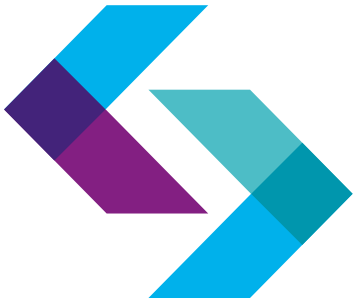

htlperg

Abteilung: Höhere Lehranstalt für Informatik

Diplomarbeit

Höhere Abteilung für Informatik

Thema:	ProcessAeye - Echtzeit Kamera Inpainting
eingereicht von:	Cedric Bauer ‹bauer.cedric@icloud.com› Stefan Czepl ‹stefan.czepl04@gmail.com›
eingereicht am:	3. April 2024
Betreuer:	Prof. Ing. Patrick Praher, MSc
In Zusammenarbeit mit:	Institut für Signalverarbeitung der JKU Assoc.-Prof. Dr. Michael Lunglmayr

1 Eidesstattliche Erklärung

Die unterfertigten Kandidaten / Kandidatinnen haben gemäß § 34 (3) SchUG in Verbindung mit § 22 (1) Zi. 3 lit. b der Verordnung über die abschließenden Prüfungen in den berufsbildenden mittleren und höheren Schulen, BGBl. II Nr. 70 vom 24.02.2000 (Prüfungsordnung BMHS), die Ausarbeitung einer Diplomarbeit mit der umseitig angeführten Aufgabenstellung gewählt.

Die Kandidaten / Kandidatinnen nehmen zur Kenntnis, dass die Diplomarbeit in eigenständiger Weise und außerhalb des Unterrichtes zu bearbeiten und anzufertigen ist, wobei Ergebnisse des Unterrichtes mit einbezogen werden können.

Die Abgabe der Diplomarbeit hat bis spätestens 03.04.2024 beim zuständigen Betreuer / der zuständigen Betreuerin zu erfolgen.

Die Kandidaten / Kandidatinnen nehmen weiters zur Kenntnis, dass gemäß § 9 (6) der Prüfungsordnung BMHS nur der Schulleiter bis spätestens Ende des vorletzten Semesters den Abbruch einer Diplomarbeit anordnen kann, wenn diese aus nicht beim Prüfungskandidaten (bei den Prüfungskandidaten) gelegenen Gründen nicht fertiggestellt werden kann.

Kandidaten / Kandidatinnen inkl. Unterschrift:

Perg, 03.04.2024	C. Bauer
Ort, Datum	Cedric Bauer
Perg, 03.04.2024	Stefan Czepl
Ort, Datum	Stefan Czepl

Signiert von: Cedric Alexander Bauer
Datum: 03.04.2024 12:31:45
Dieses Dokument ist digital signiert!
Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.
Prüfinformation: Informationen zur Prüfung der elektronischen Signatur finden Sie unter: www.a-trust.at/pdf

Signiert von: Stefan Czepl
Datum: 03.04.2024 12:40:03
Dieses Dokument ist digital signiert!
Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.
Prüfinformation: Informationen zur Prüfung der elektronischen Signatur finden Sie unter: www.a-trust.at/pdf

2 Gendererklärung

Im Sinne der besseren Lesbarkeit werden in dieser Diplomarbeit personenbezogene Bezeichnungen, die sich zugleich auf Frauen und Männer beziehen, generell nur in der im Deutschen üblichen maskulinen Form angeführt. Dies soll jedoch keinesfalls eine Geschlechterdiskriminierung oder eine Verletzung des Gleichheitsgrundsatzes zum Ausdruck bringen.

Perg, 03.04.2024	
Ort, Datum	Cedric Bauer
Perg, 03.04.2024	
Ort, Datum	Stefan Czepl

3 Danksagung

Wir bedanken uns bei sämtlichen Personen der HTL Perg und der JKU Linz, die diese Diplomarbeit ermöglicht haben.

Dabei möchten wir uns besonders bei unserem Diplomarbeitsbetreuer Herrn Professor Ing. Patrick Praher, MSc bedanken, welcher uns tatkräftig beim theoretischen und praktischen Teil dieser Arbeit unterstützt hat.

Wir bedanken uns auch bei Birgit Bauer, welche diese Diplomarbeit und die Zusammenarbeit mit der Johannes Kepler Universität Linz ermöglicht hat.

Weiters bedanken wir uns sehr herzlich beim gesamten Team des Instituts für Signalverarbeitung der Johannes Kepler Universität Linz und bei unserem Betreuer Assoc.-Prof. Dr. Michael Lunglmayr, welcher uns tatkräftig beim technischen Teil dieser Arbeit unterstützt hat.

4 Zusammenfassung

Die Grundidee von ProcessAeye ist es, fehlende oder beschädigte Teile eines Bildes in Echtzeit wiederherzustellen. Ziel ist eine Anwendung, welche Bilder einer Kamera einliest und diese anhand von fehlenden Bereichen wiederherstellt. Diese fehlenden Bereiche werden dabei vom Benutzer festgelegt.

Um dieses Ziel zu erreichen, werden drei Hauptkomponenten benötigt:

Der **NVIDIA Jetson Nano** ist ein kleiner, leistungsstarker Computer, der speziell für KI-Anwendungen entwickelt wurde und es ermöglicht, Bilder durch Machine-Learning-Modelle in Echtzeit zu verarbeiten und wiederherzustellen.

Durch ein **Deep Learning Modell** oder einen **klassischen Bildverarbeitungsalgorithmus**, werden die Bilder rekonstruiert. Im Verlauf der Arbeit sind verschiedene Modelle und klassische Algorithmen verglichen worden. Dabei sind die benötigte Zeit pro Wiederherstellungsdurchlauf und die Qualität der Ergebnisse verglichen worden und die besten Ansätze für die Echtzeit Wiederherstellung verwendet worden.

Die rekonstruierten Bilder werden in einer grafischen **Benutzeroberfläche** zusammen mit dem Ausgangsbild angezeigt. Auf dem Ausgangsbild kann der Benutzer mithilfe eines Zeichenmoduls die fehlenden Teile im Bild markieren. Um den Unterschied zwischen klassischen Algorithmen und Deep Learning Modellen anschaulich darzustellen, kann man zwischen sechs verschiedenen Ansätzen wählen, wobei drei davon klassische Algorithmen sind und drei Machine-Learning-Modelle.

Das Ergebnis ist eine Desktopanwendung, welche für den NVIDIA Jetson Nano optimiert ist und sechs Ansätze zur Bildwiederherstellung bietet. Dadurch, dass der Benutzer selbst die Maske zeichnen kann, ist die Anwendung optimal zur Veranschaulichung des Unterschieds zwischen klassischen Algorithmen und Deep Learning Modellen.

5 Abstract

The core idea of ProcessAeye is to restore missing or damaged parts of an image in real-time. The goal is to develop an application that reads images from a camera and restores them based on the missing areas, which are defined by the user.

To achieve this goal, three main components are required:

The **NVIDIA Jetson Nano** is a small, powerful computer designed specifically for AI applications, enabling real-time processing and restoration of images through machine learning models.

Through a **Deep Learning model** or a **classical image processing algorithm**, the images are reconstructed. Various models and classical algorithms have been compared in terms of the time required per restoration run and the quality of the results, with the best approaches being used for real-time restoration.

The reconstructed images are displayed in a graphical **user interface** alongside the original image. In the original image, the user can mark the missing parts of the image using a drawing module. To clearly demonstrate the difference between classical algorithms and deep learning models, users can choose from six different approaches, including three classical algorithms and three machine learning models.

The result is a desktop application optimized for the NVIDIA Jetson Nano which offers six approaches to image restoration. Because the user can draw the mask themselves, the application is ideally suited to illustrate the difference between classical algorithms and deep learning models.

Inhaltsverzeichnis

6 Einführung

6.1 Motiviation

Das Projekt „processAeye" ist ins Leben gerufen worden, um die Unterschiede von klassischen Algorithmen und künstlicher Intelligenz anschaulich darzustellen. Diese unterschiedlichen Ansätze werden anhand von Bildrekonstruktion in Echtzeit dargestellt. Für die Johannes Kepler Universität kann das Projekt bei Veranstaltungen wie z.B.: „Die lange Nacht der Forschung" ausgestellt werden, um das Interesse an künstlicher Intelligenz und Machine Learning zu wecken.

6.2 Zielsetzung

Ziel ist es, eine Anwendung zu entwickeln, welche die Bilder einer Kamera in Echtzeit rekonstruiert. Dabei legt der Benutzer die Bereiche selbst fest, welche fehlerhaft sind. Das System wird auf einem Nvidia Jetson Nano bereitgestellt und ist auch für diesen optimiert.

Das Projekt dient als eine Forschungsarbeit, mit dem Ziel, die abstrakten Themen künstliche Intelligenz und Machine Learning besser darzustellen. Es werden keine unternehmerischen oder kommerziellen Ziele verfolgt.

6.3 Projektinhalt - Überblick

Der Projektinhalt kann in drei wichtige Komponenten unterteilt werden:

6.3.1 Hardware

Um dieses Projekt erfolgreich umsetzen zu können, wurden drei Hardware-Komponenten verwendet.

- NVIDIA Jetson Nano (Abb. 6.1[1])
- Kamera (Abb. 6.2[2])
- ProcessAeye Kameragehäuse (Abb. 6.3)

Der NVIDIA Jetson Nano ermöglicht es, aufgrund der Leistung des Grafikprozessors klassische Bildverarbeitungsalgorithmen und Deep Learning Modelle schnell auszuführen, was ausschlaggebend für die Umsetzung dieser Arbeit ist. Damit die

Anwendung in Echtzeit Bilder wiederherstellen kann, muss eine Kamera zur Verfügung stehen. Mittels der Kamera kann ein Bild angezeigt werden und die Bilder können verarbeitet und repariert werden. Als Kamera kann jede beliebige Webcam verwendet werden. Je nach Qualität der Kamera ergibt sich auch die Qualität der Bildwiederherstellung. Das Kameragehäuse, welches von der Johannes Kepler Uni-

versität Linz bereitgestellt wurde, ermöglicht es eine Glasscheibe vor die Kamera zu platzieren, um besser darzustellen, wie Risse in einer Scheibe repariert werden. Auf dieser Glasscheibe kann man Risse einzeichnen oder andere Fehler mit Schreibmaterial oder Klebeband markieren. Das Kameragehäuse dient dazu, die Anwendung gut präsentieren zu können und immer eine passende Testumgebung schaffen zu können.

Abbildung 6.1: Jetson Nano Computer

Abbildung 6.2: Hardware: Kamera

Abbildung 6.3: ProcessAeye Kamera Gehäuse

[1] https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[2] https://resource.logitech.com/w_692,c_lpad,ar_4:3,q_auto,f_auto,dpr_1.0/d_transparent.gif/content/dam/products/logitech/webcams/c270-hd-webcam/gallery/c270-hd-webcam-2-0224.png?v=1

6.3.2 Bildwiederherstellung

In der Bildbearbeitung zielt der Prozess "Inpainting" darauf ab, Bilder zu reparieren oder störende Teile, die entfernt werden sollen, zu ersetzen. Ziel dabei ist es, dass das menschliche Auge nicht erkennen kann, dass Teile vom Bild bearbeitet, entfernt oder repariert wurden.

In dieser Arbeit sind zwei verschiedene Ansätze eingesetzt worden. Ziel war es, Unterschiede zwischen **klassischen Bildverarbeitungsalgorithmen** und **Deep Learning Modellen** aufzuzeigen und beide Ansätze zu testen und einzubauen.
Klassische Algorithmen setzen auf mathematische Berechnungen anhand der umliegenden Pixel oder Regionen des Bildes, um die Fehler zu reparieren.
Deep Learning Modelle hingegen lernen aus einer großen Menge an Bildern und Bilddaten. Anhand dieser Daten können bei neuen Bildern Ähnlichkeiten erkannt und Fehler besser ausgebessert werden und neue Bilddaten eingefügt werden, die gut ins Bild passen – im Gegensatz zur klassischen Variante. Deep Learning Modelle brauchen allerdings mehr Rechenleistung und dadurch dementsprechend mehr Zeit, was bei einem Echtzeitvideo zu stockenden Bildsequenzen führen kann.

Wenn man aus einem Gesicht zum Beispiel ein Auge entfernt, würde die klassische Variante nur die Hautfarbe einsetzen aufgrund der umliegenden Pixel. Ein Deep Learning Modell, welches mit Gesichtern trainiert wurde, kann ein neues Auge einsetzen.

Hier eine Gegenüberstellung

Abbildung 6.4: Klassische Bildverarbeitungsalgorithmen

Abbildung 6.5: Deep Learning

In dieser Gegenüberstellung wurde das rechte Auge als Fehler markiert. In Abbildung 6.4 wurde dies durch die klassische Variante repariert und in Abbildung 6.5 von einem Deep Learning Modell, welches mit Gesichtern trainiert wurde.

6.3.3 Benutzeroberfläche

Das Ziel ist es, eine benutzerfreundliche, moderne und leicht verständliche Oberfläche zu erschaffen. Es muss möglichst einfach sein, den Inpainting-Prozess zu starten, einen Algorithmus auszuwählen und eine selbstdefinierte Maske zu setzen. Es dürfen nicht zu viele technische Komponenten oder Optionen zur Verfügung stehen, welche Personen ohne technisches Fachwissen verwirren würden. Es wurden verschiedenste Prototypen entworfen, mit verschiedenen Ansätzen. Für eine Desktopapplikation, die ein ansprechendes Design hat und trotzdem die Funktionalität gewährleistet, wurden verschiedenste UI-Bibliotheken verglichen und sich für die PyQt5 Bibliothek entschieden, bei der alle nötigen Komponenten wie z.B. eine Zeichenfläche zum Zeichnen einer Maske, bereits bereitgestellt sind. Um eine noch ansprechendere Oberfläche zu bieten, wurden die bereits vorhanden Komponenten noch mit eigenen Stylings angepasst, mit abgerundeten Ecken und einer modernen Farbkombination. Die Oberfläche enthält keine verwirrenden Zahlen oder anderen technischen Daten, diese werden zur Laufzeit in der Konsole ausgegeben und Statistiken werden beim Beenden der Applikation generiert und angezeigt. So konnte eine Oberfläche geschaffen werden, die auch leicht verständlich ist für Personen, die keine Fachkenntnisse im Bereich Informatik haben.

6.4 Projektumfeld

6.4.1 Projektteam

Das Projektteam setzt sich aus Cedric Bauer und Stefan Czepl zusammen, beide Schüler der Höheren technischen Lehranstalt Perg. Cedric Bauer hat bereits viel Erfahrung auf Linux Betriebssystemen und war deshalb für den Hardware-Implementierungsabschnitt und die klassischen Bildverarbeitungsalgorithmen in dieser Arbeit verantwortlich. Stefan Czepl hat mit großem Interesse an maschinellem Lernen die Einbindung von verschiedenen Deep-Learning-Modellen übernommen.

Abbildung 6.6: Cedric Bauer (links) und Stefan Czepl (rechts)

6.4.2 Betreuung

Die Diplomarbeit wurde im technischen und theoretischen Teil durch Ing. Patrick Praher, MSc., mit seinem Wissen über Algorithmen und Machine Learning tatkräftig unterstützt.

6.4.3 Auftraggeber

Auftraggeber dieser Arbeit ist das Institut für Signalverarbeitung der Johannes Kepler Universität Linz. Der Hauptfokus des Instituts ist die Forschung an algorithmischen und hardwareorientierten Aspekten von Signalverarbeitungssystemen. [3]
Die Betreuung am Institut wurde tatkräftig von Assoc.-Prof. Dr. Michael Lunglmayr durchgeführt. Während der technischen Umsetzung wurde die Arbeit auch durch Jakob Winkler, MSc., welcher sich hauptsächlich mit Machine Learning und Linux auseinandersetzt, unterstützt. Herr Winkler hat bei der Implementierung von Deep Learning Modellen Hilfe anbieten und dem Projektteam Wissen weitergeben können.

Abbildung 6.7: Logo des Instituts für Signalverarbeitung

[3] https://www.jku.at/institut-fuer-signalverarbeitung/

7 Theoretische und fachpraktische Grundlagen und Methoden

7.1 Grundlegende Fachbegriffe

7.1.1 Inpainting

Inpainting beschreibt den Prozess der Rekonstruktion von fehlenden oder beschädigten Teilen eines Bildes. Es wird oft verwendet, um Rauschen zu entfernen, beschädigte oder unvollständige Teile wiederherzustellen oder um ganze Objekte aus Bildern zu entfernen. In Abbildung 7.1 ist ein Beispiel für eine Objektentfernung und ein Beispiel für die Wiederherstellung von Rissen in dem Bild, wobei die Ground Truth das Ausgangsbild ist, der Input das Bild mitsamt der zu rekonstruierenden Bereiche ist und der Output das rekonstruierte Bild zeigt.

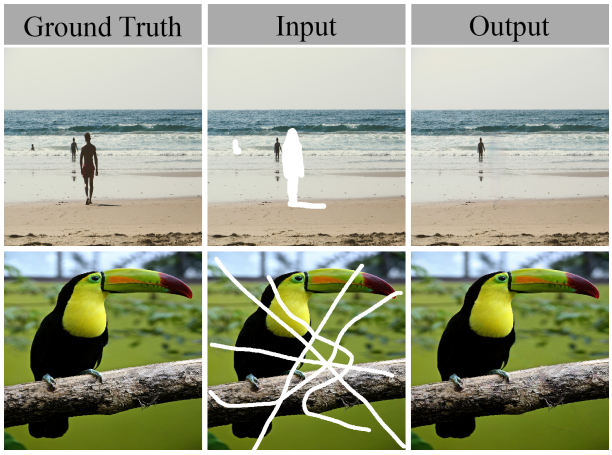

Abbildung 7.1: Beispiel für Image Inpainting

7.1.2 Maske

Die Maske, in Bezug auf Image Inpainting, gibt an, welcher Teil des Bildes kaputt ist, beziehungsweise welcher wiederhergestellt werden soll. Diese Maske ist ein binäres Bild, wobei Pixel mit dem Wert 1 Bereiche markieren, welche repariert werden sollen und Pixel mit dem Wert 0 Bereiche markieren, welche unverändert bleiben. Aufgrund dessen wird die Maske häufig weiß dargestellt, wie man in Abbildung 7.1 auf den mittleren Bildern sehen kann.

7.1.3 Machine Learning

Ein zentraler Bestandteil dieser Arbeit sind die Deep-Learning-Modelle, mit welchen Bilder wiederhergestellt werden. Generell versteht man unter Machine-Learning, dass Computer aus Daten etwas lernen. Dabei kann der Computer Sachen einordnen, Objekte erkennen oder Objekte wiederherstellen. Deep-Learning ist dabei eine spezielle Art von Machine-Learning und trägt seinen Namen, weil ein Deep-Learning Netz ein tiefes neuronales Netz ist.

Neuronale Netze sind an dem menschlichen Gehirn orientiert und bestehen aus vielen Neuronen, welche miteinander verbunden sind und Daten austauschen. Diese Neuronen sind in Schichten aufgeteilt, wobei es eine Eingabe- und Ausgabeschicht gibt, und dazwischen beliebig viele verborgene Schichten. Jede Verbindung der Neuronen hat ein Gewicht, welche angibt, wie stark ein Neuron das andere beeinflusst.

Ein neuronales Netz lernt, indem es die Gewichte, basierend auf den Eingabedaten und den gewünschten Ausgaben, anpasst. Das passiert durch verschiedenste Methoden, welche die gewünschten Ausgaben mit den tatsächlichen vergleichen.

7.2 Verwendete Technologien

7.2.1 Python

Für die technische Umsetzung dieser Arbeit wurde die Programmiersprache Python verwendet. Python ist eine weit verbreitete Programmiersprache, die viele Möglichkeiten bietet für Projekte mit Machine Learning. Auch anderen Themen wie klassischen Algorithmen oder Datenbankschnittstellen sind leicht umsetzbar. Python ist eine objektorientierte Programmiersprache, aber sowohl auch interpretiert und unterstützt Konzepte wie Module, Exceptions und mehrere dynamische Datentypen. Der Python-Interpreter und die Menge an Standard-Bibliotheken sind für jeden frei verfügbar, was die Produktivität beim Programmieren steigert.[1] Python ist aufgrund der vielen Pakete und der großen Community im Bereich Machine Learning die erste Wahl für diese Arbeit gewesen. Außerdem zeichnet sich Python durch eine übersichtliche Syntax aus, welche für gut lesbaren und aufgeräumten Code sorgt.

7.2.2 Pip

Pip[2] ist ein Paketverwaltungsprogramm für Python. Es dient dazu, Pakete aus dem "Python Package Index" zu installieren. Mit Pip kann man diese Pakte installieren, deinstallieren und ihre Versionen verwalten. Pip bietet die einfache Möglichkeit, per Befehl im Terminal oder der Kommandozeile Pakete zu suchen und zu installieren.

Beispiel

```
pip install PyQt5
```

mit diesem Befehl kann man das Paket PyQt5 installieren und dann in einem Python Projekt benutzen.
Es gibt auch Alternativen von Pip wie zum Beispiel Conda oder Pipenv.

7.2.3 Conda

Conda[3] ist ein Paket und Umgebungsverwaltungs-Werkzeug, welches für Python entwickelt worden ist, aber auch viele andere Programmiersprachen unterstützt. Mit Conda können Umgebungen erstellt werden, welche verschiedene Python Versionen und alle möglichen Pakete enthalten können. Der Vorteil von Conda liegt in der kompakten Paketverwaltung, aber auch in der leichten Verteilung von Umgebungen auf andere Geräte. In diesem Projekt ist Conda verwendet worden, um eine eigene Umgebung mit allen nötigen Python Paketen zu erzeugen. Diese Umgebung konnte dann auf alle Geräte, mit welchen entwickelt worden ist, übertragen werden.

[1] https://www.python.org/doc/essays/blurb/
[2] https://pip.pypa.io/
[3] https://docs.conda.io/

7.2.4 Git und GitLab

Git[4] ist eine Versionsverwaltungs-Software, die dazu dient, gemeinsam an Projekten zu arbeiten und den Verlauf des Projektes zu verfolgen. Dabei werden die Dateien gespeichert und bei Änderungen vom Entwickler hochgeladen. Der Unterschied zu anderen Versionierungssystemen ist, dass nicht immer die gesamten Dateien bei jeder Änderung neu gespeichert werden. Bei Git werden nur die veränderten Dateien gespeichert und für die bereits vorhanden Dateien wird ein Verweis auf die bereits vorhandene Datei erstellt.

GitLab bietet dazu eine passende Webapplikation, um diesen Prozess zu vereinfachen und zusätzliche Möglichkeiten zu bieten, wie das Erzeugen von Issues, Meilensteinen und einem Issue-Board.

In dieser Arbeit wurde das GitLab der HTL-Perg verwendet.

7.2.5 Cuda

CUDA steht für Compute Unified Device Architecture und ist eine Programmierschnittstelle, welche von NVIDIA[5] bereitgestellt wird, um einen direkten Zugriff auf den Grafikprozessor zu ermöglichen. CUDA ermöglicht es Entwickler Programme zu entwickeln, die neben der CPU auch parallel mit der Grafikkarte arbeiten, um schnellere Verarbeitungen zu ermöglichen. In dieser Arbeit war es notwendig, diese Grafikbeschleunigung zu nutzen, um die Echtzeitwiederherstellung zu ermöglichen und ein flüssiges Bild zu erhalten.

7.2.6 PyQt5

PyQt5 ist eine Sammlung des Toolkits Qt, welches es ermöglicht, grafische Benutzeroberflächen zu entwickeln. Ursprünglich ist die Qt-Bibliothek eine C++ Bibliothek und wurde von der Qt Company[6] Open-Source veröffentlicht. PytQt bringt diese Bibliothek in die Python-Welt, was es ermöglicht, auch in Python solche Desktopapplikationen zu entwickeln. Für diese Arbeit war es notwendig, solch eine Bibliothek zu verwenden, um eine anspruchsvolle und stabile Benutzeroberfläche zu erschaffen. Außerdem bietet PyQt die Möglichkeit, die bereits vorhanden Komponenten noch anzupassen, mit eigenen Farbschemas, Größen und Schriftarten, was zu einer noch besseren Erfahrung für Benutzer beitragen kann. Außerdem bietet PyQt eine Reihe an Funktionen wie die Kommunikation mit SQL-Datenbanken, Web-Toolkits und XML-Verarbeitung, was in dieser Arbeit allerdings keine Verwendung gefunden hat.

[4] https://git-scm.com/book/en/v2
[5] https://www.nvidia.com/
[6] https://www.qt.io/

7.3 Verwendete Entwicklungssysteme

7.3.1 PyCharm

PyCharm[7] ist eine leistungsstarke Entwicklungsumgebung für Python, welche von JetBrains entwickelt worden ist. Die Funktionalitäten von PyCharm umfassen Code-Vervollständigung, frühzeitige Fehlererkennung und diverse Entwickler-Tools wie einen Debugger.

Im Projekt ist die Anwendung mit PyCharm entwickelt worden.

7.3.2 Visual Studio Code

Visual Studio Code[8] ist ein kostenloser, Open-Source-Code-Editor, welcher von Microsoft entwickelt worden ist. VS Code bietet die Möglichkeit, Extension herunterzuladen, wie zum Beispiel IntelliSense für Python. Für das Projekt relevant ist die Einfachheit von Visual Studio Code, welche verwendet wurde, um einzelne Python Files mitsamt der verschiedenen Algorithmen zu testen.

[7] https://www.jetbrains.com/pycharm/
[8] https://code.visualstudio.com/

7.4 Verwendete Bibliotheken und Plug-Ins

7.4.1 PyTorch

PyTorch ist ein Open-Source Framework für Machine Learning, basierend auf der Torch-Bibliothek, welches in Python geschrieben ist und somit bestens in Python integriert ist. Seine Anwendung findet PyTorch vor allem in den Bereichen Computer Vision und Natural Language Processing. PyTorch bietet zwei Hauptfunktionalitäten, Tensor Berechnung mit GPUs für eine erhebliche Beschleunigung und ein "tape-based autograd" System zum Erzeugen von neuronalen Netzen. Beim "tape-based autograd" handelt es sich um einen Mechanismus, welcher automatisch die Gradienten, also die Änderung der Gewichte und Bias-Werte, eines neuronalen Netzes berechnet. Durch diese Gradienten werden dann die Parameter des Netzes angepasst, um die Genauigkeit zu optimieren.[2]

Mit PyTorch sind alle drei Deep-Learning-Modelle dieses Projektes trainiert worden: Generative-Inpaiting, MISF und DeepFillv2.

7.4.2 Torchvision

Torchvision ist ein Teil vom PyTorch Projekt, muss jedoch extra heruntergeladen werden und ist für Computer Vision Aufgaben entwickelt worden. Es bietet Funktionen wie effiziente Bild Transformation, beliebte Datasets und einige vortrainierte Modelle. Torchvision kommt mit GPU-Unterstützung, wodurch es sehr schnell ist.[6]

Im Projekt wurde Torchvision verwendet, um die Bilder zu verkleinern oder zu vergrößern und diese auf Tensors umzuwandeln.

7.4.3 OpenCV

OpenCV ist eine Open-Source Bibliothek mit Algorithmen für die Bildverarbeitung. Verfügbar ist diese Bibliothek für mehrere Programmiersprachen wie Java, C++, Python und mehr. Diese Bibliothek kann für Objektidentifikation, Videoverarbeitung und Inpainting verwendet werden. OpenCV enthält über 2500 Algorithmen und deckt somit verschiedenste Anwendungsfälle in der Bildverarbeitung ab, wie z.B. bei Street View oder Überwachungskameras.

7.4.4 NumPy

NumPy ist die wichtigste Bibliothek, wenn es um die Berechnung von Matrizen, Vektoren oder generell um mehrdimensionale Arrays geht. Sie wird als Standard für viele andere Bibliotheken wie Pandas oder Matplotlib verwendet. Auch Tensors, mit welchen in PyTorch und TensorFlow gearbeitet wird, werden in ndArrays(n-dimensionale Arrays) abgebildet. NumPy stellt eine Vielzahl von mathematischen Funktionen bereit, welche auf den Datenstrukturen ausgeführt werden können.[1]

Im Projekt sind NumPy Arrays für Tensors verwendet worden, aber auch für jegliche andere Berechnungen mit den Bildern als drei-dimensionalem Array zum Beispiel für die Berechnung der Frobeniusnorm, siehe Frobeniusnorm: 8.4.1.1.

7.4.5 Pillow

Pillow ist ein Fork der PIL Bibliothek und bietet umfangreiche Funktionen zum Verwalten von Bildern. Darunter befindet sich die Unterstützung von vielen Bildformaten und das Öffnen und Manipulieren von Bildern.[3]

In diesem Projekt ist Pillow hauptsächlich dafür verwendet worden, Bilder zu öffnen und zu speichern sowie zum Konvertieren von Arrays zu Bildern.

7.4.6 Pyyaml

Pyyaml[9] ist eine Python Bibliothek zum Parsen und Emitten von yaml Dateien. Das heißt, es ermöglicht uns *.yml und *.yaml Dateien einzulesen und zu verwenden. Yaml ist eine Datenserialisierungssprache, welche leicht für Menschen lesbar ist und für den Computer effizient zum Verarbeiten ist. Seinen Anwendungsbereich findet Yaml bei Konfigurationsdateien.

Für das Projekt ist pyyaml verwendet worden, um Config-Dateien von verschiedenen neuronalen Netzen zu laden. In diesen Dateien befinden sich die relativen Pfade, zu den vortrainierten Modell-Dateien und mithilfe dieser Pfade konnten die Modelle in die Anwendung geladen werden.

[9] https://pyyaml.org/wiki/PyYAMLDocumentation

7.4.7 Matplotlib

Matplotlib ist eine Python Bibliothek, die dazu dient, um mathematische Statistiken und Darstellungen zu erzeugen[10]. In dieser Arbeit wurde diese dazu verwendet, um den Vergleich zwischen verschiedenen Algorithmen und den beiden Ansätzen, klassische Bildverarbeitungsalgorithmen und Deep Learning, zu ermöglichen. Man kann nach einer Berechnung oder gewissen Dauer verschiedene Datenpunkte speichern lassen und visuell darstellen, um ein gutes Verständnis für die Leistung der einzelnen Prozesse zu schaffen. Man kann bereits mit wenig Aufwand verschiedene Diagramme wie Liniendiagramme und Fehlerdiagramme für die Analyse der eigenen Applikation erzeugen.

7.4.8 PyVirtualcam

Pyvirtualcam[11] ist eine Python Bibliothek, die dazu dient, Bilder an eine virtuelle Kamera zu senden. Diese virtuelle Kamera kann dazu verwendet werden, um sie für Videoanrufe oder Videoaufnahmen zu nutzen. In dieser Arbeit wurde diese Bibliothek verwendet, um es zu ermöglichen, das reparierte Echtzeitbild, als Kamera auszuwählen. Diese Bibliothek ist auf eine bereits installierte virtuelle Kamera angewiesen und kann nur an diese Bilder schicken.
Dabei werden mehrere Betriebssysteme und Programme unterstützt:

- Windows
  - OBS
  - Unity Capture
- MacOS
  - OBS
- Linux
  - v4l2loopback

[10] https://matplotlib.org/
[11] https://github.com/letmaik/pyvirtualcam

7.5 Sonstige verwendete Software

7.5.1 Balena Etcher

Balena Etcher[12] ist ein Open-Source-Programm, das das Erzeugen von Bilddateien wie .ISO oder .IMG sowie von Live-SD-Karten und USB-Flash-Laufwerken ermöglicht. In dieser Arbeit wurde die Software verwendet, um die SD-Karte mit dem von NVIDIA bereitgestellten Image zu beschreiben, das vom Jetson Nano verwendet wird. Dabei wird die SD-Karte formatiert und in der Software die zu beschreibende SD-Karte ausgewählt sowie das Image, das man verwenden möchte.

7.5.2 OBS Virtual Camera

Die OBS Virtual Camera ist eine Funktion von OBS Studio, die es ermöglicht, Videos, Bildschirmaufnahmen oder andere Medien als virtuelle Kamera in anderen Anwendungen, wie z.B. Microsoft Teams, zu nutzen. In dieser Arbeit wurde diese Anwendung verwendet, um die Ausgabe des reparierten Kamerabildes als virtuelle Kamera zur Verfügung zu stellen. So kann man beispielsweise im Hintergrund ein unaufgeräumtes Regal durch eine weiße Wand ersetzen und bei einem Videoanruf teilnehmen.

7.5.3 v4l2loopback

v4l2loopback ist ein Linux-Treiber, der es ermöglicht, virtuelle Video4Linux2-Geräte zu erzeugen. Diese virtuellen Geräte funktionieren wie normale Videogeräte, mit dem Unterschied, dass die Quelle eine Datei, ein Livestream oder ähnliches ist, anstatt von einer physischen Kamera. Die Funktionsweise ist dabei ähnlich wie bei der OBS Virtual Camera (siehe Abschnitt 7.5.2).

[12] https://etcher.balena.io/

7.6 Verwendete Hardware

7.6.1 Nvidia Jetson Nano

Das Jetson Nano Entwicklerkit ist ein kleiner, aber leistungsstarker Computer der amerikanischen Firma NVIDIA.

In Abb. 7.2[13] kann man das Entwicklerkit sehen und die Schnittstellen, die zur Verfügung gestellt werden.

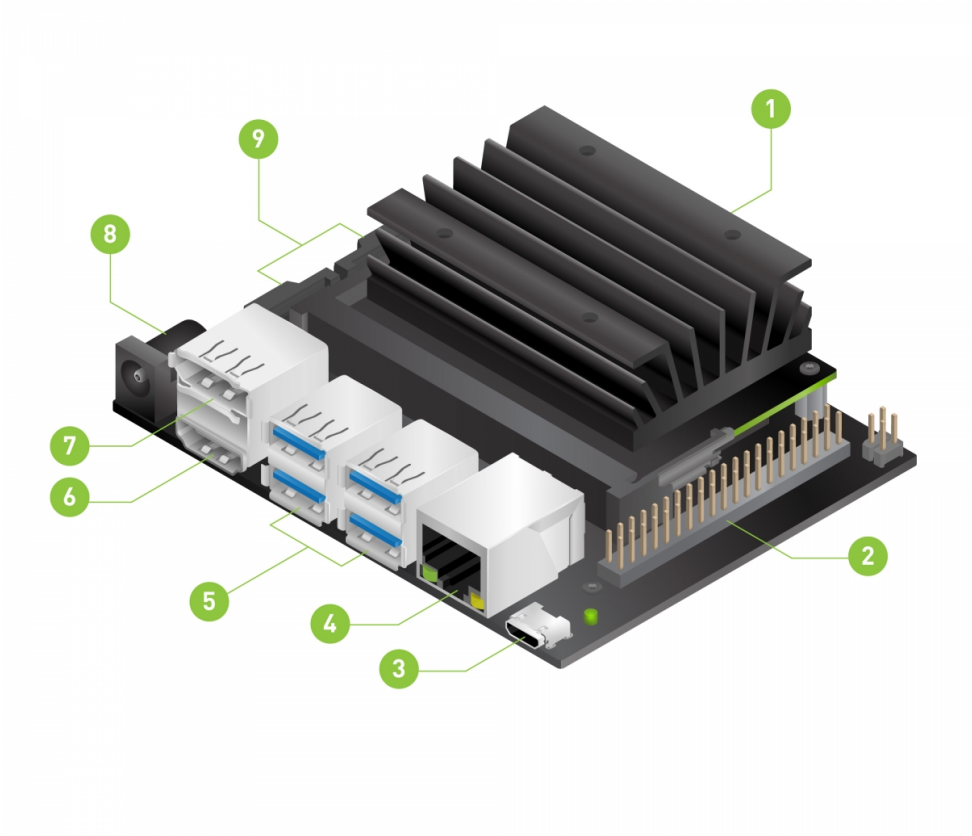

Abbildung 7.2: Jetson Nano Entwicklerkit

1	microSD-Kartensteckplatz für den Hauptspeicher
2	40-polige Erweiterungsleiste
3	Micro-USB-Anschluss für 5-V-Stromversorgung oder für den Gerätemodus
4	Gigabit-Ethernet-Anschluss
5	USB 3.0-Anschlüsse (x4)
6	HDMI-Ausgangsanschluss
7	DisplayPort-Anschluss
8	DC-Buchse für 5V Stromeingang
9	MIPI CSI-2 Kamera-Anschlüsse

[13] https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit

Das Entwicklerkit ist mit einem Quad-Core ARM Cortex-A57 Prozessor ausgestattet, der auf bis zu 1,43 GHz getaktet ist, einem NVIDIA-Grafikprozessor mit Maxwell-Architektur und 128 CUDA-Recheneinheiten sowie 4 GB DDR4 Arbeitsspeicher. Der Jetson Nano verfügt über keinen eingebauten Festplattenspeicher, bietet aber einen microSD-Slot. Diese Komponenten ermöglichen auf der kompakten Fläche von 70 × 45 mm das Ausführen von KI-Anwendungen und bringen große Vorteile in mehreren Branchen wie Smart Cities und der Bildverarbeitung.

Von Nvidia wird eine Linux Distribution bereitgestellt, die bereits mit einer Reihe Bibliotheken und zahlreichen Projekten ausgestattet ist. Durch diese Ausstattung soll es Nutzern ermöglicht werden, schnell praktisches Wissen für den Umgang mit dem Entwicklerkit zu sammeln. Die Website von Nvidia bietet auch viele Informationen und eine genaue Anleitung zur Inbetriebnahme, um den Einstieg möglichst einfach zu gestalten. Um das Entwicklerkit in Betrieb zu nehmen, muss wie man in Abb. 7.3[14] sehen kann die microSD-Karte mit dem installierten Betriebssystem in den Kartensteckplatz eingesetzt werden.

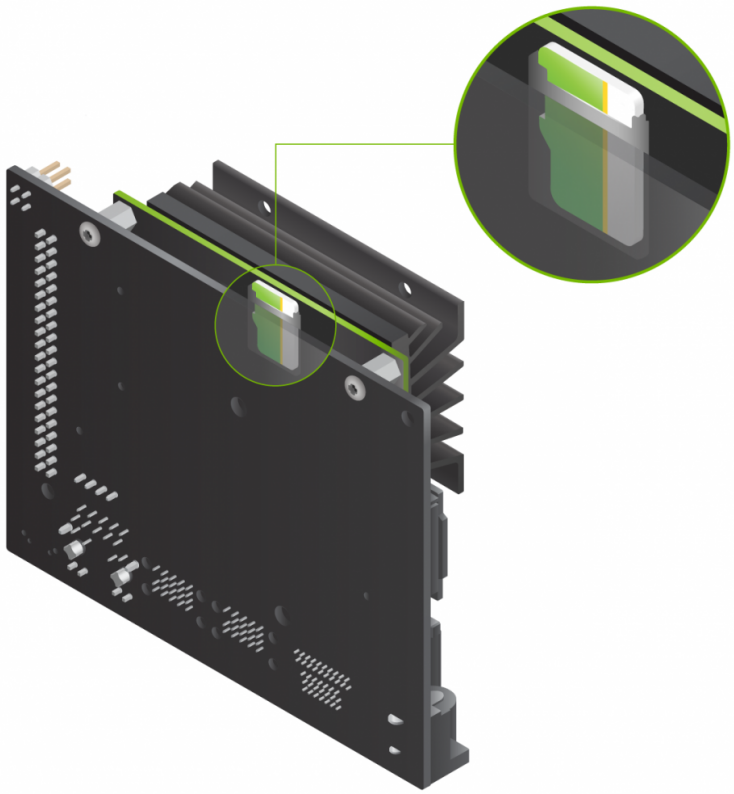

Abbildung 7.3: Jetson Nano: microSD einsetzen

[14] https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit

7.7 Verwendete klassische Bildverarbeitungsalgorithmen

7.7.1 OpenCV Telea

OpenCV Telea ist ein Algorithmus zur Wiederherstellung von Bildern und Videos und basiert auf der Fast Marching Methode und dem Paper "An Image Inpainting Technique Based on the Fast Marching Method" von Alexandru Telea in 2004 [8]. Diese Methode funktioniert so, dass eine Bildregion, die repariert werden soll, markiert wird. Die Bearbeitung beginnt am äußeren Rand dieser Region und arbeitet sich dann nach innen vor, um den Bereich vollständig auszufüllen. Um einzelne Pixel oder Bildpunkte zu reparieren, wird dabei ein Umkreis um den jeweiligen Punkt analysiert. Dieser Bildpunkt wird dann durch eine gewichtete Summe der umliegenden, bereits bekannten Pixel ersetzt.

Sobald ein Pixel repariert wurde, führt der Algorithmus dies beim nächstliegenden Punkt fort. Die Fast-Marching-Methode hat den Zweck, dass zuerst die Punkte repariert werden, wo bereits bekannte Bereiche existieren.
Der Vorteil dieses Algorithmus ist die einfache Einbindung, da man nur die OpenCV Bibliothek installieren muss und die Funktion mit einem Bild, einer Maske, einem Radius und der Methode implementieren muss, wie man in Listing 7.1 sehen kann.

```
import cv2 as cv
img = cv.imread('Bild.png')
mask = cv.imread('Maske.png', cv.IMREAD_GRAYSCALE)
dst = cv.inpaint(img,mask,3,cv.INPAINT_TELEA)
```

Listing 7.1: Implementierung von OpenCV Telea

In Abb. 7.4 ist links ein Bild von einem Hund zu sehen mit sichtbar fehlenden Bildsegmenten. Diese wurden durch die Maske markiert. Auf der rechten Seite sieht man das Ergebnis durch die Methode von Listing 7.1.

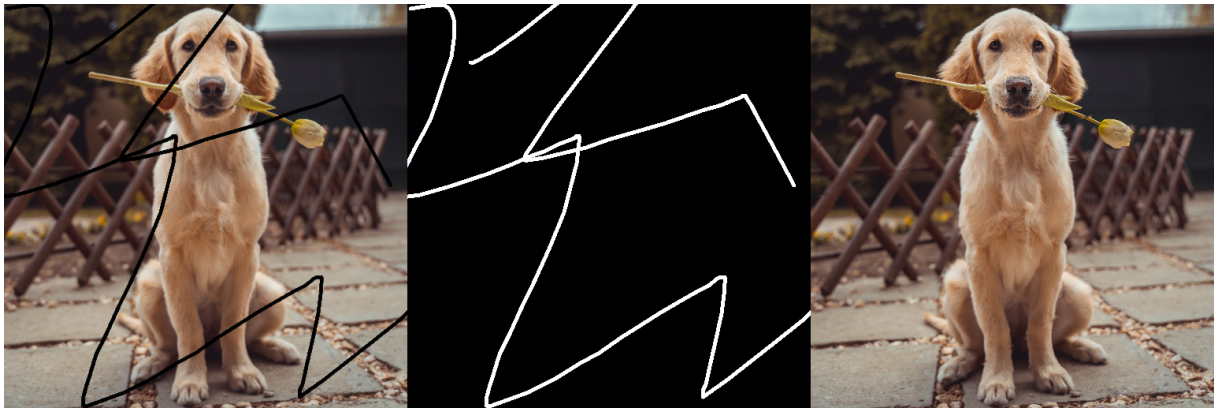

Abbildung 7.4: Beispiel TELEA

Die Stärken dieses Algorithmus sind kleine bis mittelgroße Fehler. Fehler oder Lücken können schnell ausbessert werden und durch die umliegenden Pixel kann ein natürliches Gesamtbild entstehen. Bei größeren Flächen, die fehlen oder repariert werden sollen, kann der Telea-Algorithmus zu weniger optimalen Ergebnissen führen, da die Informationen der unmittelbar angrenzenden Pixel verwendet wird und das dazu führt, dass identische Pixel wiederholt eingesetzt werden.

Ein wesentliches Problem des Telea-Algorithmus ist Blurring, also die Unschärfe, die beim Inpainting-Prozess entstehen kann. Der Algorithmus tendiert dazu, bei größeren Bereichen, die größer als 10 bis 15 Pixel sind, die Farbe der umliegenden Pixel zu verwischen. Das liegt daran, dass der Algorithmus für jedes Pixel, welches repariert werden muss, eine gewichtete Durchschnittsberechnung auf Basis der umliegenden, bereits bekannten Pixel durchführt. Diese Methode zielt darauf ab, die Farben und Helligkeiten der benachbarten Pixel so zu mischen, sodass der reparierte Bereich einheitlich aussieht.

7.7.2 OpenCV NS

OpenCV NS ist ein Algorithmus zur Wiederherstellung von Bildern und Videos und basiert auf der Arbeit "Navier-Strokes, Fluid Dynamics, and Image and Video Inpainting" von M. Bertalmio, A. L. Bertozzi und G. Sapiro aus dem Jahr 2001[5].
Dieser Algorithmus basiert auf der Fluiddynamik und auf partiellen Differenzialgleichungen. Die Fluiddynamik ist ein Teilgebiet der Physik und untersucht, wie Flüssigkeiten und Gase fließen und auf Kräfte reagieren.
Die Idee des Algorithmus ist, dass Bildinformationen ähnlich wie Flüssigkeiten fließen können und so Lücken und beschädigte Bereiche auffüllen zu können. Dazu werden die Bewegung und das Verhalten der Pixel in einer fluidähnlichen Dynamik modelliert. Dabei werden Differenzialgleichungen genutzt, um die Bewegung der Bildinformation zu steuern und so die fehlenden Bereiche zu ersetzen.
Anders als der Telea-Algorithmus der sich nur auf die Informationen rund um den zu ersetzenden Bereich begrenzt, nutzt NS ein komplexeres Verfahren, wobei das gesamte Bild berücksichtigt wird.
Der Vorteil ist bei diesem Algorithmus die einfache Einbindung, da man nur die OpenCV Bibliothek installieren muss und die Funktion mit einem Bild, einer Maske, einem Radius und der Methode implementieren kann, wie man in Listing 7.2 sehen kann.

```
import cv2 as cv
img = cv.imread('Bild.png')
mask = cv.imread('Maske.png', cv.IMREAD_GRAYSCALE)
dst = cv.inpaint(img,mask,3,cv.INPAINT_NS)
```

Listing 7.2: Implementierung von OpenCV NS

In Abb. 7.5 ist links ein Bild von einem Hund zu sehen mit sichtbar fehlenden Bildsegmenten. Diese wurden durch die Maske markiert. Auf der rechten Seite sieht man das Ergebnis durch die Methode von Listing 7.2.

Abbildung 7.5: Beispiel NS

Durch diese Methode können größere Bildbereiche effektiv wiederhergestellt werden. Durch den Ansatz der Fluiddynamik können Informationen aus weiter entfernten Bildbereichen miteinbezogen werden und so für eine natürliche Wiederherstellung sorgen.

Durch die komplexe Berechnung der Fluiddynamik und der partiellen Differenzialgleichungen nimmt dieser Algorithmus mehr Rechenleistung in Anspruch. In einer Echtzeitanwendung kann dies schnell zu stockenden Bildsequenzen führen.

7.7.3 Skimage

Skimage ist ein Paket innerhalb der "scikit-image"-Bibliothek für Bildverarbeitung in Python. Diese Bibliothek wurde vom scikit-image Team entwickelt [7]. Diese Bibliothek bietet eine Reihe an Algorithmen zur Bildbearbeitung wie Filter und Farbraummanipulation. Skimage bietet eine Inpaint Funktion, welche auf der biharmonischen Gleichung basiert.
Die biharmonische Gleichung wird in verschiedenen Bereichen der Physik und Technik angewandt, um zum Beispiel bei der Berechnung von Biegeplatten in der Mechanik. In der Bildverarbeitung ermöglicht diese Gleichung eine Art von Inpainting bei der glatte Übergänge in den reparierten Bereichen erzeugt werden können. Dabei werden Informationen aus der Umgebung benutzt, um diesen Bereich zu reparieren.
Im Gegensatz zu den vorher beschriebenen Methoden, die auf umliegenden Pixeln oder auf der Fluid-Dynamik basieren, nutzt die biharmonische Methode von Skimage ein mathematisches Modell, das darauf abzielt, die Bildinformationen über einen glatteren, natürlichen Weg wiederherzustellen.
Der Vorteil ist auch bei diesem Algorithmus die einfache Einbindung, da man nur die Bibliothek installieren muss und die Funktion mit einem Bild und einer Maske implementieren kann, wie man in Listing 7.3 sehen kann.

```
from skimage import io, color
from skimage.restoration import inpaint
img = io.imread('Bild.png')
mask = io.imread('Maske.png', as_gray=True)

mask_bool = mask > 0

dst = inpaint.inpaint_biharmonic(img, mask_bool, multichannel=True)
```

Listing 7.3: Implementierung von Skimage

In Abb. 7.6 ist links ein Bild von einem Hund zu sehen mit sichtbar fehlenden Bildsegmenten. Diese wurden durch die Maske markiert. Auf der rechten Seite sieht man das Ergebnis durch die Methode von Listing 7.3

Abbildung 7.6: Beispiel Skimage

7.8 Verwendete Deep-Learning Modelle

In der Arbeit sind verschiedene Deep Learning Modelle verwendet worden. Dabei sind vortrainierte State of the Art Modelle aus Open-Source Github Repositorys verwendet worden.

7.8.1 Generative Inpainting

Das Generative Inpainting Modell stammt aus einem Github Repository[15], wobei dieses nur eine PyTorch Implementierung des originalen Paper ”Generative Image Inpainting with Contextual Attention”von Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu und Thomas S. Huang[10] basierend auf der TensorFlow Implementierung[16] ist. Der Generative Inpainting Ansatz verwendet Generative Adversarial Networks(GANs) und Convolutional Neural Networks(CNNs), mit welchen sowohl die Texturen als auch der Kontext des Bildes berücksichtigt werden können, um ein semantisches Ergebnisbild zu schaffen.

Convolutional Neural Networks

Convolutional Neural Networks[17] sind eine spezielle Art von neuronalen Netzen, welche für die Verarbeitungen von Bildern entworfen worden sind. Dabei kann man sich CNNs wie einen Künstler vorstellen, welcher ganz genau die einzelnen Merkmale eines Bildes analysiert – wie Linien, Formen oder Muster. Also sind CNNs zuständig für die Merkmale Erkennung eines Bildes.

Der Computer sieht Bilder als 2-dimensionale-Arrays, wobei jeder Wert einen Pixel widerspiegelt. In dem Fall von Bildwiederherstellung werden Bilder meistens in Farben betrachtet, und somit gibt es für jeden der 3 RBG Kanäle ein eigenes 2-dimensionales-Array.
Um die Merkmale aus den Bildern zu erkennen, nutzen CNNs zwei Hauptschritte:

- **Convolution:** Diese Technik verwendet Filter oder auch Kernels genannt. Diese Filter werden Schritt für Schritt (Blau -> Rot -> Grün siehe Abb. 7.7(vlg[18])) über das Bild gelegt. Man kann es sich wie einen Stempel vorstellen, welcher über jede Stelle des Bildes gedrückt wird, um bestimmte Formen zu erkennen. Jedes Mal, wenn eine Form erkannt wird, wird die Stelle und wie genau die Form erkannt worden ist, in eine sogenannte Feature-Maps geschrieben. Ein Convolution Schritt umfasst meistens mehrere Filter, und somit werden auch mehrere Feature-Maps produziert.

[15] https://github.com/daa233/generative-inpainting-pytorch?tab=readme-ov-file
[16] https://github.com/JiahuiYu/generative_inpainting/tree/v2.0.0?tab=readme-ov-file.
[17] https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/
[18] https://saturncloud.io/images/blog/convoluting-a-5x5x1-image-with-a-3x3x1-kernel-to-get-a-3x3x1-convolved-f
gif

- **Pooling:** Der Pooling Prozess findet statt, nachdem CNN mehrere Merkmale erkannt hat. Diese Technik nimmt die erkannten Merkmale und fasst sie in vereinfachter Form zusammen. Dieser Schritt hilft dem Netzwerk, sich auf die wesentlichen Merkmale und Informationen zu konzentrieren.

  In Abb. 7.8(vlg.[19]) ist der Pooling Prozess abgebildet. Hierbei wird das sogenannte Max. Pooling angewandt, wobei immer der maximale Wert übernommen wird. Da der Filter eine Größe von 2x2 hat, werden auch immer so viele Pixel zusammengefasst.

Indem diese Schritte mehrmals wiederholt werden, ist es für ein CNN möglich, immer komplexere Merkmale zu identifizieren. Somit kann CNN im konkreten Anwendungsfall der Bildwiederherstellung die Merkmale des Bildes extrahieren, welche noch vorhanden sind, und dann dementsprechend die fehlenden Teile wieder auffüllen.

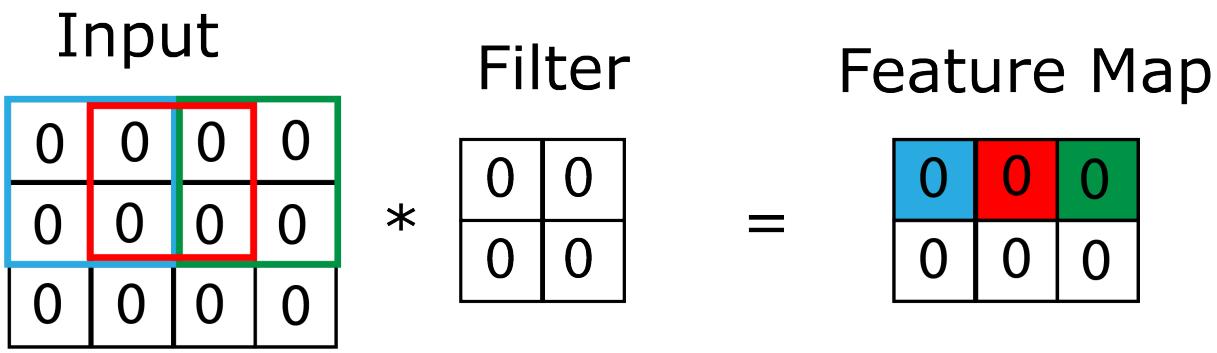

Abbildung 7.7: CNN Convolution

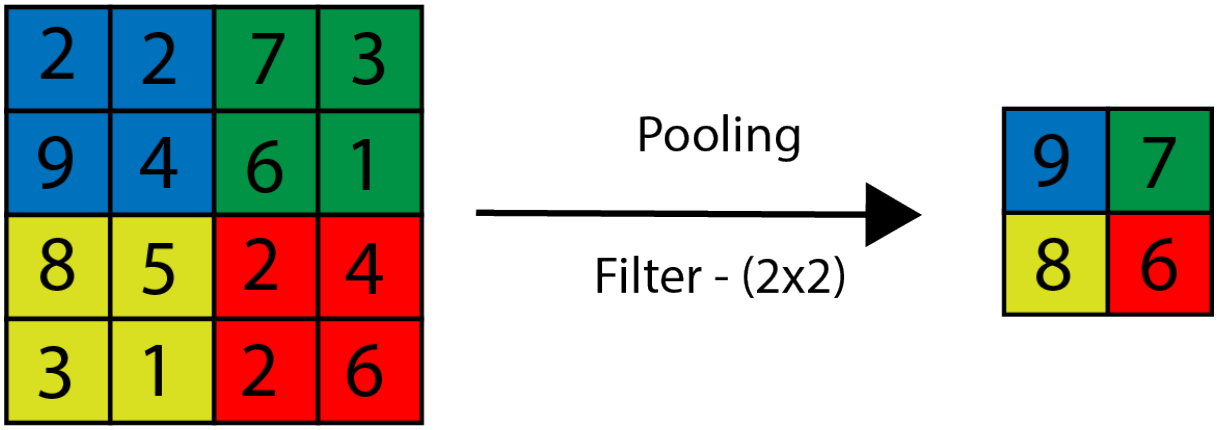

Abbildung 7.8: CNN Pooling

[19] https://media.geeksforgeeks.org/wp-content/uploads/20190721025744/Screenshot-2019-07-21-at-2.57.13-AM.png

Generative Adversarial Networks

Generative Adversarial Networks[20] sind eine Klasse von Machine-Learning Algorithmen, welche vor allem bei generativen Modellen eingesetzt werden. Ein GAN besteht aus zwei neuronalen Netzen, welche gegeneinander konkurrieren.

- **Generator:** Der Generator bekommt einen zufälligen Vektor an Daten und versucht daraus, ein Bild zu generieren. Das Ziel des Generators ist es, echte Daten zu generieren. Im Falle der Bildwiederherstellung, ist der Generator zuständig, die fehlenden Teile eines Bildes wieder zu generieren.
- **Discriminator:** Der Discriminator hat die Aufgabe, die vom Generator erstellten Daten von echten zu unterscheiden. Dabei wird der Discriminator darauf trainiert, die erstellten Daten von den echten zu unterscheiden.

Diese beiden Netzwerke konkurrieren gegeneinander, der Generator erstellt ständig neue Daten und der Discriminator versucht, diese von den Echten zu unterscheiden.

In Abb. 7.9(vlg.[21]) ist die GAN Architektur veranschaulicht. Je nachdem, ob der Discriminator die Fälschung richtig erkennt oder nicht, verliert der Generator oder der Discriminator. Das Netzwerk des jeweiligen Verlierers wird dann etwas angepasst. Dieser Prozess läuft so lange, bis der Generator täuschend echte Ergebnisse produziert.

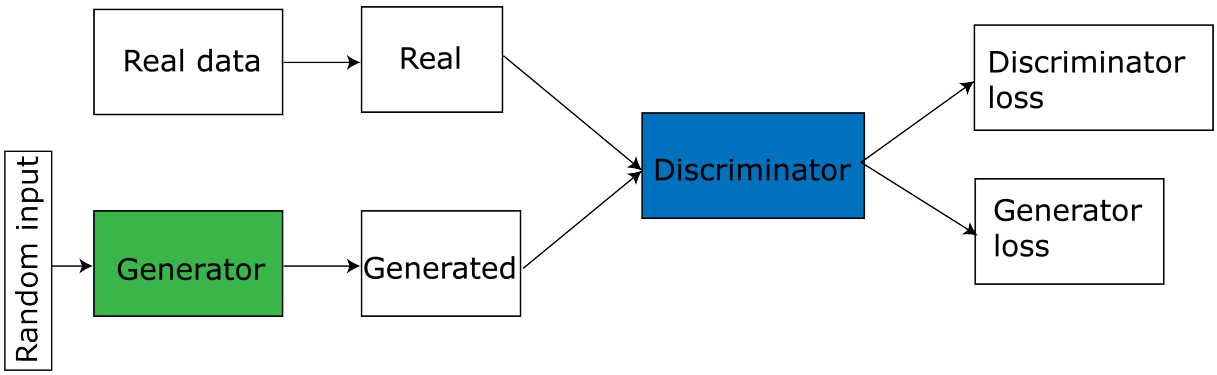

Abbildung 7.9: GAN Architektur

Diese Architektur findet sich hauptsächlich in der Trainingsphase von neuronalen Netzen. In dieser Arbeit ist hauptsächlich der vortrainierte Generator verwendet worden.

[20] https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/
[21] https://developers.google.com/machine-learning/gan/images/gan_diagram.svg

Aufbau des Generative Inpainting Netzwerks

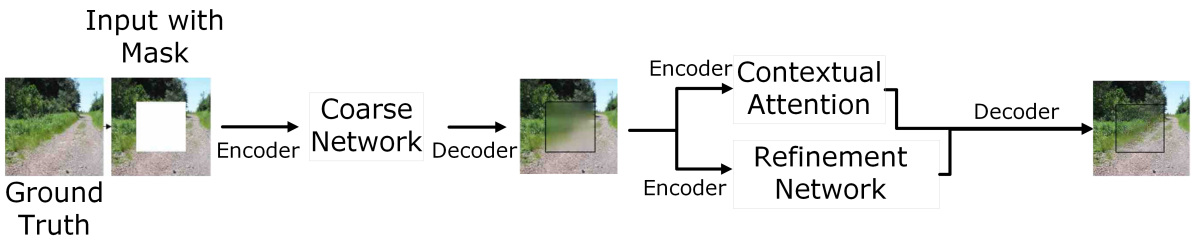

Abbildung 7.10: Generative Inpainting Netzwerk

Das Netzwerk, welches man in Abb. 7.10 sehen kann, besteht aus 2 Encoder-Decoder Netzwerken. Dabei nimmt der Encoder Daten, in unserem Fall ein Bild, und wandelt dieses in eine andere Form von Daten um. Bei einem CNN Netzwerk extrahiert der Encoder die relevanten Merkmale. Der Decoder wiederum generiert aus dem extrahierten Merkmal das Ergebnisbild. Generell gesagt, kann der Decoder auch ein anderes Datenformat als sein Eingabeformat produzieren. In dem Anwendungsfall von Bildwiederherstellung allerdings kommt ein Bild heraus, welches wieder alle Bereiche gefüllt hat. Der Teil des Netzwerkes, welcher die Bereiche füllt, ist ein Generator aus der GAN Architektur.

Das erste Netzwerk ist das sogenannte Coarse Netzwerk, welches eine grobe Füllung der fehlenden Bereiche vornimmt. Dabei fokussiert sich dieses Netzwerk, die groben Strukturen und Formen wiederherzustellen. Das Ergebnis dieses Netzwerkes ist ein wiederhergestelltes Bild, wobei alle aufgefüllten Bereiche unscharf und unrealistisch dargestellt sind.

Das zweite Netzwerk nimmt das grob wiederhergestellte Bild als Eingabe und kümmert sich um die feine Bearbeitung der fehlenden Teile. Hierbei werden feinere Texturen und Details eingefügt. Die Besonderheit dieses Netzwerks sind die 2 Encoder, wobei einer die Contextual Attention einführt. Diese beiden Encoder führen wiederum nur zu einem Decoder, welcher aus beiden extrahierten Merkmalen ein optimales Ergebnis generiert.

Diese Contextual Attention Schicht ermöglicht es dem Netzwerk, Informationen aus entfernten Bereichen des Bildes zu analysieren, um die fehlenden Teile realistischer zu rekonstruieren. Das heißt, diese Schicht identifiziert, welche Bereiche des Bildes ähnliche Strukturen und Muster aufweisen und nutzt diese Informationen, um die fehlenden Bereiche bestmöglich wiederherzustellen.

In den Netzwerken kommt außerdem eine spezielle Art von Convolution zum Einsatz, die Dilated Convolution. Diese ermöglicht es dem neuronalen Netz, den Blickwinkel auf das Bild zu vergrößern. Somit können Details von weiter weg gesehen

werden, ohne die Nähe zu den bereits vorhandenen Details zu verlieren. Das funktioniert so, dass in einen Filter Leerräume eingefügt werden und somit können größere Bereiche betrachtet werden. In Abb. 7.11[22] kann man auf der linken Seite sehen, dass nach jedem betrachteten Pixel, einer übersprungen wird.

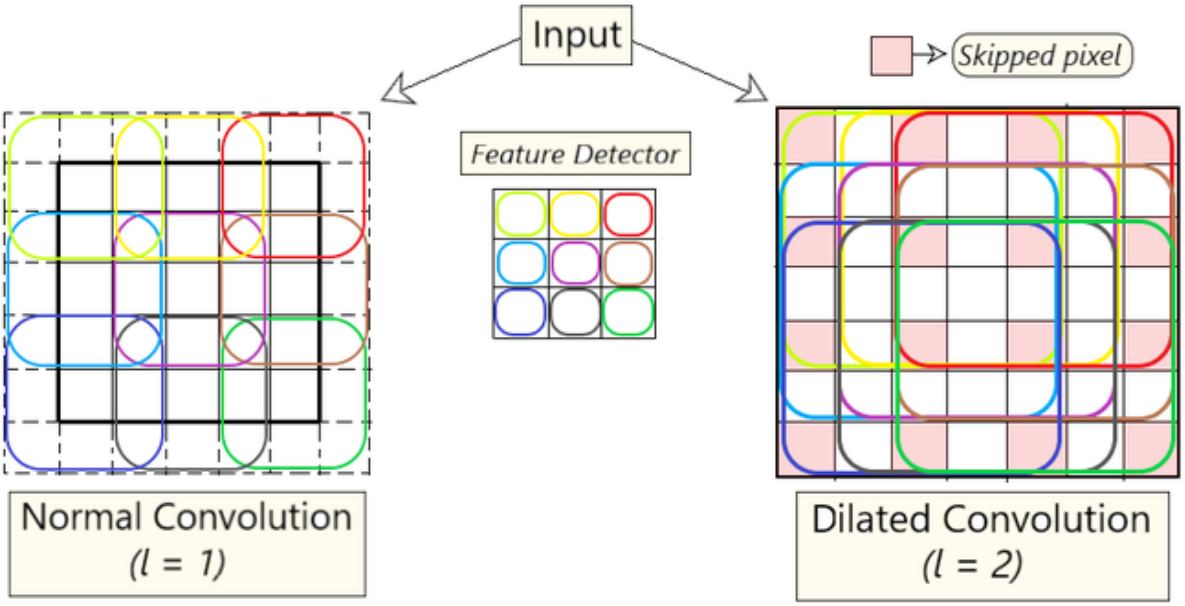

Abbildung 7.11: Dilated Convolution(left)

7.8.2 DeepFillV2

Bei dem verwendeten Modell von DeepFillV2 handelt es sich wieder um eine PyTorch Implementierung[23] der Originalen TensorFlow Implementierung[16]. Diese Implementierung basiert auf dem Paper ”Free-Form Image Inpainting with Gated Convolution”von Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu und Thomas Huang[9]. Das DeepFillV2 Netzwerk verwendet, wie das Generative Inpainting Netzwerk, die Contextual Attention Schicht und führt die Gated Convolution in Kombination mit Dilated Convolution ein.

Das Problem der normalen Convolution liegt darin, dass diese alle Pixel gleich behandelt, unabhängig, ob sie zu einem fehlenden oder einem gültigen Bereich gehören. Die Gated Convolution löst dieses Problem, indem es einen Auswahl-Mechanismus enthält, welcher entscheidet, ob die Pixel für die Wiederherstellung relevant sind oder nicht. Dies funktioniert so, dass auf die herkömmliche Convolution ein Gating-Mechanismus folgt, welcher entscheidet, welche Pixel an welcher Stelle verwendet werden sollen.

[22] https://www.geeksforgeeks.org/dilated-convolution/
[23] https://github.com/nipponjo/deepfillv2-pytorch/tree/master

In Abb. 7.12(vlg.[24]) ist ein Beispiel, wie Gated Convolution ablaufen kann. Die Gate-Matrix enthält Werte von 0 bis 1, wobei diese angeben, wie wichtige diese Pixel sind.

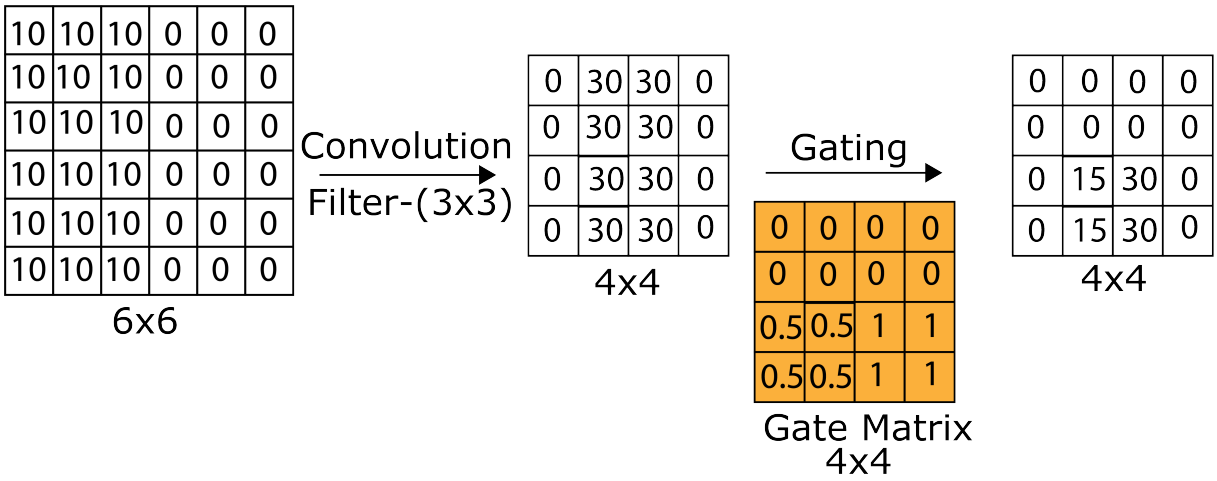

Abbildung 7.12: Gated Convolution

Aufbau des DeepFillv2 Netzwerks

Das DeepFillv2 Netzwerk ist ähnlich wie das Generative Inpainting Netzwerk aufgebaut (siehe Abb. 7.10), es umfasst auch 2 Encoder-Decoder Netzwerke. Wobei das erste wieder eine grobe Füllung durchführt, und das zweite die Contextual Attention Schicht verwendet und aus 2 Encoder und einem Decoder besteht. Der Unterschied ist, wie bereits angesprochen, dass statt der Dilated Convolution, Gated Dilated Convolution verwendet wird.

7.8.3 MISF

Das dritte verwendete Deep-Learning Netzwerk heißt MISF, und stammt aus dem Paper ”MISF: Multi-level Interactive Siamese Filtering for High-Fidelity Image Inpainting”von Xiaoguang Li, Qing Guo, Di Lin, Ping Li, Wei Feng und Song Wang[4]. In dem Modell gibt es 2 Netzwerke, welche miteinander kommunizieren, um ein optimales Ergebnis zu produzieren. Dabei gibt es den Kernel Prediction Branch(KPB) und den Semantic & Image Filtering Branch(SIFB). Diese beiden Bereiche ermöglichen es, sowohl grobe Strukturen zu erkennen, als auch feine Details und Texturen in die Wiederherstellung mit einzubauen.

- **KPB:** Dieser Teil des Netzwerkes konzentriert sich darauf, die Muster, Formen und Farben zu erkennen. Dieser Teil erkennt die fehlenden Teile des Bildes und entscheidet, wie diese gefüllt werden sollen.
- **SIFB**: Dieser Teil nimmt die Muster und Formen von KPB und kümmert sich darum, die fehlenden Bereiche zu füllen. Der SIFB kümmert sich auch darum,

[24] https://www.youtube.com/watch?v=H6ObEPfW6aw

dass die fehlenden Teile sich bestmöglich in den Rest des Bildes fügen, und kaum erkennbar ist, wo das Bild repariert worden ist.

Das, was das Netz aber so gut macht, ist die ständige Zusammenarbeit und Kommunikation dieser beiden Netze. Das SIFB Netz bekommt die Formen und Muster von KPB und versucht diese einzusetzen. Wenn das nicht optimal funktioniert, passt der SIFB diese Informationen an und teilt das dem KPB mit. Dank dieser interaktiven Zusammenarbeit und der ständigen Anpassung ist MISF in der Lage, realistische Ergebnisse zu erzielen.

7.9 Datasets

Wie bereits erwähnt worden ist, sind vortrainierte Modelle verwendet worden mit den in Abschnitt 7.8 beschriebenen Netzwerken. Diese Netzwerke müssen aber auch trainiert werden und da kommen Datasets ins Spiel. In dieser Arbeit sind drei vortrainierte Modell mit drei verschiedenen Datasets verwendet worden.

- **ImageNet:** ImageNet ist ein Dataset, welches über 14 Millionen Bilder aus mehr als 1000 Kategorien enthält. ImageNet ist also ein Dataset welches für Landschaften, Gesichter, Tiere usw. Bilder zum Trainieren eines Modells bietet. Die Größe der Bilder beträgt 224 × 224 Pixel. In dieser Arbeit ist ein Modell des Generative Inpainting Netzwerks, trainiert mit diesem Dataset, verwendet worden.
- **Places2:** Places2 enthält über 10 Millionen Bilder mit mehr als 400 einzigartigen Schauplätzen. Dieses Dataset beschränkt sich also nur auf Landschaften und die Bilder sind 256 × 265 Pixel groß. In dieser Arbeit ist ein Modell des DeepFillv2 Netzwerks, trainiert auf dem Places2 Dataset, verwendet worden.
- **CelebA:** CelebA enthält über 200.000 Bilder mit Gesichtern von über zehntausend verschiedenen Prominenten. Dieses Dataset spezialisiert sich auf die Erkennung und Wiederherstellung von Gesichtern. Die Bilder sind in einem Format von 178 × 218 Pixel vorhanden und in dieser Arbeit ist ein Modell des MISF Netzwerks, trainiert auf dem CelebA Dataset, verwendet worden.

8 Implementierung

8.1 Jetson Nano

Um die Implementierung auf dem Nvidia Jetson Nano Entwicklerkit umzusetzen, musste zuerst das passende Betriebssystem, welches von Nvidia bereitgestellt wird, auf dem Entwicklerkit installiert werden. In dieser Arbeit wurde die Version Jetson Linux 32.7.4 mit JetPack 4.6.4 gewählt. Es wurde auch Python 3.8.0 installiert, standardmäßig ist nur Python 3.6.9 installiert. Die Software Balena Etcher, welche in Abschnitt 7.5.1 beschrieben ist, wurde zum Einsatz gebracht, um das Betriebssystem auf eine SD-Karte zu schreiben. Danach musste das Betriebssystem eingerichtet werden. Dabei wurde die grafische Variante gewählt. Der erste Schritt ist es, die Nutzungsbedienungen zu akzeptieren und die Systemsprache zu wählen. Da diese Arbeit in Deutsch gehalten ist, wurde auch das Betriebssystem auf Deutsch installiert. Anschließend musste ein Benutzer erstellt, ein Computername gewählt und die Größe der Partition angegeben werden.

Folgende Bibliotheken sind bei Jetpack bereits installiert:

- CUDA 10.2.300
- cuDNN 8.2.1.32
- TensorRT 8.2.1.8

Im nächsten Schritt wurde OpenCV 4.8.0 mit Support für CUDA installiert. Dazu wurde eine Anleitung der Website "Q-Engineering[1]" verwendet.
In Abb. 8.1 kann man einen Screenshot des Terminals von dem Nvidia Jetson Nano sehen, wo mehrere Versionen aufgelistet sind.

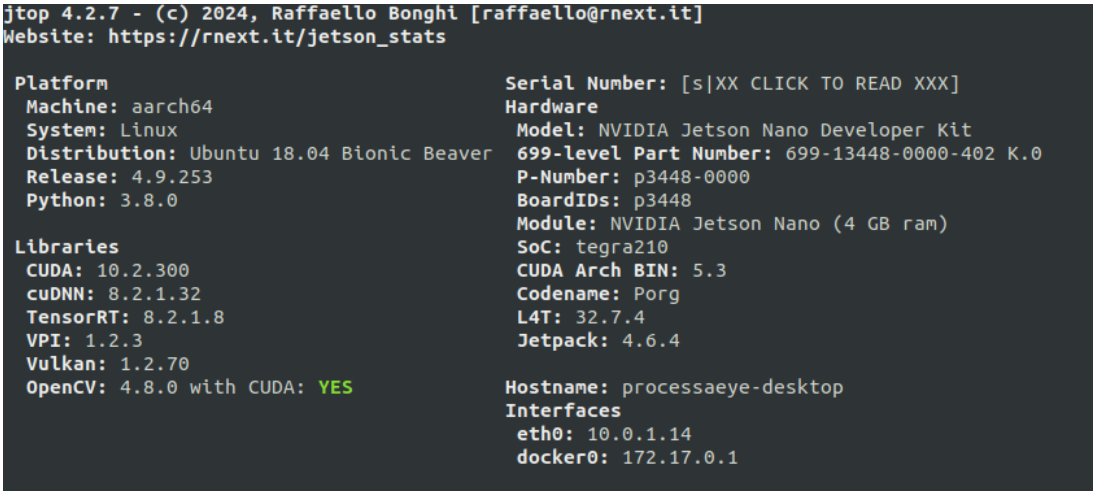

Abbildung 8.1: Versionen Jetson Nano

[1] https://qengineering.eu/install-opencv-on-jetson-nano.html

Nach der erfolgreichen Installation des Betriebssystems und den notwendigen Bibliotheken mussten mehrere Python-Bibliotheken installiert werden. Folgende Bibliotheken installiert:

- matplotlib
- numpy
- opencv_python
- Pillow
- PyQt5
- sip
- pyvirtualcam
- PyYAML
- torch
- torchvision
- scikit-image

Nachdem die Python-Bibliotheken erfolgreich installiert waren, konnte die Anwendung ausgeführt werden.

8.2 Deep-Learning Modelle

Nachfolgend sind die verwendeten Deep-Learning-Modelle aufgelistet und beschrieben, welche für die Umsetzung des technischen Teils dieser Arbeit verwendet und wie diese implementiert wurden.

Alle Algorithmen, also klassische Bildverarbeitungsalgorithmen und Deep-Learning Algorithmen, in dieser Arbeit implementieren eine abstrakte Klasse, wie man in Listing 8.1 sehen kann. Die Kernkomponente dieser Klasse ist die Inpaint Methode, welche immer die Maske und das Bild übergeben bekommt, und ein Numpy-Array als Ergebnis zurückliefert.

```
class IInpaintingAlgorithmen(ABC):

    def __init__(self, is_deep_learning=False):
        super().__init__()
        self.is_deep_learning = is_deep_learning

    @abstractmethod
    def inpaint(self, image, mask) -> np.ndarray:
        pass
```

Listing 8.1: IInpaintingAlgorithmen Interface

8.2.1 Generative Inpainting

Im Rahmen der Implementierung ist das Github Repository[15] als Grundlage verwendet worden. Von dieser Quelle stammen auch das vortrainierte Modell sowie sämtliche Netzwerk Implementierungen.

Um die Ipaint Methode zu implementieren, muss zuerst das Modell in einem Konstruktor geladen werden. Dafür wird, wie man in Listing 8.2 sehen kann, ein Pfad zu einer Config-Datei, sowie der Pfad, wo das Modell liegt, angegeben. Danach wird eine *load_config* Methode aufgerufen, wobei mithilfe von PyYaml die config.yaml Datei eingelesen wird. Nachdem das Modell geladen ist, wird noch die *check_cuda* Methode aufgerufen, welche überprüft, ob Cuda auf dem Gerät installiert ist und wenn dem so ist, wird das Modell auf die GPU geladen.

```
    script_dir = os.path.dirname(os.path.realpath(__file__))

    config_path = os.path.join(script_dir, 'config', 'config.yaml')
    self.checkpoint_path = os.path.join(script_dir, 'model')

    self.load_config(config_path)
    self.check_cuda()
```

Listing 8.2: Laden des Generative Inpainting Modells

In Listing 8.3 kann man die Logik des Wiederherstellungsprozesses sehen. Zuerst werden die beiden Bilder von dem übergebenen Pfad geladen, verkleinert oder vergrößert und in Tensors umgewandelt. Danach wird überprüft, ob auf dem Gerät Cuda installiert ist und wenn dem so ist, werden die Tensors auf die GPU verschoben.

Nachdem das Bild und die Maske im richtigen Format sind, werden diese dem Modell übergeben und ein Ergebnis wird erzeugt. Das Bild wird dann aus den Bereichen, wo es wiederhergestellt worden ist und den Bereichen vom originalen Bild, wo es nicht wiederhergestellt worden ist, zusammengesetzt.

Zum Schluss wird noch der Ergebnistensor wieder auf ein NumPy-Array umgewandelt, indem der Tensor wieder auf die CPU kommt, die Reihenfolge der Dimensionen angepasst wird, sodass die Dimension mit den 3 RBG Kanäle wieder zum Schluss ist und der Tensor wird auf ein NumPy-Array umgewandelt. Außerdem werden die Pixelwerte wieder auf 255 hochskaliert.

```
x = default_loader(image)
mask = default_loader(mask)

width, height = x.size
mask = transforms.Resize([width, height])(mask)

x = transforms.ToTensor()(x)
mask = transforms.ToTensor()(mask)

if self.cuda:
    x = x.cuda()
    mask = mask.cuda()

with torch.no_grad():
    _, x2, _ = self.netG(x, mask)
    inpainted_result = x2 * mask + x * (1. - mask)

inpainted_result_denorm = inpainted_result * 0.5 + 0.5
inpainted_result_np = inpainted_result_denorm.squeeze().cpu().numpy().transpose(1, 2, 0)
inpainted_result_np = (inpainted_result_np * 255.0).clip(0, 255).astype(np.uint8)

return inpainted_result_np
```

Listing 8.3: Inpaint Logik des Generative Inpainting Modells

8.2.2 DeepFillV2

Im Rahmen der Implementierung ist das Github Repository[23] als Grundlage verwendet worden. Von dieser Quelle stammt auch das vortrainierte Modell sowie sämtliche Netzwerk Implementierungen.

Wie bei dem ersten Deep-Learning-Algorithmus muss zuerst das Modell geladen

werden. In Listing 8.4 kann man sehen, dass zuerst ein Objekt der Inpainter Klasse erstellt wird, und dann der Pfad zu der model.yaml Datei gesetzt wird. In dieser Datei befinden sich die absoluten Pfade zu allen Modellen, welche man laden möchte, in diesem Fall ist es lediglich das Places2 Pytorch Modell. Um die Modelle dann endgültig zu laden, wird eine Methode der Inpainter Klasse aufgerufen, welche die yaml Datei einliest, dann überprüft, ob die Modell-Dateien an den angegebenen Pfaden vorhanden sind, und schlussendlich die Dateien lädt.

```
self.inpainter = Inpainter()
script_dir = os.path.dirname(os.path.realpath(__file__))
self.model_path = os.path.join(script_dir, 'model', 'models.yaml')
self.inpainter.load_models(self.model_path)
```

Listing 8.4: Laden des DeepFillv2 Modells

Die Methode, welche man in Listing 8.5 sehen kann, implementiert die Wiederherstellungslogik dieses Modells. Zuerst werden beide Bilder mithilfe der Pillow Bibliothek geöffnet. Falls die Maske ein anderes Format als das Bild hat, wird diese auf das Bild angepasst und beide Bilder werden in Tensors umgewandelt. Da diese Logik mehrere Modelle erlaubt, wird die Liste an Modellnamen iteriert und für jedes Modell wird die *infer_deepfill* Methode aufgerufen, welche die Bilder in das richtige Format bringt und danach dem Generator übergibt.

Der Generator liefert zwei Ergebnisse, deswegen wird das Output Array auch noch iteriert, wobei das Ergebnis vorher noch in ein NumPy-Array umgewandelt wird, bevor es an eine Map angehängt wird. Diese Map an Ergebnissen, wird dann wiederum in eine andere Map gegeben, wobei diese den Modellnamen als Key enthält, und dann als Value die Liste der Ergebnisse. Diese Map wird dann noch an ein *response_data* Array gehängt, welches schlussendlich dann zurückgegeben wird.

Diese Inpaint Methode ist innerhalb der Inpainter Klasse, aber nicht in der DeepFillV2 Klasse, welche die abstrakte Klasse implementiert. In dieser Klasse beziehungsweise, der Inpaint Methode, wird das erste Ergebnis des ersten Modells genommen, da in unserem Fall nur ein Modell und ein Ergebnis vorhanden sind. Das heißt, es wird die Inpaint Methode der Inpainter Klasse aufgerufen, welche man in Listing 8.5 sehen kann, und einfach der erste Eintrag der Liste, beziehungsweise auch der erste Eintrag der Ergebnisliste, als Ergebnis verwendet.

```
    image_pil = Image.open(image).convert('RGB')
    width, height = image_pil.size
    transform = T.Compose([
        T.Resize((height, width)),
        T.ToTensor()
    ])
    image = transform(image_pil)
    mask = transform(Image.open(mask))

    response_data = []
    project_dir = os.path.dirname(os.path.dirname(os.path.realpath(__file__)))

    for idx_model, model_name in enumerate(req_models):
        model_output_list = []
        outputs = infer_deepfill(
            self.loaded_models[model_name],
            image.to(self.device),
            mask.to(self.device),
            return_vals=return_vals
        )
        for idx_out, output in enumerate(outputs):
            output_np = np.array(output)

            model_output_list.append({
                'name': return_vals[idx_out],
                'data': output_np
            })
        model_output_dict = {
            'name': model_name,
            'output': model_output_list
        }
        response_data.append(model_output_dict)

    return response_data
```

Listing 8.5: Inpaint Logik des DeepFillv2 Modells

8.2.3 MISF

Im Rahmen der Implementierung ist das Github Repository[2] als Grundlage verwendet worden. Von dieser Quelle stammt auch das vortrainierte Modell sowie sämtliche Netzwerk Implementierungen.

Wie auch bei den beiden Algorithmen zuvor, muss das Modell zuerst geladen werden, bevor man es verwenden kann. Diesen Ablauf kann man in Listing 8.6 sehen. Zuerst wird wieder der Pfad für das Modell und die Config-Datei festgelegt und danach wird die *load_model* Methode aufgerufen. In dieser Methode wird ein Objekt der Klasse InpaintingModel erstellt, welche den Generator und den Discriminator enthält. Danach wird das Generator-Netz geladen und dem gerade erstellten Objekt der InpaintingModel Klasse als Membervariable gesetzt.

```
self.config = Config(os.path.join(self.script_path, 'model', 'config.yml'))
self.model_path = os.path.join(self.script_path, 'model', 'celebA_InpaintingModel_gen.pth')
self.load_model()

def load_model(self):
    self.inpaint_model = InpaintingModel(self.config)
    data = torch.load(self.model_path, map_location=self.device)
    self.inpaint_model.generator.load_state_dict(data['generator'])
```

Listing 8.6: Laden des MISF Modells

Wie man in Listing 8.7 sehen kann, werden auch hier, ähnlich wie bei den Modellen zuvor, zuerst die Bilder geöffnet, die Maske auf die Größe des Bildes gebracht und beide Bilder in Tensors umgewandelt. Die *unqueeze* Methode fügt eine Dimension zu den beiden Tensors hinzu, womit quasi ein Batch erstellt wird. Dabei gibt die neue vierte Dimension an, wie viele Bilder sich in dem Tensor befinden, was in diesem Fall nur eins ist. Danach werden Bild und Maske wieder auf die GPU geladen und in den Generator übergeben. Bei diesem Modell ist es aber außergewöhnlich, da der Generator nur einen Eingabeparameter hat, welcher das Bild mit aufgetragener Maske ist. Die beiden Bilder werden vor dem Wiederherstellungs-Durchlauf miteinander multipliziert. Da die Maske an den maskierten Stellen 1 ist, wird diese umgedreht, mit der *1 - mask_tensor Zeile*. Das maskierte Bild enthält also keine Bildinformationen an den Stellen der Maske mehr, das heißt, dort sind die Pixelwerte einfach 0. Um aber die fehlenden Teile nicht mit Teilen des Bildes zu verwechseln, welche einfach schwarz sind, wird das maskierte Bild noch mit der Maske verknüpft. Dadurch kann das Modell zwischen den fehlenden Bereichen und den komplett schwarzen Bereichen unterscheiden. Sobald der Vorgang abgeschlossen ist, wird das Bild wieder auf die CPU gegeben und in ein NumPy-Array umgewandelt, welches schlussendlich dann zurückgegeben wird.

[2] https://github.com/tsingqguo/misf

```
    img_pil = Image.open(image)
    mask_pil = Image.open(mask)
    width, height = img_pil.size

    transform = T.Compose([
        T.Resize((height, width)),
        T.ToTensor()
    ])
    img_tensor = transform(img_pil)
    mask_tensor = transform(mask_pil)
    img_tensor = img_tensor[:3].unsqueeze(0)
    mask_tensor = mask_tensor[0:1].unsqueeze(0)

    if torch.cuda.is_available():
        img_tensor = img_tensor.half().cuda()
        mask_tensor = mask_tensor.half().cuda()

    with torch.no_grad():
        img_masked = img_tensor * (1 - mask_tensor)
        input = torch.cat((img_masked, mask_tensor), dim=1)
        output_tensor = self.inpaint_model.generator(input)
    output_np = (output_tensor[0] * 255.0).clamp(0, 255).detach().cpu().numpy().transpose(1, 2,
    ↪  0).astype(np.uint8)
    return output_np
```

Listing 8.7: Inpaint Logik des MISF Modells

8.3 Klassische Bildverarbeitungsalgorithmen

Nachfolgend sind die verwendeten klassischen Bildverarbeitungsalgorithmen aufgelistet und beschrieben, welche für die Umsetzung des technischen Teils dieser Arbeit verwendet und wie diese implementiert wurden.

8.3.1 OpenCV Telea

Im Rahmen der Implementierung des Telea-Algorithmus wurde die dafür entwickelte Bibliothek von OpenCV verwendet. Wie man in Listing 8.8 sehen kann, ist es ein einfacher Prozess diesen Algorithmus zu integrieren.

```python
import cv2
import numpy as np
from utils.IInpaintingAlgorithmen import IInpaintingAlgorithmen


class TeleaInpainter(IInpaintingAlgorithmen):
    def __init__(self):
        super().__init__()

    def inpaint(self, image: str, mask: str) -> np.ndarray:
        mask = mask.convert('L')
        mask = mask.resize(image.size)

        image_np = np.array(image)
        mask_np = np.array(mask).astype('uint8')

        inpainted_image_np = cv2.inpaint(image_np, mask_np, inpaintRadius=3,
        ↪  flags=cv2.INPAINT_TELEA)

        return inpainted_image_np
```

Listing 8.8: Implementierung OpenCV Telea

Dabei werden das Bild, welches repariert werden soll, und die dazugehörige Maske als Parameter übergeben. Die inpaint Methode selbst gibt ein NumPy Array zurück, welches in der Benutzeroberfläche mithilfe einer Pixmap angezeigt werden kann. Die Maske wird auf Graustufen konvertiert, damit OpenCV sie korrekt verarbeiten kann. Anschließend wird die Größe der Maske an die des Bildes angepasst, um sicherzustellen, dass sie genau übereinstimmen. Das Bild und die Maske werden dann in NumPy-Arrays und die Maske in uint8 konvertiert, was für die weitere Verarbeitung mit OpenCV erforderlich ist. Am Ende der Methode wird die cv2.inpaint Methode aufgerufen und als Parameter wird übergeben, dass der Telea Algorithmus verwendet werden soll.

8.3.2 OpenCV NS

Im Rahmen der Implementierung des Telea-Algorithmus wurde die dafür entwickelte Bibliothek von OpenCV verwendet. Wie man in Listing 8.9 sehen kann, ist es ein einfacher Prozess diesen Algorithmus zu integrieren.

```
import cv2
import numpy as np
from utils.IInpaintingAlgorithmen import IInpaintingAlgorithmen


class NSInpainter(IInpaintingAlgorithmen):
    def __init__(self):
        super().__init__()

    def inpaint(self, image: str, mask: str) -> np.ndarray:

        mask = mask.convert('L')
        mask = mask.resize(image.size)

        image_np = np.array(image)
        mask_np = np.array(mask).astype('uint8')

        inpainted_image_np = cv2.inpaint(image_np, mask_np, inpaintRadius=3,
        ↪  flags=cv2.INPAINT_NS)

        return inpainted_image_np
```

Listing 8.9: Implementierung OpenCV NS

Dabei werden das Bild, welches repariert werden soll, und die dazugehörige Maske als Parameter übergeben. Die inpaint Methode selbst gibt ein NumPy Array zurück, welches in der Benutzeroberfläche mithilfe einer Pixmap angezeigt werden kann. Die Maske wird auf Graustufen konvertiert, damit OpenCV sie korrekt verarbeiten kann. Anschließend wird die Größe der Maske an die des Bildes angepasst, um sicherzustellen, dass sie genau übereinstimmen. Das Bild und die Maske werden dann in NumPy-Arrays und die Maske in uint8 konvertiert, was für die weitere Verarbeitung mit OpenCV erforderlich ist. Am Ende der Methode wird die cv2.inpaint Methode aufgerufen und als Parameter wird übergeben, dass der NS Algorithmus verwendet werden soll.

8.3.3 Skimage

Im Rahmen der Implementierung von Skimage wurde die dafür entwickelte Bibliothek "scikit-image" verwendet. Wie man in Listing 8.10 sehen kann, ist es ein einfacher Prozess diesen Algorithmus zu integrieren.

```
import numpy as np
from skimage.restoration import inpaint
from skimage.util import img_as_ubyte

from utils.IInpaintingAlgorithmen import IInpaintingAlgorithmen


class Skimage(IInpaintingAlgorithmen):
    def __init__(self):
        super().__init__()

        def inpaint(self, image: str, mask: str) -> np.ndarray:

        mask = mask.convert('L').resize(image.size)

        image_np = np.array(image)
        mask_np = np.array(mask)

        mask_np = np.where(mask_np > 1, 1, 0).astype(np.uint8)

        inpainted_image = inpaint.inpaint_biharmonic(image_np, mask_np, channel_axis=-1)

        return img_as_ubyte(inpainted_image)
```

Listing 8.10: Implementierung Skimage

Dabei werden das Bild, welches repariert werden soll, und die dazugehörige Maske als Parameter übergeben. Die inpaint Methode selbst gibt ein NumPy Array zurück, welches in der Benutzeroberfläche mithilfe einer Pixmap angezeigt werden kann. Die Maske wird auf Graustufen konvertiert, damit Skimage sie korrekt verarbeiten kann. Anschließend wird die Größe der Maske an die des Bildes angepasst, um sicherzustellen, dass sie genau übereinstimmen. Sowohl das Bild als auch die Maske werden in NumPy-Arrays umgewandelt. Für das Bild ist dies erforderlich, um es mit der inpaint-Funktion von scikit-image zu bearbeiten. Die Maske wird in eine binäre Maske konvertiert, dabei werden alle Werte der Maske, die größer als eins sind, auf eins gesetzt und alle anderen bleiben auf null. Das ist notwendig, da die Funktion von Skimage eine binäre Maske erwartet. Die inpaint_biharmonic Funktion wird dann aufgerufen, um das Bild basierend auf der biharmonischen Gleichung zu reparieren. Das Ergebnis ist ein repariertes Bild, das als NumPy Array zurückgegeben wird. Schließlich wird dieses Array mit der Funktion img_as_ubyte in den 8-Bit Unsigned Byte Typ konvertiert, was für die Anzeige in der Benutzeroberfläche erforderlich ist.

8.4 Vergleicher der verschiedener Bildwiederherstellungs-ansätze

8.4.1 Bewertungskriterien

Um die Algorithmen zu bewerten, ist eine Bewertungsskala bestehend aus 3 Kriterien entworfen worden. Dabei ist neben der Zeit und der Genauigkeit auch noch eine menschliche Bewertung hinzugefügt worden. Das hat den Grund, dass, auch wenn sich die Genauigkeiten nicht wirklich unterscheiden, man dennoch Unterschiede in der Qualität und Natürlichkeit des Ergebnisbildes erkennen kann.

8.4.1.1 Frobeniusnorm

Die Frobeniusnorm[3] ist eine Matrix-Norm, mit welcher die „Größe" einer Matrix berechnet werden kann. Dabei wird die Quadratwurzel der Summe der absoluten Quadrate aller Elemente aus der Matrix gezogen.

Mit der Formel ausgedrückt:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

wobei m und n die Anzahl der Zeilen und Spalten von A bezeichnen.

Mit der Frobeniusnorm kann man die "Größe" einer Matrix bestimmen. Dies kann verwendet werden, um die Differenz zwischen zwei Bildern zu berechnen, also die Differenz zwischen den Pixelwerten. Ein Bild wird als 3-dimensionales NumPy-Array dargestellt, wobei die dritte Dimension die 3 RGB Kanäle beschreibt und man somit drei Matrizen hat. Nun kann man für die drei Matrizen jeweils die Frobeniusnorm berechnen und summieren, somit erhält man die Abweichung der beiden Bilder und man kann bewerten, wie gut ein Bild anhand der Pixelwerte rekonstruiert worden ist.

Berechnung der Frobeniusnorm in Python

Zuerst wird, wie man in Listing 8.11 sehen kann, sowohl das Ausgangsbild als auch das wiederhergestellte Bild mit der Maske multipliziert. Da die Maske binär ist und an allen Stellen, wo das Bild nicht wiederhergestellt worden ist, null ist, bleiben bei den beiden Bilder auch nur jene Bereiche über, welche relevant zu betrachten sind.

[3] https://mathworld.wolfram.com/FrobeniusNorm.html

```
masked_ground_truth = ground_truth * mask
masked_inpainted = inpainted * mask
```

Listing 8.11: Maskierung der Bilder zur Vorbereitung

Danach wird, wie man in Listing 8.12 sehen kann, die Differenz der beiden Bilder berechnet. Dafür werden zuerst beide Bilder auf einen 16 Bit großen Int Datentyp umgewandelt. Das hat den Grund, dass die Bilder in *uint8* gespeichert werden, also in einem 8 Bit großen Int Datentyp, welcher keine Minus werte zulässt. Würde man diesen für die Berechnung weiterhin verwenden, kann es zu einem Unterlauf kommen. Dies tritt auf, wenn man eine größere Zahl von einer kleineren subtrahiert, also zum Beispiel 10 - 20. Da der Datentyp keine Minus werte unterstützt, würde das Ergebnis -10 auf 246 umgewandelt werden und somit würden falsche Differenzwerte herauskommen.

Nachdem die beiden Arrays umgewandelt worden sind, wird die *np.abs* Methode aufgerufen, mit welcher man alle absoluten Werte herausbekommt, also den Betrag und somit werden alle Minus Zahlen, zu positiven und das Differenzarray wird wieder auf einen *uint8* Datentyp umgewandelt.

```
masked_ground_truth = masked_ground_truth.astype(np.int16)
masked_inpainted = masked_inpainted.astype(np.int16)
error_image = np.abs(masked_ground_truth - masked_inpainted)
error_image = error_image.astype(np.uint8)
```

Listing 8.12: Berechnung der Differenz der beiden Bilder.

Zum Schluss wird, wie man in Listing 8.13 sehen kann, die Frobeniusnorm berechnet. Dabei bietet Numpy eine Funktion, wobei mit dem *ord* Parameter angegeben werden kann, welche Matrixnorm berechnet wird. Weil diese Norm aber nur die Größe einer Matrix, also eines 2-dimensionalen Arrays berechnen kann, werden die 3 RBG Kanäle durchlaufen und die Frobeniusnormen werden summiert.

Danach wird dieser absolute Fehler noch relativiert, indem man durch die Anzahl an 1 Bit in der Maske dividiert, also quasi wie viele Pixel betrachtet wurden.

```
frobenius_error = sum(np.linalg.norm(error_image[:, :, channel], ord='fro') for channel in
↪  range(error_image.shape[2]))
normalized_error = frobenius_error / np.sum(mask)
```

Listing 8.13: Berechnung der Frobeniusnorm

8.4.1.2 Menschliches Auge

Durch die Frobeniusnorm kann man die Qualität des wiederhergestellten Bildes überprüfen, allerdings kann es für das menschliche Auge Unterschiede geben. Auch wenn die Fehler nahe zugleich sind, können mit dem menschlichen Auge durchaus Unterschiede erkannt werden und deshalb wird die Frobeniusnorm mit einer Bewertung des menschlichen Auges zusammengeschlossen, um die Qualität zu bewerten.

Dabei ist von dem Projektteam eine Skala von 1 bis 10 entworfen worden.

- 10-9: Das rekonstruierte Bild ist nahezu perfekt und unterscheidet sich kaum bis gar nicht vom Originalbild. Ohne Vorwissen über die Rekonstruktion, ist diese nicht zu erkennen und die wiederhergestellten Bereiche fügen sich perfekt in das Bild ein.
- 8-7: Das rekonstruierte Bild weist leichte Ansätze der Wiederherstellung auf, sodass diese auch ohne Vorwissen über die Rekonstruktion erkennbar ist. Trotzdem, ist die Rekonstruktion sehr gut erkennbar und die wiederhergestellten Bereiche integrieren sich gut in den Rest des Bildes.
- 6-5: Man kann klar die Rekonstruktion erkennen. Die Wiederherstellung ist angemessen, mit sichtbaren Unterschieden zum Originalbild. Das Originalbild bleibt erhalten, allerdings sind vor allem Details nicht wiedererkennbar.
- 4-3: Die rekonstruierten Bereiche sind klar erkennbar und die Qualität dieser ist schlecht. Teilweise sind Sachen wiederzuerkennen, aber nur grob.
- 2-1: Die Rekonstruktion weist wenig bis gar nichts von dem Original auf.

Um die Algorithmen entsprechend zu testen, sind 20 Bilder hergenommen worden, wobei Landschafts-, Personen-, Tier- und Städtebilder verwendet worden, um ein diverses Spektrum zu haben. Die Masken, welche verwendet werden, sind willkürliche Striche durch das Bild, und hierbei teilweise etwas dicker und teilweise etwas dünner.

8.4.1.3 Wiederherstellungszeit

Bei unserer Arbeit geht es aber auch vor allem um die Echtzeit Wiederherstellung von Bildern und das auf dem NVIDIA Jetson Nano. Deshalb ist auch die Wiederherstellungszeit essenziell, um die Algorithmen zu bewerten. Dabei ist jeder Algorithmus 30 Minuten lang mit demselben Video und derselben Maske getestet worden. Dann ist die Durchschnittszeit berechnet worden, welche pro Rekonstruktionsvorgang benötigt worden ist. Dabei sind alle Algorithmen auf allen 3 Auflösungen getestet worden.

Damit immer dasselbe Videomaterial verwendet werden kann, ist von Youtube[4] ein 30 Minuten langes Video heruntergeladen worden. Es ist darauf geachtet worden, dass dieses Video kein Copyright schützt. Mithilfe der OBS Virtual Cam ist das Video wie eine Kamera simuliert worden und ist so in die Anwendung eingebunden worden.

Umsetzung im Programm

Nachdem die virtuelle Kamera gestartet worden ist, ist auch der Rekonstruktionsprozess gestartet worden und nach jedem Durchlauf wird die Anzahl der Durchläufe erhöht, sowie die benötigte Zeit einer Liste angehängt, wie man in Listing 8.14 sehen kann. Aufgrund der Deep-Learningmodelle, welche beim ersten Durchlauf zuerst ihre Modelle laden müssen, ist dieser Durchlauf ignoriert worden, um den Durchschnitt nicht zu verfälschen.

```
self.num_execution += 1
elapsed_time = time.time() - self.start_time
if self.num_execution > 1:
    self.execution_times.append(elapsed_time)
```

Listing 8.14: Addieren der benötigen-Zeiten der Wiederherstellungen

Sollte die Zahl der Gesamtsekunden 1800 überschreiten, also 30 Minuten, so wird die Durchschnitts-zeit berechnet und zusammen mit der Anzahl an Durchläufen auf dem Bildschirm ausgegeben. Wenn die Gesamtsekunden noch unter der Grenze liegen, wird der nächste Durchlauf gestartet. Der Code dafür befindet sich in Listing 8.15.

```
if self.total_execution_time < 1800:
    self.do_inpaint_image()
else:
    average_time = sum(self.execution_times) / len(self.execution_times)
    self.time_label.setText(
        f"Average Inpainting Time (over {self.num_execution - 1} runs): {average_time:.4f}
        ↪  seconds")
    self.enable_ui()
```

Listing 8.15: Ausrechnen der Durchschnittszeit und anzeigen in der UI

[4] https://www.youtube.com/watch?v=mLwlGsRhNIU&list=PLGoWuvyH709vpTCVrjaJtaaFfite9U6u8&index=2

8.5 ProcessAEye Anwendung

Die ProcessAEye Anwendung ist eine Stand Alone Python Applikation und ist in dem Framework Pyqt5 geschrieben. Die Anwendung lässt sich in die Bereiche GUI Aufbau und die Bildrekonstruktion gliedern. Die APP wird mit allen trainierten Machine-Learning-Modellen ausgeliefert und es müssen keine zusätzlichen Dateien heruntergeladen werden. Sie kann auf allen Desktop-Geräten ausgeführt werden, solange alle benötigten Python Bibliotheken installiert sind und eine Kamera vorhanden ist.

8.5.1 GUI Aufbau

Die GUI besteht aus zwei Benutzeroberflächen, ein Fenster, um in Echtzeit ein Kamerabild zu rekonstruieren (siehe Abb. 8.2) und ein Fenster, um ein einzelnes Bild hochzuladen (siehe Abb. 8.3) und wiederherzustellen. Beide Ansichten enthalten eine Auswahl des Algorithmus (siehe Abb. 8.2/8.3: I), eine Auswahl der Auflösung (siehe Abb. 8.2/8.3: II), einen Startknopf des Wiederherstellungsvorgangs (siehe Abb. 8.2/8.3: VII) und einen Knopf zum Entfernen der gezeichneten Maske (siehe Abb. 8.2/8.3: VI). Auf beiden Ansichten befindet sich das Ausgangsbild auf der linken Seite (siehe Abb. 8.2: IV / Abb. 8.3: IV) und das Ergebnisbild auf rechten Seite (siehe Abb. 8.2: VIII / Abb. 8.3: X). Auf der Seite für die Rekonstruktion eines einzelnen Bildes binden sich zusätzlich noch ein Knopf zum Hochladen eines Bildes und ein Knopf zum Hochladen einer Maske (siehe Abb. 8.3: VIII&IX). Auf beiden Ansichten befindet sich ein Knopf (siehe Abb. 8.2: X / Abb. 8.3: XI), mit welchem man zur jeweils anderen Ansicht gelangt.

8.5.1.1 Maske zeichnen

In beiden Fenstern ist ein Zeichenmodul eingebaut, um auf dem Ausgangsbild den Bereich zu markieren, welcher rekonstruiert werden soll. Um die Maske (siehe Abb. 8.2/8.3: V) unabhängig vom Bild zu speichern, wird dafür eine eigene Pixmap verwendet. Eine Pixmap speichert Bilder in Form von Pixel in einer Bitmap und kann in Pyqt5 angezeigt werden [5].

Um die UX(User Experience) zu verbessern, gibt es einen Knopf, um die gezeichnete Maske wieder zu löschen und einen Regler, mit welchem man die Größe des Zeichenstiftes anpassen kann (siehe Abb. 8.2/8.3: VI&III).

[5] https://doc.qt.io/qtforpython-5/

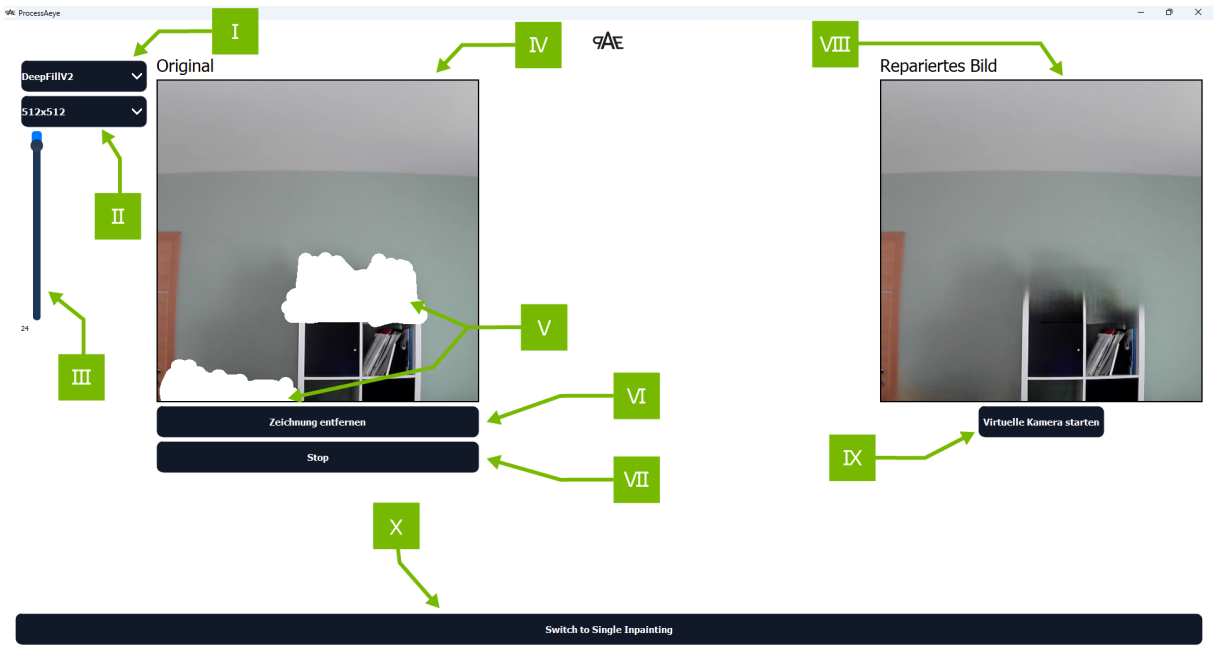

Abbildung 8.2: ProcessAEye Kamera Inpainting Ansicht

I	Algorithmusauswahl
II	Auflösungsauswahl
III	Stiftgrößenregler
IV	Ausgangsbild
V	gezeichnete Maske
VI	Knopf zum Entfernen der gezeichneten Maske
VII	Algorithmus Start/Stopp-Knopf
VIII	Wiederhergestelltes Bild
IX	Knopf zum Starten der virtuellen Kamera
X	Wechsel zu der Single Image Ansicht

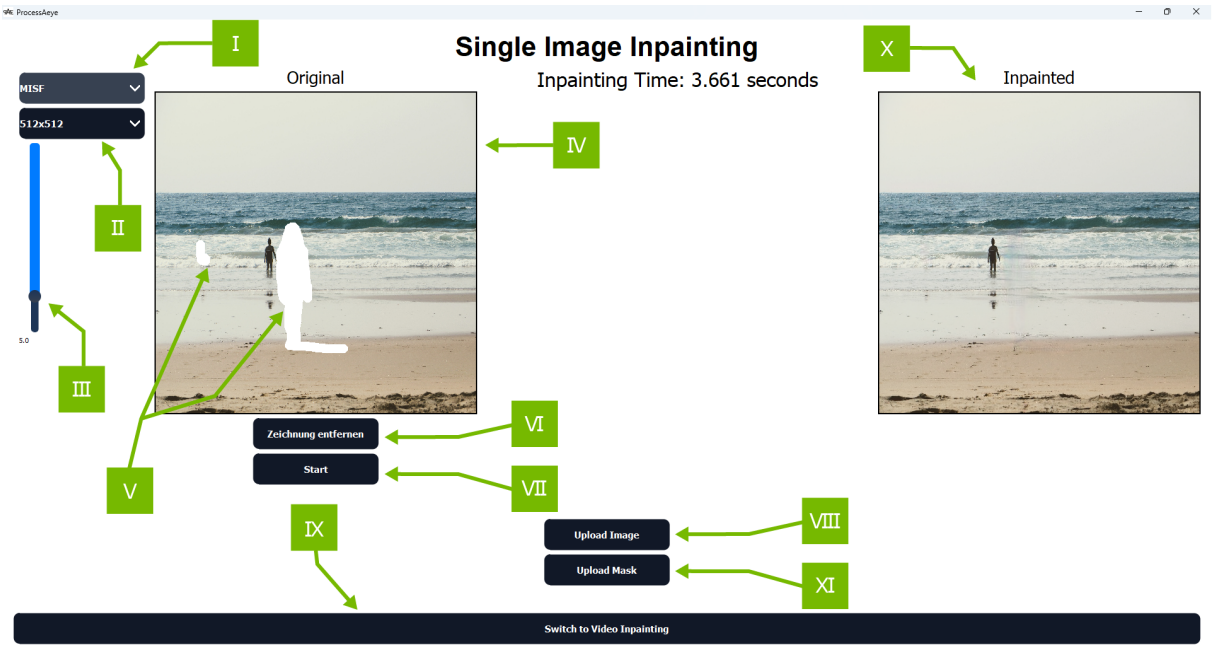

Abbildung 8.3: ProcessAEye Kamera Inpainting Ansicht

I Algorithmusauswahl

II Auflösungsauswahl

III Stiftgrößenregler

IV Ausgangsbild

V gezeichnete Maske

VI Knopf zum Entfernen der gezeichneten Maske

VII Algorithmus Start/Stopp-Knopf

VIII Knopf zum Hochladen des Bildes

IX Knopf zum Hochladen der Maske

X Wiederhergestelltes Bild

XI Wechsel zu der Kamera Inpaiting Ansicht

8.5.1.2 Algorithmus Auswahl

Damit man nicht händisch im Code den Algorithmus ändern muss, bietet die Anwendung ein Dropdown (siehe Abb. 8.2/8.3: I), in welchem zwischen allen Algorithmen gewählt werden kann. Damit es zu keinen Fehlern während der Laufzeit des Programmes kommen kann, ist das Dropdown deaktiviert, solange der Inpainting Vorgang läuft.

Umsetzung im Programm

Das Dropdown wird zuerst mit den Namen der Algorithmen als Schlüssel und einem Objekt des Algorithmus als Wert gefüllt, wie man in Listing 8.16 sehen kann.

```
self.inpaint_algorithms = {"OpenCV TELEA": TeleaInpainter(), "OpenCV NS": NSInpainter(),
↪  "Skimage": Skimage(),"DeepFillV2": DeepFillV2(), "MISF": Misf(),"Generative Inpainting":
↪  GenerativeInpainting()}
```

Listing 8.16: Befüllen des Dropdowns mit den Algorithmen

Die Anwendung funktioniert so, dass ein Exemplar eines Interfaces der aktuelle Algorithmus zugewiesen wird. Dieses Interfaces wird von allen Algorithmen implementiert und enthält eine Inpaint-Methode, wie man in Listing 8.17 sehen kann.

```
self.inpaint_algorithm: IInpaintingAlgorithmen = None
self.inpaint_algorithm = inpainting_algorithmen
```

Listing 8.17: Initialisieren und Deklarieren der Interfaceinstanz

Diese Methode wird dann von dem Objekt des Interfaces während des Inpainting-Vorgangs aufgerufen. Durch diese lose Kopplung können ganz einfach Algorithmen hinzugefügt oder entfernt werden und die Implementierung des Algorithmus ist für die Verwendung egal, da sich der Aufrufmechanismus nicht ändert.

8.5.1.3 Resolution Auswahl

Das Kamerabild in der Anwendung hat eine Auflösung von 512x512 Pixel. Für den Jetson Nano kann das Rekonstruieren eines Bildes mit dieser Auflösung mit einem Deep-Learning Netz mehrere Sekunden brauchen, und somit kann man nicht mehr von Echtzeit Bildrekonstruktion sprechen. Daher gibt es eine Auswahl der Auflösung (siehe Abb. 8.2/8.3: II), auf welche das Bild vor dem Wiederherstellungsprozess komprimiert wird. Nach dem Prozess wird das Bild wieder in die Originalgröße von 512x512 gebracht, um die Anzeige des Ausgangsbildes gleich zu der des Eingabebildes zu halten. Die Anwendung bietet die Auswahl zwischen 3 verschiedenen Auflösungen:

- 512x512
- 256x256
- 128x128

Implementierung im Programm

Wie man in 8.18 sehen kann, wird das Dropdown zuerst mit dem Namen der Auflösung als Schlüssel, und einem Tupel der Auflösung als Wert, gefüllt.

```
self.resolution_options = {"128x128": (128, 128), "256x256": (256, 256), "512x512": (512,
↪  512)}
```

Listing 8.18: Befüllen des Dropdowns mit den Auflösungen

Bevor der Wiederherstellungsvorgang beginnt, wird dann das Bild auf die gerade ausgewählte Auflösung verkleinert, wie man in Listing 8.19 sehen kann. Dafür wird die „resize" Methode der OpenCV Bibliothek verwendet - siehe OpenCV: 7.4.3.

```
dim = self.inpainting_width, self.inpainting_height
resizedImage = cv2.resize(image_arr, dim, interpolation=cv2.INTER_AREA)
```

Listing 8.19: Verkleinern des Eingabebildes

8.5.1.4 Kamera Inpainting

Die Kamera-Rekonstruktion-Ansicht ist die Hauptseite, auf welcher die Echtzeit-Wiederherstellung stattfindet. Dabei wird immer ein Bild der Kamera an einen Rekonstruktions-Thread übergeben und wird dann mit dem aktuellen Algorithmus wiederhergestellt. Sobald das Bild fertig wiederhergestellt ist, wird es angezeigt und das nächste Bild wird rekonstruiert.

Die Wiederherstellung beginnt, sobald der Startknopf (siehe Abb. 8.2: VII) gedrückt wird, danach ändert sich der Text des Knopfes auf Stopp. Während die Bilder rekonstruiert werden, sind beide Dropdowns deaktiviert, also kann weder der Algorithmus noch die Auflösung währenddessen geändert werden. Die Maske kann während des Rekonstruierens geändert werden und es wird immer die aktuelle Maske für die nächste Rekonstruktion verwendet.

Um der Echtzeit-Wiederherstellung einen Anwendungsfall zu geben, ist ein Knopf eingebunden worden, mit welchem eine virtuelle Kamera gestartet wird und auf dieser kann man das rekonstruierte Video sehen (siehe Abb. 8.2: IX). Dieses Resultat kann dann in diversen Anwendungen wie Teams verwendet werden.

8.5.1.5 Single Image Inpainting

Bei der Single Image Inpainting Ansicht handelt es sich um eine zusätzliche Ergänzung zu der Anwendung. Ursprünglich war nur die Kamera Rekonstruktionsseite geplant, allerdings hat sich das Testen so etwas schwerer herausgestellt und deswegen ist diese zusätzliche Ansicht implementiert worden. Auf dieser Ansicht kann man ein einzelnes Bild rekonstruieren. Hierbei sind 2 Knöpfe vorhanden, um ein Bild und eine Maske hochzuladen, welche rekonstruiert werden sollen. So ist es möglich, die Algorithmen anhand derselben Eingabebedingungen zu testen und die Ergebnisse besser vergleichen zu können.

8.5.2 Anpassung von PyQt Komponenten

Wie bereits in 6.3.3 erwähnt wurden die von PyQt bereitgestellten Komponenten noch mit eigenen Stylings angepasst.

Implementierung im Programm

PaeButton

PaeButton ist eine Ableitung von QPushButton und mit Stylings werden Farbe und Abrundungen angepasst, wie man in Listing 8.20 sehen kann. Diese Komponente dient dazu, verschiedene Aktionen zu starten oder zu stoppen.

```
class PaeButton(QPushButton):
    def __init__(self, parent=None, x=50, y=50, height=50):
        super().__init__(parent)
        self.x, self.y, self.height = x, y, height
        self.setMinimumSize(200, height)
        self.setGeometry(x, y, 200, height)
        self.setStyleSheet("""
            QPushButton {
                background-color: #111827;
                border-radius: 10px;
                color: #FFFFFF;
                font-size: 15px;
                text-align: center;
            }
        """)
```

Listing 8.20: Anpassung PaeButton

PaeSlider

PaeSlider ist eine Ableitung von QSlider und mit Stylings wurden die Farbe, Abrundungen und Größe angepasst, wie man in Listing 8.21 sehen kann. Diese Komponente dient dazu einzustellen, wie dick man eine Maske mit der Maus zeichnen möchte.

```
class PaeSlider(QSlider):
    def __init__(self, parent=None):
        super().__init__(Qt.Vertical, parent)
        self.setStyleSheet("""
            QSlider::groove:vertical {
                width: 10px;
                background: #1d3557;
                border-radius: 5px;
            }
        """)
```

Listing 8.21: Anpassung PaeSlider

PaeComboBox

PaeComboBox ist eine Ableitung von QComboBox und mit Stylings werden Farbe und Abrundungen angepasst, wie man in Listing 8.20 sehen kann. Diese Komponente dient dazu, in der Applikation verschiedene Algorithmen und Auflösungen auszuwählen.

```
class PaeComboBox(QComboBox):
    def __init__(self, parent=None, x=50, y=50, height=50):
        super().__init__(parent)
        self.x, self.y, self.height = x, y, height
        self.setGeometry(x, y, 200, height)
        self.setMinimumSize(200, height)
        self.setStyleSheet("""
            QComboBox {
                background-color: #111827;
                border: 1px solid transparent;
                border-radius: 10px;
                color: #FFFFFF;
                font-size: 15px;
                font-weight: 600;
                padding: .75rem 1.2rem;
                align-items: center;
            }
        """)
```

Listing 8.22: Anpassung PaeComboBox

PaeLabel

PaeLabel ist eine Ableitung von QLabel und mit Stylings wurden die Schriftfarbe und Schriftart angepasst, wie man in Listing 8.23 sehen kann. Diese Komponente dient dazu, Text in der Applikation anzuzeigen.

```
class PaeLabel(QLabel):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setStyleSheet("""
            QLabel {
            color: black;
            font-family: "Geist Mono Variable Medium";
            }
        """)
```

Listing 8.23: Anpassung PaeLabel

8.5.3 Bild Rekonstruktion

Der Ablauf der Rekonstruktion eines Bildes unterscheidet sich bei den beiden Ansichten nur in der Anzahl, wie oft der Vorgang hintereinander ausgeführt führt, sobald einmal der Startknopf gedrückt wird. Wie der Name verrät, passiert das bei der Single Inpainting Seite nur einmal und bei der Kamera Inpainting Seite so lange, bis der Knopf erneut zum Stoppen gedrückt wird.

Für die Implementierung des Prozesses sind einige Komponenten aus dem Pyqt5 Framework verwendet worden, welche zum besseren Verständnis nachfolgend kurz erklärt sind.

8.5.3.1 QThread

QThread[6] ist eine von dem PyQt-Framework bereitgestellte Klasse, um Multithreading Anwendungen zu ermöglichen. Ein Thread wird verwendet, damit das Programm mehrere Prozesse parallel voneinander ausführen kann, zum Beispiel wenn Daten von einer API geladen werden, wird dies in einem Thread erledigt, damit die UI nicht eingefroren wird. In diesem Projekt sind Threads verwendet worden, um den Rekonstruktionsprozess in einem eigenen Thread zu verwalten, damit die UI weiter bedienbar bleibt.

Die QThread Klasse bietet im Vergleich zu anderen Threading-Methoden in Python, wie dem eingebauten "threading" Modul, spezifische Vorteile, insbesondere in Zusammenarbeit mit GUI. Hierbei ist das einfache Aktualisieren der UI mittels Signalen ausschlaggebend.

8.5.3.2 PyqtSignal

Die PyQtSignal[7] Klasse ist ein zentraler Bestandteil des Qt-Frameworks und bietet einen einheitlichen Weg, die Benutzeroberfläche über Änderungen zu informieren, ohne dass die Komponenten direkt voneinander abhängen.
Der Mechanismus zum Ändern, beziehungsweise zum Benachrichtigen basiert auf dem Observer-Designmuster. Das Observer-Muster beschreibt die Vorgehensweise, eine Eins-zu-viele-Abhängigkeitsbeziehung zwischen Objekten zu erzeugen, sodass, wenn sich das Objekt ändert, alle seine Abhängigkeiten benachrichtigt werden.
Ein PyQt-Signal verwendet ein Signal, um das Objekt darzustellen, welches sich ändern kann. An dieses Signal können sich beliebig viele Slots hängen, wobei ein Slot meistens ein Python callable ist, also Code, welcher ausgeführt wird, sobald das Signal Änderungen meldet. Die Signal-Klasse bietet 3 Methoden, eine "connect(slot)", um sich mit einem Signal zu verbinden, eine "disconnect(slot)", um sich wieder vom

[6] https://doc.qt.io/qtforpython-6/PySide6/QtCore/QThread.html
[7] https://doc.qt.io/qtforpython-5/PySide2/QtCore/Signal.html

Signal zu trennen und eine "emit(*args)", um das Signal auszulösen und dabei beliebige Argumente an die verbundenen Slots zu übertragen.

8.5.3.3 QMutex und QWaitCondition

QMutex[8] und QWaitCondition[9] sind Klassen, welche zum Einsatz bei Thread-sicherer-Programmierung kommen. QMutex dient als Sperre, um den Zugriff auf geteilte Ressourcen zu regulieren, und stellt somit sicher, dass zu einem Zeitpunkt nur ein Thread auf die Ressource zugreifen kann. QWaitCondition wird verwendet, um Thread anzuhalten oder schlafen zu legen, bis eine Bedingung erfüllt worden ist. Man kann die schlafenden Threads dann mit "wakeAll()" oder mit "wakeOne()" wieder aufwecken und dann läuft der Thread wieder weiter.

8.5.3.4 Implementierung im Programm

```
if self.inpainting_thread is not None and self.inpainting_thread.isRunning():
    self.inpainting_thread.stop()
    self.inpaint_button.setText("Start")
    self.slider_vbox.enable_comobo_boxes()
    self.inpainting_thread = None
else:
    self.inpainting_thread = InpaintingThread(self.inpaint_algorithm, self.get_image,
    ↪  self.image_mutex,self.image_condition)
    self.inpainting_thread.start()
    self.inpaint_button.setText("Stop")
    self.inpainting_thread.finished_signal.connect(self.on_inpainting_finished)
    self.slider_vbox.disable_combo_boxes()
```

Listing 8.24: Starten beziehungsweise Stoppen des Inpainting Vorgangs

In Listing 8.24 wird zuerst überprüft, ob der Rekonstruktionsvorgang gerade läuft. Ist dem so, wird der Thread gestoppt, der Text wieder auf Start gestellt, die Dropdowns für die Algorithmus- und Auflösungsauswahl wieder aktiviert und der Thread auf None gesetzt.

Falls der Vorgang nicht läuft, wird ein Thread erstellt, welchem das aktuelle Exemplar des Algorithmus übergeben wird, die Methode, mit welcher das aktuelle Bild plus Maske geholt werden kann, und ein Mutex sowie eine Bedingungsvariable. Dann wird der Thread gestartet, der Text auf Stopp geändert, und die beiden Dropdowns werden deaktiviert, um Fehler während der Laufzeit zu vermeiden.

[8] https://doc.qt.io/qt-6/qmutex.html
[9] https://doc.qt.io/qt-6/qwaitcondition.html

```
def run(self):
    while not self.isInterruptionRequested():
        image_path, mask_path = self.image_getter()
        output = self.inpaint_algorithm.inpaint(image_path, mask_path)
```

Listing 8.25: Inpainting Thread

Wie man in Listing 8.25 sehen kann, läuft der Rekonstruktion-Thread, solange er nicht gestoppt wird und holt sich das Bild und die Maske für die Wiederherstellung. Danach wird von dem jeweiligen ausgewählten Algorithmus die Rekonstruktionsmethode aufgerufen und man bekommt das Ergebnis in Form eines NumPy Arrays.

Um die Benutzeroberfläche zu aktualisieren, also das Ergebnis in die Ergebnis Pixmap zu bekommen, wird ein PyQtSignal verwendet. Dafür wird zuerst ein Signal Objekt innerhalb des Threads erstellt, welches als Parameter ein NumPy Array hat (siehe Listing 8.26).

```
finished_signal = pyqtSignal(numpy.ndarray)
```

Listing 8.26: PyQtSignal Initialisierung

Danach wird, wie man im Listing 8.24 in der Startlogik sehen kann, an das Signal ein Slot gehängt, in diesem Fall eine Methode, welche das Ausgabe NumPy Array verwaltet, und in der Benutzeroberfläche anzeigt.
Im Rekonstruktions-Thread wird, nachdem der Rekonstruktionsprozess abgeschlossen ist, die Ausgabe an alle verbundenen Slots gesendet (siehe Listing 8.27).

```
self.finished_signal.emit(output)
self.image_condition.wait(self.image_mutex)
```

Listing 8.27: PyQtSignal Signal übermittlung

Nach dem Übermitteln des Signals wird der Rekonstruktions-Thread in den Warte zustand versetzt, wie man in Listing 8.27 sehen kann. Dies hat den Grund, dass der nächste Wiederherstellungsprozess erst starten soll, sobald das Bild fertig in der Benutzeroberfläche angezeigt wird. Denn bei OpenCV Telea und OpenCV NS ist der Vorgang des Wiederherstellens so schnell, dass bereits das nächste Signal gesendet wird, bevor das alte Signal fertig verarbeitet worden ist und somit können Fehler während der Laufzeit auftreten.

```
def on_inpainting_finished(self, output_np):
    self.image_mutex.lock()
    dim = display_width, display_height
    resized_image = cv2.resize(output_np, dim, interpolation=cv2.INTER_AREA)
    height, width, channel = resized_image.shape
    bytes_per_line = 3 * width
    qt_image = QImage(resized_image.data, width, height, bytes_per_line,
    ↪  QImage.Format_RGB888)

    self.inpaint_image_label.setPixmap(QPixmap.fromImage(qt_image))

    self.image_mutex.unlock()
    self.image_condition.wakeAll()
```

Listing 8.28: Numpy Array empfangen und anzeigen

Die Methode, welche man im Listing 8.28 sehen kann, wird an das Signal gehängt, und sie verwaltet somit auch das NumPy Array mit dem Ergebnisbild. Dieser ganzer Code Block wird zuerst gesperrt, damit nicht mehrere Threads gleichzeitig auf diese Methode zugreifen können. Danach wird das NumPy Array vergrößert und in ein QImage umgewandelt, welches dann wiederum in eine Pixmap umgewandelt wird, um in der Benutzeroberfläche angezeigt werden zu können. Nachdem dies geschehen ist, wird das Mutex wieder entsperrt und der Rekonstruktions-Thread aus dem Wartezustand geholt. Somit beginnt der nächste Zyklus des Rekonstruktionsprozesses.

8.5.4 PyVirtualCam

Wie in Abschnitt 7.4.8 erwähnt, wurde PyVirtualCam dazu verwendet, um eine virtuelle Kamera mit dem reparierten Ausgangsbild zu erzeugen.

Implementierung im Programm

Wie man in Listing 8.29 sehen kann, werden mehrere Methoden zur Initialisierung der virtuellen Kamera aufgerufen. Außerdem erfolgt eine Überprüfung, ob die nötigen Programme, wie in 7.4.8 beschrieben, vorhanden sind.

```
class VideoInpainting(BaseInpainting):
def __init__(self):
...
    self.runningVirtualCamera = False
    self.virtual_camera = None
    self.virtual_camera_available = False
    ...
    self.init_virtual_camera()
    self.init_start_virtual_camera_button()
    self.check_virtual_camera_availability()
    ...
    if self.virtual_camera_available:
        self.setLoadingScreen_virtualcamera()
```

Listing 8.29: Initialisierungsaufrufe von PyVirtualCam

Wie man in Listing 8.30 sehen kann, wird bei der Initialisierungsmethode versucht, eine virtuelle Kamera zu erzeugen, die Variable, welche speichert, ob eine virtuelle Kamera vorhanden ist, wird auf True gesetzt und der Name der benutzten Kamera wird ausgegeben. Sollte es nicht möglich sein, eine virtuelle Kamera zu starten, wird diese Variable auf False gesetzt und ein Fehler ausgegeben. Das Programm läuft weiter, da dieses Feature nicht zwingend notwendig ist.

```
def init_virtual_camera(self):
    try:
        self.virtual_camera = pyvirtualcam.Camera(width=512, height=512, fps=30)
        self.virtual_camera_available = True
        print(f'Benutzte Kamera: {self.virtual_camera.device}')
    except Exception as e:
        self.virtual_camera_available = False
        print(f"Fehler beim Kamera initalisieren: {e}")
```

Listing 8.30: Initialsierung von PyVirtualCam

Die Funktion init_start_virtual_camera_button initialisiert den Button, der es ermöglicht, die virtuelle Kamera zu starten und zu stoppen. Wie man in Listing 8.31 sehen kann, wird ein PaeButton erstellt und die Funktion start_virtual_camera hinterlegt. Außerdem wird direkt unter dieser Komponente ein verstecktes PaeLabel

erstellt, welches angezeigt wird, sollte kein Treiber für diese Funktion verfügbar sein.

```
def init_start_virtual_camera_button(self):
    self.start_virtual_camera_button = PaeButton("Virtuelle Kamera starten")
    self.start_virtual_camera_button.clicked.connect(self.toggle_virtual_camera)
    self.inpaint_vbox.addWidget(self.start_virtual_camera_button, alignment=Qt.AlignCenter)

    self.no_driver_label = PaeLabel("Kein Treiber für Virtuelle Kamera vorhanden")
    self.no_driver_label.hide()
    self.inpaint_vbox.addWidget(self.no_driver_label, alignment=Qt.AlignCenter)
```

Listing 8.31: Initialisierung: Startknopf der virtuellen Kamera

Wie man in Listing 8.32 sehen kann, wird der Button, welcher dazu dient, die virtuelle Kamera zu starten, deaktiviert und das Label aus Listing 8.31 angezeigt, sobald keine virtuelle Kamera vorhanden ist. Ob eine virtuelle Kamera vorhanden ist oder nicht, wird in Listing 8.30 gesetzt.

```
def check_virtual_camera_availability(self):
    if not self.virtual_camera_available:
        self.start_virtual_camera_button.setDisabled(True)
        self.no_driver_label.show()
```

Listing 8.32: Verfügbarkeit der virtuellen Kamera überprüfen

Sobald die virtuelle Kamera gestartet wird, wird die Funktion aus Listing 8.33 aufgerufen, welche die Variable auf das Gegenteil des aktuellen Zustands setzt. Sollte die virtuelle Kamera nicht laufen und die Funktion wird aufgerufen, wird die Variable auf True gesetzt. Auch der Text auf der Komponente selbst wird auf den aktuellen Zustand angepasst.

```
def toggle_virtual_camera(self):
    if self.virtual_camera_available:
        self.runningVirtualCamera = not self.runningVirtualCamera
        if self.runningVirtualCamera:
            self.start_virtual_camera_button.setText("Virtuelle Kamera stoppen")
        else:
            self.setLoadingScreen_virtualcamera()
            self.start_virtual_camera_button.setText("Virtuelle Kamera starten")
```

Listing 8.33: Starten der virtuellen Kamera

Die Funktion in Listing 8.34 wird bei jedem Inpainting-Prozess aufgerufen und sendet das Bild, welches als Parameter übergeben wird, an die virtuelle Kamera.

```
def update_virtual_camera(self, output_np):
...
            print(f'Bild wird gesendet an: {self.virtual_camera.device}')
            if output_np is not None:
                dim = display_width, display_height
                resized_image = cv2.resize(output_np, dim, interpolation=cv2.INTER_AREA)
                flipped_image = cv2.flip(resized_image, 1)
                self.virtual_camera.send(flipped_image)
...
```

Listing 8.34: Bild an virtuelle Kamera senden

Die Funktion setLoadingScreen_virtualcamera welche in Listing 8.29 aufgerufen wird, ist eine Funktion, die ein statisches Bild an die virtuelle Kamera sendet, wie man in Listing 8.35 sehen kann. Dieses Bild soll dem Nutzer signalisieren, dass die virtuelle Kamera nicht gestartet ist.

```
def setLoadingScreen_virtualcamera(self):
    placeholder_image = cv2.imread("images/A3_Copy_32x.jpg")
    resized_image = cv2.resize(placeholder_image, (512, 512), interpolation=cv2.INTER_AREA)
    self.virtual_camera.send(resized_image)
```

Listing 8.35: Standby Bild der virtuellen Kamera

8.5.5 Matplotlib

Wie in Abschnitt 7.4.7 erwähnt, wurde Matplotlib dazu verwendet, um visuelle Statistiken zu erzeugen, um die verschiedenen Ansätze und Algorithmen zu vergleichen und im schriftlichen Teil dieser Arbeit zu verwenden.

Implementierung im Programm

Wie man in Listing 8.36 sehen kann, wird bei jedem Inpainting Prozess zu jedem Algorithmus die Dauer des Prozesses gespeichert.

```
class InpaintingThread(QThread):
    ...

    def __init__(...):
        ...
        self.inpainting_times = inpainting_times
        self.cumulative_times = cumulative_times

    def run(self):
    ...
        ...
            ...
                if current_algorithm not in self.inpainting_times:
                    self.inpainting_times[current_algorithm] = []
                    self.cumulative_times[current_algorithm] = 0
                self.cumulative_times[current_algorithm] += end_time - start_time
                self.inpainting_times[current_algorithm].append((end_time - start_time))
```

Listing 8.36: Speichern der Zeitdaten aller Inpainting Algorithmen

Diese gespeicherten Daten werden, wie man in Listing 8.37 sehen kann, durch Matplotlib geladen und grafisch veranschaulicht. Dabei wird zuerst eine neue Statistik angelegt. Zuerst wird davon ausgegangen, dass keine Daten vorhanden sind und der Wert der Variable "has_data" wird auf False gesetzt. Sollte dieser False bleiben, wird beim Beenden des Programmes eine Statistik ohne Daten angezeigt. Sobald durch die Schleife Daten gefunden werden wird diese Variable auf True gesetzt und die Daten werden der Statistik hinzugefügt. Im nächsten Schritt werden zur x und y-Achse Labels hinzugefügt, um anzuzeigen, welche Werte diese anzeigen. Die x-Achse steht in diesem Beispiel für die Laufzeit eines Algorithmus. Die y-Achse zeigt an, wie lang ein Inpainting Prozess dauert. Zuletzt wird ein Titel gesetzt und die Legende aktiviert und der Raster deaktiviert.

```python
import matplotlib.pyplot as plt
...
class InpaintingThread(QThread):
    ...
    def show_graph(self):
        plt.figure()
        has_data = False
        for algo, times in self.inpainting_times.items():
            if times:
                cumulative_times = np.cumsum(times)
                plt.plot(cumulative_times, times, label=algo)
                has_data = True
        if not has_data:
            plt.text(0.5, 0.5, 'Keine Daten verfügbar', horizontalalignment='center',
            ↪  verticalalignment='center',
                     transform=plt.gca().transAxes)
        else:
            plt.xlabel('Cumulative Running Time (s)')
            plt.ylabel('Time taken for Each Inpainting (s)')
            plt.title('Inpainting Performance Comparison')
            plt.legend()
            plt.grid(False)

        plt.show()
```

Listing 8.37: Generieren einer Statistik durch Matplotlib

Durch Listing 8.37 könnte eine Statistik wie in Abb. 8.4 erstellt werden.

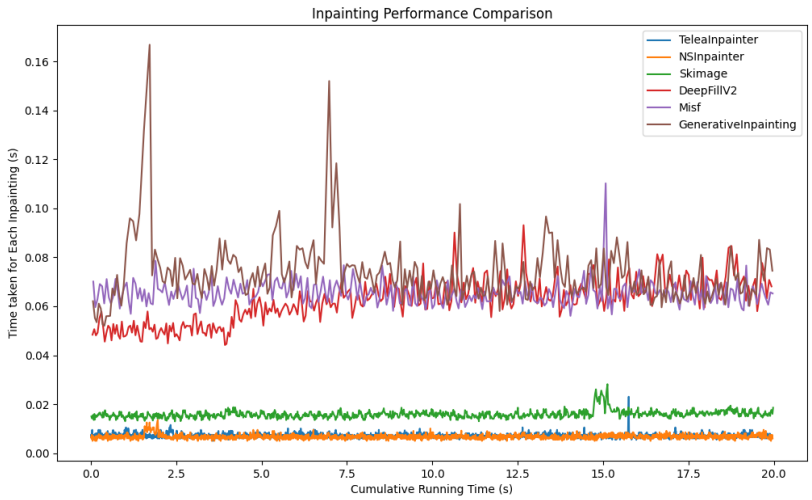

Abbildung 8.4: Statistik durch Matplotlib

9 Ergebnis

Nachfolgend sind die Ergebnisse dieser Arbeit aufgelistet und beschrieben. Diese besteht aus drei Teilen: **dem Vergleich** von verschiedenen Inpainting Ansätzen in Bezug auf deren Qualität, **einem Vergleich** der Rekonstruktionszeiten zwischen einem kostensparenden Entwicklerkit Nvidia Jetson Nano und einer teuren Nvidia RTX4070 Grafikkarte und der **ProcessAeye Anwendung**.

Die verschiedenen Ansätze werden in Geschwindigkeit (siehe 9.2) und in Qualität (siehe 9.1) getestet. Bei der Qualität gibts es einerseits die Frobeniusnorm und andererseits die menschliche Skala, wohingegen bei der Geschwindigkeit nur die Zeit gemessen wird.

9.1 Vergleich verschiedener Inpainting Ansätze

Wie man in Abb. 9.1 sehen kann, sind alle Algorithmen mit verschiedensten Bildern getestet worden. Dabei wird auf der linken Seite die sogenannte Ground Truth dargestellt, also das Bild, das im Original vorhanden ist. Als Nächstes wird die Maske dargestellt, welche den Bereich markiert, der repariert werden soll. Neben der Maske sind in jeder Zeile sechs weitere Bilder dargestellt, wo das Ergebnis jedes Algorithmus dargestellt wird.

Bei den ersten vier Reihen, wo die Maske mit eher dünnen Rissen gezeichnet wurde, kann man gut erkennen, dass keiner der Algorithmen Probleme mit dem Reparieren hat außer bei Generative Inpainting. Dieser zeichnet eher graue Striche. Das liegt daran, dass Generative Inpainting mit Masken trainiert wurde, die zentral im Bild ganze Flächen abgedeckt haben. In der Reihe mit dem Tukan, kann man hingegen gut erkennen, dass die klassischen Bildverarbeitungsalgorithmen größere Probleme haben. Die Deep Learning Modelle hingegen können hier ein deutlich besseres Bild erzeugen. Vor allem Generative Inpainting, auch wenn die Kanten zum Originalbild nach wie vor sichtbar sind, kann sich dieser Algorithmus nur im Falle einer größeren Fläche beweisen.

In nächsten drei Reihen wurde eine Maske mit größeren Rissen gewählt. Man kann auch hier wieder erkennen, dass die klassische Variante eher verschwommene Pixel einsetzt und die Deep Learning Modelle ein natürlicheres Gesamtbild schaffen. Besonders auffallend ist das Deep Learning Modell MISF in der letzten Reihe. Durch die Maske ist das rechte Auge der Person komplett verdeckt, aber MISF, ein Modell, welches mit Gesichtern trainiert wurde, hat ein Auge einsetzt.

Abbildung 9.1: Vergleich

Die genauen Fehlerwerte der Frobeniusnorm (siehe 8.4.1.1) sind in Tabelle 9.1 aufgelistet. In Tabelle 9.2 sieht man die menschliche Bewertung (siehe 8.4.1.2) dieser Ergebnisse. Alle Bilder wurden mit der Auflösung 256x256 getestet.

Wie man sehen kann, sind die Werte der Frobeniusnorm sehr nahe beieinander, aber diese Werte geben eben nicht zwangsläufig an, wie gut die Rekonstruktion wirklich ist. Genau deswegen sind die Ergebnisse auch vom menschlichen Augen bewertet worden und dort kann man sehen, dass, auch wenn die klassischen Bildverarbeitungsalgorithmen bessere oder gleich gute Frobeniusnormwerte erzielen, die Deep-Learning-Modelle die fehlenden Bereiche besser in das Bild wieder einfügen und keine verschwommenen Ergebnisse produzieren. Wenn wir uns zum Beispiel das Bild mit dem Gebäude ansehen, haben Telea und NS fast eine gleich gute Frobeniusnorm wie MISF und DeepFill, trotzdem sehen die Ergebnisse der Deep-Learning-Modelle natürlicher und besser in den Rest des Bildes integriert aus.

Bilder	Telea	NS	Skimage	DeepFill	MISF	Generative Inpainting
Person	0,1161	0,1136	0,0827	0,1182	0,0800	0,1726
Berge	0,2471	0,2459	0,2450	0,2717	0,2742	0,2618
Brücke	0,3250	0,3094	0,3087	0,3405	0,3287	0,3316
Felsen	0,1855	0,1776	0,1735	0,2005	0,1901	0,2354
Tukan	0,5059	0,4848	0,3864	0,3334	0,3038	0,3578
Gebäude	0,3332	0,3323	0,3586	0,3029	0,3277	0,3803
Hund	0,2050	0,1998	0,1728	0,1759	0,1608	0,2373
Person mit Hut	0,1818	0,1750	0,1512	0,1405	0,1247	0,2854

Tabelle 9.1: Frobeniusnorm der Algorithmen bei 256x256

Bilder	Telea	NS	Skimage	DeepFill	MISF	Generative Inpainting
Person	5	5	6	7	10	4
Berge	4	4	5	6	10	3
Brücke	6	6	6,5	6	8	3
Felsen	6	6,5	8	7	9	3
Tukan	4	4,5	3	9	8	6
Gebäude	5	5	4	7	8	3
Hund	6	6	7	9	10	3
Person mit Hut	5	5	6	7	9	3

Tabelle 9.2: menschliche Skala der Algorithmen bei 256x256

9.2 Vergleich Jetson Nano versus NVIDIA RTX4070

9.2.1 Einzelergebnisse: Jetson Nano

Nachfolgend sind die Geschwindigkeitstests der sechs Algorithmen des Jetson Nano aufgelistet.

Auf den beiden Abbildungen 9.2 und 9.3 kann man die benötigten Zeiten der Algorithmen einsehen. Auf der y-Achse befindet sich die benötigte Zeit in Sekunden, während die x-Achse die kumulative Laufzeit in Sekunden zeigt. Diese Zeiten auf der x-Achse sind möglichst gering gehalten worden, um eine überladene Darstellung zu vermeiden.

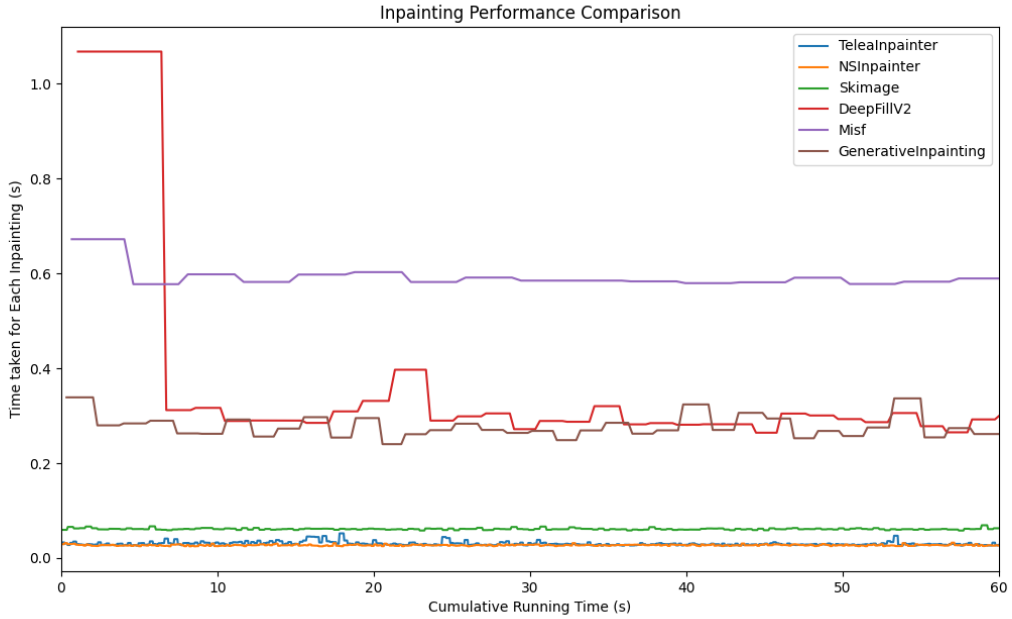

Abbildung 9.2: Jetson Nano - Auflösung: 128×128

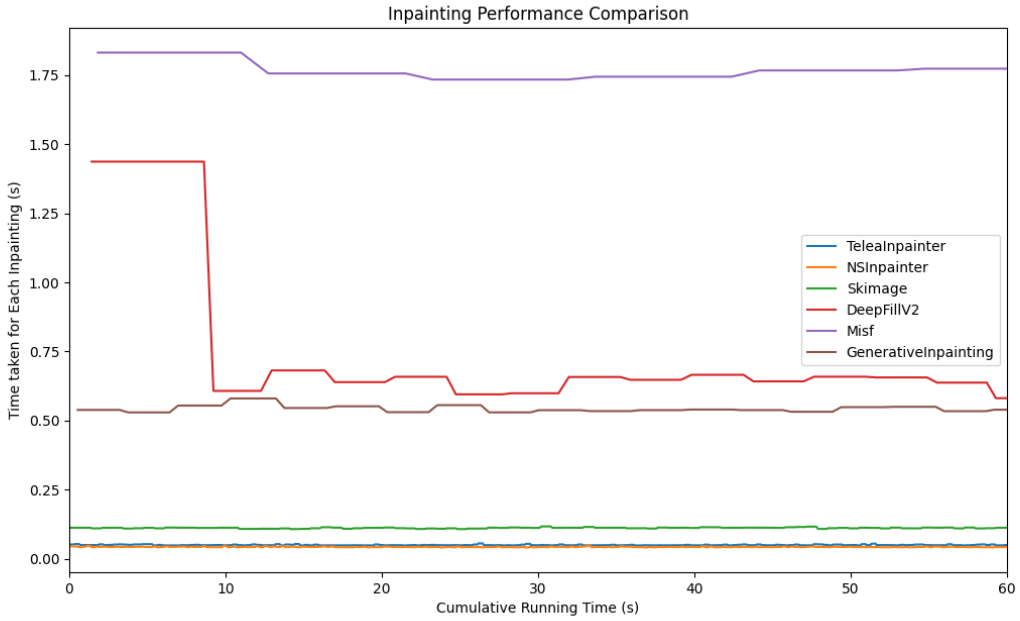

Abbildung 9.3: Jetson Nano - Auflösung: 256×256

9.2.2 Einzelergebnisse: NVIDIA RTX 4070

Nachfolgend sind die Geschwindigkeitstests der sechs Algorithmen auf einer RTX4070 aufgelistet. Die CPU des verwendeten PC ist eine AMD Ryzen 5 5600. Dies ist insofern wichtig, als die klassischen Bildverarbeitungsalgorithmen auf der CPU laufen.

Wie bereits in 7.6.1 angesprochen, verfügt der Nvidia Jetson Nano über 128 CUDA-Recheneinheiten, die RTX4070 hat dazu im Gegensatz 5.888[1]. Aufgrund dessen ist die RTX4070 auch um ein Vielfaches leistungsstärker, was man in Tabelle 9.4 sehen kann.

Auf den beiden Abbildungen 9.4 und 9.5 kann man die benötigten Zeiten der Algorithmen einsehen. Auf der y-Achse befindet sich die benötigte Zeit in Sekunden, während die x-Achse die kumulative Laufzeit in Sekunden zeigt. Diese Zeiten auf der x-Achse sind möglichst gering gehalten worden, um eine überladene Darstellung zu vermeiden.

[1] https://www.techpowerup.com/gpu-specs/geforce-rtx-4070.c3924

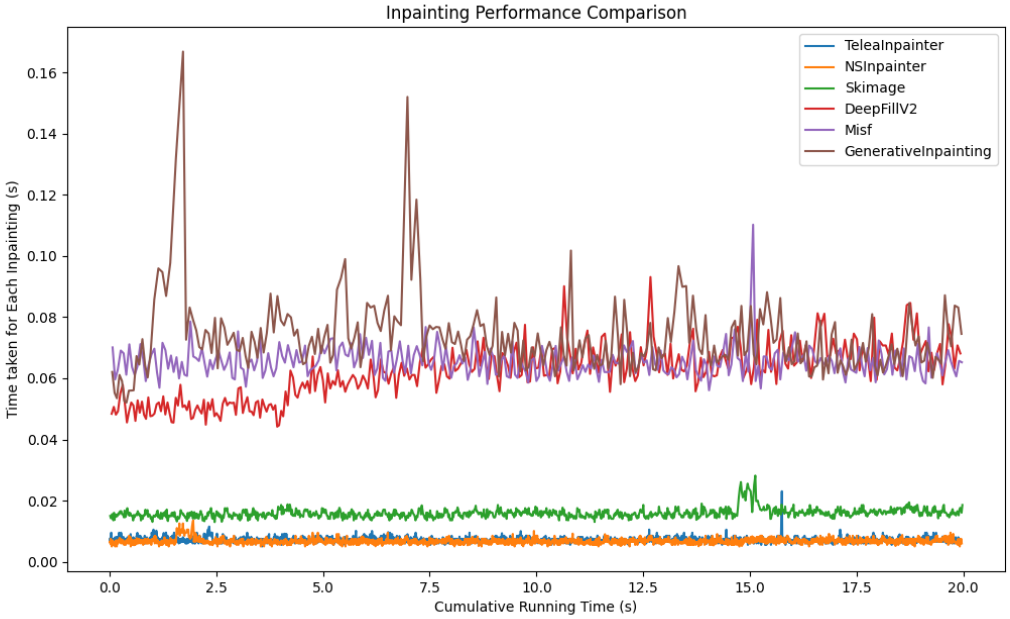

Abbildung 9.4: RTX4070 - Auflösung: 128×128

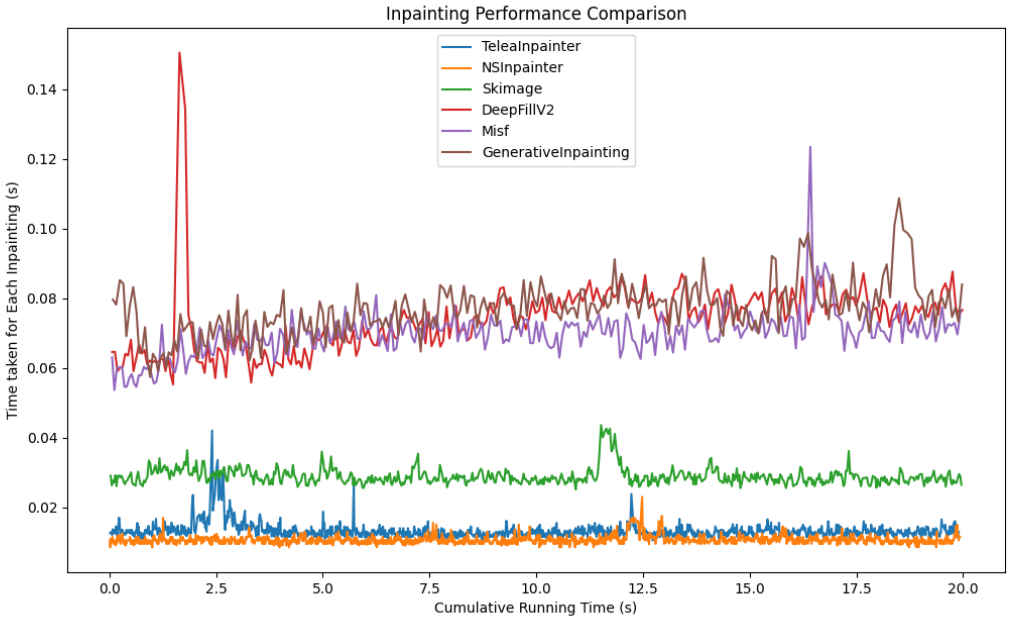

Abbildung 9.5: RTX4070 - Auflösung: 256×256

9.2.3 Vergleich von RTX4070 und Jetson Nano

Nachfolgend sind die Tabellen 9.3 und 9.4 mit den verschiedenen Geschwindigkeitswerten aller Algorithmen dargestellt. Diese Tabelle zeigt, wie lange ein Inpainting-Prozess dauert in Sekunden.

Auflösung	MISF	DeepFill	Generative	Telea	NS	Skimage
128x128	0,5956	0,2971	0,2710	0,0265	0,0227	0,0616
256x256	1,7455	0,6345	0,5588	0,0424	0,0395	0,1122
512x512	6,8354	2,5931	2,5564	0,1864	0,1515	0,2626

Tabelle 9.3: Leistungsvergleich Jetson Nano

Auflösung	MISF	DeepFill	Generative	Telea	NS	Skimage
128x128	0,0629	0,0657	0,0674	0,0084	0,0075	0,0158
256x256	0,0766	0,0756	0,0725	0,0137	0,0146	0,0288
512x512	0,0930	0,0942	0,0977	0,0329	0,0219	0,0710

Tabelle 9.4: Leistungsvergleich: AMD Ryzen 5 5600 und RTX 4070

Man kann deutlich erkennen, dass der PC mit dem AMD Ryzen 5 5600 und der RTX 4070 deutlich schneller ist als der Nvidia Jetson Nano, was in der Anwendung selbst zu einem deutlich flüssigeren Bild führt. Sobald ein Inpainting-Prozess länger als 0,2 Sekunden dauert, können nur noch maximal fünf Bilder pro Sekunde repariert werden. Das menschliche Auge sieht unter 24 Bildern pro Sekunde bereits, dass stockende Bildsequenzen vorkommen.

Auf dem PC laufen alle Algorithmen mit allen drei Auflösungen ohne Probleme in Echtzeit. Auf dem NVIDIA Jetson Nano allerdings kann man beobachten, dass die klassischen Bildverarbeitungsalgorithmen auf allen drei Auflösungen in Echtzeit laufen. Die Deep-Learningmodellen wiederum bringen selbst bei der geringsten Auflösung von 128×128 kein flüssiges Bild zustande.

9.3 ProcessAEye Anwendung

Das Hauptergebnis dieser Arbeit ist die ProcessAeye-Anwendung. Mit dieser ist es möglich, mit einer angeschlossenen Kamera in Echtzeit Bilder zu reparieren. Man kann dabei wählen, ob man eine Maske selbst zeichnet oder eine hochlädt. Besonders hervorzuheben ist, dass man dabei zwischen sechs verschiedenen Methoden wählen kann. Drei davon sind klassische Bildverarbeitungsalgorithmen und drei weitere basieren auf einer Machine-Learning-Technologie. Außerdem ist die Anwendung sehr benutzerfreundlich und kann sehr einfach bedient werden. Die Anwendung ist in Abb. 9.6 abgebildet.

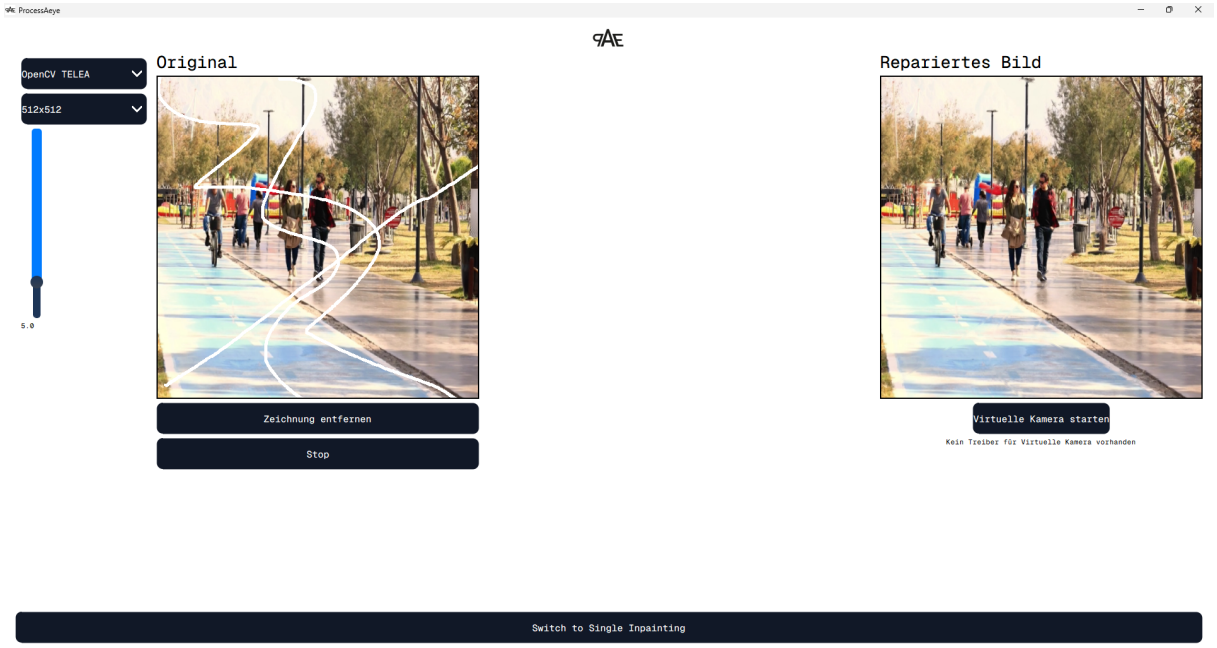

Abbildung 9.6: ProcessAEye Anwendung

10 Resümee

Durch die Erfahrungen aus dem Projektunterricht konnte die Projektplanungsphase komplett eigenständig von uns durchgeführt werden. Wir haben uns für Scrum als Entwicklungsmethode entschieden, da wir diese perfekt in dem 4-wöchigen Praktikum im Sommer 2023 an der JKU-Linz anwenden konnten.

Im Rahmen der Durchführung dieses Projekts konnten neue Technologien kennengelernt werden. Diese konnten durch Selbsterarbeitung und durch Unterstützung des Auftragsgebers in absehbarer Zeit erlernt und verwendet werden. Dazu zählen folgende Themen:

- Python
- PyQT5
- Pytorch mit verschiedensten Machine-Learning-Modellen
- OpenCV mit klassischen Bildverarbeitungsalgorithmen

Besonders der Anfang war schwierig, da wir noch keine Erfahrung im Bereich Machine Learning hatte, und uns erst mal die Grundlagen aneignen mussten. Diese Hürde ist vor allem durch unser Interesse in diesem Bereich überwunden worden. Nicht zu unterschätzen ist aber auch der Aufwand für das Evaluieren der Algorithmen gewesen. Anfangs dachten wir, dass wir einfach den Algorithmus mit den besten Ergebnissen verwenden, aber im Bereich Machine Learning haben die Trainingsdaten einen großen Einfluss auf das Ergebnis und so hat jedes Modell seine Stärken. Auch die klassischen Algorithmen unterscheiden sich in ihrer Genauigkeit, auch wenn diese Unterschiede nicht zu groß sind.

Natürlich sind die klassischen Algorithmen in der Möglichkeit größere Teile des Bildes wiederherzustellen eingeschränkt, aber für die Echt-Zeit Wiederherstellung von eher kleineren Bildfehlern optimal geeignet. Die Machine Learning-Modelle im Gegensatz brauchen auf dem Nvidia Jetson Nano zu lange, um ein flüssiges Bild zu bekommen, jedoch können diese auch größere Teile eines Bildes wiederherstellen. Diese Modelle finden aber auch deren Ende ab bestimmten Größen beziehungsweise, wenn zu viele wichtige Informationen fehlen.

11 Planung und Realisierung

11.1 Projektorganisation

Folgende Tabelle zeigt eine IVM-Matrix welche die Verantwortlichkeiten der einzelnen Projektmitglieder darstellt:

	Cedric Bauer	Stefan Czepl
Projektmanagement	V	M
Auswahl von Features	M	V
Einbindung von Deep-Learning Modellen	M	V
Einbindung von klassischen Algorithmen	V	M
Entwickeln einer Benutzeroberfläche	V	M
Vergleich von Deep-Learning und klassischen Algorithmen	M	V

Abbildung 11.1: IMV-Matrix: I... Information, M... Mitarbeit, V... Verantwortlich

11.2 Meilensteine

Folgende Meilensteine wurden für die Diplomarbeit definiert:

- Hardware Setup
- Testdaten gesammelt
- Deep-Learning und klassische Bildverarbeitungsalgorithmen fertig verglichen
- Echtzeit-Kamerabilder eingebunden
- Anzeigen des Original- und Ergebnisbildes
- Interaktiver Modus implementiert

Alle Meilensteine wurden nach dem definierten Ziel erreicht und der JKU präsentiert und vorgelegt.

11.2.1 Projektverlauf

Für jeden Meilenstein ist ein Termin für die Fertigstellung definiert worden. Dies hat sich im Verlauf des Projekts als sehr positiv herausgestellt um zu beobachten, ob man gut in der Zeit liegt. Allerdings sind aufgrund der fehlenden Erfahrung in diesem Bereich die Schätzungen etwas ungenau gewesen und Aufgabenpakete mit höher definiertem Aufwand haben sich als leichter herausgestellt und umgekehrt.

Für die Oben genannten Meilensteine sind folgende Termine gesetzt worden:

- 03.08.2023 - Hardware Setup
- 07.08.2023 - Testdaten gesammelt
- 25.08.2023 - Deep-Learning und klassische Bildverarbeitungsalgorithmen fertig verglichen
- 30.08.2023 - Echtzeit-Kamerabilder eingebunden
- 04.09.2023 - Anzeigen des Original- und Ergebnisbildes
- 10.09.2023 - Interaktiver Modus implementiert

Diese Termine wurden so geschätzt, dass alle Aufgaben, welche mit den Algorithmen zu tun haben, im Praktikum in den Sommerferien an der JKU abgeschlossen werden können. Dies hat vor allem den Grund, dass das Knowhow des Instituts zur Unterstützung dient. Aufgrund der Unerfahrenheit des Projektteams hat sich die Reihenfolge der Meilensteine etwas verändert. Der interaktive Modus, sowie die Echtzeit-Kamera Einbindung plus die Anzeige des Original- und Ergebnisbildes sind bereits im Praktikum abgeschlossen worden. Das Hardware-Setup hat sich als eine große Herausforderung herausgestellt und ist deswegen erst, 2 Tage nach dem geplanten Termin abgeschlossen worden.

Das ausführliche Testen der Algorithmen ist dafür, erst Oktober fertig geworden. Dies ist absichtlich auf die Zeit nach dem Praktikum verlegt worden, weil dafür keinerlei Hilfe des Instituts notwendig war, aber viel Zeit beansprucht hat, da jeder Algorithmus 30 Minuten lang getestet werden musste.

11.2.2 Erkenntnisse

Durch die gute Unterstützung des Instituts für Signalverarbeitung der JKU Linz war es einfacher von Anfang an einen guten Plan für das Projekt zu entwickeln. Es konnten alle Meilensteine rechtzeitig oder sogar früher abgeschlossen worden. Da beide Projektmitglieder noch keine Erfahrung im Bereich Machine Learning hatten, wäre es ohne die Unterstützung und Betreuung nicht möglich gewesen, die Meilensteine einzuhalten.

12 Aufgabenverteilung

Im folgenden Punkt ist festgehalten, wer welche Kapitel der Diplomarbeit verfasst hat. Teilweise wurden Kapitel von beiden Autoren verfasst, da der jeweilige Bereich in die Zuständigkeit beider Teammitglieder fällt.

12.1 Cedric Bauer

Inhaltsverzeichnis Cedric

12.2 Stefan Czepl

Inhaltsverzeichnis Stefan

Abbildungsverzeichnis

Literatur

[1] NumPy Developers. *NumPy documentation*. URL: https://numpy.org/doc/stable. (abgerufen am 19.02.2024).

[2] PyTorch Developers. *PyTorch Dokumentation*. URL: https://pytorch.org/. (abgerufen am 18.02.2024).

[3] Jeffrey A. Clark Fredrik Lundh und contributors. *Pillow*. URL: https://pillow.readthedocs.io/en/stable/. (abgerufen am 19.02.2024).

[4] Xiaoguang Li u. a. „MISF: Multi-level Interactive Siamese Filtering for High-Fidelity Image Inpainting". In: *CVPR* (2022).

[5] G. Sapiro M. Bertalmıo A. L. Bertozzi. *Navier-Stokes, Fluid Dynamics, and Image and Video Inpainting*. 2001. URL: https://www.math.ucla.edu/~bertozzi/papers/cvpr01.pdf. (abgerufen am 29.03.2024).

[6] TorchVision maintainers und contributors. *TorchVision: PyTorch's Computer Vision library*. https://github.com/pytorch/vision. 2016.

[7] scikit-image team. *Fill in defects with inpainting*. URL: https://scikit-image.org/docs/stable/auto_examples/filters/plot_inpaint.html#id5. (abgerufen am 30.03.2024).

[8] Alexandru Telea. *An Image Inpainting Technique Based on the Fast Marching Method*. 2004. URL: https://webspace.science.uu.nl/~telea001/uploads/PAPERS/JGT04/paper.pdf. (abgerufen am 27.03.2024).

[9] Jiahui Yu u. a. „Free-Form Image Inpainting with Gated Convolution". In: *arXiv preprint arXiv:1806.03589* (2018).

[10] Jiahui Yu u. a. „Generative Image Inpainting with Contextual Attention". In: *arXiv preprint arXiv:1801.07892* (2018).

Listings

13 Anhang

13.1 Logo

Abbildung 13.1: Logo

13.2 Diplomarbeitsplakat

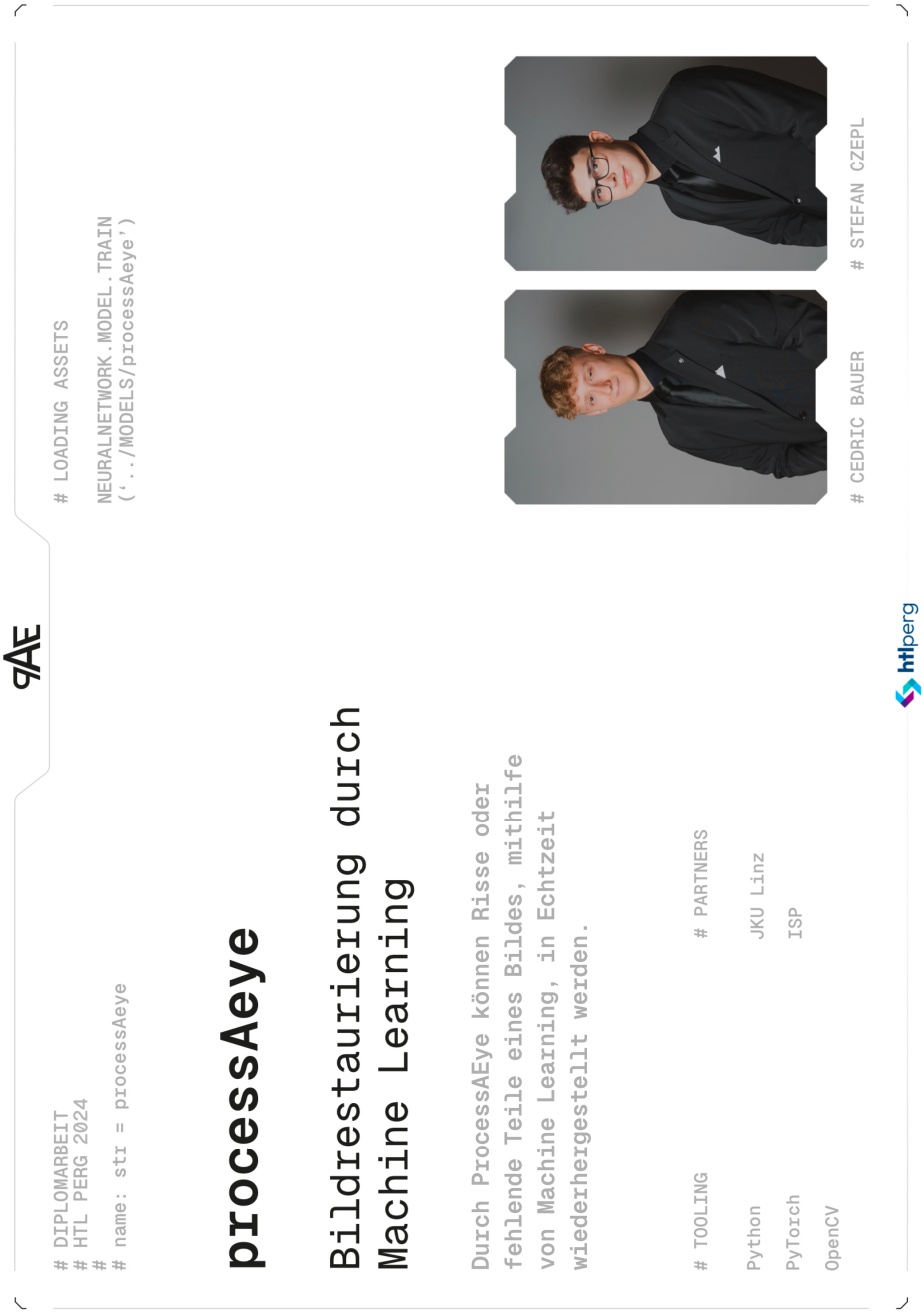

Abbildung 13.2: Diplomarbeitsplakat