnis, da nicht der gesamte Trainingsdatensatz verwendet wurde. Das Hauptziel dieses Abschnitts bestand darin, die Leistungsfähigkeit eines Schadensdetektors zu demonstrieren, wobei die Optimierung außer Acht gelassen wurde.
- Schließlich wurde die Koordinatenabbildungsfunktion auf einen Testdatensatz angewendet, um eine mögliche Darstellung der lokalisierten Schadensinformationen in Tabellenform zu zeigen. Dies ist ein Präzedenzfall für die Erstellung einer Datenbank mit ihrer genauen Position innerhalb eines BIM-Modells.

5.2. DISKUSSION

Zu Beginn dieser Arbeit wurde besonderer Wert auf den Einsatz von maschinellem Lernen zur Schadenserkennung bei der Inspektion gelegt. Bei der Entwicklung eines Arbeitsablaufs für die Klassifizierung und Lokalisierung von Schäden, die bei der Bauwerksinspektion gefunden wurden, traten jedoch zusätzliche Unbekannte auf, wie z. B. die genaue Lokalisierung der gefundenen Schäden innerhalb des BIM-Modells eines Bauwerks und die Beschaffung der erforderlichen Informationen zur Durchführung dieses Prozesses. Während die Implementierung eines Schadensdetektors allein in eine Datenbank wie die der SIB-Bauwerke integriert werden kann, hielt ich es für notwendig, die Idee der Interpretation aller von einem Machine-Learning-Modell bereitgestellten Informationen zu untersuchen. Die Bestimmung der genauen Koordinaten einer Schadensart ermöglicht die Bestimmung ihrer Geometrie, was für die Beurteilung der Schwere des Schadens sehr wichtig ist. Darüber hinaus eröffnet es die Möglichkeit, die gefundenen Schäden als Elemente in das BIM-Modell des inspizierten Gebäudes einzubinden.

Für die Erfassung von Schadensbildern wurde die Verwendung einer kalibrierten Kamera für die Anwendung der projektiven Geometrie auf die von einem neuronalen Netz erhaltenen Schadenskoordinaten in Betracht gezogen. Obwohl die Methoden zur Kalibrierung und damit zur Ermittlung der Kameraparameter in Kapitel 3.3 theoretisch erläutert werden, wird ein System benötigt, mit dem diese Parameter in eine Datenbank aufgenommen werden können, entweder direkt oder als Metainformationen in den während der Inspektion aufgenommenen Fotos. Es ist zu beachten, dass diese Kameraparameter ständig aktualisiert werden müssen, da sie von der externen und internen Position der Kamera abhängen. Es wird davon ausgegangen, dass die Kamera ständig in Bewegung ist, weil der Inspektor sich mit ihr bewegt. Daher wird ein System benötigt, das sowohl die Position als auch die Drehung der Kamera bei der Aufnahme von Bildern ständig anpasst. Der P3P-Algorithmus (Kapitel 3.3.2) ist eine auf der Photogrammetrie basierende Methode zur Ermittlung der extrinsischen Parameter, die jedoch die Kenntnis der Koordinaten von drei Punkten in jedem Bild erfordert. Auf diese Weise könnte ein System entwickelt werden, das die extrinsischen Parameter nach der Inspektion berechnet. Die Ermittlung dieser Punkte für jedes einzelne Foto könnte jedoch eine zeitaufwändige Aufgabe sein. Eine Alternative könnte sein, die eingebauten Fähigkeiten einer modernen Kamera, wie GPS und Gyroskopfunktion, zu nutzen. Bei den intrinsischen Parametern ist es wichtig, die Faktoren zu berücksichtigen, die die innere Position der Kamera beeinflussen. Eine kalibrierte metrische Kamera gilt als stabil, wenn sich die intrinsischen Parameter zwischen dem Zeitpunkt der Kalibrierung und nach der Verwendung nicht drastisch ändern. Faktoren wie Temperaturschwankungen oder Erschütterungen der Kammer können jedoch die innere Position der Kammer beeinflussen (Förstner & Wrobel, 2016, S. 696).

Idealerweise sollten nur zwei Kalibrierungen durchgeführt werden, eine vor und eine nach der Inspektion, um zu bestätigen, dass sich die intrinsischen Parameter nicht zu stark verändert haben. Eine praktische Anwendung zu diesem Thema wurde nicht durchgeführt, da es in der Aufgabenstellung ursprünglich nicht vorgesehen war.

Im Rahmen der Implementierung eines neuronalen Netzes zur Erkennung von Bauschäden war der größte Nachteil, der Mangel an öffentlich verfügbaren Datensätzen für das Training des neuronalen Netzes. Die Option, einen Datensatz von Schadensbildern zu erstellen, war nicht realisierbar, da es zu Beginn der Projektarbeit keine Testkonstruktion gab, um die Bilder zu sammeln. Die Beschriftung von Bildern ist ebenfalls zeitaufwändig und hängt von der Art der auszuführenden Machine Learning-Aufgabe ab (Sliding Window, Objekterkennung, semantische Segmentierung). Da der Schwerpunkt dieser Arbeit auf der Implementierung eines durchgängigen maschinellen Lernmodells lag, entschied ich mich, den von Ozgenel (2019) erstellten Datensatz für Betonrisse und ein Sliding-Window-Modell zu verwenden, da der Datensatz zu diesem Zweck bereits beschriftet war. Was die Leistung des implementierten Schadensdetektors betrifft, so gibt es noch viel Raum für Optimierungen. Die Verwendung von mehr Trainingsbildern wirkt sich direkt auf die Qualität der Vorhersagen eines neuronalen Netzmodells aus, wie ein Vergleich der Ergebnisse aus Abschnitt 4 mit den Ergebnissen des von Dwivedi (2019) trainierten Modells zeigt. Weitere wichtige Faktoren für die Optimierung sind die Wahl der Trainingshyperparameter, die Gesamtzahl der trainierbaren Netzparameter und die verwendete Architektur des neuronalen Netzes.

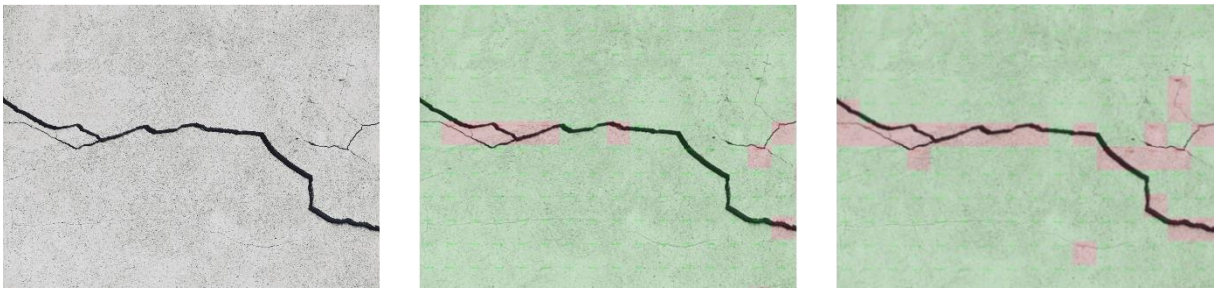


Abb. 27 Vergleich der Ergebnisse der Risserkennungsmodell in Abhängigkeit von der Anzahl der Trainingsbilder. Links: Originalbild; Mitte: Neuronales Netz trainiert mit 6000 Bildern; Rechts: Neuronales Netz trainiert mit 40000 Bildern (Dwivedi, 2019)

Schließlich dient die Erstellung von Tabellen mit den dreidimensionalen Koordinaten dazu, eine Vorstellung von der möglichen Visualisierung der Schadensinformationen in einer Datenbank zu vermitteln. Der Code, mit dem die Zuordnungsfunktion implementiert wird, ist eher rudimentär, aber er funktioniert in dem in Kapitel 4 genannten Beispiel. Die Herausforderung, die ich in diesem Fall sehe, ist die Erstellung einer Schadensdatenbank, die diese Informationen optimal integriert. Die Tabellen im CSV-Format können in

eine Datenbank mit einer Datenbanksprache wie SQL importiert werden. Durch den Zugriff auf das BIM-Modell mit den Geometrien der Bauelemente wäre es möglich, jeden der gefundenen Schäden mit dem entsprechenden beschädigten Element zu verknüpfen und so eine in das BIM-Modell integrierte Schadensdatenbank zu erhalten.

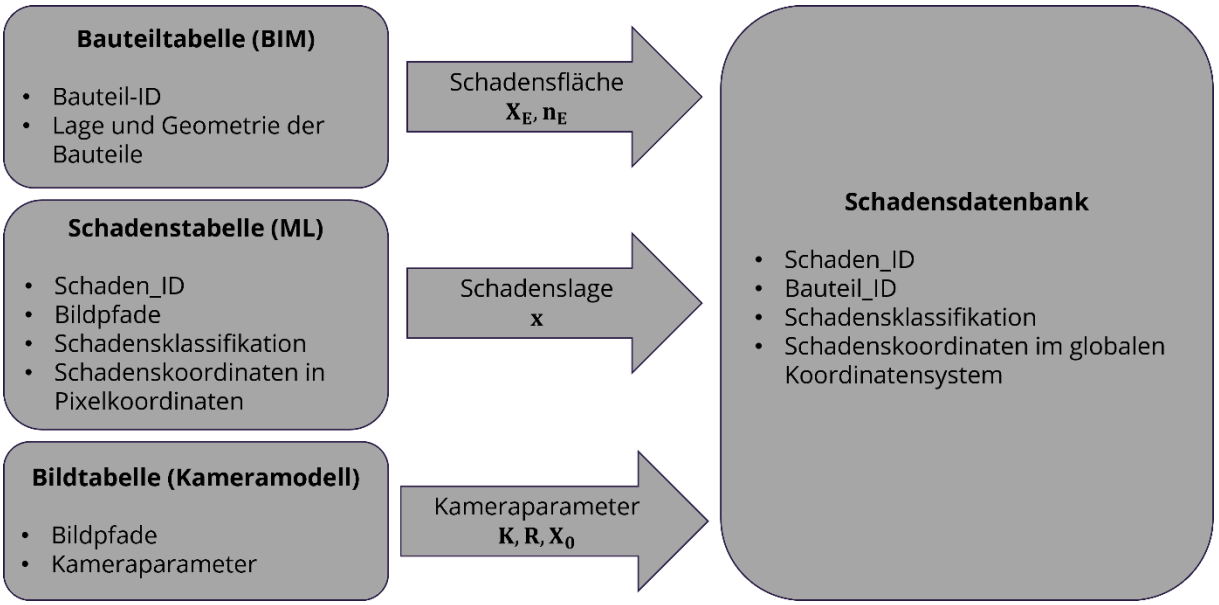


Abb. 28 Diagramm, das die Übertragung von Informationen aus dem BIM-Modell, die Tabelle der durch maschinelles Lernen gefundenen Schäden und die Tabelle mit den Kameraparametern darstellt.
Quelle: Eigene Darstellung

6 AUSBLICK

- Der Computer, der für das Training der neuronalen Netzmodelle in dieser Arbeit verwendet wird, ist in Bezug auf Rechenleistung und Speicherplatz begrenzt. Dies führte dazu, dass eine Schulungssitzung viel Zeit in Anspruch nahm. Aufgrund des fehlenden Arbeitsspeichers auf der Grafikkarte war der Einsatz von parallelem Rechnen mit einem Tool wie Nvida CUDA nicht möglich. Die Verwendung eines Computers mit besseren Funktionen ist eine einfache Lösung. Eine weitere Möglichkeit, dieses Problem zu umgehen, ist die Nutzung kostenloser Cloud-Server wie Google Collab oder Kaggle.
- Es wäre interessant, das Konzept der Koordinatenabbildung in einem Testgebäude zu testen, da es in dieser Arbeit nur theoretisch vorgeschlagen wurde. Auf der Grundlage von Experimenten mit verschiedenen Kameratypen und Kalibrierungsmethoden könnte ein System entwickelt werden, das die Kameraparameter als Metainformationen für jedes der Inspektionsfotos berechnet und speichert. Diese Informationen könnten dann in einer Tabelle protokolliert werden, ähnlich dem im praktischen Teil dieses Papiers vorgestellten Beispiel.
- Wenn ein BIM-Modell der zu untersuchenden Struktur zur Verfügung steht, ist eine vollständige Entwicklung des in dieser Arbeit vorgeschlagenen Arbeitsablaufs ebenfalls möglich. Für eine vollständige Abbildung der Koordinaten der lokalisierten Punkte wird ein Algorithmus benötigt, der die Oberflächen der beschädigten Gebäudeelemente aus ihrer Geometrie im BIM-Modell berechnet. Bei flächigen Bauteilen, wie Wänden oder Platten, ist eine Darstellung der Schadensfläche mittels eines Plans sehr gut geeignet.
- Eine durchgängige Entwicklung, die die Erfassung der Kameraparameter, die Anwendung des maschinellen Lernens zur Schadenserkennung und die Abbildung der Koordinaten im BIM-Modell umfasst, würde es ermöglichen, die Genauigkeit dieser Methode zur Schadenserkennung zu visualisieren und einen Eindruck von ihrer Durchführbarkeit bei der Inspektion von Baustellen zu vermitteln.
- Während der Entwicklung dieser Arbeit wurde der Einsatz von GPS und Tiefenkameras zur Schadenslokalisierung in Betracht gezogen. Dies könnte den Prozess der Kamerakalibrierung erleichtern, da die Kameraposition automatisch anhand von GPS-Koordinaten berechnet würde, während eine Tiefenkarte den Abstand zwischen den Pixeln des Fotos und der Kamera liefern würde.

V LITERATURVERZEICHNIS

Arthus, M., Alabassy, M., & Koch, C. (2021). IFC based Framework for Generating, Modeling and Visualizing Spalling Defect Geometries. *EG-ICE 2021 Workshop on Intelligent Computing in Engineering*, 176-186.

Atha, D. J., & Jahanshahi, M. R. (2018). Evaluation of Deep Learning Approaches Based on Convolutional Neural Networks for Corrosion Detection. *Structural Health Monitoring Vol. 17(5)*, 1110-1128.

Beckman, G. H., Polyzois, D., & Cha, Y.-J. (2019). Deep Learning-Based Automatic Volumetric Damage Quantification Using Depth Camera. *Automation in Construction 99*, 114-124.

Bolya, D., Zhou, C., Xiao, F., & Lee, Y. J. (2019). YOLACT: Real-Time Instance Segmentation. *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 9157-9166.

Bornholdt, M., Petersen, M., & Goedeke, H. (2021). *BIM in der Instandsetzungsplanung.* Ernst & Sohn.

Bundesministerium für Verkehr, B. u. (2013). Bauwerksprüfung nach DIN 1076; Bedeutung, Organisation, Kosten. 74.

Cha, Y.-J., Choi, W., & Büyüköztürk, O. (2017). Deep Learning-Based Crack Damage Detection Using Convolutional Neural Networks. *Computer-Aided Civil and Infrastructure Engineering 32*, 361-378.

Cha, Y.-J., Choi, W., Suh, G., Mahmoudkhani, S., & Büyüköztürk, O. (2018). Autonomous Structural Visual Inspection Using Region-Based Deep Learning for Detecting Multiple Damage Types. *Computer-Aided Civil and Infrastructure Engineering 33*, 731-747.

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2018). DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 834-848.

Dong, C.-Z., & Catbas, F. N. (2021). A review of computer vision-based structural health monitoring at local and global levels. *Structural Health Monitoring Vol. 20(2)*, 692-743.

Dorafshan, S., Thomas, R. J., & Maguire, M. (2018). Comparison of Deep Convolutional Neural Networks and Edge Detectors for Image-based Crack Detection in Concrete. *Construction and Building Materials 186*, 1031-1045.

Dwivedi, P. (31. December 2019). *Detection of Surface Cracks in Concrete Structures using Deep Learning.* Von Towards Data Science: https://towardsdatascience.com/detection-of-surface-cracks-in-concrete-structures-using-deep-learning-f8f85cd8ac8b abgerufen

Förstner, W., & Wrobel, B. P. (2016). *Photogrammetric Computer Vision.* Springer.

Frochte, J. (2018). *Machinelles Lernen - Grundlagen und Algorithmen in Python.* Hanser.

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow.* O'Reilly.

Grimson, E., Guttag, J., & Bell, A. (2016). *MIT 6.0002 Introduction to Computational Thinking and Data Science.* Massachusetts Institute of Technology: MIT OpenCourseWare. Von https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0002-introduction-to-computational-thinking-and-data-science-fall-2016 abgerufen

Grunert, J. A. (1841). Das Pothenot'sche Problem in erweiterter Gestalt nebst über seine Anwendungen in der Geodäsie. *Grunerts Archiv für Mathematik und Physik 1*, 238–248.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770-778.

Hubel, D. H. (1959). Single Unit Activity in Striate Cortex of Unrestrained Cats. *The Journal of Physiology 147*, 226-238.

Hubel, D. H., & Wiesel, T. N. (1959). Receptive Fields of Single Neurons in the Cat's Striate Cortex. *The Journal of Physiology 148*, 574-591.

Hubel, D. H., & Wiesel, T. N. (1968). Receptive Fields and Functional Architecture of Monkey Striate Cortex. *The Journal of Physiology 195*, 215-243.

Isailović, D., Stojanovic, V., Trapp, M., Richter, R., & Hajdin, R. (2020). Bridge Damage: Detection, IFC-based semantic enrichment and visualization. *Automation in Construction 112 (2020) 103088*, 1-22.

Katsamenis, I., Protopapadakis, E., Doulamis, A., Doulamis, N., & Voulodimos, A. (2020). Pixel-Level Corrosion Detection on Metal Constructions by Fusion of Deep

Learning Semantic and Contour Segmentation. *Advances in Visual Computing ISVC 2020*, 160-169.

Kim, H., Ahn, E., Shin, M., & Sim, S.-H. (2019). Crack and Noncrack Classification from Concrete Surface Images Using Machine Learning. *Structural Health Monitoring Vol.18(3)*, 725-738.

Kim, I.-H., Jeon, H., Baek, S.-C., Hong, W.-H., & Jung, H.-J. (2018 (6)). Application of Crack Identification Techniques for an Aging Concrete Bridge Inspection Using an Unmanned Aerial Vehicle. *Sensors*, 1881.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Proceedings of the 25th International Conference on Neural Information Processing Systems 1*, 1097-1105.

LeCun, Y. (1989). *Generalization and Network Design Strategies.* Toronto: Department of Computer Science, University of Toronto.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE 86, no.11*, 2278-2324.

Lee, J. Y., Sim, C., Detweiler, C., & Barnes, B. (2019). Computer-Vision Based UAV Inspection for Steel Bridge Connections. *Structural Health Monitoring*, 3152-3159.

Li, D., Cong, A., & Guo, S. (2019). Sewer Damage Detection from Imbalanced CCTV Inspection Data Using Deep Convolutional Neural Networks with Hierarchical Classification. *Automation in Construction 101*, 199-208.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. *Proceedings of the 14th European Conference on Computer Vision 1*, 21-37.

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully Convolutional Networks for Semantic Segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3431-3440.

Marxer, M., Bach, C., & Keferstein, C. P. (2021). *Fertigungsmesstechnik.* Springer Vieweg.

McCulloch, W. S., & Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biology 5, no. 4*, 115-133.

Nash, W., Drummond, T., & Birbilis, N. (2018). Quantity beats quality for semantic segmentation of corrosion in images. Von https://arxiv.org/abs/1807.03138 abgerufen

Ozgenel, C. F. (2019). *Concrete Crack Images for Classification.* Mendeley Data, v2. doi:10.17632/5y9wdsg2zt.2

Pohl, K. (2020). *Wissensbasiertes BIM-Schadensmanagement mittels Web Ontologien.* Institut für Bauinformatik; Technische Universität Dresden.

Rahman, A., Wu, Z. Y., & Kalfarisi, R. (2021). Semantic Deep Learning Integrated with RGB Feature-Based Rule Optimization for Facility Surface Corrosion Detection and Evaluation. *Journal of Computing in Civil Engineering Vol.35, Issue 6*.

Raschka, S., & Mirjalili, V. (2019). *Python Machine Learning.* Packt.

Raschka, S., & Mirjalili, V. (2019). *Python Machine Learning; Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2.* Birmingham: Packt.

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779-788.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *Proceedings of the 28th International Conference on Neural Information Processing Systems 1*, 91-99.

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review Vol. 65, No. 6*, 386-408.

Stachniss, C. (2015). Camera Parameters: Extrinsics and Intrinsics; Photogrammetry Course I. *University of Bonn*.

Stachniss, C. (2015). Projective 3-Point (P3P) Algorithm / Spatial Resection; Photogrammetry Course I. *University of Bonn*.

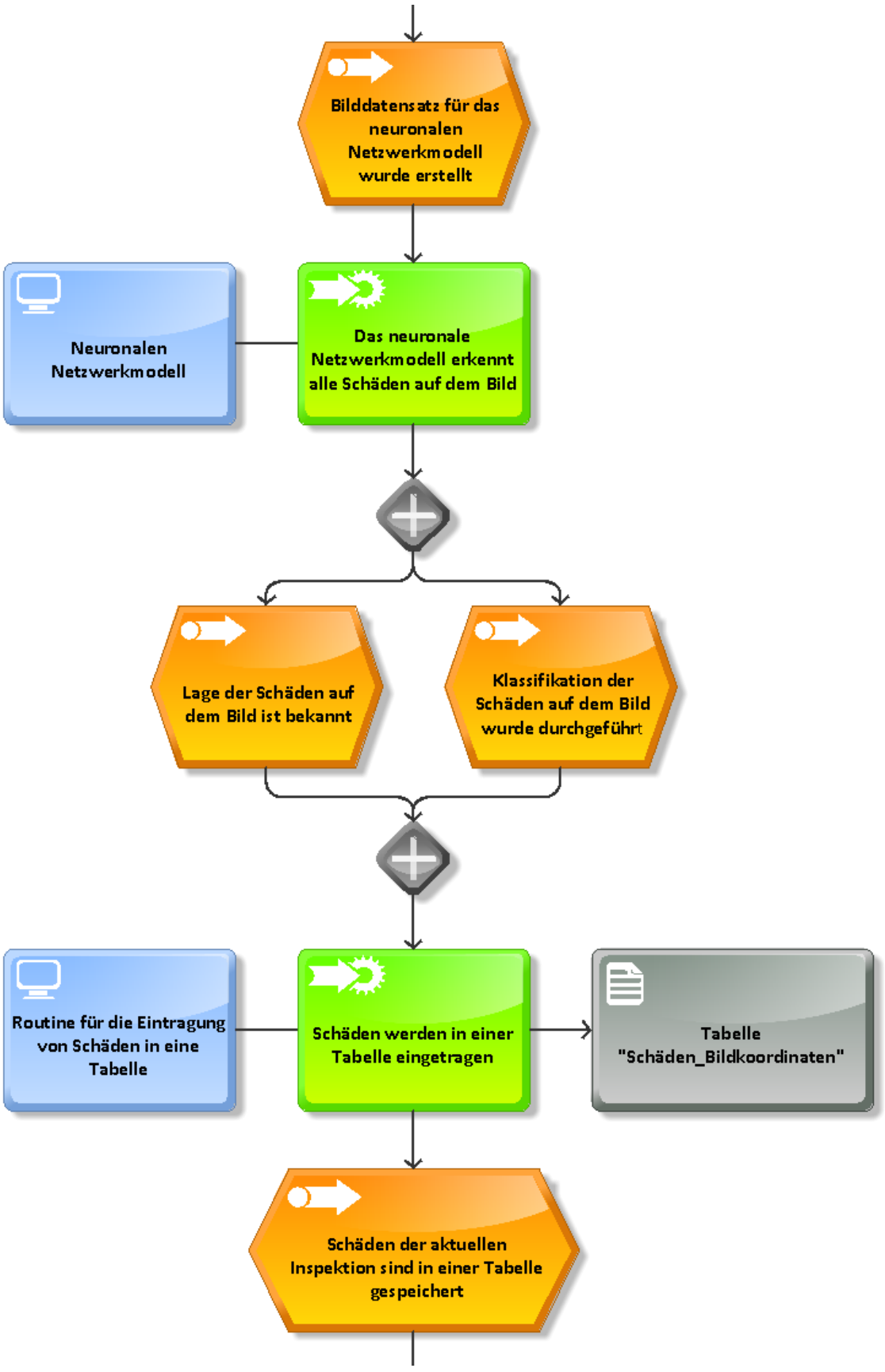
Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Rabinovich, A. (2015). Going Deeper with Convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1-9.

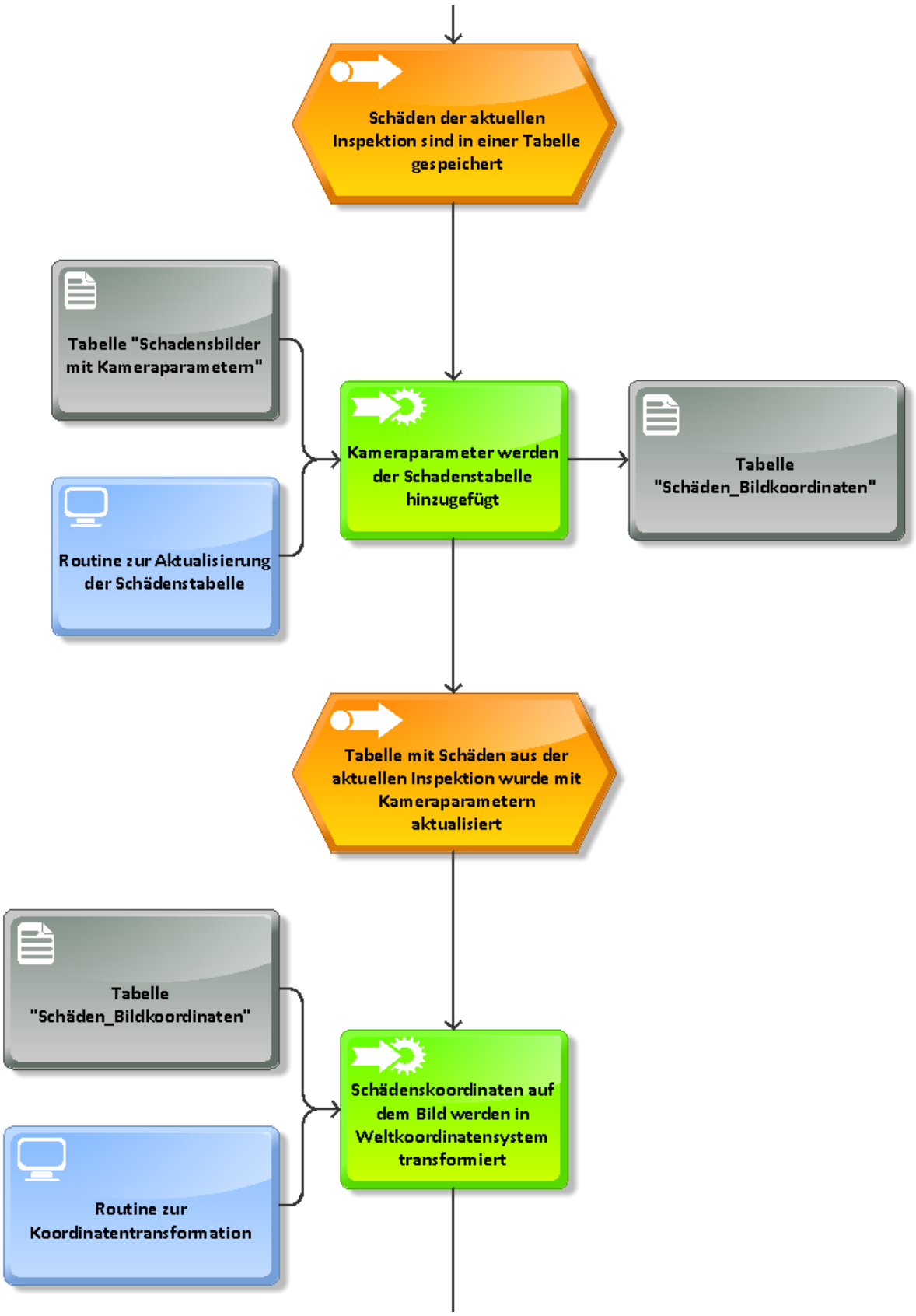
Vladimirov, L. (2020). *TensorFlow 2 Object Detection API tutorial*. Von https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/# abgerufen

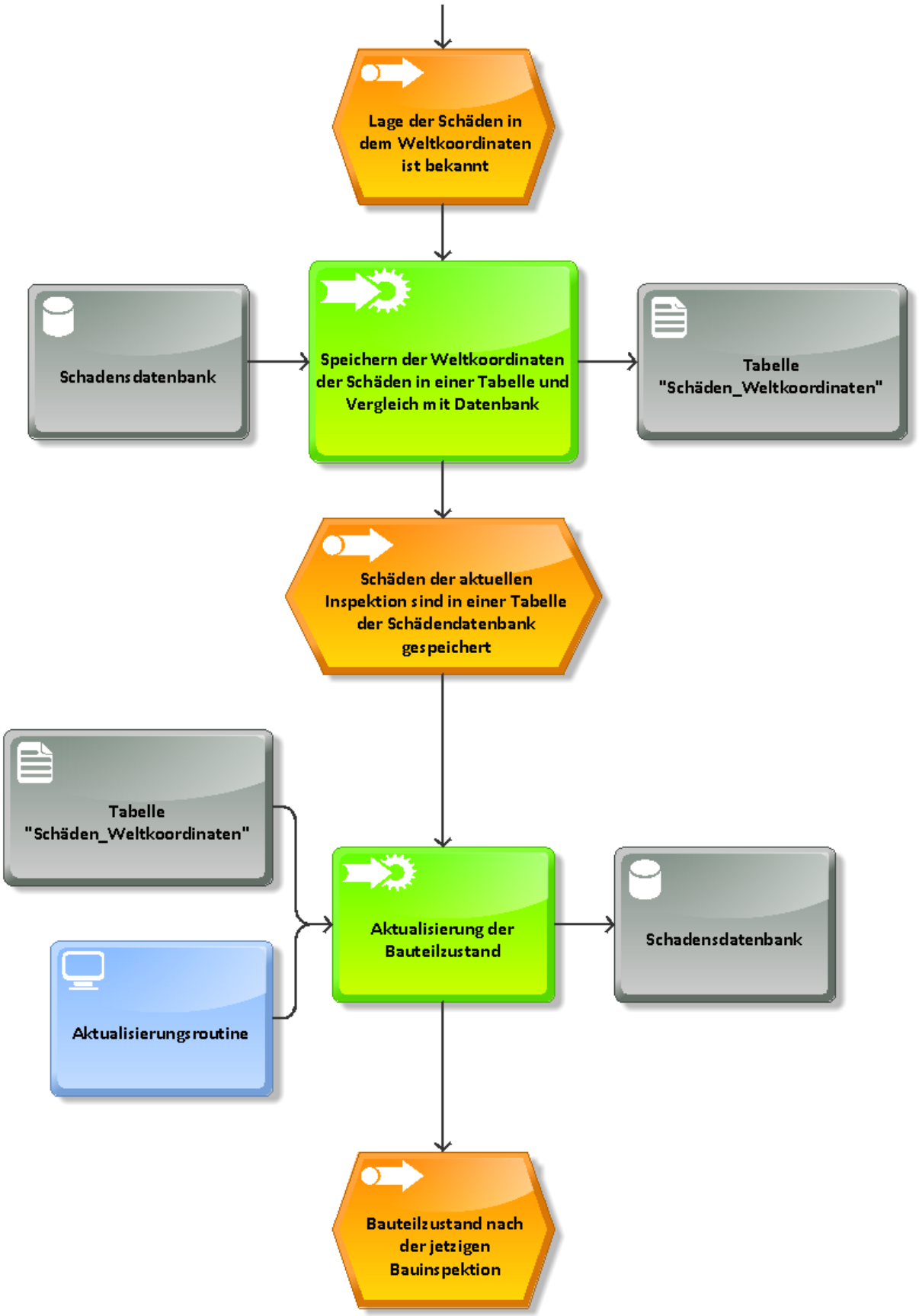
Yu, H., Chen, C., Du, X., Li, Y., Rashwan, A., Hou, L., . . . Li, J. (2020). *TensorFlow Model Garden.* Von https://github.com/tensorflow/models abgerufen

Zhang, L., Yang, F., Zhang, Y. D., & Zhu, Y. J. (2016). Road crack detection using deep convolutional neural network. *2016 IEEE International Conference on Image Processing (ICIP)* (S. 5). Phoenix, AZ, USA: IEEE.

ANLAGE 1 ERWEITERTE EREIGNISGESTEUERTE PROZESS-KETTE


Bilddatensatz für das neuronalen Netzwerkmodell wurde erstellt
Neuronalen Netzwerkmodell
Das neuronale Netzwerkmodell erkennt alle Schäden auf dem Bild
Lage der Schäden auf dem Bild ist bekannt
Klassifikation der Schäden auf dem Bild wurde durchgeführt
Routine für die Eintragung von Schäden in eine Tabelle
Schäden werden in einer Tabelle eingetragen
Tabelle "Schäden_Bildkoordinaten"
Schäden der aktuellen Inspektion sind in einer Tabelle gespeichert


Schäden der aktuellen Inspektion sind in einer Tabelle gespeichert
Tabelle "Schadensbilder mit Kameraparametern"
Routine zur Aktualisierung der Schädenstabelle
Kameraparameter werden der Schadenstabelle hinzugefügt
Tabelle "Schäden_Bildkoordinaten"
Tabelle mit Schäden aus der aktuellen Inspektion wurde mit Kameraparametern aktualisiert
Tabelle "Schäden_Bildkoordinaten"
Routine zur Koordinatentransformation
Schädenskoordinaten auf dem Bild werden in Weltkoordinatensystem transformiert


Lage der Schäden in dem Weltkoordinaten ist bekannt
Schadensdatenbank
Speichern der Weltkoordinaten der Schäden in einer Tabelle und Vergleich mit Datenbank
Tabelle "Schäden_Weltkoordinaten"
Schäden der aktuellen Inspektion sind in einer Tabelle der Schädendatenbank gespeichert
Tabelle "Schäden_Weltkoordinaten"
Aktualisierung der Bauteilzustand
Schadensdatenbank
Aktualisierungsroutine
Bauteilzustand nach der jetzigen Bauinspektion

ANLAGE 2 NEURONALEN NETZWERKMODELL FÜR DIE RISSERKENNUNG

Der nachstehende Code wurde von Priya Dwivedyi (2019) entwickelt und für diese Arbeit leicht modifiziert. Der Trainingsdatensatz wurde von Ozgenel (2019) zusammengestellt und gelabelt.

```
## Import the necessary dependencies
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
from torch.optim import lr_scheduler
from torchvision import datasets, models, transforms
import torchvision
import numpy as np
import os
cwd = os.getcwd()
from PIL import Image
import time
import copy
import random
import cv2
import re
import shutil
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline

## Load all the images with cracks
crack_images = os.listdir('Positive/')
print("Number of Crack Images: ", len(crack_images))

## Load all images without cracks
no_crack_images = os.listdir('Negative/')
print("Number of No Crack Images: ", len(no_crack_images))

## Visualize Random images with cracks
random_indices = np.random.randint(0, len(crack_images), size=4)
print("*****************Random Images with Cracks*************************")
random_images = np.array(crack_images)[random_indices.astype(int)]

f, axarr = plt.subplots(2,2)
axarr[0,0].imshow(mpimg.imread(os.path.join(cwd, 'Positive', random_images[0])))
axarr[0,1].imshow(mpimg.imread(os.path.join(cwd, 'Positive', random_images[1])))
axarr[1,0].imshow(mpimg.imread(os.path.join(cwd, 'Positive', random_images[2])))
axarr[1,1].imshow(mpimg.imread(os.path.join(cwd, 'Positive', random_images[3])))

```

```
## Visualize Random images with no cracks
random_indices = np.random.randint(0, len(no_crack_images), size=4)
print("*****************Random Images without Cracks**************************")
random_images = np.array(no_crack_images)[random_indices.astype(int)]

f, axarr = plt.subplots(2,2)
axarr[0,0].imshow(mpimg.imread(os.path.join(cwd, 'Negative', random_images[0])))
axarr[0,1].imshow(mpimg.imread(os.path.join(cwd, 'Negative', random_images[1])))
axarr[1,0].imshow(mpimg.imread(os.path.join(cwd, 'Negative', random_images[2])))
axarr[1,1].imshow(mpimg.imread(os.path.join(cwd, 'Negative', random_images[3])))

## Create training folder
base_dir = cwd
files = os.listdir(base_dir)

def create_training_data(folder_name):
    train_dir = f"{base_dir}/train/{folder_name}"
    for f in files:
        search_object = re.search(folder_name, f)
        if search_object:
            shutil.move(f'{base_dir}/{folder_name}', train_dir)

create_training_data('Positive')
create_training_data('Negative')

## Move images randomly from training to val folders
os.makedirs('val/Positive')
os.makedirs('val/Negative')

positive_train = base_dir + "/train/Positive/"
positive_val = base_dir + "/val/Positive/"
negative_train = base_dir + "/train/Negative/"
negative_val = base_dir + "/val/Negative/"

positive_files = os.listdir(positive_train)
negative_files = os.listdir(negative_train)

print(len(positive_files), len(negative_files))

for f in positive_files:
    if random.random() > 0.80:
        shutil.move(f'{positive_train}/{f}', positive_val)

for f in negative_files:
    if random.random() > 0.80:
        shutil.move(f'{negative_train}/{f}', negative_val)

## Compute mean and std deviation for the dataset
mean_nums = [0.485, 0.456, 0.406]
std_nums = [0.229, 0.224, 0.225]

## Define data augmentation and transforms
chosen_transforms = {'train': transforms.Compose([
        transforms.RandomResizedCrop(size=227),
```

```
        transforms.RandomRotation(degrees=10),
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.ColorJitter(brightness=0.15, contrast=0.15),
        transforms.ToTensor(),
        transforms.Normalize(mean_nums, std_nums)
]), 'val': transforms.Compose([
        transforms.Resize(227),
        transforms.CenterCrop(227),
        transforms.ToTensor(),
        transforms.Normalize(mean_nums, std_nums)
]),
}

## Create the data loader
def load_dataset(format, batch_size):
    data_path = os.path.join(cwd, format)
    dataset = datasets.ImageFolder(
        root=data_path,
        transform= chosen_transforms[format]
    )
    data_loader = DataLoader(
        dataset,
        batch_size=batch_size,
        num_workers=4,
        shuffle=True
    )
    return data_loader, len(dataset), dataset.classes

## Set code to run on device
#device = torch.device("cpu")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

train_loader, train_size, class_names = load_dataset('train', 8)
print("Train Data Set size is: ", train_size)
print("Class Names are: ", class_names)
inputs, classes = next(iter(train_loader))
print(inputs.shape, classes.shape)

# Visualize some images
def imshow(inp, title=None):
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([mean_nums])
    std = np.array([std_nums])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # Pause a bit so that plots are updated

# Grab some of the training data to visualize
inputs, classes = next(iter(train_loader))
```

```
# class_names = chosen_datasets['train'].classes
# Now we construct a grid from batch
out = torchvision.utils.make_grid(inputs)

idx_to_class = {0:'Negative', 1:'Positive'}
plt.figure(figsize=(20,10))
imshow(out, title=[x.data.numpy() for x in classes])

## Load pretrained model
resnet50 = models.resnet50(pretrained=True)

# Freeze model parameters
for param in resnet50.parameters():
    param.requires_grad = False

## Change the final layer of the resnet model
# Change the final layer of ResNet50 Model for Transfer Learning
fc_inputs = resnet50.fc.in_features

resnet50.fc = nn.Sequential(
    nn.Linear(fc_inputs, 128),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Linear(128, 2)
)

# Convert model to be used on GPU
resnet50 = resnet50.to(device)

from torchsummary import summary
print(summary(resnet50, (3, 227, 227)))

# Define Optimizer and Loss Function
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet50.parameters())
# optimizer = optim.SGD(resnet50.fc.parameters(), lr=0.001, momentum=0.9)
# Decay LR by a factor of 0.1 every 3 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)

dataloaders = {}
dataset_sizes = {}
batch_size = 256
dataloaders['train'], dataset_sizes['train'], class_names = load_dataset('train',
batch_size)
dataloaders['val'], dataset_sizes['val'], _ = load_dataset('val', batch_size)
idx_to_class = {0:'Negative', 1:'Positive'}

## Define Training function
def train_model(model, criterion, optimizer, scheduler, num_epochs=10):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

```

```
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            current_loss = 0.0
            current_corrects = 0

            # Here's where the training happens
            print('Iterating through data...')

            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # We need to zero the gradients, don't forget it
                optimizer.zero_grad()

                # Time to carry out the forward training poss
                # We only need to log the loss stats if we are in training phase
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()


                # We want variables to hold the loss statistics
                current_loss += loss.item() * inputs.size(0)
                current_corrects += torch.sum(preds == labels.data)

            epoch_loss = current_loss / dataset_sizes[phase]
            epoch_acc = current_corrects.double() / dataset_sizes[phase]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # Make a copy of the model if the accuracy on the validation set has improved
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()
```

```
259
260     time_since = time.time() - since
261     print('Training complete in {:.0f}m {:.0f}s'.format(
262         time_since // 60, time_since % 60))
263     print('Best val Acc: {:4f}'.format(best_acc))
264
265     # Now we'll load in the best model weights and return it
266     model.load_state_dict(best_model_wts)
267     return model
268
269 def visualize_model(model, num_images=6):
270     was_training = model.training
271     model.eval()
272     images_handeled = 0
273     fig = plt.figure()
274
275     with torch.no_grad():
276         for i, (inputs, labels) in enumerate(dataloaders['val']):
277             inputs = inputs.to(device)
278             labels = labels.to(device)
279
280             outputs = model(inputs)
281             _, preds = torch.max(outputs, 1)
282
283             for j in range(inputs.size()[0]):
284                 images_handeled += 1
285                 ax = plt.subplot(num_images//2, 2, images_handeled)
286                 ax.axis('off')
287                 ax.set_title('predicted: {}'.format(class_names[preds[j]]))
288                 imshow(inputs.cpu().data[j])
289
290                 if images_handeled == num_images:
291                     model.train(mode=was_training)
292                     return
293         model.train(mode=was_training)
294
295 ## Start Training
296 base_model = train_model(resnet50, criterion, optimizer, exp_lr_scheduler, num_epochs=6)
297 visualize_model(base_model)
298 plt.show()
299
300 ## Inference
301 # Define prediction function
302 def predict(model, test_image, print_class = False):
303
304     transform = chosen_transforms['val']
305
306     test_image_tensor = transform(test_image)
307
308     if torch.cuda.is_available():
309         test_image_tensor = test_image_tensor.view(1, 3, 227, 227).cuda()
310     else:
311         test_image_tensor = test_image_tensor.view(1, 3, 227, 227)
312
```

```
    with torch.no_grad():
        model.eval()
        # Model outputs log probabilities
        out = model(test_image_tensor)
        ps = torch.exp(out)
        topk, topclass = ps.topk(1, dim=1)
        class_name = idx_to_class[topclass.cpu().numpy()[0][0]]
        if print_class:
            print("Output class :  ", class_name)
    return class_name

# Full-image crack detection function
def predict_on_crops(input_image, height=227, width=227, save_crops = False):
    im = cv2.imread(input_image)
    imgheight, imgwidth, channels = im.shape
    k=0
    output_image = np.zeros_like(im)
    for i in range(0,imgheight,height):
        for j in range(0,imgwidth,width):
            a = im[i:i+height, j:j+width]
            ## discard image cropss that are not full size
            predicted_class = predict(base_model,Image.fromarray(a))
            ## save image
            file, ext = os.path.splitext(input_image)
            image_name = file.split('/')[-1]
            folder_name = 'out_' + image_name
            ## Put predicted class on the image
            if predicted_class == 'Positive':
                color = (0,0, 255)
            else:
                color = (0, 255, 0)
            cv2.putText(a, predicted_class, (50,50), cv2.FONT_HERSHEY_SIMPLEX , 0.7, color,
1, cv2.LINE_AA)
            b = np.zeros_like(a, dtype=np.uint8)
            b[:] = color
            add_img = cv2.addWeighted(a, 0.9, b, 0.1, 0)
            ## Save crops
            if save_crops:
                if not os.path.exists(os.path.join('real_images', folder_name)):
                    os.makedirs(os.path.join('real_images', folder_name))
                filename = os.path.join('real_images', folder_name,'img_{}.png'.format(k))
                cv2.imwrite(filename, add_img)
            output_image[i:i+height, j:j+width,:] = add_img
            k+=1
    ## Save output image
    cv2.imwrite(os.path.join('real_images','predictions', folder_name+ '.jpg'), output_im-
age)
    return output_image

# Visualize images
plt.figure(figsize=(10,10))
output_image = predict_on_crops('real_images/concrete_crack1.jpg', 128, 128)
plt.imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))

```

```
plt.figure(figsize=(10,10))
output_image = predict_on_crops('real_images/concrete_crack2.jpg')
plt.imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))

plt.figure(figsize=(10,10))
output_image = predict_on_crops('real_images/road_surface_crack1.jpg', 128,128)
plt.imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))

plt.figure(figsize=(10,10))
output_image = predict_on_crops('real_images/road_surface_crack3.jpg',128,128)
plt.imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))

# Save the Model Parameters
#Example for saving a checkpoint assuming the network class named #Classifier

torch.save(base_model,'saved_model')
```

ANLAGE 3 CODE ZUM ZUSAMMENFÜHREN VON BILD- UND SCHADENSINFORMATIONEN

```
import csv
# open the file in the write mode
f = open('damage_in_images.csv', 'w')
# create the csv writer
writer = csv.writer(f)
# write a row to the csv file
writer.writerow(['img_path','la-
bel','x_min','x_max','y_min','y_max','x_cam','y_cam','z_cam','alpha','beta','gamma'])

camera_params = []
with open('camera_parameters/camera_parameters.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        camera_params.append(row)
dd_results = []
with open('results/results.csv','r') as file:
    reader = csv.reader(file)
    for row in reader:
        dd_results.append(row)

damage_in_images = [['img_path','la-
bel','x_min','x_max','y_min','y_max','x_cam','y_cam','z_cam','alpha','beta','gamma']]
for dd_row in range(1,len(dd_results)):
    for cp_row in range(1,len(camera_params)):
        if dd_results[dd_row][1] == camera_params[cp_row][0]:
            #print(dd_results[dd_row][1])
            line = [dd_results[dd_row][1],dd_results[dd_row][2],dd_re-
sults[dd_row][3],dd_results[dd_row][4],dd_results[dd_row][5],dd_results[dd_row][6],cam-
era_params[cp_row][1],camera_params[cp_row][2],camera_params[cp_row][3],cam-
era_params[cp_row][4],camera_params[cp_row][5],camera_params[cp_row][6]]
            damage_in_images.append(line)

print(damage_in_images)

# open the file in the write mode
f = open('damage_in_images.csv', 'w', newline='')
# create the csv writer
writer = csv.writer(f)
writer.writerows(damage_in_images)
# write a row to the csv file
#for row in damage_in_images:
#    writer.writerow(row)

# close the file
f.close()
```

ANLAGE 4 2D-ZU-3D-PUNKT-MAPPING-CODE

```
# Import libraries
import pandas as pd
import numpy as np
import csv

# Import Extrinsics and Intrinsics
# Camera intrinsics
# c = 1.0     # camera constant
# hx = 0.0    # principal point, x coordinate
# hy = 0.0    # principal point, y coordinate
# m = 0.0     # scale difference m in x- and y-axes
# s = 0.0     # shear compensation s

def get_calibration_matrix(c, hx, hy, m, s):
    calibration_matrix = np.matrix([[c,         c*s,        hx],
                                    [0.0,       c*(1+m),    hy],
                                    [0.0,       0.0,        1.0]])
    return calibration_matrix

# Import damage information
# Step 1: import the CSV File with the extrinsics
def get_damage_information():
    data = pd.read_csv(r'damage_in_images.csv')
    dinfo_df = pd.DataFrame(data, col-
umns=['x_min','x_max','y_min','y_max','x_cam','y_cam','z_cam','alpha','beta','gamma'])

    # Step 2: transform dataframe to numpy array
    dinfo_np = dinfo_df.to_numpy()
    return dinfo_np

def get_damage_classes():
    data = pd.read_csv(r'damage_in_images.csv')
    dclasses_df = pd.DataFrame(data, columns=['img_path','label'])

    return dclasses_df

# Build the rotation matrix
# Extrinsic rotation whose (improper) Euler angles are alpha, beta and gamma about axes x,
y and z (https://en.wikipedia.org/wiki/Rotation_matrix#General_rotations)
def build_rotationmatrix(a, b, c):
    z_rotation = np.array([[np.cos(c), -np.sin(c), 0],[np.sin(c), np.cos(c), 0],[0, 0, 1]],
np.float32)
    y_rotation = np.array([[np.cos(b), 0, np.sin(b)],[0, 1, 0],[-np.sin(b), 0, np.cos(b)]],
np.float32)
```

```
    x_rotation = np.array([[1, 0, 0],[0, np.cos(a), -np.sin(a)],[0, np.sin(a), np.cos(a)]],
np.float32)
    rotation_matrix = z_rotation * y_rotation * x_rotation
    return rotation_matrix

# Mapping function
# Attributes: Camera Positionmatrix X0, Rotationmatrix R, Calibrationmatrix K, Point coor-
dinates x
def point_mapping():
    # Get the calibration matrix FOR AN IDEAL CAMERA (c=1, other parameters = 0)
    calibration_matrix = get_calibration_matrix(1, 0, 0, 0, 0)
    damage_info_np = get_damage_information()

    damage_3DCoordinates = []
    for x in damage_info_np:
        # Rotation Matrix, Center of Damage Box
        rotation_matrix = build_rotationmatrix(x[7],x[8],x[9])
        box_centerpoint = [(x[0]+x[1])/2,(x[2]+x[3])/2,1]
        #print(box_centerpoint)
        #print(rotation_matrix)

        # Matrix multiplication and Compute inverse matrix
        matmul = np.matmul(calibration_matrix,rotation_matrix)
        inv_matmul = np.linalg.inv(matmul)

        # Mapping function
        newbox_centerpoint = np.matmul(inv_matmul,box_centerpoint)

        # Reshape coordinate vector
        #row = np.array(newbox_centerpoint).reshape(1,3)

        # Append to point list
        damage_3DCoordinates.append(newbox_centerpoint)

    # Reshape matrix to list
    damage_3DCoordinates = np.array(damage_3DCoordinates).reshape(9,3).tolist()

    return damage_3DCoordinates

    #print("This function maps the 2D coordinates of the detected damage to the 3D coordi-
nates")

# Output 3D Information in a new csv File
def outputCSVFile():
    # Import classes and 3D Coordinate values
    classes_df = get_damage_classes()      # Dataframe
    camera_np = get_damage_information()   # Numpy Array
    delta_np = point_mapping()             # Numpy Array

    # Transform "classes" Dataframe into a list
    classes_list = classes_df.values.tolist()

    # Get Camera position
    data = pd.read_csv(r'damage_in_images.csv')
```

```
    df = pd.DataFrame(data, columns=['x_cam','y_cam','z_cam'])
    camera_np = df.to_numpy()

    damage_in_world = [['img_path', 'label', 'X_cam', 'Y_cam', 'Z_cam', 'X_delta', 'Y_del-
ta', 'Z_delta']]
    # damage_in_world = [['img_path', 'label', 'X_cam', 'Y_cam', 'Z_cam']]
    for i in range(len(classes_list)):
        #print(classes_list[i])
        for j in range(len(camera_np)):
            for k in range(len(delta_np)):
                if i==j and i==k and j==k:
                    print(delta_np[k][0])
                    row = [classes_list[i][0],classes_list[i][1], camera_np[j][0], cam-
era_np[j][1], camera_np[j][2], delta_np[k][0], delta_np[k][1], delta_np[k][2]]
                    damage_in_world.append(row)
                else:
                    continue

    #for row in damage_in_world:
    #    print(row)

    # Write output in a CSV File
    # open the file in the write mode
    f = open('damage_in_world.csv', 'w', newline='')

    # create the csv writer
    writer = csv.writer(f)

    writer.writerows(damage_in_world)
    # write a row to the csv file
    #for row in damage_in_images:
    #    writer.writerow(row)

    # close the file
    f.close()


if __name__== '__main__':

    #delta_np = point_mapping()
    #for row in delta_np:
    #    print(row)
    outputCSVFile()
```