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

1.1. Übersicht über die Applikation

Bei dieser Applikation handelt es sich um ein singleplayer Kartenspiel. Hier versucht ein Spieler auf einer einsamen Insel zu überleben und von ihr zu flüchten, indem er Karten zieht und Ressourcen (*wood*, *plastic*, *metal*) sammelt, um Gegenstände zu bauen (*Scavange*), die ihn vor seinen Feinden (*Encounter*) schützen oder ihm eine Rettung (*Endeavor*) von der Insel ermöglichen. Während des Spiels trifft der Spieler auf Tiere (*spider*, *tiger*, *snake*), die ihm gesammelte Ressourcen kosten können, oder Katastrophen, die Ressourcen zerstören. Das Spiel endet, wenn der Spieler sich retten konnte oder keine Aktionen mehr möglich sind.

Das Kartenspiel hat verschiedene Spielphasen (s. Abbildung 1.1), in denen der Spieler unter bestimmten Bedingungen Karten ziehen und Gegenstände bauen kann (*Scavange*). Wenn ein Gegenstand aus der Kategorie Rettungen (*Endeavor*) gebaut wurde, muss der Spieler würfeln, um zu entscheiden, ob die Rettung erfolgreich ist. Ein Segelboot und ein Hanggleiter stellen eine Rettung dar, wenn beim Würfeln eine bestimmte Augenzahl erzielt wird. Wenn der Rettungsversuch fehlschlägt, kann der Spieler weiter Karten ziehen und Gegenstände bauen, solange Karten im Stapel vorhanden sind. Das Bauen eines Dampfschiffs und eines Heißluftballons ist nur mit einer Feuerstelle möglich und garantiert eine erfolgreiche Rettung. Wenn ein Tier gezogen wird, muss der Spieler gegen es kämpfen (*Encounter*) und auch hier bestimmt Würfeln über Sieg oder Niederlage. Für unterschiedliche Aktionen werden unterschiedliche Würfel (*vier-*, *sechs-*, *acht-seitig*) verwendet. Das Spiel endet, wenn keine weiteren Aktionen mehr möglich sind oder wenn eine garantierte Rettung erfolgt ist. Wenn das Spiel endet (*End*), bleibt es in diesem Zustand, bis ein neues Spiel gestartet wird, bis das Spiel neu initialisiert wird oder die Anwendung beendet wird.

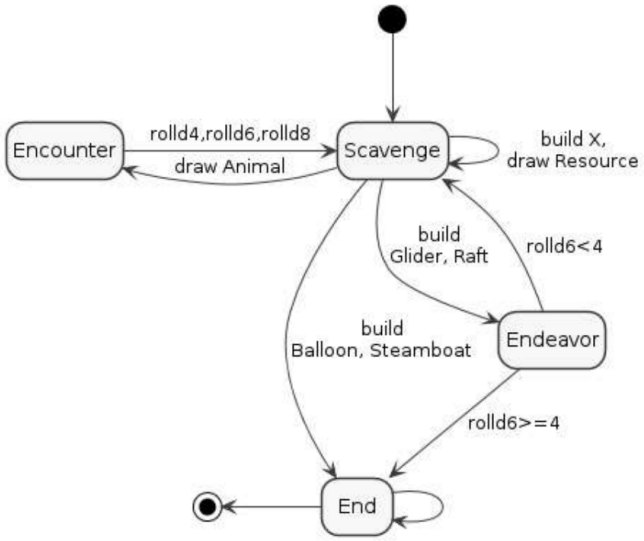

Abbildung 1.1.: Spielphasen.

1.2. Wie startet man die Applikation?

Vorraussetzung ist **Java 17**[1].
Wahlweise wird eine gängige Java-IDE[2] oder eine *maven*-Installation benötigt.

Über Maven:

Im Wurzelverzeichnis folgende Maven-Befehle ausführen:

```
mvn compile
mvn package
java -jar target/ase-1.0-SNAPSHOT.jar
```

[1] Check mit `java -version`.
[2] Hier wurde *Jetbrains' IntelliJ* verwendet.

Über IDE:

Projekt mit IDE im Wurzelverzeichnis öffnen. Zu folgendem Pfad navigieren
`./src/main/java/de/dhbw/karlsruhe/ase/plugin/cli`
und dort die *main*-Methode der Klasse *Main* ausführen lassen.

Erste Schritte:

Nun läuft das Command Line Interface der Applikation und es kann mit
`help`
eine Übersicht über die möglichen Befehle (Name, Regex, Beschreibung) aufgerufen werden, oder mit
`start?`
ein Spiel mit zufälligem Kartenstapel gestartet werden. Daraufhin kann mit
`draw`
begonnen werden Karten zu ziehen.

1.3. Wie testet man die Applikation?

Über Maven:

Im Wurzelverzeichnis:

```
mvn test
```

Über IDE:

Die Tests befinden sich unter
`./src/test/java`
und können dort mit einer IDE-Funktion ausgeführt werden.
Es existieren Blackbox-Integration-Tests und Unit-Tests, die aber keine besonderen Anforderungen außer eine **JUnit5**-Abhängigkeit haben.

2. Clean Architecture

2.1. Was ist Clean Architecture?

Clean Architecture ist ein allgemeines Konzept zum Design von Softwarearchitekturen, das darauf abzielt, die Struktur einer Anwendung so zu gestalten, dass sie unabhängig von ihren Benutzerschnittstellen, Datenquellen und anderen äußeren Faktoren bleibt. Dies bedeutet, dass die verschiedenen Komponenten und Funktionen einer Anwendung in abgekapselten Schichten angeordnet werden (*Onion*-Architektur), um sicherzustellen, dass Veränderungen in einer Schicht die tieferliegenden Schichten nicht beeinflussen. Außere Schichten haben dabei Abhängigkeiten von tieferen Schichten aber niemals umgekehrt. Aufrufe von tieferen Schichten an äußere Schichten werden über Abstraktionen (*Dependency Inversion* und *Injection*) realisiert. Tiefere Schichten sind dabei langlebiger als Schichten weiter außen.
Dies führt zu einer Anwendung, die flexibler und leichter zu entwickeln und zu warten ist, da Änderungen an einem Teil der Anwendung die tieferliegenden Schichten nicht berühren.

2.2. Analyse der Dependency Rule

Schichten im Sinne der Dependency Rule sind hierp `plugin`, `application`, `domain` und `abstraction` aufsteigend geordnet nach zunehmender Tiefe in der Onion-Architektur[1]. Sie werden durch gleichnamige Packages, die parallel unter
`de.dhbw.karlsruhe.ase`
zu finden sind, erschöpfend repräsentiert. Es sind Abhängigkeiten von äußerden Schichten in tiefere Schichten erlaubt, aber nicht umgekehrt. Diese Regel wird für die oben genannten Schichten immer eingehalten, was die folgenden beiden Beispiele illustrieren.

[1] Siehe **??** für mehr Informationen.

(Positiv) Beispiel 1:

Abbildung 2.1 zeigt das verlangte UML-Diagramm der Beispielklasse 1 *Camp* der Schicht *domain*, welches die Dependency Rule einhält. Wie im Diagramm zu sehen ist hängt Camp ausschließlich von Klassen in der selben Schicht (*domain*) ab. *Crafting* ist dabei keine Schicht sondern ein *Aggregat*[2]. Schichten sind ausschließlich die oben genannten. Weiter zu sehen ist, dass ausschließlich Klassen in *application* von Camp abhängen, aber keine aus höheren Schichten (was in diesem konkreten Fall nur *abstraction* sein könnte).

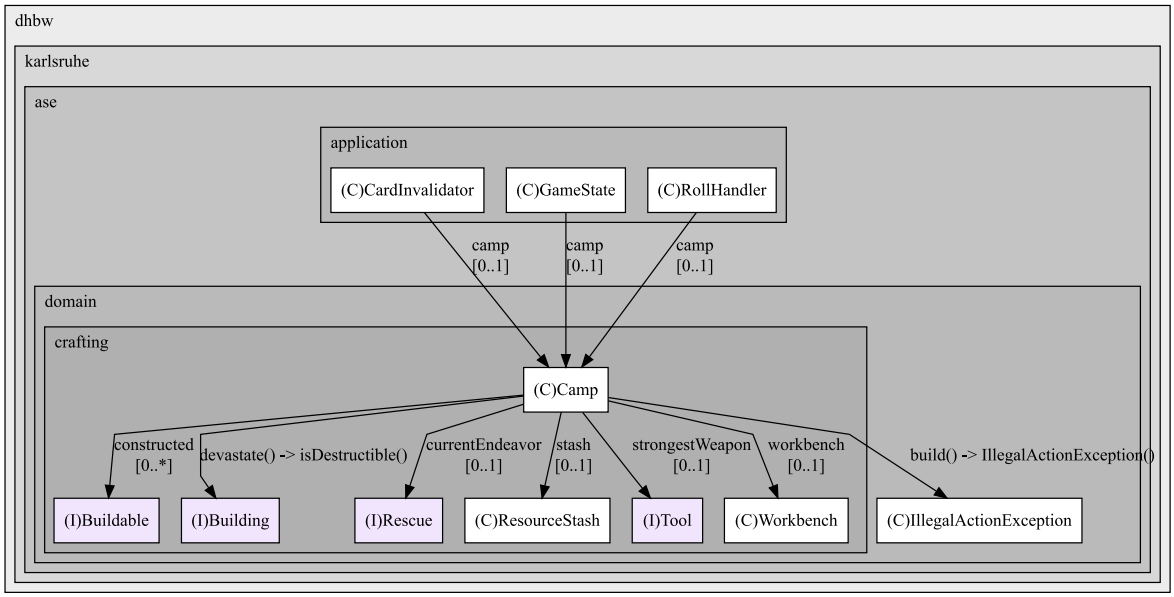

Abbildung 2.1.: UML-Diagramm 1 zum Einhalten der Dependency Rule von *Camp*.

(Positiv) Beispiel 2:

Abbildung 2.2 zeigt ein weiteres positives Beispiel für die Dependency Rule, da diese in dieser Softwarearchitektur nicht verletzt wird. Abgebildet ist die Klasse *Game*, welche Lediglich von Klassen aus der eigenen Schicht (*application*) und Klassen aus der tieferen Schicht *domain* abhängt. Abhänig von Game sind nur Klassen aus der Schicht *plugin*, bzw. dem konkreten *cli*-Plugin.

[2]Siehe Kapitel 6.

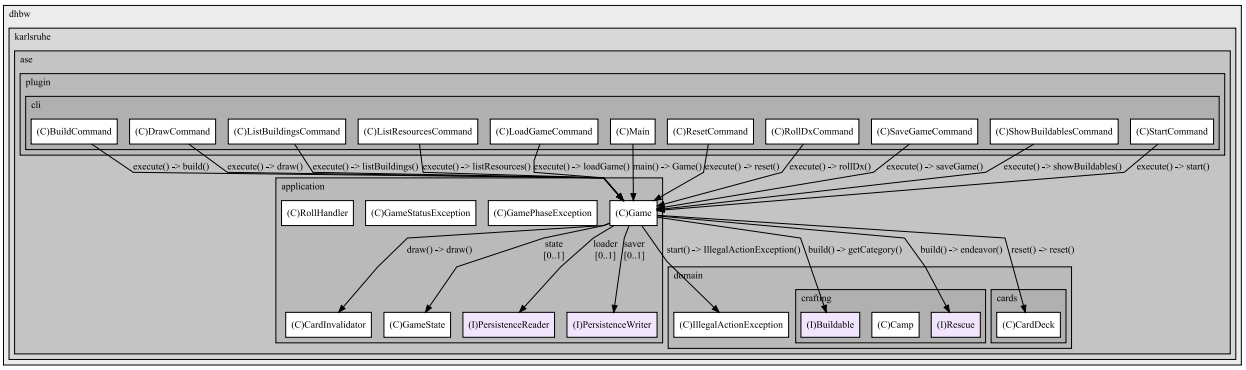

Abbildung 2.2.: UML-Diagramm 2 zum Einhalten der Dependency Rule von *Game*.

2.3. Analyse der Schichten

In dieser Softwarearchitektur gibt es folgende Schichten aufsteigend sortiert nach Tiefe: `plugin`, `application`, `domain` und `abstraction`. Hierbei stellt *plugin* die äußerste und *abstraction* die tiefste Schicht dar.

Schicht Plugin:

Bei einem Refactoring der Plugin-Schicht ist aufgefallen, dass viele der (Fehler-)Hinweise, die aufgrund von falschem Befehlssyntax oder einem falschen Befehl an den User über das CLI ausgegeben werden, keine gemeinsame einheitliche Form aufwiesen, da diese direkt als String ausgegeben wurden. Um eine einheitliche Fehlermeldung zu gewährleisten wurde der *ErrorBuilder* eingefügt, dessen Aufgabe es ist, die Fehler einheitlich zu formatieren und auszugeben. Abbildung 2.3 zeigt das UML-Klassendiagramm dieser Klasse. Sie wird im gesamten CLI-Plugincode verwendet um (Fehler-)Hinweise auszugeben.
Über verschiedene Konstruktoren kann ein Grund für den (Fehler-)Hinweis und eine mögliche Abhilfeemfehlung eingegeben werden. Mit der *print*-Methode kann der erzeugte (Fehler-)Hinweis dann formatiert an den User ausgegeben werden. Hierzu wird aus Gründen der Testbarkeit an die Funktion *printError* des Proxys *Terminal* delegiert.
Diese Klasse befindet sich im Plugin-Layer (insb. im CLI-Plugin), da die konkrete Ausgabe an den User in Form von Text von dem CLI abhängt. Die high-level Fehlermeldungen die von den tieferen Schichten über Exceptions realisiert sind können von jedem Plugin

anders verarbeitet und an den User weitergegeben werden. In der CLI funktioniert dies mit dem ErrorBuilder, während es mit einem denkbaren GUI-Plugin über bspw. ein Popup funktionieren könnte. Der ErrorBuilder ist lediglich für die Ausgabe an den Nutzer zuständig und enhält keine Fehler-Logik und gehört somit in die Plugin-Schicht.

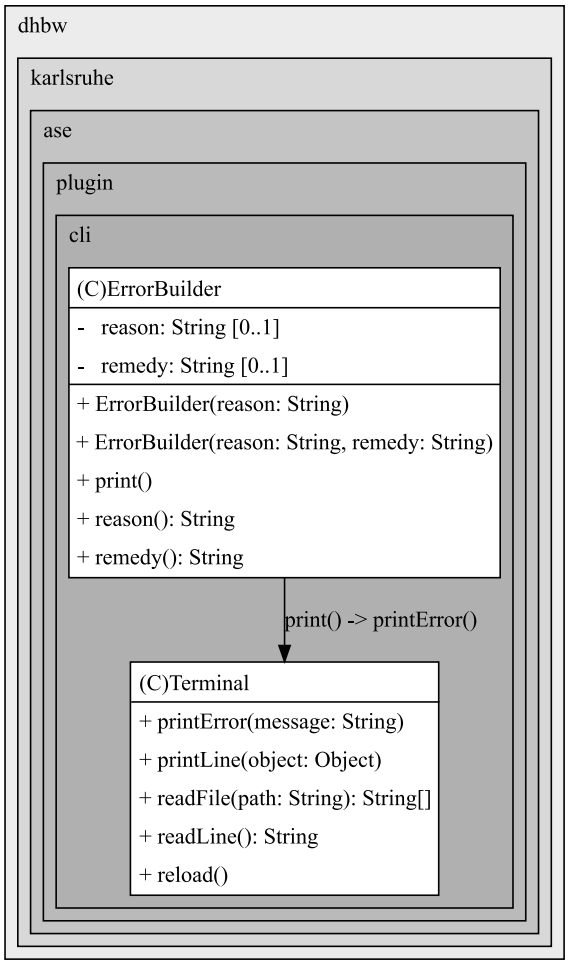

Abbildung 2.3.: Beispielklasse der Plugin-Schicht: ErrorBuilder.

Schicht Domain:

Der *ResourceStash* ist fester Teil der Domain-Schicht und stellt hier ein Wrapper mit Methodennamen in Domänensprache für eine Resource-Deque dar. Abbildung 2.4 zeigt das zugehörte UML-Klassendiagramm. Der ResourceStash beinhaltet und verwaltet alle Ressourcen, die der Spieler im Verlauf des Spiels ansammelt. Durch eine Katastrophe

oder das Verlieren gegen ein Tier wird der ResourceStash zerstört (*devastate*) und alle ungeschützten Ressourcen gelöscht. Das Erbauen eines *Shacks*[3] erlaubt es die obersten $n \in [0; \inf)$ Ressourcen zu schützen (*protectTopMostNResources*), sodass diese nach einem *devastate* übrig bleiben. Zusätzlich können Ressourcen hinzugefügt, konsumiert oder deren Vorhandensein überprüft werden. *Camp* und *Workbench* teilen sich eine Referenz auf denselben Stash.
Die Klasse ist als Teil des Kartenspiels (der Domäne) im Code natürlich in der Domain-Schicht angesiedelt und wird von anderen Domänenklassen genutzt. Es gibt keine Möglichkeit die Klasse in einer der anderen Schichten sinnvoll anzusiedeln.

[3]nicht in der Abb.

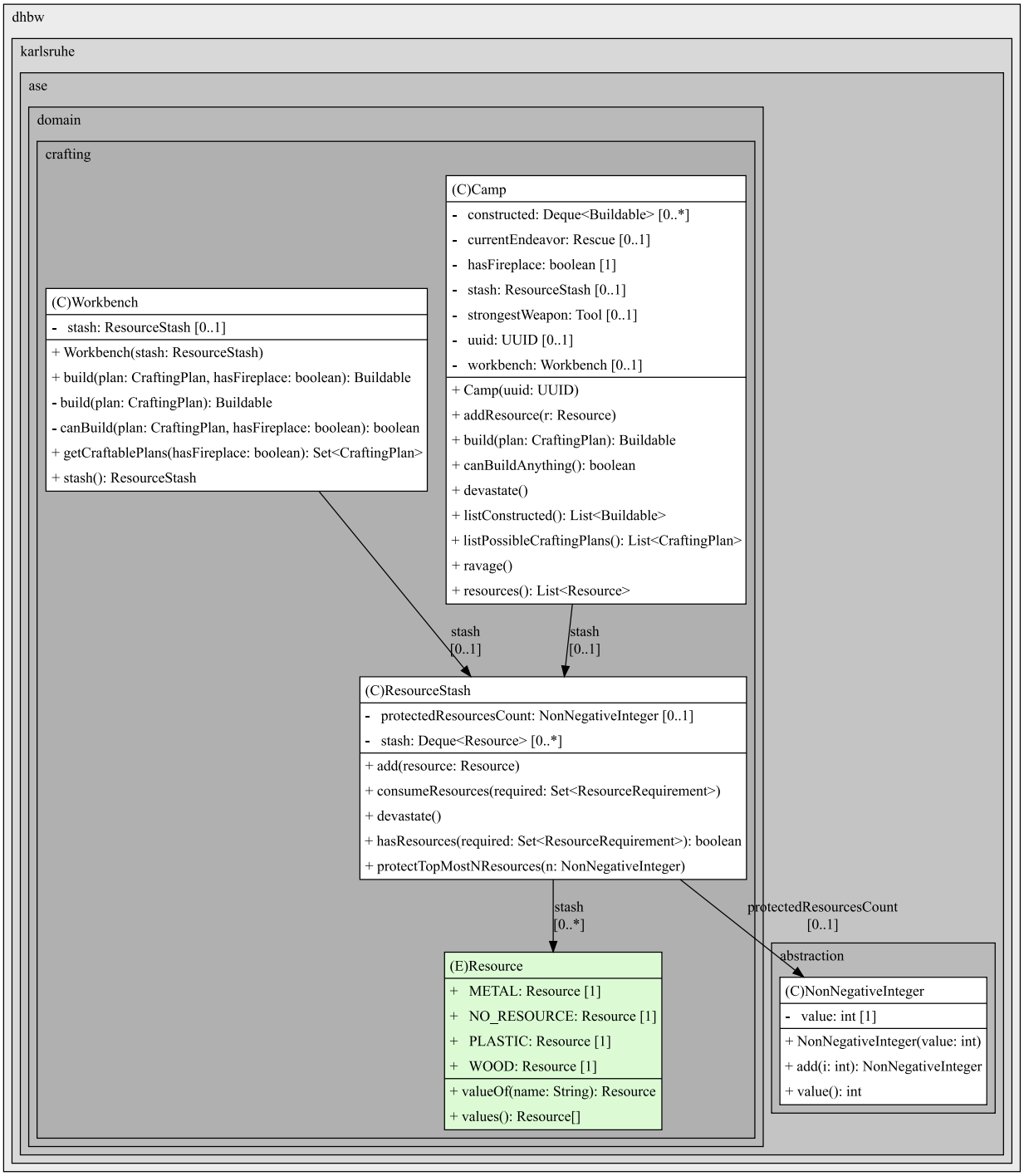

Abbildung 2.4.: Beispielklasse der Domain-Schicht: ResourceStash.

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 **??** 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%.

?? 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.

	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.

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

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.

?? zeigt eine beispielhafte Augmentierung des Bildes und der dazugehörigen Maske aus **??**, 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 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!** (**GSD!**) generalisiert werden.

[1] Die Spiegelachse ist hierbei *vertikal*.

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 **??** 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 **??** 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 **??** dargelegt sind alle relevanten Modelle zur Straßenerkennung basierend auf der U-Net-Archtiektur (vgl. **??**). 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 **??** vorgestellten Datensätzen zur Straßenerkennung vorzutrainieren, was zu allgemein besseren Ergebnissen führen kann (s. **??**). 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 (**CNN!**s) (s. **??**) als Backbones für ein U-Net verwendet, da **??** 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

?? zeigt die von uns verwendete Architektur für das ***BUNet!*** *(**BUNet!**)*, bzw. genauer das ***BUNet2!*** *(**BUNet2!**)*. Hierfür wurde das in **??** 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 **??** 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 **??** 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 [1]. 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 **??** beschriebenen Netze ebenfalls 15-25 Mio. Parameter haben, soll ein weiteres U-Net mit mehr Parametern getestet werden. Dieses ***BUNet15!*** *(**BUNet15!**)* ist abgebildet in **??** 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 **??** 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!** (**ELU!**) aktiviert, anstatt durch **ReLU!** (**ReLU!**), wie im Original-U-Net. Hierdurch können die meisten der in **??** herausgearbeiteten Vorteile von **ReLU!**, wie Robustheit gegen das Vanishing-Gradient-Problem, genutzt werden. Jedoch wurde bereits im ursprünglichen U-Net-Paper ([1]) 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 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 **??** 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 **??** 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. **??**) 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 **??** 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. **??**), 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 **??** 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 **??** und insbesondere **??**. Es wird sich ebenfalls eine bessere Performanz dieses Modells bei der Radwegerkennung erhofft.

DenseNet121-Bike-U-Net

DBUNet! *(**DBUNet!**)* nutzt das in **??** 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 **??**. 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 **??** 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!** (**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 wahr-positiven *tp* nicht stärker gewichtet werden, sodass das Netz nicht besonders die wahr-positiven lernt. Des Weiteren ist die Quality aufgrund der in **??** 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! (**BCE!**) ist eine beliebte Verlustfunktion für Klassifikationsprobleme, allerdings in der Grundform wie in **??** 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 [2]. 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 **??**. 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. [2]) 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.

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] 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.

[2] 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.

A. Anhang

A.1. Architekturen

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.