DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Advanced Software Engineering

Projektarbeitsdokumentation

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges

Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Dominik Ochs

Abgabedatum:	XX. Mai 2023
Bearbeitungszeitraum:	01.10.2022 - XX.05.2023
Matrikelnummer, Kurs:	2847475, TINF20B2
Gutachter der Dualen Hochschule:	Dr. Lars Briem

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1. Einführung

Die Navigation über Apps wie beispielsweise Google Maps ist inzwischen aus dem Alltag nicht mehr wegzudenken. Auch für das Zurücklegen einer Strecke mit dem Fahrrad werden entsprechende Routen vorgeschlagen. Allerdings liegen diese häufig neben großen Straßen und berücksichtigen als vorrangiges Ziel die schnellste Strecke zum Zielpunkt. Touren müssen eigenständig über das Hinzufügen weiterer Wegpunkte angepasst werden, wenn entspannter ans Ziel gelangt werden möchte. Um dieses Umplanen zu erleichtern, sollen im Rahmen dieser Studienarbeit städtische Radwege über Luftaufnahmen ausfindig gemacht werden. Gefundene Radwege sollen dann im Anschluss anhand ihres Umfeldes in verschiedene Kategorien eingruppiert werden, die Rückschlüsse auf die Fahrradfreundlichkeit zulassen.

Die Extraktion von Straßen aus Luftaufnahmen ist bereits ein gut untersuchtes Teilgebiet der Computer-Vision, wozu es eine Vielzahl an wissenschaftlichen Publikationen gibt und praxistaugliche Resultate erzielt werden konnten.
Allerdings gibt es keinerlei Untersuchungen oder Datensätze zur Erkennung, geschweige denn Extraktion, von Radwegen aus Luftbildern. Hierzu ist es nötig neben dem Erkennen eines befestigten Weges diesen auch von Straßen oder Gehwegen abzugrenzen. Selbst für Menschen stellt es oft ein schwieriges Unterfangen dar, aufgrund der für das Problem geringen **GSD!** (**GSD!**), wobei selbst bei modernen Aufnahmen ein Radweg nur wenige Pixel breit ist (1 Pixel entspricht 0,20 bis 0,50 Meter), oder ein Radweg von Vegetation wie Bäumen oder von Schatten verdeckt wird. Diese Arbeit kann also darüber Aufschluss geben, inwiefern eine Abgrenzung zwischen unterschiedlichen Straßentypen oder -bestandteilen mithilfe von Computer Vision grundsätzlich möglich ist und ob in diesem Anwendungsfall eine Maschine eventuell sogar übermenschliche Performanz erreichen kann, um so zum Beispiel auch sehr dunkle, überschattete oder überwachsene Wege zu erkennen. Des Weiteren kann diese Arbeit als Anhaltspunkt für verwandte Probleme, wie zum Beispiel das Erkennen von Busspuren oder Gehwegen genutzt werden. Auch wird eine Methode vorgeschlagen, wie für solche Fälle ein Datensatz erstellt werden kann, mit dem ein Computer-Vision-Modell trainiert werden kann.

1.1. Ziele der Arbeit

Das Ziel der Studienarbeit liegt in der Erkennung von Fahrradwegen mithilfe von Computer Vision. Hierfür werden Satellitenaufnahmen mit einer Bodenauflösung von 20 *cm* verwendet. Da es für die Erkennung von Fahrradwege keine vorliegenden Datensets gibt, muss dieses selbst erstellt werden. Die Umsetzung des neuronalen Netzes ist über ein U-Net realisiert. Im ersten Schritt erfolgt eine reine Klassifikation in Fahrradweg bzw. Nicht-Fahrradweg. Über das Verändern von Parametern wird versucht, das Netz zu optimieren. Die Überführung in verschiedene Kategorien soll erst erfolgen, wenn ausreichend zuverlässig Wege identifiziert werden können.

1.2. Strukturierung

Zu Beginn der Arbeit wird auf den aktuellen Stand der Technik eingegangen. Der Fokus liegt hierbei auf Sachverhalten, die Bestandteil der Studienarbeit oder wichtig für das Verständnis sind. So werden zunächst verschiedene Arten von Bilderkennung aufgegriffen. Im späteren Verlauf verwendete Metriken werden vorgestellt. Die Struktur und Architektur eines U-Nets wird beschrieben. Da es bereits Datensätze und Implementierungen zum Segmentieren von Straßen gibt, folgt eine Erläuterung des Transfer-Learning-Ansatzes. Die Datensätze werden ebenfalls kurz vorgestellt und auf die Unterschiede zwischen ihnen eingegangen.

Bei der Konzeption wird ein Pretraining mithilfe der Straßen-Datensätze durchgeführt. Im Anschluss erfolgt die Erstellung eines eigenen Datensatzes für Fahrradwege. Die Architektur des zu implementierenden Netzes wird mit seinen Hyperparametern erläutert. Zu messende Werte zum Einschätzen der Güte werden festgelegt. Die Implementierung realisiert die in der Konzeption entworfenen Netze und generiert den Datensatz.

Die Ergebnisse werden anschließend vorgestellt und die einzelnen Varianten miteinander verglichen. Als Bewertungsbasis werden die definierten Metriken verwendet. Wichtige Erkenntnisse werden hervorgehoben. Zum Schluss folgt eine kritische Reflexion in Zusammenhang mit einer Zusammenfassung sowie ein Ausblick der Arbeit.

2. Clean Architecture

Dieses Kapitel beschäftigt sich mit dem Stand der Technik, der Einordnung dieser Arbeit in die Literatur und den Grundlagen der Konzeption. Hierfür sind zunächst die Aufgabenkategorien der Computer-Vision zu vergleichen, um eine geeignete für das gegebene Problem finden zu können. Ebenso wird für verschiedene state-of-the-art Aktivierungsfunktionen und Ähnlichkeitsmaße verfahren, sodass später geeignete ausgewählt und diskutiert werden können. Daraufhin sind die Vorteile einiger relevanter Architekturkomponenten zu betrachten und einige state-of-the-art Klassifikationsmodelle des ImageNet-Datensatzes, welche ggf. als vortrainierte Bestandteile der späteren Architektur verwendet werden können, was genauer im Abschnitt *Transfer-Learning mit U-Net* betrachtet ist. Schließlich wird der aktuelle Stand bei der verwandten Domäne der Straßenerkennung aus Luftaufnahmen untersucht.

2.1. Aufgabenkategorien

Im Bereich der Computer Vision gibt es verschiedene Problemstellungen, die unterschiedlich gelöst werden können. Die einzelnen Lösungsansätze werden jeweils einer Aufgabenkategorie zugeordnet. Nachfolgend wird ein Überblick über die Kategorien *Klassifizierung*, *Object Detection* und *Image Semantic Segmentation* gegeben. Eine Visualisierung ist in Abbildung 2.1 gegeben. Die aufgeführten Kategorien nehmen in der genannten Reihenfolge in ihrer Komplexität zu.

- *Klassifizierung:* Eine reine Klassifizierung ist die einfachste Lösungskategorie. Es soll lediglich erkannt werden, welche Klasse in einem Bild enthalten ist. Die Lokalität des erkannten Objektes wird vernachlässigt, wichtiger ist die Zuordnung zu einer Klasse. Dies lässt sich auch entsprechend auf die Erkennung von mehreren Klassen erweitern.
- *Object Detection:* Eine Erweiterung der Klassifizierung um die Lokalität des erkannten Objektes wird *Image Localization* genannt. Für mehrere Objekte wird der

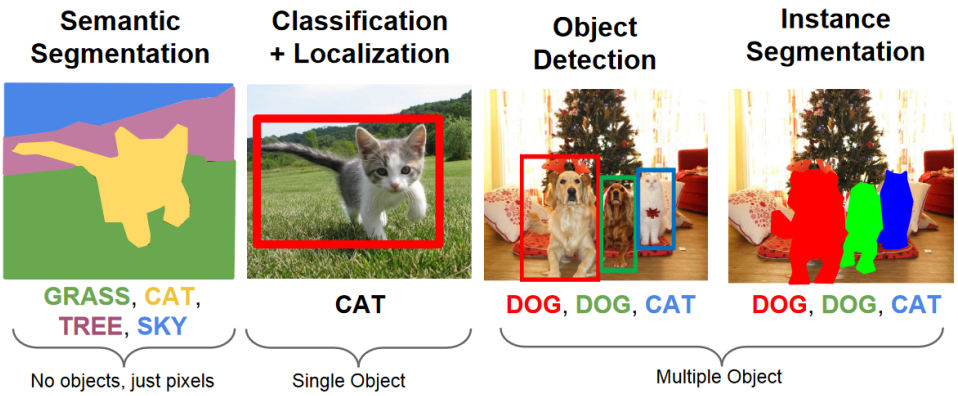

Abbildung 2.1.: Verschiedene Aufgabenkategorien [2].

Begriff Object Detection verwendet. Die Darstellung erkannter Objekte wird über Bounding Boxes realisiert, welche die Objekte jeweils umschließen.

- *Image Semantic Segmentation:* Die bei der Object Detection verwendeten Bounding Boxes geben keine Rückschlüsse auf die konkrete Form der Klasse. Je nach Lage des Objektes kann nur ein Bruchteil der Bounding Box von der erkannten Klasse gefüllt sein. Abhilfe schafft die Verwendung einer pixelweisen Maske anstelle einer Bounding Box, das Objekt kann von seiner Umgebung abgegrenzt werden. Zusätzlich kann gefordert werden, dass einzelne Instanzen verschiedener Objekte unterschieden werden sollen. Diese Forderung wird dann *Instance Segmentation* genannt [1].

2.2. Aktivierungsfunktionen

Im Folgenden wird der Stand der Technik für Aktivierungsfunktionen in **ML!** (**ML!**)-Modellen dargelegt, hierzu werden einige Funktionen kurz vorgestellt und deren Vor- und Nachteile bewertet. Die besprochenen Funktionen sind dargestellt in Abbildung 2.2.

2.2.1. Sigmoid

Die *Sigmoid*-Funktion bildet die Eingabe auf einen Wert im Intervall $(0;1)$ ab und wird durch Gleichung (2.1) beschrieben. Der Vorteil hier ist, dass Aktivierungen nie groß werden können, wodurch einzelne Neuronen nicht den Gradienten dominieren können und

eine fortlaufende Normalisierung durchgeführt wird. Die Probleme hingegen sind, dass die Funktion eher aufwendig zu berechnen ist und dass die Ableitung für betragsmäßig größere Werte verschwindet. Hierdurch kann es zum *Vanishing-Gradient-Problem* kommen, wobei Neuronen in den flacheren Schichten eines neuronalen Netzes kaum noch geupdated werden [3, S. 191–192]

$$Sigmoid(z) = \frac{1}{1 + e^{-z}} \ . \tag{2.1}$$

2.2.2. ReLU! (ReLU!)

Die ***ReLU!*** *(**ReLU!**)* ist eine Aktivierungsfunktion, welche durch Gleichung (2.2) ausgedrückt wird. **ReLU!** ist inzwischen der de-facto Standard von Aktivierungsfunktionen in den verborgenen Schichten eines Deep-Learning-Modells. Dies liegt vor allem daran, dass das Vanishing-Gradient-Problem adressiert wird. Allerdings hat **ReLU!** das Problem, dass Neuronen auf 0 gesetzt werden, wodurch sie kein Gradienten-Update mehr erhalten und dauerhaft genullt bleiben und somit nichts mehr zum Netzwerk beitragen - die Neuronen sterben. Das Problem wird *Dying-**ReLU!**-Problem* genannt [3, S. 189–191]

$$ReLU(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases} \ . \tag{2.2}$$

ReLU! erzielt eine höhere Inferenz- und Trainingsgeschwindigkeit bei gleicher oder besserer Performanz, als Sigmoid. Dies liegt zum einen an der Verbesserung des Vanishing-Gradient-Problems und zum anderen an der simpleren und damit leichter zu berechnenden Funktion [3, S. 226].

2.2.3. ELU! (ELU!)

Die ***ELU!*** *(**ELU!**)* ist eine Aktivierungsfunktion, die das Dying-**ReLU!**-Problem adressieren soll. Ausgedrückt wird die Funktion durch Gleichung (2.3). Die negative Komponente lässt zu, dass Neuronen nicht von einem Satz auf 0 zurückkommen können, und weiterhin etwas zur Zielfunktion beitragen können. Außerdem ist die mittlere Aktivierung näher an

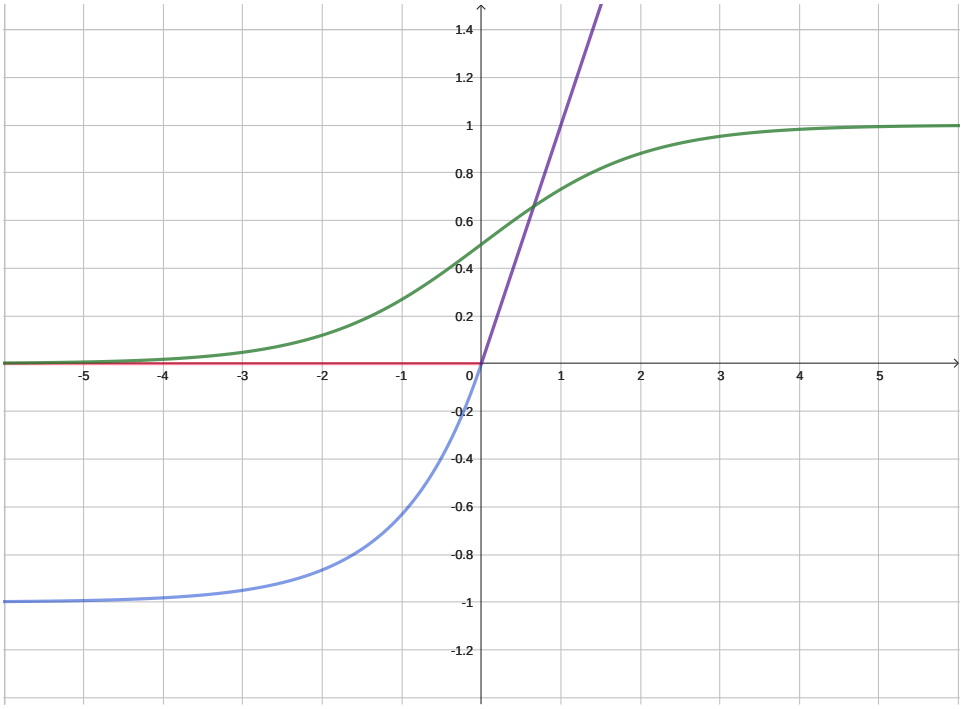

Abbildung 2.2.: Verschiedene Aktivierungsfunktionen. Grün: Sigmoid. Rot: **ReLU!**. Blau: **ELU!** mit $\alpha = 1$.

0 als bei **ReLU!**, was zur Folge hat, dass das Training schneller konvergiert [4]

$$ELU(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z \leq 0 \end{cases} . \tag{2.3}$$

Sowohl **ReLU!** als auch **ELU!** haben das Problem, dass Aktivierungen beliebig groß werden können, wodurch einige wenige Neuronen den Gradienten dominieren können, was zu langsamen Training und suboptimaler Performanz führen kann.

2.3. Ähnlichkeitsmaße

Im Folgenden sollen Ähnlichkeitsmaße zwischen Mengen, insbesondere solche, die als Kostenfunktion bzw. Bewertungsmetrik für einen binären Klassifikator verwendet werden können, untersucht werden. Hierzu wird zunächst Cross-Entropy betrachtet, gefolgt

von dem Dice- bzw. F-Maß. Weiter werden **IoU!** (**IoU!**) und das darauf aufbauende Quality-Maß betrachtet.

2.3.1. BCE! (BCE!)

Die *Cross-Entropy* (bzw. dt. *Kreuzentropie*) ist ein Maß des Unterschieds zweier Wahrscheinlichkeitsdistributionen. Im Spezialfall einer binären Wahrscheinlichkeitsvariable kann die Cross-Entropy zur ***BCE!*** *(**BCE!**)* spezialisiert werden, um auf ein binäres Klassifikationsproblem angewandt zu werden. Gleichung (2.4) zeigt die Kalkulation von **BCE!**, wobei $p \in [0;1]$ die Prediction eines binären Klassifikators und $y \in \{0,1\}$ der Wert des Labels darstellen

$$BCE = -[p \cdot \log(y) + (1-p) \cdot \log(1-y)] \ . \tag{2.4}$$

Um über n Prediction-Label-Paare $(p_i; y_i)$ den **BCE!** zu berechnen, wird das arithmetische Mittel nach Gleichung (2.5) gebildet [5, S. 82, 6, S. 57–59]

$$BCE = -\frac{1}{n}\sum_{i=1}^{n}[p_i \cdot \log(y_i) + (1-p_i) \cdot \log(1-y_i)] \ . \tag{2.5}$$

Im Sinne einer differenzierbaren Kostenfunktion im Kontext von **ML!** sind **BCE!**, *negative Log-Likelihood* und *Logistic-Regression* synonym [6, S. 249].

Aus Gleichung (2.5) geht hervor, warum **BCE!** gut geeignet für Klassifikationsprobleme ist. Im Gegensatz zum mittleren absoluten Fehler, bei dem ein Fehler linear eingeht, und zum mittleren quadratischen Fehler, bei dem ein Fehler quadratisch eingeht, geht ein Fehler bei **BCE!** exponentiell ein. Ein größerer Fehler wiegt also exponentiell stärker als ein kleinerer Fehler. Hierdurch werden die Fehler pro Datenpunkt und Klasse sehr klein, wodurch gute Performance und gute Generalisierung bei **ML!**-Modellen erreicht werden können.

Bei stark ungleichmäßiger Klassenverteilung kann es jedoch dazu kommen, dass die unterrepräsentierte Klasse kaum noch geschätzt wird, da der Fehler einer falsch geschätzten überrepräsentierten Klasse zu stark bestraft wird. Dadurch lernt der Algorithmus, die un-

terrepräsentierte Klasse kaum zu schätzen. Eine Abhilfe dagegen schafft eine Gewichtung der unterschiedlichen Klassen [7, S. 4].

2.3.2. Dice- und F-Maß

Das *Dice*-, oder auch *Sorensen-Dice*-Maß D wurde 1945 bzw. 1948 erstmals vorgestellt und genutzt, um die Ähnlichkeit zweier botanischer Stichproben zu ermitteln. Verallgemeinert auf diskrete Mengen X, Y kann deren Ähnlichkeit nach Dice D beschrieben werden durch Gleichung (2.6). Es gilt $D \in [0;1]$ [8, S. 33, 9, S. 297]

$$D = \frac{2 \cdot |X \cap Y|}{2 \cdot |X \cap Y| + |Y \setminus X| + |X \setminus Y|} = \frac{2 \cdot |X \cap Y|}{|X| + |Y|} \ . \tag{2.6}$$

Angewandt auf boolesche Mengen und binäre Klassifikatoren ist das Dice-Maß gleich dem F_1-Maß, das ein Maß für die Qualität eines statistischen Tests darstellt. Dafür sei X nun die Menge der positiven Elemente und Y die Menge der als positiv eingestuften Elemente. Dann ist die *Genauigkeit* oder auch *Precision* gegeben durch

$$precision = \frac{|X \cap Y|}{|Y|} \ , \tag{2.7}$$

der Anteil der richtig eingestuften Elemente an allen positiv eingestuften Elementen und die *Trefferquote* oder auch *Recall* gegeben durch

$$recall = \frac{|X \cap Y|}{|X|} \ , \tag{2.8}$$

der Anteil der richtig eingestuften Elemente an allen positiven Elementen.
Das F-Maß, bzw. genauer das F_1-Maß, ist dann gegeben durch das harmonische Mittel aus Precision und Recall, wobei tp die Anzahl von wahr-positiven, fp die Anzahl von falsch-positiven und fn die Anzahl von falsch-negativen Elementen ist [10]:

$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall} = \frac{2 \cdot tp}{2 \cdot tp + fp + fn} \ . \tag{2.9}$$

Precision und Recall können mit einem Faktor α unterschiedlich zueinander gewichtet werden, um mit F_α unterschiedliche Aspekte zu fokussieren.

Das Dice-, bzw. F_α-Maß kann leicht für eine differenzierbare Kostenfunktion genutzt werden mit Dice-Loss $D_L(X,Y) = 1 - D(X,Y)$, bzw. F_α-Loss $F_{\alpha L}(X,Y) = 1 - F_\alpha(X,Y)$.

2.3.3. IoU! (IoU!)

Die ***IoU!*** *(**IoU!**)*- bzw. *Jaccard-Ähnlichkeitsmetrik* ist ein weit verbreitetes Maß zur Bestimmung der Ähnlichkeit zwei diskreter Mengen. Hierzu seien X und Y diskrete Mengen. Dann ist die IoU gegeben durch

$$IoU = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X \cap Y| + |Y \setminus X| + |X \setminus Y|} \ . \tag{2.10}$$

Für ein binäres Klassifikationsproblem lässt sich die IoU ausdrücken durch

$$IoU = \frac{tp}{tp + fp + fn} \ , \tag{2.11}$$

wobei tp, fp, fn wie in Gleichung (2.9) [11].

Auffällig ist die Ähnlichkeit zum Dice- bzw. F_1-Maß. Es ist allerdings anzumerken, dass bei Dice/F_1 die tp, also die wahr-positiven, stärker gewichtet werden, als bei der **IoU!**. Die augenscheinliche Ähnlichkeit lässt sich durch die Beziehungen

$$IoU = \frac{D}{2 - D} \tag{2.12}$$

$$D = \frac{2 \cdot IoU}{1 + IoU} \ , \tag{2.13}$$

beschreiben. Im Gegensatz zur **IoU!** wird beim Dice-Maß eine höhere Gewichtung auf die wahr-positiven Elemente gelegt.

2.3.4. Quality

Bei der *Quality* handelt es sich um eine gepufferte Form des **IoU!**, die toleranter bezüglich der Lokalität der Elemente der verglichenen Mengen, oder konkreter, der Pixel einer semantischen Segmentierung, ist, wobei für dieselbe Eingabe $Quality \geq IoU$; $Quality \in [0;1]$ gilt, abhängig von der Puffergröße. Die Quality wird analog zur **IoU!** über eine gepufferte Precision - die *Correctness* - und über einen gepufferten Recall - die *Completeness* -

berechnet. Insbesondere werden einige Elemente, die zuvor als fp und fn eingeordnet wurden, hiermit zu tp konvertiert.
Die Quality soll einige Probleme der **IoU!** beheben, um ein Ähnlichkeitsmaß darzustellen, was näher an der praktischen und vom Menschen wahrgenommen Leistung eines **ML!**-Modells zur semantischen Segmentierung liegt. So soll relativiert werden, dass vor allem im Randbereich einer Segmentierung einzelne abweichende Pixel nicht als falsch anerkannt werden, sodass die Segmentierung im Großen und Ganzen als richtig anerkannt wird [12].

In dieser Arbeit für den Spezialfall von semantischer Segmentierung wird die Quality wie folgt berechnet: Sei P die Prediction der Segmentierungsmaske für ein Eingabebild, Y die tatsächliche Segmentierungsmaske (Ground-Truth / Label) und $P, Y \in \{0,1\}^{w \times h}$, wobei w und h die Breite und Höhe des Eingabebildes darstellen. Die Einträge mit Wert eins von P und Y sind positiv, die Einträge mit Wert null negativ. Einträge repräsentieren Pixel. B sei die Puffergröße in Pixeln.
Zunächst sei die Hilfsfunktion $dilate_B : \{0,1\}^{w \times h} \mapsto \{0,1\}^{w \times h}$ definiert als

$$dilate_B(A) = \left(\begin{cases} 1 & \exists a_{kl} \in A : a_{kl} = 1 \wedge \left\| \begin{pmatrix} k \\ l \end{pmatrix} - \begin{pmatrix} i \\ j \end{pmatrix} \right\|_2 \leq B \\ 0 & else \end{cases} \right)_{ij} . \tag{2.14}$$

Die Funktion $dilate_B$ bildet eine Matrix auf eine weitere derselben Ordnung ab, sodass alle Einträge gleich eins sind, die in der Eingabematrix eins sind. Zusätzlich sind alle Einträge eins, die im Umkreis von B Pixel um eine Eins in der Eingabematrix liegen. Somit werden die positiven Pixel der Eingabematrix um die Bufferzone erweitert.
Damit lassen sich die gepufferten, fehlertoleranteren wahr-positiv tp_B, falsch-positiv fp_B und falsch-negativ fn_B Werte berechnen:

$$tp_B = \| dilate_B(Y) \odot P \|_{1,1} , \tag{2.15}$$

$$fn_B = \| Y - (Y \odot dilate_B(P)) \|_{1,1} , \tag{2.16}$$

$$fp_B = \| P \|_{1,1} - tp_B . \tag{2.17}$$

Beispielhaft und intuitiv dargestellt für tp_B: Es sollen auch alle Eins-Pixel als wahr-positiv aufgefasst werden, die leicht von der Ground-Truth abweichen, also sich noch in der

Pufferzone befinden. Hierfür wird die Pufferzone um die Ground-Truth Y gelegt und das Hadamard-Produkt mit der Predicition P gebildet. Da alle Matritzen als Einträge nur 0 oder 1 enthalten, findet durch das Hadamard-Produkt eine Art Verundung statt, sodass im Produkt nur Einträge eins sind, die in beiden Matritzen eins sind. Der Rest ist null. Durch die zuvor durchgeführte Pufferung mit $dilate_B$ sind allerdings auch Einträge gleich eins, die in Y zuvor null waren. Hierdurch besitzt das Hadamard-Produkt mehr Einträge mit eins, als bei der Berechnung der **IoU!**. Diese Zahl ist genau um die Anzahl an positiven Pixeln aus P größer, die in der Pufferzone liegen. Schließlich wird mit der $1,1$-Höldernorm für Matritzen die Anzahl an Einsen gezählt, was dann tp_B ergibt.
Die Formel für die Quality ist analog zur **IoU!**, bis auf die Verwendung der jeweils gepufferten Variablen: [1]

$$quality = \frac{tp_B}{tp_B + fp_B + fn_B} \ . \tag{2.18}$$

2.4. Architekturkomponenten

Im Folgenden werden verschiedene Architekturkomponenten diskutiert, die im **ML!** allgemein bzw. bei semantischer Segmentierung im Speziellen verwendet werden. Hierzu werden zunächst Dropout-Layer und Batch-Normalization-Layer begutachtet und dann die U-Net-Architektur zur semantischen Segmentierung vorgestellt.

2.4.1. Dropout

Dropout ist eine ressourcenschonende Regularisierungstechnik für **ML!**-Modelle. Hierbei werden einzelne Neuronen mit einer Wahrscheinlichkeit von *Rate r* während des Trainings deaktiviert, also deren Output auf 0 gesetzt.
Da bei der Inferenz Dropout dazu führen kann, dass wichtige Features ignoriert werden, ist Dropout während der Inferenz unerwünscht. Ohne Dropout während der Inferenz sind allerdings alle Gewichte aktiv, was zu einer höheren Summe der Gewichte während der Inferenz, als während des Trainings führt. Deswegen müssen die Gewichte für die Inferenz nach unten skaliert werden. Alternativ können während des Trainings alle Gewichte nach oben skaliert werden, die nicht deaktiviert wurden. Somit muss für die Inferenz keine

[1] Code-Ausschnitt A.1 zeigt die dazugehörige Python-Implementation.

Anpassung vorgenommen werden. Nach jedem Trainings-Batch werden die aktivierten Neuronen dann um Faktor $\frac{1}{1-r}$ skaliert [3, S. 255–258, 13].

Es hat sich gezeigt, dass Dropout eine effektivere Regularisierungstechnik zur Minderung von Overfitting ist, als andere ressourcenschonende Techniken, wie *Weight-Decay*, *Filter-Norm-Constraints* oder *Sparse-Activity-Regularization*, wobei Dropout mit diesen kombiniert werden kann, für noch bessere Regularisierung [3, S. 265].

2.4.2. Batch-Normalization

Batch-Normalization normalisiert und standardisiert den Output von Neuronen auf Basis des Mittelwerts und der Streuung einer Batch während des Trainings. Für die Inferenz werden Durchschnittswerte des Mittelwerts und der Streuung der Batches des Trainingsdatensatzes herangezogen und angewandt.
Batch-Normalization führt zu einer schnelleren Konvergenz im Training, sodass die Anzahl an benötigten Epochen in manchen Fällen bis zu halbiert werden können. Des Weiteren führt Batch-Normalization zu einer gewissen Regularisierung, da es die Kostenfunktion zu einem gewissen Grad glättet [3, S. 317–320, 14]. Besonders gut funktioniert Batch-Normalization für **CNN!**s und Netzwerke mit Sigmoid-Aktivierungsfunktion [3, S. 425].

2.4.3. U-Net

Die *U-Net-Architektur* beschreibt eine *Fully-Convolutional-Network-Architektur*, die erstmals in Freiburg 2015 vorgestellt wurde und herausragende Ergebnisse für verschiedene Benchmarks, insbesondere zur semantischen Segmentierung kleiner Datensätze, liefert.
Aus Abbildung 2.3 geht die namensgebende Architektur der U-Net hervor. Die folgenden Besonderheiten führen zu der sehr guten Performanz des Netzes bei semantischer Segmentierung [7]:

- Das Netz besteht aus einem kontrahierenden Encoder-Teil (linke Hälfte) und einem expandierenden und symmetrisch aufgebauten Decoder-Teil (rechte Hälfte). Der Encoder erzeugt feinere *Feature-Maps* mit zunehmender Netztiefe, während der Decoder diese wieder extrapoliert, was zu einer besseren Lokalisierung führt.

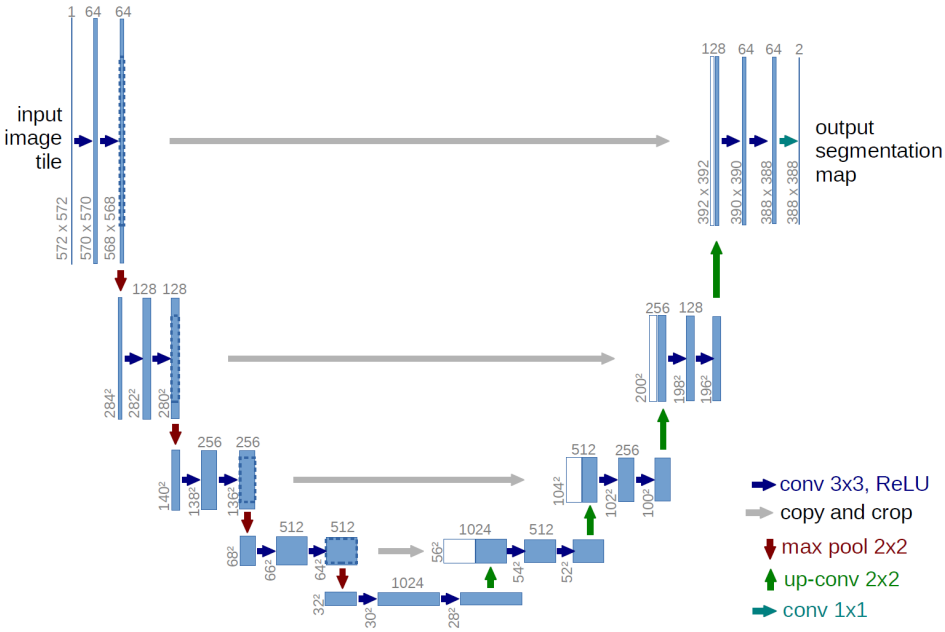

Abbildung 2.3.: Ursprüngliche U-Net-Architektur [7].

- Zwischen den jeweiligen symmetrischen Encoder- bzw. Decoder-Blöcken befinden sich *Skip-Connections*[2]. Zusammen mit dem vorherigen Punkt erhöht dies weiter die Lokalisierung und Performanz, da der jeweilige Decoder-Block feinere Features von der *Up-Convolution*[3], wie auch den größeren Kontext von früheren Blocks mittels Skip-Connection erhält.
- Zur Mitte des Netzes hin erhöht sich die Anzahl der Convolution-Filter und damit die Anzahl der *Channel*, während sich die Dimensionen der einzelnen Feature-Maps durch das *Downsampling* verringert. Hierbei ist anzumerken, dass bei der Implementation kein *Padding* für die Convolutions verwendet wurde, wodurch sich die Dimensionen der Feature-Maps nach jeder Convolution verringert. Hieraus folgt ein Zuschneiden für die Skip-Connections.

[2] *copy and crop* in der Abbildung.
[3] implementiert als *Transposed Convolutions*.

2.5. Vortrainierte CNN!-Modelle

Im Folgenden wird eine Auswahl an **CNN!**-Modellen vorgestellt, von denen es öffentlich zugängliche, vortrainierte Instanzen zur freien Verwendung gibt. Diese können zu Transfer-Learning-Zwecken genutzt werden, indem sie zum Beispiel als Teilnetz in eine **CNN!**-Architektur eingebaut werden (mehr dazu in Abschnitt 2.6).
Die Klassifikations-Modelle sind auf dem Bildklassifikationsdatensatz *ImageNet*, der über 14 Mio. Bilder mit 1000 Objektklassen enthält, trainiert. Der ImageNet-Datensatz ist der größte und bekannteste Datensatz seiner Art und ist sehr beliebt für Benchmarks und Pre-Training.

2.5.1. VGG16

VGG16 ist ein **CNN!**, welches 2015 von der *Visual Geometry Group* vorgestellt wurde und zu der Zeit bahnbrechende Ergebnisse bei der Bilderklassifizierung lieferte [15]. Inzwischen ist es ein Standard-Netz, auf welchem viele weitere Architekturen aufbauen, um verschiedene Verbesserungen zu realisieren. Dadurch ist es auch sehr beliebt für Transfer-Learning.

VGG16 besteht aus fünf Blöcken mit $3x3$-Convolution-Schichten, gefolgt von drei Fully-Connected-Schichten. Die ersten beiden Convolution-Blöcke haben zwei Convolution-Schichten, die letzten drei Blöcke dagegen jeweils drei. Jeder Convolution-Block wird gefolgt von einer Maxpool-Schicht und jeder bis auf den vorletzeten verdoppelt die Filteranzahl in den Convolution-Schichten, beginnend bei 64 Filtern im ersten Block bis hin zu 512 Filtern in den letzten beiden Blöcken [15][4].

Das gesamte Netz besitzt 138 Mio. Parameter. Werden die drei Fully-Conntected-Schichten am Ende ausgelassen, sind es ungefähr 14,7 Mio. Parameter.

2.5.2. ResNet34

Damit Modelle mehr und feinere Features lernen können, müssen sie in der Lage sein, sehr komplexe Funktionen darzustellen. Das führt dazu, dass tiefere Netze auf dem

[4] Abbildung A.1 veranschaulicht VGG16.

ImageNet-Datensatz tendenziell bessere Ergebnisse erzeugen, als flachere. Hierin lag auch der Erfolg von VGG16 bzw. VGG19. Jedoch führt das einfache Vertiefen von Netzen zu dem Degredation-Problem, wonach flachere Netze besser performen, als tiefere, da die tieferen sehr schwer zu trainieren sind.
Dieses Problem adressiert *ResNet* erfolgreich, indem *Residual-Blocks* eingeführt werden. Hierdurch können mit *ResNet* tiefere Netze erstellt werden als bisher, die trotzdem noch effizient trainiert werden können. Ein Residual-Block besteht dabei aus zwei Convolutional-Layer mit einer Skip-Connection zwischen Input der ersten Layer und Output der zweiten Layer. Durch die Skip-Connections kann bei der Backpropagation der Gradient ungehindert rückwärts laufen, um so auch die flachen Layer upzudaten. So kann z.B. *ResNet34* 34 trainierbare Layer haben, während zuvor mit VGG nur 16 bzw. 19 effektiv waren [16].

Die Architektur ist dabei sehr simpel. Sie stellt - im Falle von ResNet34 - im Prinzip 33 Convolutional-Layer mit wiederholtem Downsampling und Skip-Connections zw. jeweils zwei Convoltuional-Layern, gefolgt von einer Fully-Connected-Layer dar [16][5].

ResNet34 hat 63,5 Mio. Parameter. Wird auf die letzte Fully-Connected-Layer verzichtet, sind es 21,2 Mio. Parameter.

2.5.3. DenseNet121

DenseNet121 wurde 2018 erstmals vorgeschlagen und wurde designed, um das Vanishing-Gradient-Problem zu verbessern, bessere Feature-Übertragung in tiefere Netzschichten zu ermöglichen und das Wiederverwenden von Features zu unterstützen, um somit die Anzahl der benötigten Parameter deutlich zu reduzieren, bei gleichbleibender Performanz [17].

Das Netz besteht aus mehreren sogenannten Dense-Blöcke, die wie folgt aufgebaut sind: Jede angegebene Convolution-Schicht besteht aus einer Batch-Normalization-Schicht mit **ReLU!**-Aktivierung, gefolgt von der angegebenen Convoltuion-Schicht. Außerdem - und hier liegt die große Erweiterung des DenseNet - sind alle Convolution-Schichten eines Dense-Blocks mit allen nachfolgenden Convolution-Schichten des Dense-Blocks

[5] Abbildung A.2 veranschaulicht ResNet34.

konkateniert, anstatt nur mit dem einen Nachfolgenden, wie es zum Beispiel bei VGG16 der Fall ist[6].

DenseNet121 enthält circa 7,6 Mio. Parameter. Ohne die letzte Fully-Connected-Schicht sind es noch circa 6,9 Mio. Bei nur halb so vielen Parametern erreicht DenseNet121 leicht bessere Ergebnisse als VGG16 auf dem ImageNet-Datensatz (93% ggü. 92% Accuracy für ImageNet-Top5) [17].

2.6. Transfer-Learning mit U-Net

Transfer-Learning beschreibt das Übertragen von trainierten Gewichten eines **ML!**-Modells auf ein anderes, bestenfalls ähnliches Problem und Modell. Das Modell wird dann via *Fine-Tuning* verfeinert mit dem neuen Datensatz und unterschiedlichen Trainingsmethoden. Häufig wird dafür ein Teil des Modells eingefroren, sodass sich die eingefrorenen Gewichte nicht verändern können. Dies verhindert, dass die bereits vortrainierten Gewichte durch die erste Trainings-Batch ruiniert und unbrauchbar werden. Eine weitere Möglichkeit ist, mit einer sehr geringen Lernrate das gesamte Modell zu trainieren. Oft werden beide Ansätze auch verbunden.

Der Vorteil von Transfer-Learning liegt darin, dass das Training deutlich kürzer dauert, weil direkt mit einer höheren Genauigkeit eingestiegen wird, mit kleineren Datensätze bessere Ergebnisse erzielt werden können und auch insgesamt eine höhere Genauigkeit am Ende des Trainings erreicht wird, als bei herkömmlichem Training. Diese Effekte sind verstärkt, abhängig davon, wie ähnlich der Datensatz des Pre-Trainings und des eigentlichen Trainings sind [18].
Insbesondere bei Computer-Vision ist Transfer-Learning effektiv, da bei Bildern high-level Features wie Clustering ähnlichfarbener Pixel oder Kantenerkennung oftmals sehr ähnlich zwischen unterschiedlichen Datensätzen ausfallen und damit schon vorhanden sind [18].

Für Transfer-Learning mit U-Nets gibt es verschiedene Strategien: Backbone-Netze als Encoder, partielles Einfrieren verschiedener Netzbereiche und direktes Trainieren mit geringer Learning-Rate. Der Stand der Wissenschaft diesbezüglich wird im Folgenden vorgestellt.

[6] Abbildung A.3 veranschaulicht DenseNet121.

2.6.1. Training mit Backbones

Im Kontext von Transfer-Learning bei U-Nets bezeichnet ein *Backbone* ein etabliertes vortrainiertes **CNN!**, welches, leicht modifiziert, als Encoder für das U-Net verwendet wird. Hierbei wird der Decoder-Teil des U-Net symmetrisch dem Encoder nachempfunden und an passenden Stellen Skip-Verbindungen zwischen En- und Decoder eingebaut. Hierdurch kann eine geeignete **CNN!**-Architektur für das spezifische Problem ausgewählt werden. Des Weiteren sind diese Modelle auf sehr großen Datensätzen, wie *ImageNet*, vortrainiert öffentlich zugänglich.

Bei der semantischen Segmentierung von medizinischen Lungen-Ultraschall-Bildern, wurden die besten Ergebnisse von einem Dice-Maß-Standpunkt aus, mit einem U-Net mit auf ImageNet trainierten *VGG16*-Backbone erzielt. Das Vergleichsnetz, welches zuerst auf dem *Salien Object* Datensatz vortrainiert wurde, erzielte schlechtere Ergebnisse von dem Dice-Maß her, wobei allerdings das VGG16-U-Net kleine falsch-positive Regionen erkannte, die weit außerhalb der Ground-Truth lagen. Die falsch-positiven beim Vergleichsnetz, lagen direkt an der Ground-Truth, dies lässt auf eine sensitivere Kantenerkennung beim VGG16-U-Net schließen, die manchmal aber auch übersensitiv war [19].

Für das Training des VGG16-U-Nets wurde der Encoder-Teil eingefroren und somit nur der Decoder trainiert.

2.6.2. Partielles Einfrieren und Training mit geringer Lernrate

Im oben beschriebenen Problem zur semantischen Segmentierung wurde das vortrainierte Vergleichsnetz auf zwei Weisen fein-trainiert:

1. Ohne Einfrieren mit einer Lernrate von 10^{-5}
2. und mit Einfrieren des mittleren Blocks, welcher ungefähr $14 \cdot 10^6$ der insgesamt $31 \cdot 10^6$ Parameter enthielt.

Die 5-fache Kreuzvalidierung ergab sowohl für den besten Lauf, als auch für den durchschnittlichen Lauf, ein besseres Dice-Maß für das Training aller Parameter. In keinem der Fälle gab es, anders als beim VGG16-U-Net, segmentierte Regionen ohne Zusammenhang mit der Ground-Truth [19].

In einem weiteren Paper wurde untersucht, welche Layer eines U-Net am besten eingefroren werden sollten, für das Fine-Tuning von medizinischen Bildern - zum einen von Lungen-Ultraschall-Bildern und zum anderen von Brust-Röntgen-Aufnahmen. Hier wurde wieder mit dem *Salien Objetcs* Datensatz vortrainiert. Dann wurden für beide Anwendungsfälle folgende Tests durchgeführt:

1. Einfrieren der linken Hälfte (Encoder) des Netzes,
2. Einfrieren der rechten Hälfte (Decoder) des Netzes,
3. gesamtes Netz, bis auf den ersten Block eingefroren und dann nach jeweils fünf Epochen sukkzessive weitere Blöcke freigeben und
4. dasselbe allerdings von hinten nach vorne.

Für die Röntgenaufnahmen gab es keine Unterschiede, wobei der Dice-Score hier allerdings auch bei 0.98 lag. Für die Ultraschallbilder lieferte Methode (1) die schlechtesten Ergebnisse (Dice: 0.72), gefolgt von Methode (2) (Dice: 0.80) und gleichermaßen (3) und (4) (Dice: 0.82), wobei (3) deutlich schneller konvergierte [20].
Es ist ersichtlich, dass die Methodik beim Fine-Tuning, abhängig vom Datensatz, einen erheblichen Einfluss auf die abschließende Performanz des Modells haben kann und stets berücksichtigt und gegen Alternativen abgewogen werden sollte, um beste Ergebnisse zu erzielen. Die vorgestellten Methodiken können später genutzt werden, um eine Transfer-Learning-Methode für das Problem der Radwegerkennung zu konzipieren.

2.7. Aktueller Stand bei der Straßendetektion

Die Erkennung von Fahrradwegen soll auf der bereits existierenden Straßenextraktion aus Satellitenbildern aufbauen. Hierfür werden vorhandene Datensätze für Straßen vorgestellt. Im Anschluss werden auf den aktuellen Stand und Benchmark-Optionen eingegangen.

2.7.1. Datensätze

Im Folgenden sind alle relevanten Datensätze zur Straßenerkennung aus Luftbildern aufgeführt. Eine Auswahl der später zum Pre-Training verwendeten Datensätze findet im Abschnitt 3.6 statt.

Ein Testdatensatz ist hierbei ein meist sehr kleiner Datensatz, der ausschließlich zum Testen und Bewerten eines bereits trainierten Modells verwendet werden soll. Im Rahmen der Straßenerkennung ist dies besonders relevant, da sich unterschiedliche Städte in unterschiedlichen Ländern und Kulturen stark unterscheiden können, während die Trainingsdaten meist nur aus einer Stadt, die in sich ein eher homogenes Bild aufweist, bestehen. Der Testdatensatz soll also die Generalisierungsfähigkeit des Netzes testen, um zu sehen, ob das Modell, das bspw. auf einem Datensatz einer modernen amerikanischen Stadt trainiert wurde, auch auf Bildern einer mitteleuropäischen Altstadt funktioniert. Deswegen ist es wichtig, dass keine Bilder des Testdatensatzes im Training verwendet werden.

Beim *Massachusetts Roads Dataset* liegen 1171 Luftaufnahmen aus dem Gebiet Massachusetts vor. Jedes der Bilder besteht aus 1500×1500 Pixeln, was einer Fläche von $2.25km^2$ entspricht. Die gesamte Abdeckung sind $2600km^2$ bestehend aus städtischer und ländlicher Gegend. Der Split erfolgt im Verhältnis $95:1:4$. Über den Split wird sichergestellt, dass keine Durchmischung der Test- und Validierungsdaten mit den Trainingsdaten erfolgt und die Bewertung des Netzes verfälscht wird. Es gibt also 1108 Trainings-, 14 Validierungs- und 49 Testbilder. Die Zielbilder sind über eine Rasterung der Straßenmittellinien aus dem OpenStreetMap-Projekt generiert worden. Die Liniendicke der gelabelten Straßen beträgt $7px$ und ist ungeglättet, da hiermit bessere Ergebnisse erzielt worden sind [21, S. 85f]. Der Zuschnitt der Bilder sorgt teilweise für einen hohen Anteil weißer Stellen in den Bildern.

Das *Buffalo Roads Dataset* ist als zusätzliches Testset für das Massachusetts Roads Dataset zu sehen. Hintergrund ist die Überprüfung des Netzes mit Daten aus einem anderem Gebiet unter anderen Bedingungen. Es besteht lediglich aus 30 Orthophotos $(5:5:20)$ mit einer Größe von 609×914 und einer Auflösung von $1\frac{px}{m^2}$. Die Generierung der gelabelten Daten erfolgt analog zum Massachusetts Road Set über OpenStreetMap. Größter Unterschied ist die Verdeckung von Straßen durch Bäume [21, S. 86–88].

LandCover.ai kann für die Erkennung von Gebäuden, Waldgebieten, Gewässern und Straßen verwendet werden. Die Aufnahmen stammen aus Polen und Zentraleuropa in einer Größe von $21627km^2$. 33 Orthophotos haben eine Auflösung von $25\frac{cm}{px}$ mit 9000×9500 Pixeln, 8 Bilder von $50\frac{cm}{px}$ mit 4200×4700 Pixeln [22]. Die Bilder sind hand-annotiert und beinhalten einen großen Anteil an ländlichen Gebieten.

Mithilfe des *Deep Globe Road Extraction Datasets* können in Kathastrophgebieten Informationen über Karten und Zugänglichkeitsmöglichkeiten für die Krisenbewältigung gesammelt werden. Der Datensatz besteht aus 6226 Satellitenbildern in RGB mit einer Größe von 1024×1024. Die Auflösung der Aufnahmen beträgt $50cm$. Mit 1243 Validierungs- und 1101 Testdaten ist der Split in Training, Validierung und Test zu $73:15:13$ erfolgt. Der hohe Aufwand zur Erstellung der Labels führt zu Ungenauigkeiten, insbesondere in ländlichen Regionen. Kleine Straßen innerhalb von Landwirtschaftsflächen sind bewusst unbeschriftet geblieben. Die Bilder sind ursprünglich über Thailand, Indonesien und Indien mit je 19584×19584 Pixeln aufgenommen worden. Insgesamt entspricht es einer Fläche von $2220km^2$ [23].

Mit finanzieller Unterstützung des Chesapeak Bay Programms wurde ein Datensatz *Land Cover* zur Landnutzung und Bodenbedeckung erstellt. Es lassen sich Einblicke in die Landschaft und die Verwaltung der Regionen gewinnen. Das Einzugsgebiet umfasst $250.000km^2$ mit einer Auflösung von $1m$. Damit handelt es sich bei diesem Datensatz um den größten bei der Analyse von der Landnutzung im Vergleich zur Bodenbedeckung [24].

Bei den nachfolgenden drei Datensätzen handelt es sich um Testdatensätze. Das *Toronto Dataset* umfasst $1,45km^2$ im Umkreis der Stadt Toronto in Canada. Darin enthalten sind ca. $1km$ Straßen mit einer Bodenauflösung von $15cm$. Zusätzlich werden Laserscanner mit $6\frac{points}{m^2}$ bereitgestellt. Im Datensatz ist eine hohe Varianz bezüglich der Gebäudehöhe, Struktur der Dächer und verschiedener Straßen vorzufinden. Der Testbestand kann für die Überprüfung von Objektextraktion und Gebäuderekonstruktionstechniken in zwei weitere Teilgebiete unterteilt werden. Für Straßen können alle Testdaten verwendet werden. Die Labels sind manuell mithilfe von CloudCompare erstellt worden [25, 26].

Der Testdatensatz *Vaihingen/Enz* besteht aus einer Untermenge der Daten, welche für einen Test von digitalen Luftbildkameras verwendet wurden. Er ist in drei Testgebiete für verschiedene Objektklassen und in einen größeren Bereich für das Überprüfen von Straßenextrakion aufgeteilt, wobei die drei Gebiete hier mit inbegriffen sind. Die einzelnen Gebiete sind in verschiedene Kategorien unterteilt: Hochhäuser, Wohngebiete mit kleineren Einfamilienhäusern und Innenstadt mit dichter Bebauung. Die Bodenauflösung beträgt $8cm$ [25].

Für die Stadt *Potsdam* gibt es einen ähnlichen Datensatz. Dieser zeigt eine typsiche Altstadt mit großen Gebäudeblöcken, kleinen Straßen und einer dichten Besiedelung.

Die **GSD!** beträgt $5cm$. Zur Vermeidung von Randbereichen ohne Daten, sind zentrale Ausschnitte verwendet worden. Übrig gebliebene Datenlücken wurden interpoliert [27, 28].

2.7.2. Aktueller Stand und Benchmarks

Die aktuellen State-Of-The-Art-Methoden zur Extraktion von Straßen aus Satelliten- und Luftbildern verwenden alle Computer-Vision mittels **CNN!**s und dabei ausschließlich Netze aus der U-Net-Familie - also angepasste, erweiterte, abgeänderte U-Nets [29, 30, 31, 32].
Dies liegt vor allem daran, dass das U-Net, welches ursprünglich für biomedizinische Bilder entworfen wurde, genau die Probleme adressiert, die auch beim Segmentieren von Straßen auf Luftbildern auftreten. Wie auch in medizinischen Bildern leiden die Luftbilder-Datensätze häufig unter großer Klassen-Imbalance und unter ähnlichen Annotationsfehlern. Außerdem haben viele medizinische Bilder und Luftbildaufnahmen von Straßen ähnliche Topologien. Komplexere Architekturen zur semantischen Segmentierung, wie zum Beispiel solche für Bodenaufnahmen-Datensätze (vgl. KITTI [33]) erreichen nicht die Performance von U-Nets bei Luftbild-Benchmarks [29].
Weiter wird die Extraktion von Straßen aus Satelliten- bzw. Luftbildern zumeist in zwei Phasen eingeteilt:

1. das Erstellen einer binären Maske zu den Luftbildern mittels semantischer Segmentierung, welche die Straßen pixelweise induziert,
2. das Extrahieren eines topologischen Graphens, welcher die Straßen-Zentrumslinien beschreibt.

Schritt (1) wird dabei zumeist von U-Net-Derivaten behandelt, wobei es allerdings auch Vorschläge von Netzen gibt, die direkt Schritt (2) ausführen [29]. Im Folgenden wird weiter Schritt (1) betrachtet.

Der Vergleich von unterschiedlichen Netzen, Papern und Ergebnissen zur Straßenerkennung ist häufig erschwert, da es trotz gleicher Bewertungsmaße wie Dice oder **IoU!**, zwei unterschiedliche gängige Kodierungen gibt:

1. Das verwendete **CNN!** hat je Input-Neuron bzw. -Pixel p genau ein Output-Neuron o_p, welches den jeweiligen Input-Pixel p als Straße identifiziert, genau dann, wenn der Wert des Output-Neuron o_p einen gewissen Grenzwert l überschreitet ($o_p > l$), ansonsten ($o_p \leq l$) gilt der Pixel p *implizit* als Nicht-Straße, bzw. *Hintergrund*.
   In diesem Fall stellt ein korrekt als Hintergrund klassifizierter Pixel ein wahr-negatives Ergebnis dar. Wahr-Negative werden von Dice, bzw. **IoU!** allerdings nicht berücksichtigt (vgl. Gleichung (2.9) und 2.11). Ein so klassifizierter Pixel ändert also nichts an dem Score. Es werden nur Straßen betrachtet, nicht aber der Hintergrund.

2. Das verwendete **CNN!** hat je Input-Neuron bzw. -Pixel p *zwei* Output-Neuronen $\mathbf{o_p} = (o_{p_S}, o_{p_H})$, wobei dies eine One-Hot-Kodierung von Punkt (1) darstellt. Ein Hintergrund-Pixel wird nun *explizit* als solcher klassifiziert. Dies ändert allerdings die Berechnungsgrundlage von Dice und **IoU!** erheblich, da diese nun (zumeist) als Mittelwert der Scores zu o_{pS} und o_{pH} berechnet werden. Hierdurch fließen die zuvor unberücksichtigten wahr-negativen Hintergrundpixel als wahr-postive in den Dice- bzw. **IoU!**-Score ein. Die Verzerrung wird noch verstärkt, da eine starke Imbalance zwischen Straßen- und Hintergrundpixel herrscht, wodurch der Dice-/**IoU!**-Score bei der One-Hot-Kodierung viel mehr eine Aussage darüber trifft, wie viele Hintergrundpixel, von denen es ja viel mehr gibt, richtig klassifiziert wurden. Die so erzielten Dice-/IoU-Werte sind hier rein numerisch deutlich höher.

Im Weiteren werden aktuelle Ergebnisse zu den Massachusetts- und Deep-Globe-Datensätzen vorgestellt, um zu Ermitteln, was für diese Datensätze gut funktioniert, sodass diese state-of-the-art Erkenntnisse beim späteren Entwurf eines oder mehrerer Netze zum Erkennen von Fahrradwegen (s. Abschnitt 3.3) genutzt werden können. Die beiden Datensätze sind für die Literaturrecherche ausgewählt, da sie sehr häufig als Benchmark zum Erkennen von Straßen genutzt werden, es dazu sehr viel Literatur und Forschung gibt und da beide Datensätze sich stark unterscheiden, soweit das im Rahmen der Straßenerkennung möglich ist. Haben die State-Of-The-Art-Lösungen trotz des starken Unterschieds der Datensätze ähnliche gut funktionierende Komponenten, stärkt das die Annahme, dass diese Komponenten auch für das Erkennen von Radwegen nützlich sein könnten.
Tabelle 2.1 zeigt Resultate zum Massachusetts- und Deep-Globe-Datensatz von sogenannten *Basline*-Modellen. Hierbei handelt es sich um verschiedene Architekturen, die allerdings nicht extrem auf den jeweiligen Datensatz optimiert sind, was die Hyper-Parameter oder manche Architektur-Anpassungen angeht. Ein Beispiel dafür wäre das

Model	Massachusetts		Deep Globe	
	Dice	IoU	Dice	IoU
DeepLabv3+*	69,35	52,95	75,19	59,65
D-LinkNet50	71,01	54,90	74,04	58,12
U-Net*	71,91	55,92	76,82	61,97
Res-U-Net50	72,74	56,93	78,62	64,55
Dense-U-Net-121	**73,03**	**57,12**	**79,19**	**65,13**

Tabelle 2.1.: Baseline-Resultate verschiedener Modelle auf dem Massachusetts- bzw. Deep-Globe-Datensatz in Prozent, aufsteigend sortiert. Nicht one-hot-kodiert. * nicht vortrainiert. [29].

Ausprobieren von verschiedenen (Kompositions-)Kostenfunktionen. Mit den Baseline-Resultaten können diese Modelle zu Vergleichszwecken verwendet werden und es bestehen Daten zu mehreren Benchmarks [29].

Tabelle 2.2 zeigt hingegen die top-drei optimierten Modelle, die derzeit die Leaderboards des jeweiligen Datensatzes anführen. Alle hiervon sind U-Net-Derivate. Mit der Modell-Optimierung kann *Dense-U-Net-121* aus Tabelle 2.1 beispielsweise beim Massachusetts-Datensatz bis zu 66,61% erreichen, anstelle von den 57,12% wie im Baseline-Fall. Dense-U-Net-121 ist dabei ein U-Net mit einem vortrainierten DenseNet121 als Backbone [29]. Im Falle vom RDRCNN (Refined Deep Residual Convolutional Neural Network), konnte die große Verbesserung erzielt werden, indem das U-Net so angepasst wurde, dass die Encoder-Blöcke mit Residual-Blöcken[7] ersetzt wurden und der Bottleneck-Block um Dilation erweitert wurde [34]. Beim EOSResUNet wurden ebenfalls die Encoder-Blöcke mit Residual-Blöcken ersetzt und zusätzlich nach **IoU!** statt nach Dice oder **BCE!** optimiert [35].
Die besten Netze beider Datensätze haben die Gemeinsamkeit der U-Net-Struktur, wie auch die Residual-Blöcke (bzw. sogar Dense-Blöcke, was als eine Erweiterung der Residual-Blöcke aufgefasst werden kann). Diese Informationen können zur Konzeption eines Netzes zur Radwegerkennung verwendet werden.

[7]Zwei Convolutional-Layer bei dem eine Skip-Connection zwischen Input der ersten und Output der zweiten Schicht besteht (s. Unterabschnitt 2.5.2).

	Massachusetts			Deep Globe		
Model	*RDRCNN*	*Dense-U-Net-121*	*WRAU-Net*	*EOSResUNet*	*D-LinkNet*	*U-Net-like ResNet34*
IoU	**67,10**	66,61	64,58	**65,60**	64,12	64,00

Tabelle 2.2.: Optimierte Resultate verschiedener Modelle auf dem Massachusetts- bzw. Deep-Globe-Datensatz in Prozent, absteigend sortiert von links nach rechts. Nicht one-hot-kodiert. [29].

3. SOLID

hier kurz beschreiben, was so grob abgeht, also was wir so machen wollen und was für netze wir vergleichen wollen, um das beste zu finden

3.1. Datensatz für Fahrradwege

3.2. Pre-Processing

Dieser Abschnitt befasst sich mit dem Pre-Processing, welches auf den in **??** beschriebenen Datensatz angewandt wird. Insbesondere ist die Klassenimbalance, Eingabegröße, Training-Validation-Test-Split und die Daten-Augmentation zu diskutieren.
Für die Verwendung im Modell werden die Pixelwerte aller Bilder von $[0; 255] \subset \mathbb{N}$ auf $[0; 1] \subset \mathbb{R}$ abgebildet, indem alle Kanäle - bei RBG drei, bei den Graustufen-Masken einer - durch 255 geteilt werden. Dies vermindert die betragsmäßige Größe der Eingaben und aufgrund der Masken auch Ausgaben.

3.2.1. Eingabegröße und Klassenimbalanceausgleich

Durch die Art des Problems besteht bei der Erkennung der Fahrradwege ohnehin schon eine starke Klassenimbalance zwischen den Radweg-Pixel und den Hintergrundpixel. Diese Diskrepanz ist noch ein mal drastischer als bei den Straßendatensätzen, wo es schon ein Problem darstellt. So ist auf einem Bild mit Radweg tendenziell sehr wenig Radweg, aber sehr viel sonstige Strukturen, wie Vegetation, Gebäude und Straßen. Durch die automatische Generierung des Radweg-Datensatzes sind zudem viele Bilder von kleinen Orten, Industriegebieten, Feldern und Wäldern vorhanden, die keinerlei eingezeichnete Radwege besitzen. Selbst in Bildern der Innenstadt haben oft nur größere Straßen dedizierte Radwege, während der Großteil des Bildes mit Wohngebiet gefüllt ist. Somit ist nur ein verschwindender Anteil aller Pixel des Datensatzes als Radweg markiert.

Um den Anteil an Radwegpixel zu erhöhen, sollen nur Bilder zum Training verwendet werden, die überhaupt Radwege enthalten. Da es technisch zu ressourcenaufwendig wäre ein ganzes 10.000×10.000-Pixel-Bild einzugeben, sollen die Bilder in kleinere Stücke zerteilt werden. Dies ist auch bei den Modellen zur Straßenerkennung aus Unterabschnitt 2.7.2 die Praxis. Vorgeschlagene Ausschnittsgrößen könnten hierbei 256×256, 512×512 und 1024×1024 sein. Wird ein größerer Ausschnitt gewählt, steigt potentiell die Klassenimbalance, da häufig viele überflüssige Hintergrundpixel eingeschlossen werden, allerdings kein weiterer Radweg im Bild liegt. Wird eine kleine Größe gewählt, könnte es sein, dass nur noch Radwege im Bild vorhanden sind, wodurch die Gefahr besteht, dass jede beliebige Straße einen Radweg seitlich angezeichnet bekommt. Es wären hauptsächlich große Straßen abgebildet – kleine Straßen würden kaum gelernt werden. Außerdem würde es häufiger dazu kommen, dass Radwege nur sehr klein in den Ecken eines kleinen Bildausschnittes vorkämen und die lange zusammenhängende Struktur, die zu einem Radweg gehört, schwierige zu erlernen wäre. Aus diesen Gründen wird als Kompromiss die 512×512-Größe gewählt. Somit entspricht eine Bildkante $512 \cdot 0,2m = 102,4m$. Die Zweierpotenz bzw. eher das Vielfache von 32 wird daher gewählt, sodass das Bild nicht zu klein in der U-Net-Struktur wird, bzw. damit eine saubere Teilung der Max-Pool-Schichten möglich ist, die die Bildkantenlänge stets halbieren.

Jedes 10.000×10.000-Pixel-Bild und die dazugehörige Maske wird in $\left\lceil \frac{10.000}{512} \right\rceil^2 = 400$ Teile geschnitten. Die 39 Ausschnitte, die nur partiell im Bild liegen, werden im überstehenden Bereich mit Schwarz ($RGB = (0,0,0)$) gefüllt. Damit ergeben sich bei 143 Bildern $143 \cdot 400 = 57.200$ 512×512-Bildausschnitte. Weiter werden alle Ausschnitte entfernt, die keine oder fast keine als Radweg annotierten Pixel beinhalten. Die Quote wird auf 1% festgelegt. Sollte also ein Bildausschnitt weniger als 1% Fahrradweg-Pixel beinhalten, wird es entfernt. Durch diese Maßnahme sinkt die Anzahl an Bildausschnitten von 57.200 auf 10.181 Bildausschnitte - dieser Anteil entspricht ca. 17,7%.

Abbildung 3.1.: Beispiel 512×512-Ausschnitt aus Bike-Datensatz mit roter Maske.

Abbildung 3.1 zeigt beispielhaft ein auf 512×512 Pixel zugeschnittenes Bild aus dem Bike-Datensatz mit der dazugehörigen Maske, die mit 50% Transparenz in rot überlagert

ist. Es ist zu erkennen, dass die Klassenimbalance noch recht stark ist, allerdings deutlich geringer, als in den ursprünglichen Bildern, und dass auch trotzdem genügend Straßen ohne Radwege vorhanden sind und die Struktur und der Verlauf des Radweges bei der gewählten Ausschnittgröße weiterhin gut zu erkennen ist.

3.2.2. Training-Validation-Test-Split

Zunächst werden die zerschnittenen Bilder aller Städte gemischt. Dann werden diese Ausschnitte disjunkt in Training-, Validation- und Test-Daten aufgeteilt. Damit sollten die unterschiedlichen Städte gleichermaßen in allen Teil-Datensätzen auftauchen, sodass der Test- und Validationdatensatz gut die Generalisierungsfähigkeit des Netzes überprüft. Durch die bereits höhere Zahl an Bildausschnitten (10.181 Stück) ist es eher unwahrscheinlich, dass eine ungleiche Verteilung der Städte vorkommt. Außerdem ist das Mischen wichtig, damit die Randausschnitte, die große schwarze Flächen beinhalten, anteilig gleich in jedem Teildatensatz repräsentiert sind, damit dies das Ergebnis nicht verfälscht, sollten diese zum Beispiel nur in dem Validation- oder Testdatensatz vorkommen.

Tabelle 3.1 zeigt den im Folgenden verwendeten Trainings-Validation-Test-Split. Hierbei sind zwei Dinge zu bemerken, die nicht aus der Tabelle hervorgehen: Zunächst wurde der Testdatensatz abgespaltet. Hierzu wurden 15% vom Hannover-Teil und 20% von den jeweiligen restlichen Städten abgespaltet und vereinigt. Da der Hannover-Teil so groß ist, wie die restlichen Städte zusammen, läuft dies auf einen eher unkonventionellen Split von 17,5% heraus. Diese Verteilung wurde so getroffen, um den Hannover-Teil etwas weniger zu gewichten, um im Test die Generalisierungsfähigkeit besser zu testen und den Einfluss der kleineren Städte zu erhöhen. Der Validation-Datensatz wurde danach aus 8%, bzw. der Trainingsdatensatz aus 92% vom Rest gebildet.
Der Trainingsdatensatz ist mit ca. 75% der gesamten Daten eher groß gewählt. Diese Entscheidung wurde getroffen, um möglichst viele und unterschiedliche Bilder zu verwenden, da der Datensatz mit automatischer Synthese generiert wurde und daher qualitativ eher unvorteilhaft ist, wodurch eine höhere Anzahl an Trainingsdaten nötig wird.

Der Validationdatensatz wird genutzt, um nach jeder Trainingsepoche den Verlust dieses Datensatzes zu bestimmen. Verbessert sich der Validation-Verlust nach fünf Epochen nicht, wird die Lernrate um Faktor 10 verringert. Verbessert sich der Validation-Verlust

	Training	Val.	Test	Summe
Absolut	7738	672	1771	10181
Anteil	75,9	6,6	17,5	100

Tabelle 3.1.: Training-Validation-Test-Split des Bike-Datensatzes in gefilterten 512×512-Ausschnitten.

nach sieben Epochen nicht, wird der beste Stand, mit der Epoche mit geringstem Validation-Verlust, wiederhergestellt und das Training vorzeitig beendet.

3.2.3. Augmentation

Die Bild-Augmentation wird nicht vor dem Training angewandt, um den Datensatz künstlich zu vergrößern, sondern während dem Training pseudo-zufällig mit festem Seed, um die Ergebnisse reproduzieren zu können. Somit wird jedes Bild während des Trainings, bevor es in das Netz eingegeben wird zufällig augmentiert und dann eingegeben. Somit erhält ein und dasselbe Bild über die verschiedenen Epochen jedes Mal unterschiedliche Anpassungen auf Basis des Zufallsgenerators. Dies erhöht deutlich die Generalisierungsfähigkeit bzw. verringert die Gefahr von Overfitting drastisch. Das Netz sieht tendenziell viel mehr Varianten der Bilder. Der einzige Nachteil ist, dass das Training tendenziell länger dauert, da jedes Bild vor Eingabe bearbeitet wird. Dieser Nachteil ist vor allem evident, wenn mehrere Netze auf demselben Datensatz trainiert werden, da jedes mal dieselben Augmentationen (da gleicher Seed) erneut vorgenommen werden müssen. Dieses Problem könnte allerdings durch eine Vorausberechnung der aufeinanderfolgenden Augmentationen behoben werden. Die Verlängerung pro Epoche lag allerdings lediglich bei zwei Minuten beim Bike-Datensatz und fünf Minuten beim Pre-Training auf der Straßenerkennung, daher wurde auf den zusätzlichen Aufwand einer solchen Implementation verzichtet (vgl. eine Epoche benötigt im Schnitt 10-17 min bzw. 40-50 min).

Jede der folgenden Augmentationen wird in ihrem Wertebereich pseudo-zufällig angewandt. Sollte es die Veränderung erfordern, werden fehlende Bildteile mit Schwarz gefüllt. Schwarz wurde ausgewählt, da es ohnehin Bilder mit schwarzen Rändern im Datensatz gibt, weswegen das hierbei mit nichts Neuem konfrontiert ist.

1. Rotation im Intervall von $[-90°; 90°]$. Eine Gradzahl wird für jedes Bild zufällig ausgewählt und angewandt. Die dabei entstehenden leeren Bildbereiche werden mit

Schwarz ($RGB = (0, 0, 0)$) gefüllt. Diese Augmentation ist realistisch und nützlich, da Straßen in beliebiger Ausrichtung vorkommen und eine natürliche Orientierung bei Bildern aus der Vogelperspektive nicht besteht.

2. Horizontale Spiegelung. Es wird zufällig entschieden, ob ein Bild an der vertikalen Spiegelachse gespiegelt wird. Diese Augmentation ist ebenfalls realistisch und hat gegenüber der Rotation den Vorteil keine schwarzen Flächen einzuführen aber den Nachteil nicht so viele neue Bilder erzeugen zu können.

3. Vertikale Spiegelung. Wie horizontale Spiegelung, allerdings wird an der horizontalen Achse gespiegelt. Beide Spiegelungen können simultan auftreten, es gibt also vier Spiegelungs-Permutationen, die mit gleicher Wahrscheinlichkeit auftreten.

4. Horizontale Verschiebung im Intervall von $[-0,1 \cdot w; 0,1 \cdot w]$ Pixel entlang der horizontalen Achse, wobei w die Breite des Bildes ist. Es wird eine zufällige Länge aus dem Intervall ausgewählt und verschoben; die entstehenden Ränder werden mit Schwarz gefüllt. Auch diese Augmentation ist realistisch und erhöht lediglich die Möglichkeit der Veränderung der Bilder.

5. Vertikale Verschiebung im Intervall von $[-0,1 \cdot h; 0,1 \cdot h]$ Pixel entlang der vertikalen Achse, wobei h die Höhe des Bildes ist. Rest wie bei der horizontalen Verschiebung.

Abbildung 3.2 zeigt eine beispielhafte Augmentierung des Bildes und der dazugehörigen Maske aus Abbildung 3.1, wobei eine horizontale Spiegelung[1], eine Rotation um 13° und eine Verschiebung um 8%, also $\lceil 0,08 \cdot 512 \rceil = 41$ Pixel, nach links dargestellt ist. Es ist keine vertikale Spieglung oder vertikale Verschiebung vorhanden. Die Randbereiche, wofür aufgrund der Verschiebung und Rotation keine Daten vorhanden sind, sind mit Schwarz gefüllt.

Die angewandten Augmentationen sind eher konservativ gewählt. An den Bildern wird kaum etwas verändert, außer die Pixel leicht umzusetzen. Dadurch würde ein augmentiertes Bild im Original-Datensatz nicht auffallen. Auf weitergehende Veränderungen wie Gamma-Korrektur, Helligkeitsanpassung, Zoom und Scherung wird hingegen verzichtet. Diese Veränderungen werden nicht vorgenommen, um mit dem Testdatensatz besser Abschätzen zu können, wie gut die Erkennung der Radwege allgemein - unter optimalen Bedingungen - funktioniert, ohne zunächst auf maximale Generalsierungsfähigkeit zu

[1]Die Spiegelachse ist hierbei *vertikal*.

Abbildung 3.2.: Beispielhafte Datenaugmentation mit Originalbild und Maske in Rot links und augmentiertes (horizontale Spiegelung, Rotation, Verschiebung nach links) Bild mit Maske in Rot rechts.

untersuchen. Diese Abschätzung wird dann mithilfe des Testdatensatzes von Karlsruhe, welcher im Erscheinungsbild stark vom Bike-Testdatensatz abweicht, vorgenommen, indem die Ergebnisse für den Bike-Testdatensatz und den Karlsruhe-Datensatz verglichen werden.
An dieser Stelle sei jedoch gesagt, dass insbesondere die Zoom-Augmentation sehr nützlich sein könnte, sollte das Modell für Orhtofotos mit unterschiedlicher **GSD!** generalisiert werden.

Am Ende jeder Epoche wird der Trainingsdatensatz gemischt, sodass in der nächsten Epoche neue zufällige Batches entstehen.

3.3. Architektur

Im Folgenden werden die untersuchten Architekturen genauer beschrieben und ebenfalls begründet, warum die jeweiligen architektonischen Entscheidungen getroffen wurden.
In Unterabschnitt 2.7.2 wurde ausführlich der Stand der Technik und Wissenschaft im Bereich der Straßenerkennung und -extraktion mittels Computer-Vision-Modellen beschrieben. Aufgrund der Ähnlichkeit der Problemdomäne lässt sich vermuten, dass ähnliche Verfahren wie zum Erkennen von Straßen auch für das Erkennen von Fahrradwegen nützlich sein könnten. Die Problemdomänen sind ähnlich, da Fahrradwege auch Straßen sind und eben jene Eigenschaften, wie große Klassenimbalance zwischen Fahrradweg und Hintergrund, Beschattung durch andere Objekte, Verdeckung durch z.B. Bäume und

ähnliche Annotationsfehler teilen. Schwierigkeit hier wird sein, Fahrradwege von Straßen zu unterscheiden.
Die Radwege sollen, wie Straßen auch, mittels Image-Semantic-Segmentation von einem Computer-Vision-Modell markiert werden. Andere Klassifizierungsarten, wie die in Abschnitt 2.1 beschriebene Objektdetektion würde zu grobe Bounding-Boxes um schräg verlaufende Radwege legen, sodass im Prinzip eine Straße markiert werden würde, die zwar ein Radweg hat, aber nicht klar wäre, wo dieser verläuft, bzw. ob ein Radweg in beide Richtungen existiert. Aus demselben Grund ergibt eine reine Klassifikation, ob ein Bild ein Radweg enthält oder nicht ebenfalls keinen Sinn. Auf der anderen Seite würde Instanz-Segmentierung keine weiteren relevanten Informationen hinzufügen, wonach semantische Segmentierung völlig ausreicht, um das Problem zu lösen.
Wie bereits in Unterabschnitt 2.7.2 dargelegt sind alle relevanten Modelle zur Straßenerkennung basierend auf der U-Net-Archtiektur (vgl. 2.4.3). Folglich sollen die hier betrachteten Modelle ebenfalls als angepasste U-Nets entworfen werden. Insbesondere ermöglicht dies auch die Modelle zur Fahrradwegerkennung auf den verschiedenen in Unterabschnitt 2.7.1 vorgestellten Datensätzen zur Straßenerkennung vorzutrainieren, was zu allgemein besseren Ergebnissen führen kann (s. Abschnitt 2.6). Außerdem können so die erzielten Ergebnisse vom Pre-Training mit den öffentlichen Benchmarks verglichen werden, um deren Ergebnisse zu validieren und früh Fehler in den eigenen Entscheidungen und Implementationen zu entdecken.

Zunächst soll so ein nur leicht modifiziertes U-Net, welches im Folgenden *Bike-U-Net* genannt wird, entworfen werden, welches als Baseline- und Vergleichs-Netz dienen soll. Dann soll eine zweite Klasse an U-Nets beschrieben werden, die verschiedene vortrainierte **CNN!**s (s. Abschnitt 2.5) als Backbones für ein U-Net verwendet, da Unterabschnitt 2.7.2 gezeigt hat, dass im Falle der Straßendetektion die Performanz eines Netzes stark verbessert werden konnte, indem auf Techniken und Methoden von Modellen aus anderen Teilgebieten der Computer-Vision zurückgegriffen wurde. Die einfachste Ausprägung hiervon ist das Dense-U-Net-121, welches einfach ein DenseNet121 als Backbone verwendet und somit von dessen Pre-Training und Architektur profitieren konnte und damit 2-5% bessere Resultate in der Basline-Bewertung und bis zu 19% bessere Ergebnisse in der optimierten Version erzielen konnte, als ein herkömmliches U-Net. Gegebenenfalls ist das auch für die Radwegerkennung möglich. Dazu sollen mehrere Backbones untersucht werden.

3.3.1. Bike-U-Net

Abbildung 3.3 zeigt die von uns verwendete Architektur für das ***BUNet!*** *(**BUNet!**)*, bzw. genauer das ***BUNet2!*** *(**BUNet2!**)*. Hierfür wurde das in Abbildung 2.3 dargestellte originale U-Net auf den vorliegenden Anwendungsfall angepasst. Diese Anpassungen sind in der nachfolgenden Liste erklärt.

- Zunächst werden Input-Bilder der Größe $width \times height \times 3$ verwendet. Wobei $width$ und $height$ variabel in der Architektur sind, mit der Einschränkung, dass diese Vielfache von 32 sein sollten, damit die mittlere Schicht nicht zu klein wird. In jedem Fall wird ein drei-kanal RGB-Bild verwendet. In der Abbildung ist exemplarisch $512 \times 512 \times 3$ gewählt.
  Der Output verwendet zunächst keine explizite One-Hot-Kodierung für Fahrradweg und Hintergrund, sondern eine implizite, wie in Unterabschnitt 2.7.2 beschrieben, um eine intuitivere Bewertung zu erhalten.

- Im Gegensatz zum originalen U-Net, wird bei den Convolutional-Layern Padding eingesetzt, um die Dimensionen der Feature-Maps nicht nach und nach zu verkleinern und so einen exakt symmetrischen Aufbau zu gewährleisten. Ebenso wurde bei den Up-Convolutions Padding eingefügt, um auch hier eine Verkleinerung der Feature-Maps zu verhindern. Diese Anpassung wurde eingesetzt, um kein Zuschneiden bei den Skip-Connections zu benötigen, was die Lokalisierung verbessern soll, und so bei den in Unterabschnitt 2.7.2 beschriebenen Netzen gewöhnlich ist.

- Im originalen U-Net beginnt der erste Convolutional-Block mit 64 Filtern, welche sich jeden weiteren Block mit zum Mittel-Block verdoppeln, der dann 1024 Filter besitzt. Daraufhin halbieren sich die Filter mit jedem weiteren Decoder-Block wieder.
  Im Bike-U-Net-2 beginnen die Filter bei 16 und verdoppeln sich bis 256, was zu ungefähr 2 Mio. trainierbaren Parametern führt. Die Zahl an Parametern ist somit weitaus geringer als im Original-U-Net mit ca. 24 Mio. Parameter [7]. Hiermit soll getestet werden, ob auch ein recheneffzienteres und kleineres U-Net gute Ergebnisse liefert, und somit auch weniger Regularisierung notwendig ist.
  Da aber die in Unterabschnitt 2.7.2 beschriebenen Netze ebenfalls 15-25 Mio. Parameter haben, soll ein weiteres U-Net mit mehr Parametern getestet werden. Dieses ***BUNet15!*** *(**BUNet15!**)* ist abgebildet in Abbildung A.4 und hat 15 Mio.

Parameter. Dies wurde erzielt, indem die Filter ab Block zwei dreimal mehr sind, als im Bike-U-Net-2. So haben wir hierbei 16 Filter in Block eins, dann 96 Filter in Block zwei, die sich verdoppeln bis hin zu 768 Filtern im mittleren Block und danach wieder halbieren bis zum vorletzten Block. Dabei hat der letzte Block wieder 16 Filter. Der erste und letzte Block haben dabei jeweils 16 Filter, da damit der Rechen- und Speicheraufwand erheblich reduziert werden kann. Bis auf die Filteranzahl und die daraus resultierende Tiefe der Feature-Maps, sind Bike-U-Net-2 und Bike-U-Net-15 identisch.

- Auf jede Convolution-Schicht folgt eine Batch-Normalization-Schicht. Dies hat mehrere Gründe: Zum einen übernimmt so das Netz selbst die Normalisierung und Standardisierung der Daten, wodurch sich das beim Vorverarbeiten gespart werden kann und zum anderen kann von den in Unterabschnitt 2.4.2 beschriebenen Vorteilen, wie schnellerem Training und leichter Regularisierung profitiert werden, ohne, dass dafür zusätzlicher Aufwand betrieben werden muss und keine Nachteile entstehen.

- Zusätzlich zu den Batch-Normalization-Schichten wurden pro Block eine Dropout-Schicht nach der jeweils ersten Convolution-Schicht eingezogen. Diese sollen als Hyperparameter eingebaut werden, um einfach kontrollierbar Regularisierung anzuwenden, sollten die Modelle Probleme mit Overfitting durch zu hohe Komplexität bekommen. Aufgrund der eher wenigen Parameter wird dies aber gegebenenfalls nur beim Bike-U-Net-15, oder überhaupt nicht nötig.

- Die Convolution-Schichten werden durch **ELU!** aktiviert, anstatt durch **ReLU!**, wie im Original-U-Net. Hierdurch können die meisten der in Abschnitt 2.2 herausgearbeiteten Vorteile von **ReLU!**, wie Robustheit gegen das Vanishing-Gradient-Problem, genutzt werden. Jedoch wurde bereits im ursprünglichen U-Net-Paper ([7]) auf ein Problem hingewiesen, worunter oftmals **CNN!**s leiden: Häufig kommt es vor, dass Netzteile dauerhaft nicht oder nur kaum aktiviert werden und so kaum etwas beitragen. Dieses Problem ist insbesondere für **ReLU!** relevant, da diese Aktivierungsfunktion unter dem Dyning-Neuron-Problem leidet. Deshalb wurde hier auf **ELU!** zurückgegriffen, da damit alle Netzteile zum Beitrag animiert werden sollen. Der einzige Nachteil ist hierbei, dass die Berechnung von **ELU!** etwas aufwendiger ist. Das Problem von unbeschränkt großen positiven Aktivierungen unter denen

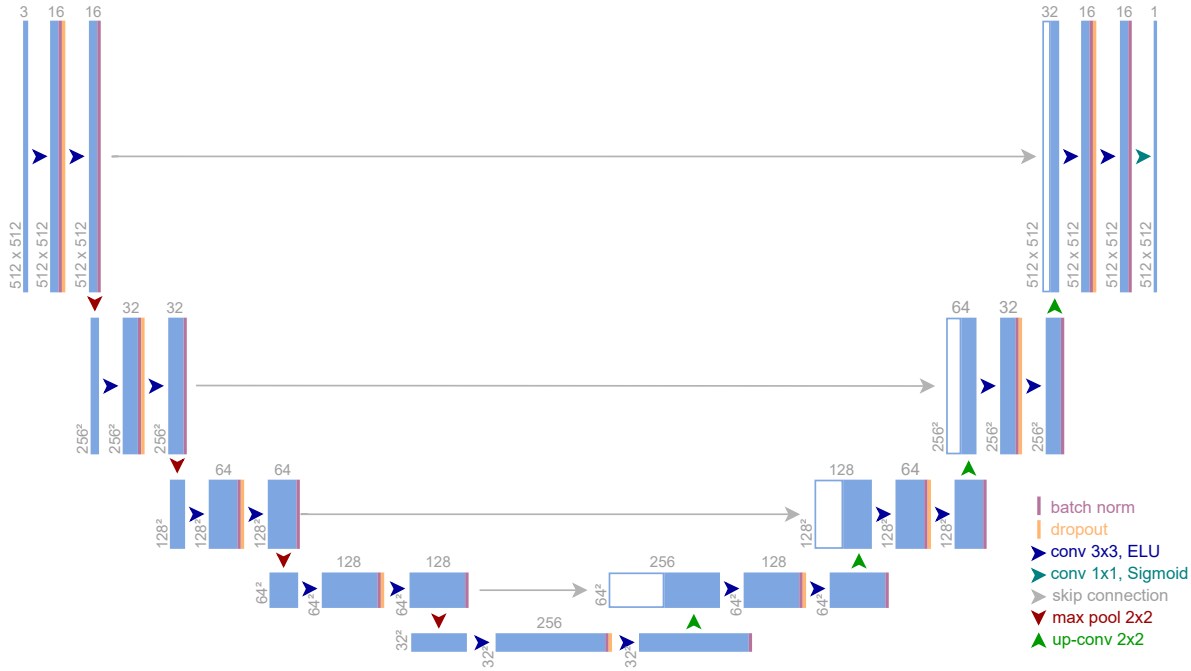

Abbildung 3.3.: Bike-U-Net-2 mit 1,946 Mio. Parametern.

sowohl **ELU!** als auch **ReLU!** leiden, wird in diesem Netz durch die wiederholte Batch-Normalisierung abgefedert.

- Für das Ouptut-Layer, nach der 1×1-Convolution, wird die Sigmoid-Funktion verwendet. Im Falle einer One-Hot-Kodierung würde hier Softmax herangezogen werden.

3.3.2. Backbone-U-Nets

Dieser Abschnitt beschreibt kurz die als U-Net mit vortrainiertem Backbone entworfenen Architekturen.
Prinzipiell dienen die Convolution-Schichten aller in Abschnitt 2.5 der Feature-Erkennung und -Extraktion. Lediglich die letzten Schichten, die bei den beschriebenen Klassifikationsmodellen allesamt Fully-Connected-Schichten waren, sind für die schlussendliche Zuordnung der Features zu den Objektklassen im ImageNet-Datensatz notwendig. Werden diese Schichten mit einem Decoder-Teil, der den Encoder - hier also das vortrainierte Klassifikationsnetz - spiegelt, entsteht ein Modell zur semantischen Segmentierung. Weiter

müssen lediglich an geeigneter Stelle Skip-Connections zwischen Encoder und Decoder eingefügt werden, um die U-Net-Form nachzubilden. Im Folgenden werden die auf diese Weise konstruierten Netze beschrieben.

VGG16-Bike-U-Net

VBUNet! *(**VBUNet!**)* nutzt das in Unterabschnitt 2.5.1 beschriebene Netz VGG16 mit auf ImageNet vortrainierten Gewichten als Backbone.
Hierzu werden die drei Fully-Connected-Layer (und Soft-Max-Aktivierung) am Ende entfernt und mit drei neuen *conv3-512*-Schichten ersetzt. Diese bilden den mittleren („untersten“) Block im U-Net. Darauf folgt eine direkte Spieglung von VGG16 als Decoder mit abschließender 1×1-Convolution. Im Gegensatz zum Bike-U-Net (s. Unterabschnitt 3.3.1) und ursprünglichen U-Net (s. Unterabschnitt 2.4.3) wurden allerdings keine *up-conv3*-Layer mit trainierbaren Gewichten zum Upsampling genutzt, sondern 2D-Upsampling-Layer, ohne trainierbaren Parameter. Diese Entscheidung wurde getroffen, um VGG16 besser nachzuempfinden, da hier Maxpool-Schichten zum Downsampling verwendet werden, die ebenfalls keine trainierbaren Parameter enthalten. Weiter sind nach jeder nicht-vortrainierten Convolution-Schicht, also nach jeder, die nicht zu VGG16 gehören, Batch-Normalization-Layer eingezogen, aus den in Unterabschnitt 3.3.1 und Unterabschnitt 2.4.2 genannten Gründen wie schnellerem Training, leichter Glättung und fehlender sonstiger Standardisierung und Normalisierung. Die für U-Net charakteristischen Skip-Connections werden vor jeder Maxpool-Schicht außer der ersten eingesetzt und mit dem korrespondierenden Decoder-Block verbunden. Dies resultiert in genau vier Skip-Connections, wie im Bike-U-Net und originellen U-Net. Abgeschlossen wird durch eine Sigmoid-Aktivierung, so wie im Bike-U-Net.
VGG16-Bike-U-Net enthält 23,7 Mio. Parameter. Grob die Hälfte davon (14,7 Mio.) sind von VGG16.

VGG16 wird als Backbone ausgewählt und getestet, da es als eine Art Basis- bzw. Standardversion der nachfolgenden Backbone-Netze aufgefasst werden kann, da diese auf VGG16 aufbauen und dieses - zumindest für den ImageNet-Datensatz - verbessern und nachvollzogen werden soll, ob diese Anpassungen dienlich sind für den konkreten Anwendungsfall der Radwegerkennung. Zudem ist VGG16 sehr beliebt als vortrainiertes Netz, weswegen es viele Vergleichsmöglichkeiten gibt. Des Weiteren wird VGG16 als

Backbone in Research zum Pre-Training mit U-Nets eingesetzt (s. Unterabschnitt 2.6.1), auf dessen Ergebnisse diese Arbeit aufbaut. Demnach ist es wiederum für Vergleichszwecke sinnvoll VGG16 als Backbone für die Radwegerkennung zu verwenden.

ResNet34-Bike-U-Net

RBUNet! *(**RBUNet!**)* nutzt das in Unterabschnitt 2.5.2 beschriebene Netz ResNet34 mit auf ImageNet vortrainierten Gewichten als Backbone.
Hierzu wird die eine Fully-Connected-Layer am Schluss des Netzes entfernt und um einen Decoder ersetzt. Anders als bei **VBUNet!** ist der mittlere Block der U-Net-Struktur von ResNet34 und nicht zusätzlich neu ergänzt. Die Begründung hierfür liegt in der höheren Anzahl an Parametern von ResNet34, welche ohne die Fully-Connected-Schicht bereits bei 24,4 Mio. liegt, im Gegensatz zu VGG16. Da der mittlere Block die meisten Parameter beinhaltet wird keine weitere Vertiefung des Netzes vorgenommen, um die Anzahl der Parameter nicht drastisch zu erhöhen. An Stelle dessen wird nach Ende der Convolution-Schichten von ResNet34 direkt mit dem Upsampling begonnen, welches wie bei **VBUNet!** mit Upsampling-Schichten bewerkstelligt wird. Auch wird im Widerspruch zu **VBUNet!** der Encoder - also ResNet34 - nicht gespiegelt, um eine Verdoppelung der Parameteranzahl zu vermeiden. Stattdessen wird ein einfacher, U-Net-ähnlicher Decoder aus fünf Blöcken á Upsampling-Schicht und zwei Convolution-Schichten verwendet, wobei in jedem Block die Filter-Anzahl von 512 an bis schließlich 16 halbiert wird. Abgeschlossen wird wie bei **VBUNet!** und **BUNet!** mit 1×1-Convolution und Sigmoid-Aktivierung. Aus den gleichen Gründen wie bei **VBUNet!** sind nach jeder Convoltuion-Schicht Batch-Normalization-Schichten eingezogen. Es erfolgt wiederum das Einsetzen von vier Skip-Connection nach U-Net-Vorbild ergänzend zu den Skip-Connections der Residuen-Blöcke an folgenden Stellen: nach der 7×7-Convolution zu Beginn von ResNet34, am Ende der Blöcke mit 64 Filtern, am Ende der Blöcke mit 128 Filtern und am Ende der Blöcke mit 256 Filtern. Im Decoder werden diese jeweils zu Beginn der ersten vier Decoder-Blöcke konkateniert.
ResNet34-Bike-U-Net enthält 24,4 Mio. Parameter. Davon sind 21,2 Mio. von ResNet34 und lediglich 3,2 Mio. vom Decoder, welcher somit eher unterrepräsentiert ist im Netz.

ResNet34 wird als Backbone ausgewählt und getestet, aufgrund der hohen Präsenz von Residual-Blöcken in den leistungsstärksten Modellen zur Straßenerkennung aus

Unterabschnitt 2.7.2 und insbesondere Tabelle 2.2. Es wird sich ebenfalls eine bessere Performanz dieses Modells bei der Radwegerkennung erhofft.

DenseNet121-Bike-U-Net

DBUNet! *(**DBUNet!**)* nutzt das in Unterabschnitt 2.5.3 beschriebene Netz DenseNet121 mit auf ImageNet vortrainierten Gewichten als Backbone.
Hierzu wird die Classification-Layer bestehend aus einer 7×7-Average-Pool-Schicht und einer Fully-Connected-Layer entfernt und durch einen Decoder ersetzt. Wie schon bei **RBUNet!** erstreckt sich das DenseNet bis in den mittleren U-Net-Block. Beim **DBUNet!** liegt der Grund dafür allerdings nicht in der hohen Parameterzahl, sondern in der hohen sonstigen Komplexität des Netzes. So benötigt DenseNet121 trotz deutlich geringerer Parameterzahl für die Inferenz und das Training länger als sowohl **RBUNet!**, als auch **VBUNet!**. Vermutlich ist hierfür die große Tiefe von 64 Schichten, wovon 60 Convolution-Schichten sind, verantwortlich. Um die Inferenz- und Trainingszeit also nicht weiter zu belasten, ist der Decoder eher schmal gehalten. Für den Decoder wird die gleiche Architektur wie im **RBUNet!** verwendet. Auch hier sind die Batch-Normalization-Schichten eingezogen. Der einzige Unterschied besteht darin, dass die erste Convolution-Schicht anders als bei **RBUNet!** nicht von 512 Kanälen auf 256 abbildet, sondern von 1536 auf 256. Dies liegt an der Konkatenierung der DenseNet-Architektur. Wieder wird das Netz abgeschlossen durch eine 1×1-Convolution gefolgt von einer Sigmoid-Aktivierung. Auch hier werden wieder vier Skip-Connections eingebaut, um die U-Net-Architektur nachzuahmen. Diese Skip-Connections sind im Falle von **DBUNet!** im Encoderteil nach der ersten 7×7-Convolution und dann jeweils nach jeder weiteren 1×1-Convolution der DenseNet-Transition-Zonen eingesetzt. Im Decoder-Teil sind diese wie bei **RBUNet!** zu Beginn jedes Decoder-Blocks bis auf den letzten eingebaut.
DenseNet121-Bike-U-Net enthält 12,1 Mio. Parameter, wovon 6,9 Mio. zum DenseNet121 und 5,1 Mio. neu hinzugefügt wurden. Trotz doppelter bzw. sogar vierfacher Tiefe besteht **DBUNet!** nur aus ungefähr halb so vielen Parametern wie **RBUNet!** und **VBUNet!**.

DenseNet121 wird als Backbone ausgewählt, wegen der guten Ergebnisse von Dense-U-Net-121 aus Unterabschnitt 2.7.2. Darüber hinaus werden zusammen mit VGG16 und ResNet34 Netze mit verhältnismäßig flacher, mitteltiefer und tiefer Architektur getestet, was eine Vielfalt an Architekturen abbildet.

3.4. Evaluationsmaße

Dieser Abschnitt beschäftigt sich mit der Auswahl geeigneter Bewertungsmaße für die Performanz des Netzwerkes. Insbesondere wird eine geeignete Verlustfunktion ermittelt. Die folgenden Diskussionen und Betrachtungen stützen sich auf Abschnitt 2.3 und die dort beschriebenen Maßzahlen. Weiter ist zu beachten, dass sich sämtliche Maßzahlen auf eine implizite Bestimmung der Hintergrundpixel durch ein Output-Neuron pro Pixel beziehen und nicht auf eine One-Hot-Kodierung.

Als eine direkte und intuitiv sehr anschauliche Metrik zum Bewerten der Modellperformanz bei Segmentierungsproblemen kann **IoU!** herangezogen werden. Die **IoU!** ist dabei auch ein recht verbreitetes Maß in der semantischen Segmentierung, weswegen es viele Vergleichswerte gibt, um die Modellperformanz bewerten zu können.
Problematisch an der **IoU!** dagegen ist, dass alle möglichen Klassifikationen (tp, fp, fn, tn) gleich stark gewichtet werden, wobei die wahr-positiven tp zunächst Priorität haben sollten, während die Fehler der falsch-positiven fp und falsch-negativen fn im Gleichgewicht bleiben sollten, sodass es nicht zu einer trivialen Klassifikation als rein positiv oder rein negativ kommt.
Ein weiteres Problem ist, dass die **IoU!** sehr streng und intolerant gegenüber leichter Verschiebung der Erkennung bewertet. Wenn ein Fahrradweg, der nur wenige Pixel breit ist, um die Hälfte der Breite verschoben segmentiert würde aber ansonsten komplett dem Radweg entspricht, würde die **IoU!** bereits von 1 auf 0,5 sinken, obwohl *qualitativ* der Weg exakt erkannt wurde. Diese Intoleranz gegenüber leichter lokaler Verschiebung ist insbesondere in dem in dieser Arbeit vorgestellten Bike-Datensatz aus **??** problematisch, da dieser über **OSM!** (**OSM!**) annotiert ist und für viele Straßen die Lage des zugehörigen Radwegs heuristisch über die Spuranzahl ermittelt wird.
Aufgrund der genannten Einschränkungen ist die **IoU!** ungeeignet als Verlustfunktion, kann jedoch als eine Dimension in der Bewertung mit einfließen.

Das Quality-Maß adressiert das Verschiebungs-Problem der **IoU!** mit einem Buffer zur relaxierten Ermittlung der tp, fp und fn. Hierbei ist allerdings die Buffergröße in Pixel eine wichtige Stellschraube. Wird diese zu groß gewählt, werden gegebenenfalls positive Pixel, die weit abseits liegen und keinen Radweg andeuten, fälschlicherweise als tp erkannt. Wird die Buffergröße hingegen zu klein gewählt, wird lediglich eine Toleranz gegenüber ungeraden Kanten in der Prediction aufgebaut, nicht aber die Verschiebung des gesamten

Radwegs akzeptiert. Da ein Radweg im Mittel ca. 11,8 Pixel und eine Straßenfahrbahn ca. 17,6 Pixel breit ist (s. **??**), wird der Buffer als ganze Zahl auf das arithmetische Mittel $\frac{11,8+17,6}{2} = 14,7 \approx 15$ festgelegt. Dies erlaubt eine Verschiebung des Radwegs um eine volle Breite zzgl. einer Toleranz für die leicht variierende Radwegbreite, nicht aber um die Breite einer Straßenfahrbahn. Hiermit soll ein realistisches Bild der Performanz des Netzes bezüglich der qualitativen Erkennung der Radwege gezeichnet werden und wird ergänzend zur **IoU!**-Metrik als Bewertungsfunktion eingesetzt.
Als Verlustfunktion ist die Quality allerdings ungeeignet, da wie bei der **IoU!** die wahrpositiven *tp* nicht stärker gewichtet werden, sodass das Netz nicht besonders die wahrpositiven lernt. Des Weiteren ist die Quality aufgrund der in Gleichung (2.14) beschriebenen Dilate-Funktion nicht trivial differenzierbar, was für die Verlustfunktion nötig wäre. Gegebenenfalls kann eine Ableitung für die Quality gefunden werden, die Untersuchung dessen geht allerdings über den Umfang dieser Arbeit hinaus. Zuletzt würde das Verwenden der Quality als Loss-Funktion - unter der Prämisse, dass dies möglich wäre - einen weiteren Hyperparameter - die Buffergröße - einführen, was weiter die Komplexität des Modells erhöht.

BCE! ist eine beliebte Verlustfunktion für Klassifikationsprobleme, allerdings in der Grundform wie in Gleichung (2.4) oftmals eher ungeeignet für semantische Segmentierung, die generell häufiger unter Klassenimbalance leidet, aufgrund der Annotation jeden Pixels als eine Klasse. Durch Gewichtung kann dem zwar entgegengewirkt werden, dabei entsteht allerdings zusätzlicher Aufwand für die Gewichtsbestimmung und zusätzliche Hyperparameter, die die Optimierung des Modells komplexer machen. Im speziellen Fall der Erkennung von Fahrradwegen ist eine größere Klassenimbalance als selbst bei der Straßenerkennung zu erwarten, da Radwege im Sinne dieser Arbeit seltener sind als Straßen und dazu noch schmaler, wodurch nur ein sehr kleiner Teil aller Pixel auf einem Bild einem Radweg zugehörig ist. Deswegen ist eine Gewichtung der **BCE!** hierbei unvermeidlich, was allerdings Zusatzaufwand in der Konzeption und Optimierung des Modells bedeutet. Darüber hinaus wird bereits bei der Straßenerkennung eher auf **BCE!** verzichtet bzw. mit anderen Verlustfunktionen kombiniert verwendet, wie bei [29]. Dort erzielt es allerdings schlechtere Ergebnisse als Dice. Daher wird **BCE!** nicht weiter als Verlustfunktion in Betracht gezogen. Auch als reine Bewertungsfunktion ist **BCE!** aufgrund des geringen Interpretationsmehrwerts eher uninteressant.

Der Dice-Koeffizient ist eine weitere Möglichkeit für eine Verlustfunktion. Insbesondere ist Dice als Verlustfunktion sehr beliebt in der semantischen Segmentierung und ist auch stark vertreten in seiner Reinform oder gemischt mit anderen Maßen in den Modellen zur Straßensegmentierung aus Unterabschnitt 2.7.2. Außerdem setzt Dice eine höhere Gewichtung auf die wahr-positiven *tp* als **IoU!**, wodurch Dice besser zum Erlernen der *tp* geeignet ist.
Im Gegensatz zur Quality und genauso wie die **IoU!** hat Dice jedoch keine Toleranz ggü. geringfügig verschobenen Radwegen. Damit wäre Dice optimal geeignet für von Hand annotierte Radwege, da hier keine heuristischen Fehler in Form von Radwegen, die um ca. eine Fahrradwegbreite versetzt auf zum Beispiel einem Grünstreifen verlaufen. Trotz dieser Unzulänglichkeit bei Verwendung auf dem Bike-Datensatz stellt Dice die beste Alternative für eine Verlustfunktion dar und wird folglich hier angewandt.

Zuletzt sei noch erwähnt, dass eine kombinierte Verlustfunktion aus zum Beispiel **BCE!** und Dice ggf. zu besserer Performanz führen kann, wie auch die Forschung zur Straßenerkennung (z.B. [29]) zeigt. Für diese Arbeit wird sich jedoch zunächst auf ein alleinstehendes Maß bezogen. Eine Mischform kann Gegenstand weiterer Forschung sein.

3.5. Hyperparameter

Batch size: kompromiss aus klein wie in u-net paper und batch normalization und trainingspeed/genauigkeit

3.6. Pre-Training auf Straßendatensätzen

3.7. Testkonzeption

4. Weitere Prinzipien

Zunächst wird eine Basisimplementation gegeben, die den Algorithmus zur Subproblemerzeugung und Similarity-Berechnung aus **??** umsetzt und diesen dann auf Geschwindigkeit optimiert.

4.1. Basisimplementation

Algorithmus 1 Vereinfachter Algorithmus zum Erstellen der Subprobleme aus den Submodellen

```
 1: procedure CREATE SUBPROBLEMS(F)
 2:     for S ∈ atomicSubmodels do
 3:         S.graph ← calculateGraph(S)
 4:         S.graph ← reduceGraph(S.graph, F)
 5:     end for                                          ▷ n := |atomicSubmodels|
 6:     while |atomicSubmodels| ≠ 0 do                   ▷ O(n(n+1)/2) · O(t_sim) = O(n² · t_sim)
 7:         M ← selectOne(atomicSubmodels)
 8:         M.set ← ∅
 9:         while complexity(M) < minSubproblemComplexity do
10:             highestSim ← −1
11:             for S ∈ atomicSubmodels do
12:                 sim ← computeSim(M, S)
13:                 if sim > highestSim then
14:                     mostSimilar ← S
15:                     highestSim ← sim
16:                 end if
17:             end for
18:             M ← M ∪ mostSimilar
19:             M.set ← M.set ∪ mostSimilar.set
20:             atomicSubmodels ← atomicSubmodels ∖ mostSimilar
21:         end while
22:         subproblems ← subproblems ∪ {M}
23:     end while
24: end procedure
```

5. Unit Tests

Mittlere Szenarien

Die Messungen haben ergeben, dass für alle mittelgroßen Szenarien $s \in S_{Anon_M}$ gilt, dass $t_{sub_{MatFlowG}}(s) < t_{sub_{IdObj}}(s)$. Hierbei haben sich für Identical Objects die Laufzeiten über die 13 Szenarien von S_{Anon_M} zwischen 11s und 1min 39s bewegt, während die Laufzeiten von Material Flow Graph zwischen 1s und 9s lagen. Mit $p = 2,744 \cdot 10^{-9}$ ist Material Flow Graph *statistisch signifikant* schneller[1].

Szenario	$t_{sub_{IdObj}}$	$t_{sub_{IdMat}}$	$t_{sub_{MatFlowG}}$	$f_{IdObj \to MatFlowG}$
s_{Krit}	9h 41min 20s	14min 47s	3min 31s	$168,15$
s_{Ult}	21h	2h 45min	15min	$83,\bar{3}$

Tabelle 5.1.: Ergebnisse der Laufzeitmessungen von S_L nach verschiedenen Methoden. Außerdem der Beschleunigungsfaktor f von Identical Objects zu Material Flow Graph. Szenario s_{Ult} wurde auf einem Server durchgeführt (Speicherbedarf zu hoch), s_{Krit} auf dem in **??** beschriebenen Rechner.

S_X	$\bar{m}_{pre}$	$\bar{m}_{post}$	f
small	$45,11$	$5,94$	$7,6$
medium	$395,82$	$120,3$	$3,29$
large	$4711,24$	$50,01$	$94,19$

Tabelle 5.2.: Durchschnittliche Knotenanzahl vor der Graphenreduktion $\bar{m}_{pre}$, nach der Graphenreduktion $\bar{m}_{post}$ und deren mittlerer Reduktionsfaktor f nach Szenario-Klassen.

Abschließend zur Untersuchung der praktischen Tests lässt sich also festhalten, dass die erwarteten Ergebnisse erzielt, sowie die theoretischen Überlegungen praktisch bestätigt werden konnten.

[1] Einseitiger Zwei-Stichproben-t-Test mit unterschiedlicher Varianz; Normalverteilung angenommen; $n = 12$; Szenario z mit $t_{sub_{IdObj}}(z) = 1min39s$ und $t_{sub_{MatFlowG}}(z) = 9s$ als Ausreißer von Test exkludiert.

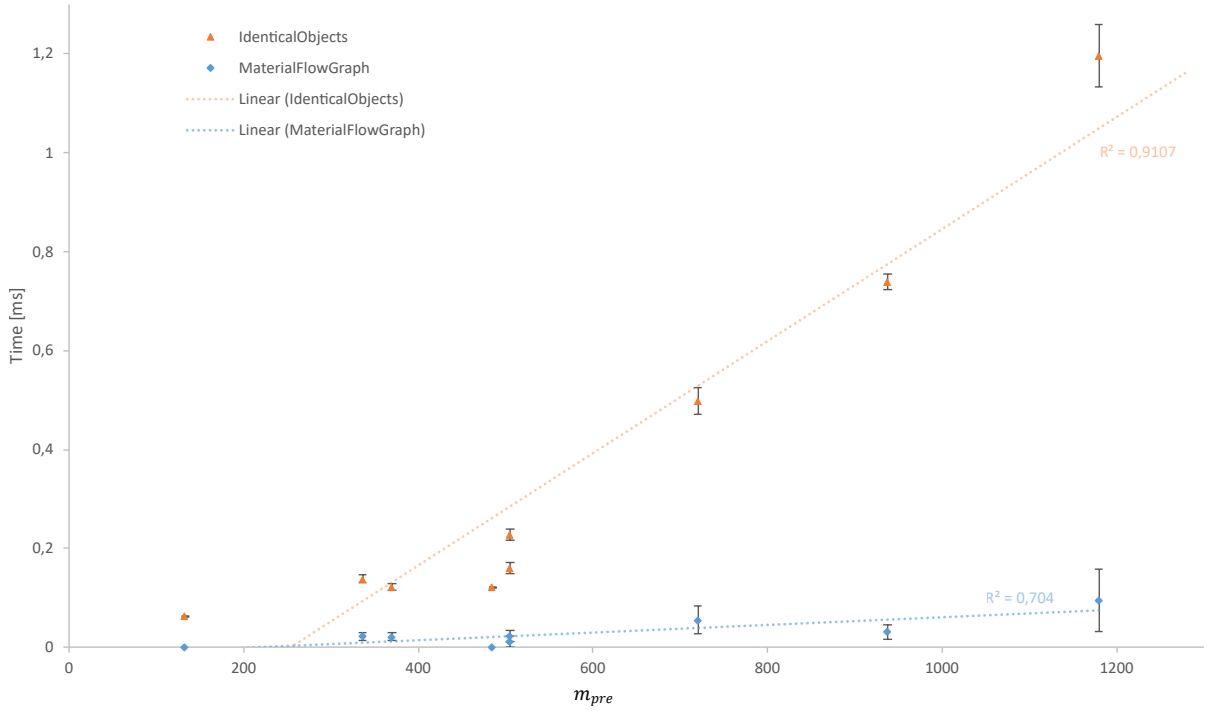

Abbildung 5.1.: Mittlere Graphengröße m mittelgroßer Szenarien abgebildet auf die durchschnittliche Laufzeit von t_{sim} in ms nach Identical Objects (orange), Material Flow Graph (blau) mit Trendlinien nach linearer Regression.

6. Domain Driven Design

7. Refactoring

In diesem Kapitel wird das Vorgehen in dieser Arbeit zusammengefasst, eine kritische Reflexion vorgenommen, in der unter Anderem die Schwierigkeiten des Projekts aufgeführt werden, sowie eine Aussicht auf die Zukunft des Projekts gegeben.

7.1. Zusammenfassung

7.2. Kritische Reflexion

7.3. Zukunftsaussicht

8. Entwurfsmuster

Literaturverzeichnis

[1] Sharma, P. „Image Classification vs. Object Detection vs. Image Segmentation“. In: *Analytics Vidhya* (21. 08. 2019). URL: https://medium.com/analytics-vidhya/image-classification-vs-object-detection-vs-image-segmentation-f36db85fe81 (Einsichtnahme: 03. 11. 2022).

[2] *Index of /slides/2017*. 10.11.2022. URL: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf (Einsichtnahme: 10. 11. 2022).

[3] Goodfellow, I./ Bengio, Y./ Courville, A. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016.

[4] Clevert, D.-A./ Unterthiner, T./ Hochreiter, S. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. Published as a conference paper at ICLR 2016. 23/11/2015. URL: https://arxiv.org/pdf/1511.07289.

[5] Cybenko, G./ O'Leary, D. P./ Rissanen, J. *The mathematics of information coding, extraction, and distribution*. Bd. 107. The IMA volumes in mathematics and its applications. New York: Springer, 1999. URL: http://www.springer.com/gb/%20BLDSS.

[6] Murphy, K. P. *Machine learning. A probabilistic perspective / Kevin P. Murphy*. Adaptive computation and machine learning series. Cambridge, Mass. und London: MIT Press, 2012.

[7] Ronneberger, O./ Fischer, P./ Brox, T. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. conditionally accepted at MICCAI 2015. 18/05/2015. URL: https://arxiv.org/pdf/1505.04597.

[8] Sørenson, T. *A Method of Establishing Groups of Equal Amplitude in Plant Sociology Based on Similarity of Species Content and Its Application to Analyses of the Vegetation on Danish Commons*. Biologiske skrifter. I kommission hos E. Munksgaard, 1948. URL: https://books.google.de/books?id=rpS8GAAACAAJ.

[9] Dice, L. R. „Measures of the Amount of Ecologic Association Between Species“. In: *Ecology* 26.3 (1945), S. 297–302.

[10] Yutaka Sasaki. *The truth of the F-measure.* 2007. URL: https://www.researchgate.net/publication/268185911_The_truth_of_the_F-measure.

[11] Fletcher, S./ Islam, M. Z. „Comparing sets of patterns with the Jaccard index“. en. In: *Australasian Journal of Information Systems* 22 (2018). URL: http://journal.acs.org.au/index.php/ajis/article/view/1538.

[12] Christian Wiedemann u. a. *Empirical Evaluation Of Automatically Extracted Road Axes.* 1998.

[13] Nitish Srivastava u. a. „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15.1 (2014), S. 1929–1958. URL: https://www.researchgate.net/publication/286794765_Dropout_A_Simple_Way_to_Prevent_Neural_Networks_from_Overfitting.

[14] Ioffe, S./ Szegedy, C. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 11/02/2015. URL: https://arxiv.org/pdf/1502.03167.

[15] Simonyan, K./ Zisserman, A. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* 04/09/2014. URL: https://arxiv.org/pdf/1409.1556.

[16] He, K. u. a. *Deep Residual Learning for Image Recognition.* Tech report. 10/12/2015. URL: https://arxiv.org/pdf/1512.03385.

[17] Huang, G. u. a. *Densely Connected Convolutional Networks.* CVPR 2017. 25/08/2016. URL: https://arxiv.org/pdf/1608.06993.

[18] Ruder, S. „Transfer Learning - Machine Learning's Next Frontier“. In: *Sebastian Ruder* (). URL: https://ruder.io/transfer-learning/ (Einsichtnahme: 07. 11. 2022).

[19] Cheng, D./ Lam, E. Y. *Transfer Learning U-Net Deep Learning for Lung Ultrasound Segmentation.* 14 pages, 8 figures. 05.10.2021. URL: https://arxiv.org/pdf/2110.02196.

[20] Amiri, M./ Brooks, R./ Rivaz, H. *Fine tuning U-Net for ultrasound image segmentation: which layers?* 19.02.2020. URL: https://arxiv.org/pdf/2002.08438.

[21] *Road and Building Detection Datasets.* 06.04.2014. URL: https://www.cs.toronto.edu/~vmnih/data/ (Einsichtnahme: 06. 10. 2022).

[22] *LandCover.ai.* 20.04.2022. URL: https://landcover.ai.linuxpolska.com/ (Einsichtnahme: 06. 10. 2022).

[23] Ashwath, B. *DeepGlobe Road Extraction Dataset*. 10.11.2020. URL: https://www.kaggle.com/datasets/balraj98/deepglobe-road-extraction-dataset (Einsichtnahme: 06. 10. 2022).

[24] Chesapeake Conservancy. *Chesapeake Bay Program Land Use/Land Cover Data Project*. 02.06.2022. URL: https://www.chesapeakeconservancy.org/conservation-innovation-center/high-resolution-data/lulc-data-project-2022/ (Einsichtnahme: 06. 10. 2022).

[25] Englich, M. *Detection and Reconstruction*. 06.10.2022. URL: https://www.isprs.org/education/benchmarks/UrbanSemLab/detection-and-reconstruction.aspx#VaihigenDataDescr (Einsichtnahme: 06. 10. 2022).

[26] Tan, W. u. a. *Toronto-3D: A Large-scale Mobile LiDAR Dataset for Semantic Segmentation of Urban Roadways*. 2020. URL: https://arxiv.org/pdf/2003.08284.

[27] Englich, M. *2D Semantic Labeling*. 17.11.2022. URL: https://www.isprs.org/education/benchmarks/UrbanSemLab/semantic-labeling.aspx (Einsichtnahme: 17. 11. 2022).

[28] Englich, M. *2D Semantic Labeling Contest - Potsdam*. 17.11.2022. URL: https://www.isprs.org/education/benchmarks/UrbanSemLab/2d-sem-label-potsdam.aspx (Einsichtnahme: 17. 11. 2022).

[29] C. Henry/ F. Fraundorfer/ E. Vig. „Aerial Road Segmentation in the Presence of Topological Label Noise“. In: *2020 25th International Conference on Pattern Recognition (ICPR)*. 2020 25th International Conference on Pattern Recognition (ICPR). 2021, S. 2336–2343.

[30] Constantin, A./ Ding, J.-J./ Lee, Y.-C. „Accurate Road Detection from Satellite Images Using Modified U-net“. In: *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS) (Chengdu). IEEE, 2018, S. 423–426.

[31] Kamiya, R. u. a. „Road Detection from Satellite Images by Improving U-Net with Difference of Features“. In: *Proceedings of the 7th International Conference on Pattern Recognition Applications and Methods*. 7th International Conference on Pattern Recognition Applications and Methods (Funchal, Madeira, Portugal). SCITEPRESS - Science and Technology Publications, 2018, S. 603–607.

[32] Yerram, V. u. a. „Extraction and Calculation of Roadway Area from Satellite Images Using Improved Deep Learning Model and Post-Processing“. eng. In: *Journal of Imaging* 8.5 (2022). PII: jimaging8050124 Journal Article, S. 124. eprint: 35621888.

[33] Geiger, A. u. a. „Vision meets robotics: The KITTI dataset“. In: *The International Journal of Robotics Research* 32.11 (2013), S. 1231–1237.

[34] Gao, L. u. a. „Road Extraction from High-Resolution Remote Sensing Imagery Using Refined Deep Residual Convolutional Neural Network“. In: *Remote Sensing* 11.5 (2019). PII: rs11050552, S. 552.

[35] O. Filin/ A. Zapara/ S. Panchenko. „Road Detection with EOSResUNet and Post Vectorizing Algorithm“. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). 2018, S. 201–2014.

A. Anhang

A.1. Architekturen

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Abbildung A.1.: Die ursprünglichen VGG-Architekturen. Spalte D zeigt VGG16 wie in Unterabschnitt 2.5.1 beschrieben. Abb. aus [15].

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Abbildung A.2.: Die ursprünglichen ResNet-Architekturen. Zwischen jeweils zwei Convolutional-Layer befindet sich eine Skip-Connection [16].

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Abbildung A.3.: Die ursprünglichen DenseNet-Architekturen [17].

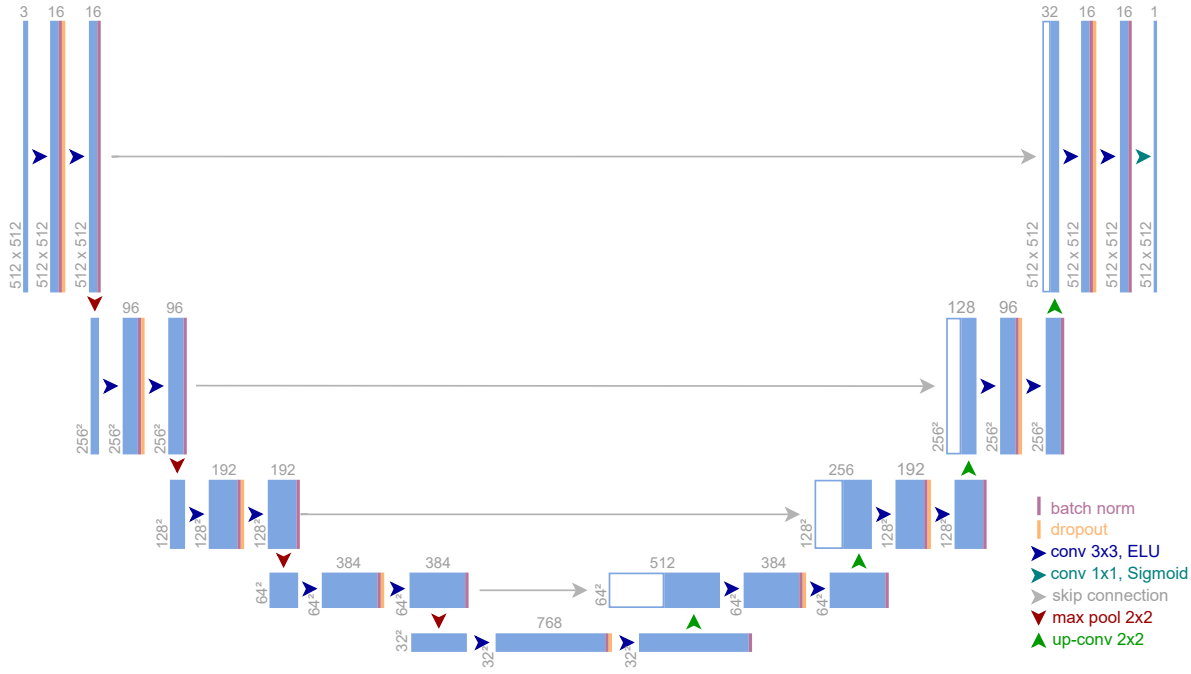

Abbildung A.4.: Bike-U-Net-15 mit 15,165 Mio. Parametern.

A.2. Quelltext-Implementation

```
1  import cv2
2
3  import numpy as np
4
5
6  def quality(buffer:int = 15, threshold:float = 0.1) -> any:
7      r"""Returns the quality metric function for keras's compile.
8      Implementation following this paper ↙
           ↳ https://www.researchgate.net/publication/2671242_Empirical_Evaluatio
9
10     Args:
11         buffer: size of buffer, i.e. max euclidean distance ↙
               ↳ between point x and y so that x and y are still ↙
               ↳ considered to be in each other's buffer area.
12         threshold: number from which on an input will be ↙
               ↳ considered as activated (1; else 0).
13
14     Returns:
15         function taking tensor y_true and y_pred
16     """
17
18     diameter = 2 * buffer + 1
19     kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, ↙
           ↳ (diameter, diameter))
20     dilate_i = 1
21
22     def quality(y_true_batch, y_pred_batch):
23         y_true_batch_numpy = y_true_batch.numpy()
24         y_pred_batch_numpy = y_pred_batch.numpy()
25         batch_size = y_true_batch_numpy.shape[0]
26         width, height = (y_true_batch_numpy.shape[1], ↙
               ↳ y_pred_batch_numpy.shape[2])
27
28         qual_sum = 0
```

```
29        for i in range(0, batch_size):
30            y_true = y_true_batch_numpy[i][:, :, 0]
31
32            y_pred_binary = np.copy(y_pred_batch_numpy[i])
33            y_pred_binary[y_pred_binary < threshold] = 0.
34            y_pred_binary[y_pred_binary >= threshold] = 1.
35            y_pred_binary = y_pred_binary.reshape((width, height))
36
37            y_true_buffered = cv2.dilate(y_true, kernel, ↙
                  ↳ iterations=dilate_i)
38            y_pred_binary_buffered = cv2.dilate(y_pred_binary, ↙
                  ↳ kernel, iterations=dilate_i)
39
40            tp = np.sum(np.multiply(y_true_buffered, ↙
                  ↳ y_pred_binary))
41            fp = np.sum(y_pred_binary) - tp
42            fn = np.sum(y_true - np.multiply(y_true, ↙
                  ↳ y_pred_binary_buffered))
43
44            denom = tp + fp + fn
45            qual = tp / denom if denom > 0.0001 or denom < ↙
                  ↳ -0.0001 else 1
46            qual_sum += qual
47
48        qual_avg = qual_sum / batch_size
49        return qual_avg
50    return quality
```

Code-Ausschnitt A.1: Implementation des Quality-Maßes in Python zur Verwendung im Training und Testen von Keras-Modellen.