

Untersuchungen von Style Transfer Methoden für die Verwendung mit
leistungssarmer Hardware

Abschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft Berlin

Fachbereich IV: Informatik, Kommunikation und Wirtschaft

Studiengang: Angewandte Informatik

1. Prüfer: Prof. Dr. Christin Schmidt
2. Prüfer: Patrick Baumann

Eingereicht von: Christoph Stach

Immatrikulationsnummer: 0555912

Eingereicht am: 28.08.2018

Danksagung

Ich danke allen Personen, die mich bei der Erstellung dieser Abschlussarbeit unterstützt haben. Besonderer Dank gilt meinen Eltern, Annegret Stach und Rüdiger Stach sowie meiner Schwester Pauline Stach, die mich während der gesamten Zeit meines Studiums unterstützt haben.

Weitere Danksagungen gehen an meine Freunde und meine Betreuer Frau Prof. Dr. Christin Schmidt und Herrn Patrick Baumann, die mir bei der Erstellung dieser Arbeit mit Rat und technischen Ressourcen zur Seite standen.

Abstract

Die Abschlussarbeit im Bereich der angewandten Informatik handelt davon, verschiedene Style Transfer Methoden zu implementieren, die Konzepte zu verstehen und sie auf Tauglichkeit für Systeme mit leistungsarmer Hardware zu testen. Style Transfer ist ein Bereich des Deep Learning. Es werden zwei oder mehrere willkürliche Bilder zu einem neuen Bild kombiniert. Beim ersten Bild wird der Inhalt des Bildes herausgefiltert, z.B. aus einem Foto, auf dem ein Vogel zu sehen ist. Aus dem zweiten Bild wird der Stil herausgefiltert, z.B. aus einem abstrakt gezeichneten Gemälde. Das Ergebnis ist die Mischung aus Objekt und Stil. Die Ausrichtung, ob das Ergebnis mehr Objekt oder eher Stil ist, kann durch Parameter beeinflusst werden. Es werden unterschiedliche Algorithmen sowie ihre Hyperparameter untersucht. Anschließend wird ihre Performanz gemessen und die Ergebnisse miteinander verglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis	iv
Tabellenverzeichnis	v
Quellcodeverzeichnis	vi
1 Einleitung	1
1.1 Idee und Motivation	1
1.2 Zielsetzung	1
1.3 Vorgehensweise und Aufbau der Arbeit	2
2 Grundlagen	3
2.1 Feedforward Neural Network	3
2.2 Fully-Connected Neural Network	4
2.3 Aktivierungsfunktionen	5
2.3.1 LeakyReLU, PReLU	6
2.3.2 Hardtanh	6
2.3.3 Sigmoid	7
2.4 Convolutional Neural Network	8
2.5 Backpropagation	9
2.6 Verwandte Arbeiten	11
2.6.1 Neural Style Transfer	11
2.6.2 Fast Neural Style Transfer	14
3 Methodologie	15
3.1 Vorherige Arbeiten	15
3.2 Neural Style Transfer	16
3.3 Fast Neural Style Transfer	18
3.3.1 Aufbau	18
3.3.2 Unterschiedliche Netzwerkgrößen	20
4 Implementierung	21
4.1 Loss-Funktion	21
4.1.1 Zugriff auf Hidden-Layer-Ergebnisse	22

4.1.2	Content-Loss	22
4.1.3	Style-Loss	23
4.1.4	Total-Variation-Loss	24
4.1.5	Perceptual-Loss	24
4.2	Neural Style Transfer	24
4.3	Fast Neural Style Transfer	25
4.3.1	COCO-Datensatz	25
4.3.2	Netzwerkarchitektur	26
4.3.3	Aktivierungsfunktionen	26
4.3.4	ConvBlock	27
4.3.5	ResidualBlock	27
4.3.6	UpSampleBlock	27
4.3.7	TransformerNet	27
4.3.8	Weitere Implementierungen	28
5	Tests und Experimente	29
5.1	Auswirkungen der Hyperparameter	29
5.1.1	Experiment 1: Starry Night	30
5.1.2	Experiment 2: The Scream	31
5.1.3	Interpretation der Ergebnisse	32
5.2	Auswirkungen der Netzwerkarchitekturen	32
5.2.1	Verwendete Geräte	32
5.2.2	Durchführung der Experimente	33
5.2.3	Interpretation der Ergebnisse	34
5.2.4	Weitere Ergebnisse	34
6	Evaluation	35
6.1	Grundlagen	35
6.2	Methodologie	35
6.3	Implementierung	36
6.4	Tests und Experimente	36
7	Fazit	37
7.1	Zusammenfassung	37
7.2	Kritischer Rückblick	37
7.3	Ausblick	38
7.3.1	Verwendung anderer Layer-Kombinationen	38
7.3.2	Kombination aus mehreren Stilen	38
7.3.3	Superresolution	38
7.3.4	Alternativen zum ResidualBlock	39

Inhaltsverzeichnis

7.3.5	Video Stylization	39
7.3.6	Ausführung im Webbrowser	39
Glossar		40
Quellen- und Literaturverzeichnis		41
Anhang		I
A.1	Notebook Neural Style Transfer	I
A.2	Script Activation-Functions	III
A.3	Script ConvBlock	IV
A.4	Script ResidualBlock	IV
A.5	Script UpsampleBlock	V
A.6	Script TransformerNet	VI
B.1	Netzwerkarchitekturen	VII
B.2	Performanz-Test mit 1920 * 1080 (Full HD) Pixel Bildern	VIII
B.3	Performanz-Test mit 1024 * 768 Pixel Bildern	IX
B.4	Performanz-Test mit 640 * 480 Pixel Bildern	X
B.5	Loss: The Starry Night	XI
B.6	Loss: The Scream	XII
B.7	Ergebnisse: The Starry Night	XIII
B.8	Ergebnisse: The Scream	XIII
B.9	Trainierte Modelle	XIV

Abbildungsverzeichnis

2.1	Fully-Connected Neural Network [Kar19a]	4
2.2	ReLU-Aktivierungsfunktion [PyT19c]	5
2.3	LeakyReLU-, PReLU-Aktivierungsfunktion [PyT19b]	6
2.4	Hardtanh-Aktivierungsfunktion [PyT19a]	7
2.5	Sigmoid-Aktivierungsfunktion [PyT19d]	7
2.6	Beispiel CNN Architektur [Com15]	8
2.7	Berechnungsschritt während einer Convolution [Vel16]	8
2.8	Visualisierung eines Neurons (auch Perceptron [Gal90] genannt) [Jus16]	9
2.9	Forward-Pass in Grün und Backward-Pass in Rot dargestellt [Kar19b]	10
2.10	Architektur des VGG16-Netzwerks [Fer17]	11
2.11	Visualisierung des Receptive Field über drei CNN-Layer [Hie17]	12
2.12	Trainingsprozess eines Image Transformer Networks [JAL16]	14
3.1	Komponenten für den Neural Style Algorithmus, eigene Darstellung	16
3.2	Programmablaufplan für den Neural Style Algorithmus, eigene Darstellung	17
3.3	Netzwerkarchitektur des Transformer Net, eigene Darstellung	18
3.4	Aufbau eines Residual Block [He+15a]	19
5.1	HTW kombiniert mit <i>The Starry Night</i> [Nig89]	30
5.2	<i>The Starry Night</i> mit $\alpha = 1, \beta = 10^6 - 10^9, \gamma = 0$	30
5.3	<i>The Starry Night</i> mit $\alpha = 1, \beta = 10^8, \gamma = 10^{-6} - 10^{-3}$	30
5.4	HTW kombiniert mit <i>The Scream</i> [Høs93]	31
5.5	<i>The Scream</i> mit $\alpha = 1, \beta = 10^6 - 10^9, \gamma = 0$	31
5.6	<i>The Scream</i> mit $\alpha = 1, \beta = 10^7, \gamma = 10^{-6} - 10^{-3}$	31
B.1	Loss: The Starry Night	XI
B.2	Loss: The Scream	XII
B.3	Ergebnisse: The Starry Night	XIII
B.4	Ergebnisse: The Scream	XIII
B.5	Eigene Abbildungen kombiniert mit verschiedenen Stilen: [Bac19], [Jos89]	XIV
B.6	Eigene Abbildungen kombiniert mit verschiedenen Stilen: [Art18], [Liq09]	XV
B.7	Eigene Abbildungen kombiniert mit verschiedenen Stilen: [TP19], [Tre89]	XVI

Tabellenverzeichnis

5.1 Spezifikation: Dell XPS 15 9550	33
5.2 Spezifikation: Jetson TX2	33
B.1 Unterschiedliche Netzwerkgrößen und ihre Parameter	VII
B.2 Berechnungsgeschwindigkeit in Sekunden für Bilder der Größe 1920 * 1080 Pixel	VIII
B.3 Berechnungsgeschwindigkeit in Sekunden für Bilder der Größe 1024 * 768 Pixel .	IX
B.4 Berechnungsgeschwindigkeit in Sekunden für Bilder der Größe 640 * 480 Pixel . .	X

Quellcodeverzeichnis

4.1	Extrahierung der Activation-Maps mit PyTorch	22
4.2	Berechnung des Content-Loss, vgl. Gleichung (2.12)	22
4.3	Berechnung der Gram-Matrix, vgl. Gleichungen (2.13) u. (2.14)	23
4.4	Berechnung des Style-Loss, vgl. Gleichung (2.15)	23
4.5	Berechnung des Total-Variation-Loss [Lee17], vgl. Gleichung (2.16)	24
4.6	Berechnung des gesamten Perceptual-Loss, vgl. Gleichung (2.17)	24
4.7	Vereinfachter Code einer Trainingsschleife in PyTorch	25
4.8	Optimierung der Gewichte des Netzwerks	25
4.9	Der PyTorch-Dataloader lädt Bilder in der gewünschten Batchgröße	26
A.1	Notebook zur Durchführung des Neural Style Transfer Algorithmus	III
A.2	Activation-Functions	IV
A.3	ConvBlock	IV
A.4	ResidualBlock	V
A.5	UpsampleBlock	V
A.6	TransformerNet	VI

Kapitel 1

Einleitung

Im Kapitel Einleitung wird die Idee und persönliche Motivation, Zielsetzung und Vorgehensweise sowie der Aufbau der Arbeit beschrieben.

1.1 Idee und Motivation

Die Idee, einen Algorithmus für Neural Style Transfer zu entwickeln, der auf Geräten mit leistungssarmer Hardware funktioniert, entstand durch mein persönliches Interesse an der Webentwicklung und an Machine Learning Verfahren. Vor dem Studium an der HTW Berlin konnte ich Erfahrungen im Bereich der Web-/Frontendentwicklung sammeln und habe bereits professionell in diesem Bereich gearbeitet. Im Rahmen des Hochschulstudiums lernte ich, Verfahren in den Bereichen Machine- und Deep Learning anzuwenden. Besonders der Bereich Computervision war für mich interessant. Ich nutze die Bachelorarbeit dazu, diese beiden Interessensgebiete zu kombinieren und möchte die Möglichkeiten erforschen, Style Transfer Methoden auf Geräten mit leistungssarmer Hardware auszuführen.

Style Transfer ist ein interessantes Forschungsobjekt und erfordert die intensive Auseinandersetzung mit Convolutional Neural Networks [LeC+98] und dem Backpropagation Algorithmus [LeC+89]. Außerdem müssen benutzerdefinierte Loss-Funktionen, die nicht in Problemstellungen der Bildklassifizierung verwendet werden, benutzt werden.

1.2 Zielsetzung

Ziel der Arbeit ist es, den Prototypen eines Softwaresystems zu erstellen, der in der Lage ist, Style Transfer auf Geräten mit leistungssarmer Hardware zu ermöglichen. Es sollen Mechanismen des Style Transfer untersucht und sukzessiv verbessert werden.

1.3 Vorgehensweise und Aufbau der Arbeit

Nach der Einarbeitung in die Grundlagen von Style Transfer [GEB15; JAL16] Methoden müssen entsprechende Algorithmen gefunden werden. Da es sich um die Manipulation von Bildern handelt, werden Convolutional Neural Networks [LeC+98] zum Einsatz kommen, die sich dazu eignen, Features aus Bildern zu extrahieren. Im nächsten Schritt muss ein Datensatz wie ImageNet [Den+09] oder COCO [Lin+14] benutzt oder in entsprechender Größe generiert werden, mit dem ein solches Netzwerk trainiert werden kann. Letztendlich muss das Netzwerk auf dem verwendeten Datensatz durch Backpropagation [LeC+89] optimiert werden. Die Implementierung des Modells in den Prototypen eines Softwaresystems soll der Veranschaulichung der Ergebnisse dienen.

Abschließend werden umfangreiche Experimente mit verschiedenen Einstellungen und Netzwerkarchitekturen durchgeführt. Die Ergebnisse werden miteinander verglichen und in einem Fazit kritisch betrachtet.

Kapitel 2

Grundlagen

Im Kapitel Grundlagen werden die erforderlichen Grundlagen erläutert, die notwendig sind, um die Funktionsweise von Style Transfer zu verstehen. Außerdem wird auf bestehende Arbeiten in diesem Bereich eingegangen.

2.1 Feedforward Neural Network

Feedforward Neural Networks werden im Bereich des Deep Learning eingesetzt [GBC16, S. 164–223]. Es werden unterschiedliche Arten von Layern zu einem Neuronalen Netzwerk zusammengesetzt. Je nach Aufgabenstellung sind verschiedene Arten von Layer-Kombinationen sinnvoll.

Ein Feedforward Neural Network definiert eine Abbildung von $y = f(x; \phi)$, wobei ϕ (auch manchmal mit w notiert) Gewichtungsparameter des Netzwerks sind. Es wird Feedforward genannt, da die Informationen x von vorne nach hinten durch die einzelnen Layer des Networks fließen und am Ende y ausgeben. Die Layer des Netzwerks sind wiederum Unterfunktionen, die in einer Kettenstruktur ineinander übergehen. Bei einem Neuronalen Netzwerk mit den drei Layern $f^{(1)}, f^{(2)}, f^{(3)}$ ergibt sich der Aufbau $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$.

Der erste Layer eines Netzwerks nennt sich Input-Layer, der letzte Output-Layer. Alle dazwischenliegenden Layer werden Hidden-Layer genannt. Je mehr Layer ein Netzwerk besitzt, desto komplexere mathematische Funktionen kann es abbilden und desto tiefer ist es. Daher röhrt der Name Deep Neural Network und Deep Learning.

2.2 Fully-Connected Neural Network

Fully-Connected Neural Networks bestehen aus mehreren hintereinander geschalteten Fully-Connected-Layern. Der Name stammt daher, dass zwischen zwei aufeinanderfolgenden Layern alle Neuronen miteinander verbunden sind.

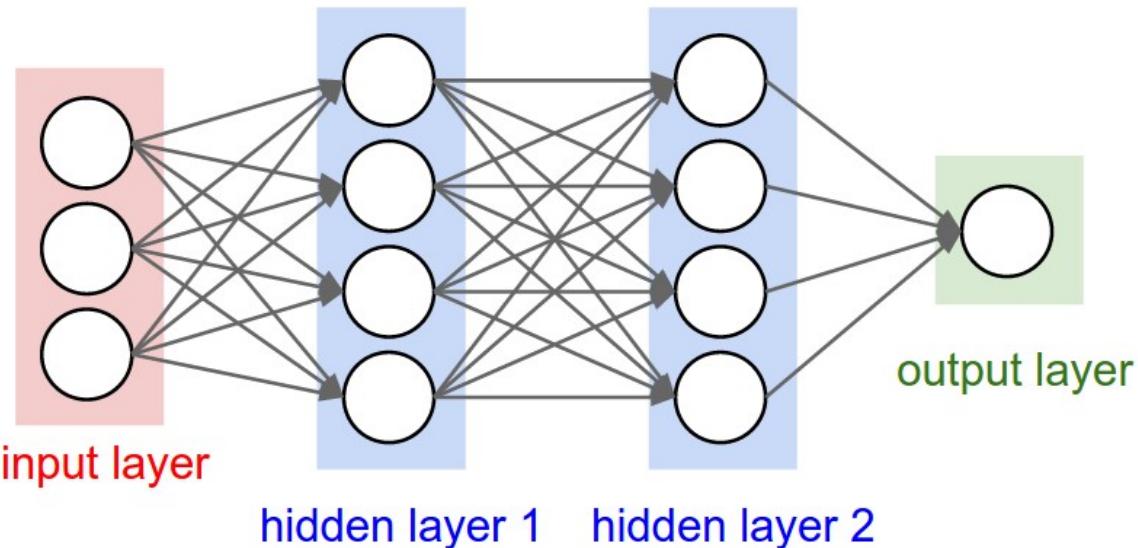


Abbildung 2.1: Fully-Connected Neural Network [Kar19a]

Die Kreise stellen die Neuronen dar und die Kanten sind die Gewichtungsfaktoren. Eine solche Struktur kann mit einer einzigen Matrix-Multiplikation berechnet werden.

$$f(x) = \sigma(x * w + b) \quad (2.1)$$

In der Formel steht x für die Eingangsdaten, w für die Gewichtungsfaktoren, b für einen Biaswert. Letztlich läuft das berechnete Ergebnis durch eine Aktivierungsfunktion σ , die nachfolgend in 2.3 beschrieben wird.

Ausgehend von einem Neuronalen Netzwerk mit einem Input-Layer mit zwei Neuronen, einem Hidden-Layer mit drei Neuronen, einem Output-Layer mit einem Neuron, einer Minibatchgröße

von drei und der Aktivierungsfunktion $\sigma(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ ergibt sich folgendes Beispiel:

$$x = \begin{pmatrix} 0.0 & 0.0 \\ 1.0 & 1.0 \\ 2.0 & 2.0 \end{pmatrix} w_1 = \begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \\ 0.5 & 0.5 \end{pmatrix} w_2 = \begin{pmatrix} 1.0 \\ 1.0 \\ 0.5 \end{pmatrix} \quad (2.2)$$

Die Werte der letzten Zeilen in den Gewichtsmatrizen w_1 und w_2 sind die Gewichtungsfaktoren für den Bias-Wert. Den Eingabedaten x wird vor Durchlauf durch einen Layer ein fester Wert (Bias) von 1 hinzugefügt. Nach Durchlauf durch den ersten Layer ergibt sich dementsprechend folgendes Zwischenergebnis:

$$\sigma \left(\begin{pmatrix} 0.0 & 0.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 1.0 \end{pmatrix} \times \begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \\ 0.5 & 0.5 \end{pmatrix} \right) = \begin{pmatrix} 0.5 & 0.5 \\ 2.5 & 2.5 \\ 4.5 & 4.5 \end{pmatrix} \quad (2.3)$$

Wiederholt wird dieser Vorgang mit der Gewichtsmatrix w_2 des zweiten Layers:

$$\sigma \left(\begin{pmatrix} 0.5 & 0.5 & 1.0 \\ 2.5 & 2.5 & 1.0 \\ 4.5 & 4.5 & 1.0 \end{pmatrix} \times \begin{pmatrix} 1.0 \\ 1.0 \\ 0.5 \end{pmatrix} \right) = \begin{pmatrix} 1.5 \\ 5.5 \\ 9.5 \end{pmatrix} \quad (2.4)$$

2.3 Aktivierungsfunktionen

Die im vorherigen Beispiel verwendete Aktivierungsfunktion nennt sich ReLU [NH10].

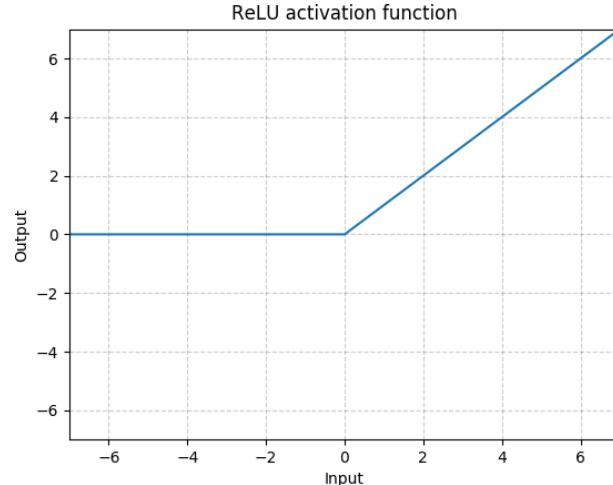


Abbildung 2.2: ReLU-Aktivierungsfunktion [PyT19c]

Neben der ReLU-Aktivierungsfunktion existieren weitere Aktivierungsfunktionen, die im Zusammenhang mit Neuronalen Netzwerken verwendet werden. Im Folgenden werden die in dieser Arbeit verwendeten Aktivierungsfunktionen beschrieben.

2.3.1 LeakyReLU, PReLU

Die Aktivierungsfunktionen LeakyReLU und PReLU sind Abwandlungen von ReLU und werden um den Parameter α ergänzt. Bei LeakyReLU handelt es sich bei α um einen fixen, einstellbaren Wert. Bei PReLU wird dieser im Laufe der Trainingsphase gelernt [Xu+15]. ReLU hat gegenüber PReLU und LeakyReLU den Nachteil, dass Neuronen deaktiviert werden können, da die linke Seite der Aktivierungsfunktion 0 beträgt. Somit ergeben sich auch bei der Berechnung der Gradienten keine Veränderungen.

$$\sigma(\alpha, x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad (2.5)$$

Visualisiert besitzen beide Aktivierungsfunktionen folgende Form:

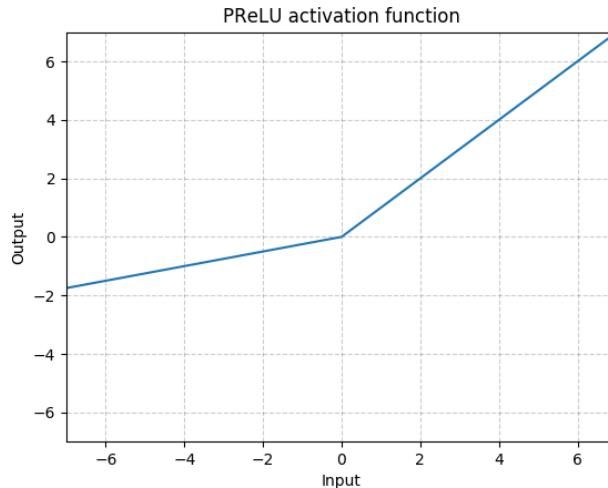


Abbildung 2.3: LeakyReLU-, PReLU-Aktivierungsfunktion [PyT19b]

2.3.2 Hardtanh

Die Hardtanh Funktion wird in dieser Arbeit als abschließende Aktivierungsfunktion genutzt [Nwa+18; Col+11].

$$\sigma(x) = \begin{cases} 1 & \text{if } x > 1 \\ -1 & \text{if } x < -1 \\ x & \text{otherwise} \end{cases} \quad (2.6)$$

Die Funktion ist in der Lage, die Ergebnisse auf einen festgelegten Bereich zu beschränken.

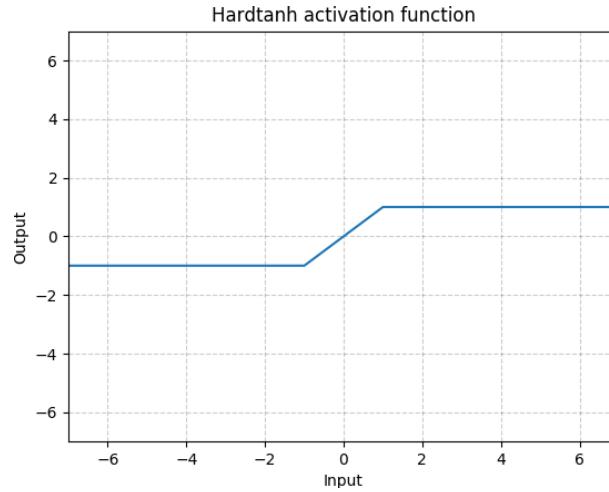


Abbildung 2.4: Hardtanh-Aktivierungsfunktion [PyT19a]

2.3.3 Sigmoid

Die Sigmoid-Funktion wird in dieser Arbeit als alternative, abschließende Aktivierungsfunktion genutzt [Nwa+18].

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.7)$$

Die Ergebnisse der Sigmoid-Funktion konvergieren zu 0 und 1, erreichen diese jedoch nie komplett.

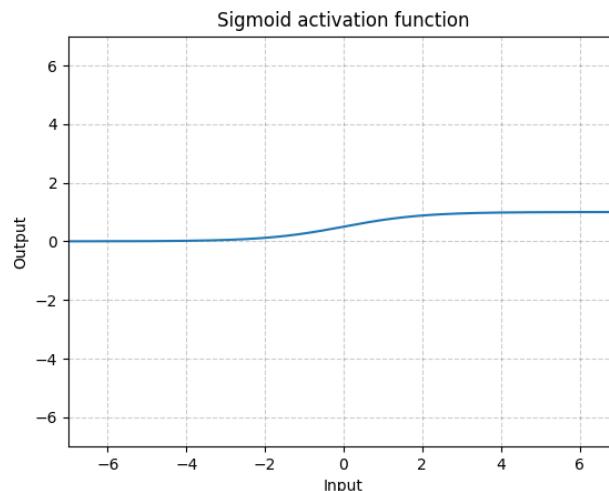


Abbildung 2.5: Sigmoid-Aktivierungsfunktion [PyT19d]

2.4 Convolutional Neural Network

Ein Convolutional Neural Network [LeC+89; GBC16, S. 326–366], auch CNN genannt, besteht aus mehreren aufeinanderfolgenden Convolutional-Layern sowie abschließend beliebig vielen Fully-Connected-Layern (siehe 2.6). Convolutional-Layer dienen der Featureextraktion. Die Fully-Connected-Layer sind für die Klassifizierung zuständig. Nachfolgend wird das Prinzip einer Convolution (zu dt. Faltung) erläutert.

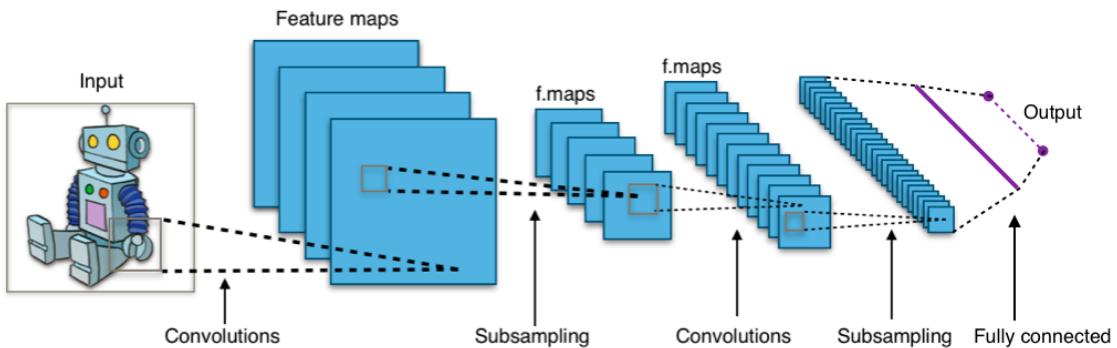


Abbildung 2.6: Beispiel CNN Architektur [Com15]

Ein Convolutional-Layer akzeptiert Daten in Form eines Bildes I in den Dimensionen $W_1 \times H_1 \times C_1$. Optional wird das Bild mit P Nullen an den Rändern aufgefüllt. Über das Bild fahren N Filter der Größe F mit einer Schrittweite S und führen an jeder Stelle ein Skalarprodukt mit den Daten der Bildmatrix durch. Daraus ergibt sich eine neue Matrix mit den Dimensionen $W_2 \times H_2 \times C_2$:

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$
- $C_2 = N$

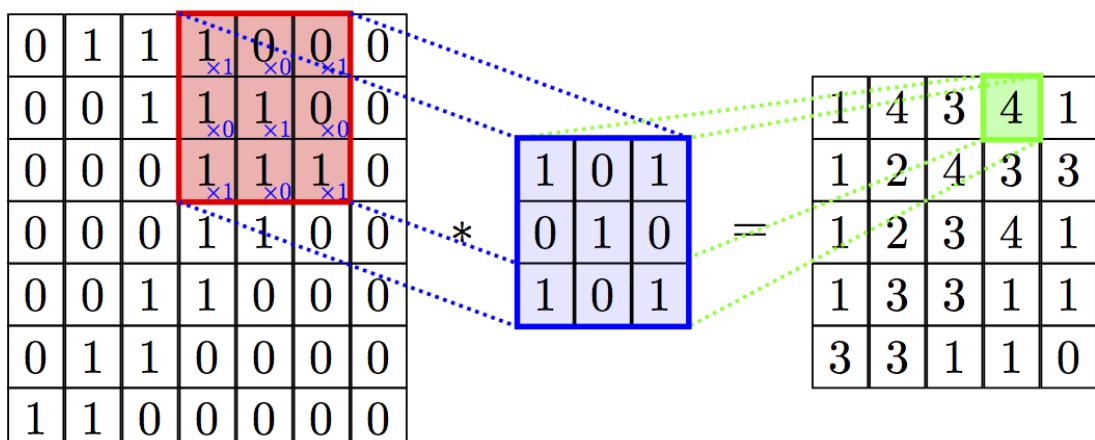


Abbildung 2.7: Berechnungsschritt während einer Convolution [Vel16]

Die Ergebnisse nach Durchlauf eines Convolutional-Layer werden Feature Maps oder Activation Maps genannt. Im trainierten CNN erkennen frühe Convolutional-Layer Kanten, Kurven und feine Muster. Spätere Layer erkennen komplettete geometrische Figuren bis hin zu komplexen Mustern sowie Schemen von realen Objekten und Gegenständen.

2.5 Backpropagation

Um ein Neuronales Netzwerk zu trainieren, muss das Loss, welches durch das Neuronale Netzwerk erzeugt wird, minimiert werden. Eine Funktion berechnet das Loss des Netzwerks. Beispielsweise berechnet sich das MSE-Loss aus der Differenz zwischen y und \hat{y} , wobei y die echten gelabelten Ergebnisse bezeichnet und \hat{y} die Vorhersage, die das Neuronale Netzwerk erzeugt. Anhand dieser Daten kann ein Optimizer, wie beispielsweise Adam [KB15], die Gewichte des Netzwerks anpassen, um das Loss zu minimieren.

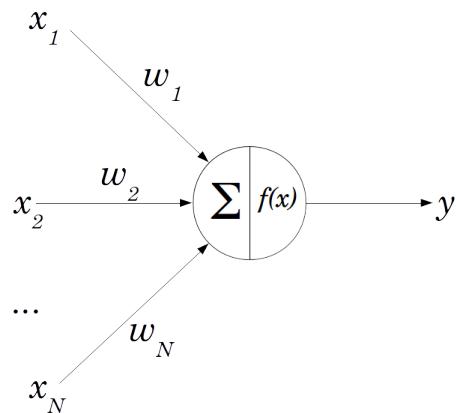


Abbildung 2.8: Visualisierung eines Neurons (auch Perceptron [Gal90] genannt) [Jus16]

Als Beispiel dient die mathematische Funktion eines 2-dimensionalen Neurons mit angehängter Sigmoid-Aktivierungsfunktion. Die Eingabewerte werden durch die Variable x und die Gewichte des Neurons durch die Variable w beschrieben.

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} \quad (2.8)$$

Folgender Graph veranschaulicht die Berechnung der Gradienten der Funktion.

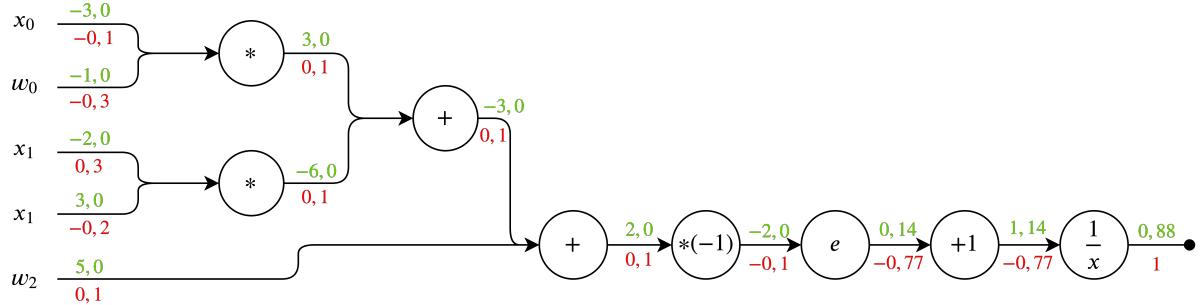


Abbildung 2.9: Forward-Pass in Grün und Backward-Pass in Rot dargestellt [Kar19b]

Die Funktion wird in beliebig viele Unterfunktionen zerteilt und in Form eines gerichteten Graphen aufgeschrieben. In einem Forward-Pass werden die Zwischenergebnisse (hier in Grün dargestellt) oberhalb der Kanten festgehalten.

Im Backward-Pass (Zwischenergebnisse in Rot dargestellt) wird von jedem Knoten die erste Ableitung gebildet. Die Zwischenergebnisse des Forward-Pass werden in die Ableitung eingesetzt und danach mit dem Backward-Passergebnis des Folgeschritts multipliziert.

Beispielrechnungen des Backward-Pass der letzten drei Knoten:

$$f(x) = \frac{1}{x} \rightarrow \frac{\partial f}{\partial x} = \frac{-1}{x^2} \quad -0,77 \approx 1 * \frac{-1}{1,14^2} \quad (2.9)$$

$$f(x) = x + 1 \rightarrow \frac{\partial f}{\partial x} = 1 \quad -0,77 = -0,77 * 1 \quad (2.10)$$

$$f(x) = e^x \rightarrow \frac{\partial f}{\partial x} = e^x \quad -0,1 \approx -0,77 * e^{-2} \quad (2.11)$$

Durch die Berechnung der Gradienten der Gewichtsvariablen w ist der Optimizer in der Lage, w Schritt für Schritt anzupassen und somit das Loss zu verringern.

2.6 Verwandte Arbeiten

Diese Arbeit basiert auf den bereits bestehenden Lösungen von Johnson et. al. und Gatys et al., welche in den folgenden Abschnitten erklärt werden.

2.6.1 Neural Style Transfer

Bei Neural Style Transfer handelt es sich um eine Technik, die den Stil von artistischen Bildern auf ein anderes willkürliche Bild überträgt. Das Verfahren nutzt einen optimierenden Ansatz. Es werden die Pixel eines Eingangsbildes Schritt für Schritt den Pixeln des gewünschten Ausgangsbildes angepasst.

Der Neural Style Transfer Algorithmus, beschrieben im Paper [GEB15], optimiert direkt die Pixel eines Bildes anhand eines sogenannten Perceptual-Loss. Die entsprechende Loss-Funktion besteht aus mehreren Teilen, einem Content-Loss und einem Style-Loss. Optional kann zusätzlich ein Total-Variation-Loss verwendet werden. Das gesamte Perceptual-Loss wird über Hyperparameter konfiguriert, welche die Gewichtung der Unterfunktionen auf das Gesamt-Loss wiederspiegeln.

Content-Loss und Style-Loss benutzen ein bereits auf einem großen Datensatz vortrainiertes Modell. In dieser Arbeit wird das VGG16-Modell [SZ14] verwendet. Das Content-Loss vergleicht Features eines Inhaltsbildes mit den Features des Ausgangsbildes. Das Style-Loss vergleicht Features eines Stilbildes mit den Features des Ausgangsbildes.

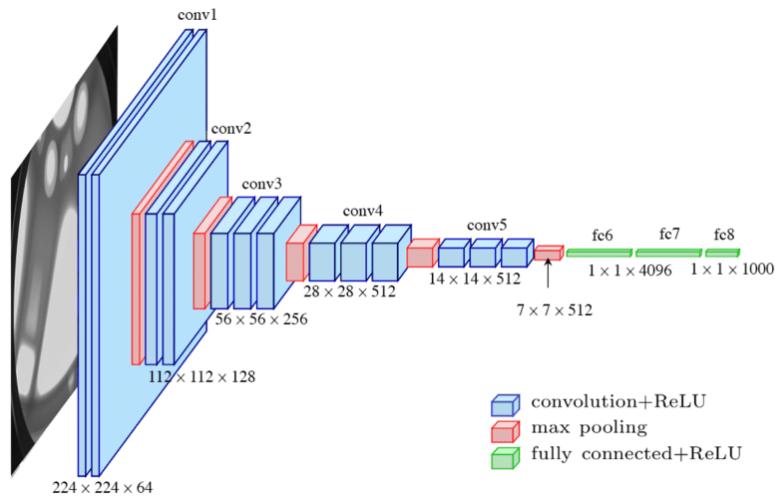


Abbildung 2.10: Architektur des VGG16-Netzwerks [Fer17]

Content-Loss

Um das Content-Loss zu berechnen, wird nicht direkt das Contentbild mit dem Ausgangsbild verglichen, stattdessen werden die Activation Maps nach bestimmten Layern im VGG16-Netzwerk (auch Loss-Network genannt) verwendet. Es ist auch möglich, die Activation Maps mehrerer Layer zu benutzen, da unterschiedliche Layer ein anderes Receptive Field haben und somit unterschiedliche Informationen speichern.

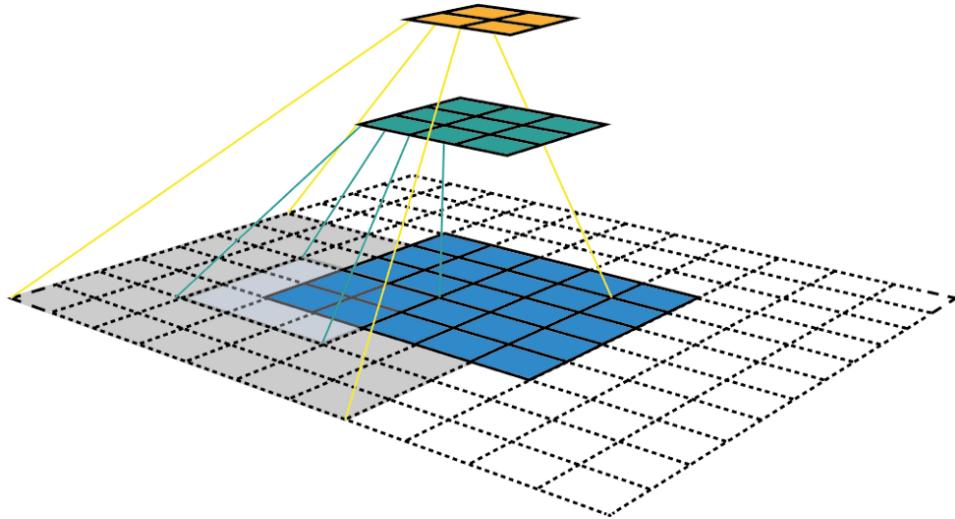


Abbildung 2.11: Visualisierung des Receptive Field über drei CNN-Layer [Hie17]

Zwischen den Activation Maps P des Contentbildes \vec{p} und den Activation Maps F des generierten Bildes \vec{x} wird das MSE-Loss anhand der gegebenen Layer l berechnet.

$$L_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2 \quad (2.12)$$

Style-Loss

Das Style-Loss verwendet die Activation Maps nicht direkt, sondern bildet vorher deren Gram-Matrix. Das hat zur Folge, dass räumliche Informationen verworfen werden, jedoch Farben und Muster erhalten bleiben, die beim Übertragen des Stils signifikant sind.

Um die Gram-Matrix mehrerer Activation Maps zu berechnen, werden die Matrizen geflächt und Height- und Width-Dimension werden zu einer Dimension zusammengefügt. Daraus ergibt

sich eine 2-dimensionale Matrix der Form $C \times HW$. Diese wird transponiert und mit sich selbst multipliziert.

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \quad (2.13)$$

Es wird die Gram-Matrix, G aus dem Stilbild und A aus dem generierten Bild, anhand der gegebenen Layer l berechnet, anschließend normalisiert und das MSE-Loss gebildet.

$$E_l = \frac{1}{4N_l^2 M_l^2} = \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2 \quad (2.14)$$

Das Style-Loss ergibt sich aus der gewichteten Summe der Layer-Losse.

$$L_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^L w_l E_l \quad (2.15)$$

Total-Variation-Loss

Bei der Total-Variation-Loss-Funktion handelt es sich um einen Signal-Denoising-Algorithmus [ROF92; EMS16]. Sie sorgt dafür, dass die Pixel der generierten Bilder sanft in einander überlaufen. Farblich sehr unterschiedliche, nah aneinanderliegende Pixel erzeugen ein hohes Loss. So können Verpixelungseffekte vermieden werden. Das Total-Variation-Loss kann optional eingesetzt werden, um das Resultat der generierten Bilder zu verbessern.

$$L_{tv}(\vec{x}) = \sum_{i,j} |y_{i+1,j} - y_{i,j}| + |y_{i,j+1} - y_{i,j}| \quad (2.16)$$

Perceptual-Loss

Das Perceptual-Loss ist die Kombination aus Content-, Style- und Total-Variation-Loss und wird um die Parameter α , β und γ ergänzt, mit denen die Gewichtung der Losse eingestellt wird.

$$L_{perceptual}(\vec{p}, \vec{a}, \vec{x}) = \alpha L_{content}(\vec{p}, \vec{x}) + \beta L_{style}(\vec{a}, \vec{x}) + \gamma L_{tv}(\vec{x}) \quad (2.17)$$

2.6.2 Fast Neural Style Transfer

Im Gegensatz zum im Kapitel 2.6.1 vorgestellten Algorithmus, in dem die Pixel des Eingangsbildes direkt optimiert werden, wird ein Neuronales Netzwerk darauf trainiert, die Konzepte eines Stils zu erlernen. Dieses Verfahren wird im Paper [JAL16] genauer beschrieben. Pro Stil wird ein Modell trainiert. Die verwendete Loss-Funktion ist das bereits zuvor beschriebene Perceptual-Loss 2.6.1.

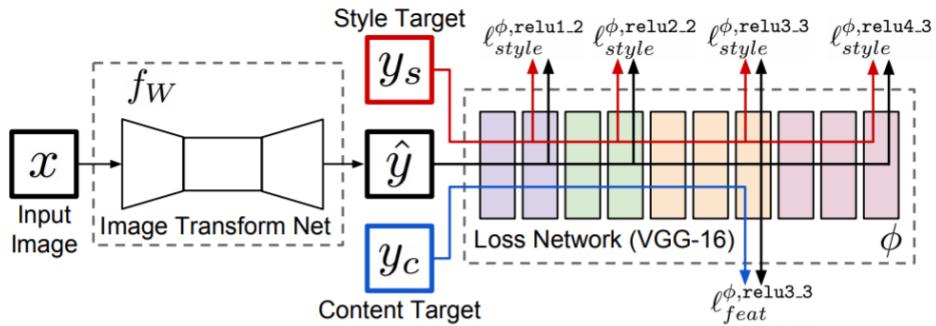


Abbildung 2.12: Trainingsprozess eines Image Transformer Networks [JAL16]

Durch die Verwendung eines Image Transformer Networks wird die Berechnungszeit verbessert, da das neu gestylte Bild mit einem Forward-Pass durch das Netzwerk erstellt wird. Die rechenintensive Optimierung und Berechnung der Pixel-Gradienten durch den Backpropagation-Algorithmus 2.5 entfällt. Das Training eines Image Transformer Networks nimmt jedoch viel Zeit in Anspruch.

Kapitel 3

Methodologie

Das Kapitel Methodologie umfasst das allgemeine Vorgehen, das notwendig ist, um die in Kapitel 2 vorgestellten Algorithmen zu implementieren.

3.1 Vorherige Arbeiten

Bereits vor Erstellung dieser Arbeit gab es Ansätze, den Neural Style Transfer zu realisieren. Diese Arbeit orientiert sich an den bereits bestehenden Entwicklungen und versucht diese zu verbessern sowie auf Geräten mit leistungssarmer Hardware zu testen. Im GIT-Repository von Justin Johnson [Joh15] ist der Neural Style Transfer 2.6.1 auf Basis des Papers [GEB15] mit dem Framework Torch [CKF11] implementiert. Außerdem existiert ein online Tutorial des Frameworks PyTorch [Jac18].

Der Fast Neural Style Transfer 2.6.2 orientiert sich ebenfalls an einem GIT-Repository von Justin Johnson [Joh16], dass den Algorithmus im Framework Torch implementiert. Des weiteren ist er als Beispielalgorithmus im PyTorch GIT-Repository [PyT18] vorhanden.

Neben den vorgestellten Implementierungen existieren viele weitere Implementierungen in unterschiedlichen Frameworks verschiedener Autoren sowie entsprechende Blogeinträge auf beispielsweise <https://towardsdatascience.com/>.

3.2 Neural Style Transfer

Als erster Schritt wird die Loss-Funktion entwickelt. Da das Loss aus drei Teilen besteht, eignet es sich, dieses als Klasse zu realisieren. Style-Loss, Content-Loss sowie Total-Variation-Loss können als einzelne Methoden dieser Klasse implementiert werden und als Gesamt-Loss mit Gewichtung zusammengefasst werden. Außerdem können die notwendigen Hyperparameter des Perceptual-Loss der Klasse bei Instanzierung übergeben werden. Des weiteren werden unterschiedliche Hilfsfunktionen benötigt, um die Gram-Matrix zu berechnen sowie das Laden und Speichern von Bildern und deren Umwandlung in und von Tensoren¹ zu ermöglichen.

Um mathematische Operationen durchzuführen, die bei Berechnung der Gram-Matrix oder des Loss verwendet werden, benötigt man außerdem unterschiedliche Funktionen, die das Rechnen mit Vektoren und Matrizen ermöglichen.

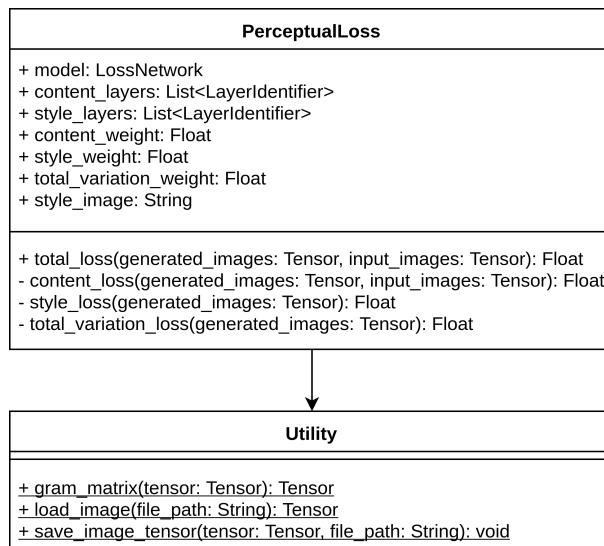


Abbildung 3.1: Komponenten für den Neural Style Algorithmus, eigene Darstellung

Der Algorithmus wird in einer interaktiven Umgebung erstellt, damit unterschiedliche Kombinationen der Hyperparameter schnell verändert und getestet werden können. Folgender Programmablaufplan skizziert ein entsprechendes Script. Für die Berechnung der mathematisch aufwendigen Aufgaben (z.B. Backpropagation 2.5) wird eine entsprechende externe Library, die für diesbezügliche Aufgaben optimiert wurde, benutzt.

¹Als Tensoren werden Matrizen/Vektoren jeglicher Dimensionen bezeichnet

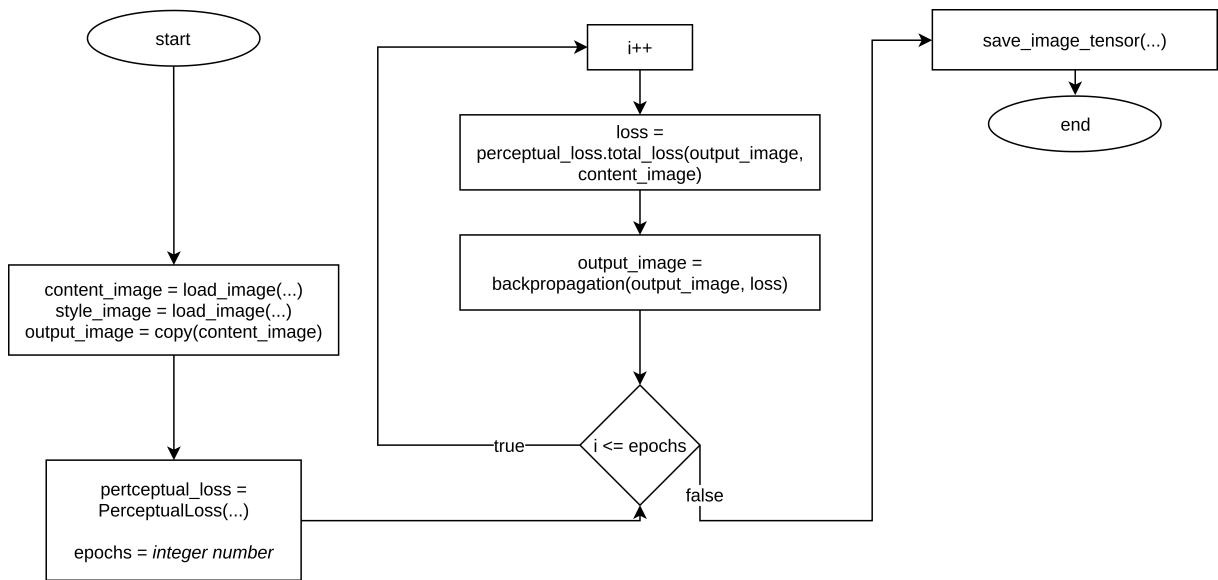


Abbildung 3.2: Programmablaufplan für den Neural Style Algorithmus, eigene Darstellung

Bei Start des Programms werden das Content-Bild `content_image` und das Style-Bild `style_image` geladen. Danach wird eine Kopie von `content_image` in `output_image` erstellt. Die Bilder liegen als Tensoren der Form *Channel* × *Height* × *Width* vor, wobei *Channel* = 3 beträgt, da Bilder im RGB-Farbraum drei Farbkanäle aufweisen.

Es wird ein Objekt der Klasse `PerceptualLoss` erstellt. Bei der Instanziierung werden die folgenden, benötigten Hyperparameter als Argumente dem Konstruktor übergeben:

- `model`: Das Netzwerk, hier VGG16, dessen Activation Maps genutzt werden.
- `content_layers`: Eine Liste von Layern für die Berechnung des Content-Loss.
 $\{relu3_3\}$
- `style_layers`: Eine Liste von Layern für die Berechnung des Style-Loss.
 $\{relu1_2, relu2_2, relu3_3, relu4_3\}$
- `content_weight`: Die Gewichtung des Content-Loss. 1
- `style_weight`: Die Gewichtung des Style-Loss. 1×10^7
- `total_variation_weight`: Die Gewichtung des Total-Variation-Denoising Faktors. 1×10^{-5}
- `style_image`: Der Tensor des gewünschten Style-Bildes.

Die Werte orientieren sich an den im Paper [JAL16] verwendeten Hyperparametern. Je nach gewähltem Stil sind abweichende Einstellungen erforderlich, um optisch ansprechende Ergebnisse zu erzielen. Auf unterschiedliche Kombinationen der Hyperparameter wird im Kapitel 5 eingegangen.

3.3 Fast Neural Style Transfer

Die zuvor konzipierte Loss-Funktion kann beim Fast Neural Style Transfer wieder verwendet werden. Im Gegensatz zum vorherigen Verfahren werden nicht die Pixel als Parameter anhand der Loss-Funktion optimiert, sondern die Gewichtungsparameter eines Neuronalen Netzwerks. Dieses Verfahren wird auch Training genannt. Nach beendeter Trainingsphase ist das Netzwerk in der Lage, willkürliche Bilder in Bilder mit dem mit `style_image` ausgewählten Stil umzuwandeln.

Im Zuge dieser Arbeit soll eine Netzwerkarchitektur gefunden werden, die in der Lage ist, diese Aufgabe zu bewältigen. Außerdem wird die Netzwerkarchitektur im Kapitel 5 auf ihre Performanz auf Geräten mit leistungssarmer Hardware getestet.

3.3.1 Aufbau

Die Netzwerkarchitektur, die beim Fast Neural Style Transfer verwendet wird, nachfolgend Transformer Net genannt, orientiert sich an dem Github-Repository [PyT18], das den Fast Neural Style Transfer implementiert. Sie besteht aus drei Teilen: Einem Down-Sampling-Teil, einem Computational-Teil (auch Bottleneck genannt) und einem Up-Sampling-Teil.

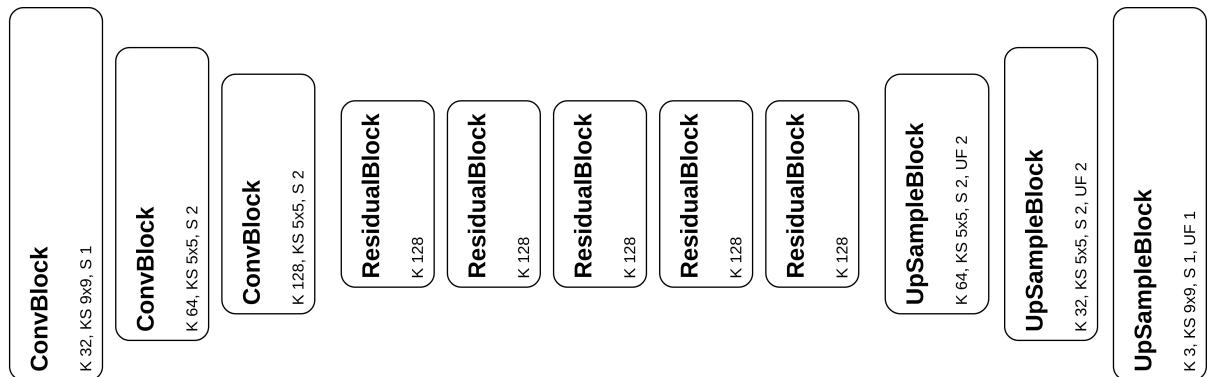


Abbildung 3.3: Netzwerkarchitektur des Transformer Net, eigene Darstellung

Die Abkürzungen in der Grafik stehen für folgende Werte:

- K = Anzahl Kernel
- KS = Kernel Size
- S = Stride
- UF = Upsampling Factor

Im ersten Teil, bestehend aus Convolutional Blöcken, wird durch die Verwendung des Stride Parameters das Eingangsbild hinter jedem Layer verkleinert. Mit einem Stride von 2 wird das Bild ungefähr auf die Hälfte der ursprünglichen Größe verkleinert, vgl. Kapitel 2. Die Anzahl der Channels wird mit jedem Block vergrößert, um den Informationsverlust, der durch die verkleinerten Bilder entsteht, auszugleichen.

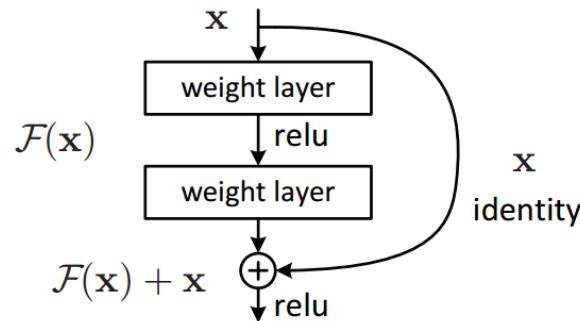


Abbildung 3.4: Aufbau eines Residual Block [He+15a]

Der zweite Teil besteht aus Residual Blöcken, beschrieben in [He+15b]. Diese bestehen aus zwei aufeinanderfolgenden Convolutional Layern, die mit einer sogenannten Skip-Connection mit ihren Eingangsdaten verbunden sind. Das hat den Vorteil, dass die ursprünglichen Eingangsdaten nach jedem Layer noch vorhanden sind. Dies beugt dem Vanishing Gradient Problem vor. In diesem Teil des Netzwerks entsteht die meiste Berechnung und die meisten Daten des Stils werden hier in den Gewichten der Kernels gespeichert.

Im dritten Teil wird das Bild durch Upsampling auf seine ursprünglichen Maße vergrößert. Außerdem werden die Channels auf drei reduziert, um es als RGB-Bild anzeigen zu können. Die Vergrößerung wird durch Nearest Neighbor Interpolation kombiniert mit einem Convolutional Layer realisiert. Es wird zusammen mit seinen Vorteilen in [ODO16] näher beschrieben.

3.3.2 Unterschiedliche Netzwerkgrößen

Wie auf der Grafik 3.3 zu erkennen, ergibt sich die Anzahl der Channels jeweils aus einem Potenzwert der Basis 2 multipliziert mit einem Multiplikator m , in diesem Fall der Wert 32.

$$c_1(m) = m * 2^0 \quad 32 = 32 * x^0 \quad (3.1)$$

$$c_2(m) = m * 2^1 \quad 64 = 32 * x^1 \quad (3.2)$$

$$c_3(m) = m * 2^2 \quad 128 = 32 * x^2 \quad (3.3)$$

Durch den Multiplikator m lässt sich Netzwerkgröße und somit die Anzahl der Parameter einstellen. Unterschiedliche Netzwerkgrößen sind in der Lage, die Stile unterschiedlich gut festzuhalten. Auch wirkt sich die Netzwerkgröße auf die Berechnungs- und Trainingszeit des Netzwerks aus, was unter dem Gesichtspunkt der Verwendung auf Geräten mit leistungssarmer Hardware eine Rolle spielt.

Neben dem Multiplikator gibt es einen weiteren Parameter, der die Netzwerkgröße verändert. Der Parameter s legt die Anzahl der ResidualBlocks im Computational-Teil fest. Im Kapitel 5 werden unterschiedliche Netzwerkgrößen und ihre Performanz getestet.

Kapitel 4

Implementierung

Im Bereich des Machine Learning und Deep Learning gibt es viele verschiedene Frameworks, die bei der Umsetzung und dem Training von Neuronalen Netzwerken behilflich sind. Beispiele hierfür sind Tensorflow¹ von Google, PyTorch² von Facebook und Apache MXNet³. Da bereits im Vorfeld dieser Arbeit Erfahrungen mit PyTorch gesammelt wurden, wurde dieses Framework ausgesucht.

4.1 Loss-Funktion

Im ersten Schritt wird die Loss-Funktion implementiert, die das Kernstück des modellbasierten und des optimierenden Ansatzes ist. PyTorch beinhaltet bereits vortrainierte Versionen verschiedener Modellarchitekturen. In dieser Arbeit wird das VGG16-Modell benutzt, auf dieses wird mit `model = torchvision.models.vgg16(pretrained=True).features` zugegriffen. Das PyTorch-Modell ist in die Bereiche *features* und *classifier* aufgeteilt. Es wird der Bereich *features* benötigt, welcher die bereits vortrainierten Convolutional-Layer enthält. Der Bereich *classifier* enthält die Fully-Connected-Layer des Modells, welche für die Aufgabenstellung ungenutzt bleiben.

¹<https://www.tensorflow.org/>

²<https://pytorch.org/>

³<https://mxnet.apache.org/>

4.1.1 Zugriff auf Hidden-Layer-Ergebnisse

Um bei PyTorch auf Zwischenergebnisse der Hidden-Layer zuzugreifen, werden hinter diesen Hooks implementiert. Das kann in eine Funktion ausgelagert werden, welche die Activation Maps während der Berechnung des Forward-Pass durch das VGG16-Modell extrahiert.

```

1  def extract_activation_maps(image_batch, model, layers, detach=False):
2      class SaveActivationMap:
3          def __init__(self):
4              self.activation = []
5          def hook(self, model, inpt, outpt):
6              self.activation.append(outpt.detach() if detach else outpt)
7
8      sam = SaveActivationMap()
9      handles = []
10
11     for layer in layers:
12         i = list(dict(model.named_children()).keys()).index(str(layer))
13         handles.append(model[i].register_forward_hook(sam.hook))
14     model(image_batch)
15
16     for handle in handles:
17         handle.remove()
18
19     return sam.activation

```

Code snippet 4.1: Extrahierung der Activation-Maps mit PyTorch

4.1.2 Content-Loss

Um das Content-Loss zu bilden, wird mit der zuvor definierten Funktion auf die Zwischenergebnisse der Hidden-Layer zugegriffen. Das Ergebnis ist das MSE-Loss aller Zwischenergebnisse der Eingangsdaten (Inhaltsbild) verglichen mit den Zwischenergebnissen der Ausgangsdaten (generierte Bilder), vgl. Kapitel 2.6.1.

```

1  def content_loss(self, y, x):
2      y = extract_activation_maps(y, self.model, self.c_layers)
3      x = extract_activation_maps(x, self.model, self.c_layers, detach=True)
4      return sum([
5          F.mse_loss(generated_feature, input_feature)
6          for generated_feature, input_feature in zip(y, x)
7      ])

```

Code snippet 4.2: Berechnung des Content-Loss, vgl. Gleichung (2.12)

4.1.3 Style-Loss

Für das Style-Loss wird eine Funktion benötigt, um die Gram-Matrix eines Tensors zu bilden. Die Anzahl der Dimensionen des Eingangstensors wird auf zwei Dimensionen geflächtet. Diese Matrix wird mit einer von sich selbst transponierten Version multipliziert. Danach wird das Ergebnis über das Produkt der Anzahl der Channel, Höhe und Breite des Eingangstensors normalisiert.

```

1 def gram_matrix(tensor):
2     b, c, h, w = tensor.shape
3     normalizer = c * h * w
4     tensor_flat = tensor.flatten(2)
5
6     return torch.div(
7         torch.bmm(tensor_flat, tensor_flat.transpose(1, 2)),
8         normalizer
9     )

```

Code snippet 4.3: Berechnung der Gram-Matrix, vgl. Gleichungen (2.13) u. (2.14)

Die Gram-Matrix-Funktion wird dazu benutzt, das Style-Loss zu berechnen. Wie beim Content-Loss wird das MSE-Loss gebildet. Dazu werden vorher die Gram-Matrizen aller Activation Maps berechnet. Auf diese Weise wird das berechnete Ausgangsbild mit dem vorher eingestellten Stilbild verglichen. In dieser Implementierung wurde die Gewichtung der einzelnen Layer nicht berücksichtigt. Die Implementierung weicht daher geringfügig von der in den Grundlagen verwendeten Gleichung (2.15) ab.

```

1 def style_loss(self, y):
2     y = extract_activation_maps(y, self.model, self.s_layers)
3     y_grams = [gram_matrix(row) for row in y]
4
5     return sum([
6         F.mse_loss(y_gram, style_gram)
7         for y_gram, style_gram in zip(y_grams, self.style_grams)
8     ])

```

Code snippet 4.4: Berechnung des Style-Loss, vgl. Gleichung (2.15)

4.1.4 Total-Variation-Loss

Das Total-Variation-Loss berechnet die Summe der Abweichungen eines Pixels zum nächsten Pixel eines Bildes. Dies wird für Höhe und Breite des Bildes durchgeführt und addiert.

```

1 def total_variation_loss(self, y):
2     return torch.add(
3         torch.sum(torch.abs(y[:, :, :, :-1] - y[:, :, :, 1:])),
4         torch.sum(torch.abs(y[:, :, :-1, :] - y[:, :, 1:, :]))
5     )

```

Code snippet 4.5: Berechnung des Total-Variation-Loss [Lee17], vgl. Gleichung (2.16)

4.1.5 Perceptual-Loss

Das Perceptual-Loss ist das Gesamt-Loss und addiert die einzelnen Teil-Losse mit entsprechend einstellbaren Gewichtungsfaktoren.

```

1 def forward(self, y, x):
2     content_loss = self.c_weight * self.content_loss(y, x)
3     style_loss = self.s_weight * self.style_loss(y)
4     total_variation_loss = self.tv_weight * self.total_variation_loss(y)
5
6     loss = content_loss + style_loss + total_variation_loss
7
8     return loss

```

Code snippet 4.6: Berechnung des gesamten Perceptual-Loss, vgl. Gleichung (2.17)

4.2 Neural Style Transfer

Die Implementierung des Neural Style Transfer Algorithmus wird in einem Jupyter Notebook⁴ durchgeführt. Dadurch können im späteren Verlauf unterschiedliche Tests durchgeführt werden, da die Hyperparameter schnell angepasst werden können.

Das Notebook, vgl. A.1, orientiert sich am zuvor erstellten Programmablaufplan, vgl. Abbildung 3.2. Es bietet außerdem die Auswahl zwischen den PyTorch-Optimizern Adam und L-BFGS [LN89]. Folgender Code zeigt die Optimierung von Parametern mit PyTorch.

⁴<https://jupyter.org/>

```
1 criterion = PerceptualLoss(...)
2 optimizer = optim.Adam([outputs])
3
4 for epoch in range(epochs):
5     optimizer.zero_grad()
6
7     loss = criterion(outputs, inputs)
8     loss.backward()
9
10    optimizer.step()
```

Code snippet 4.7: Vereinfachter Code einer Trainingsschleife in PyTorch

4.3 Fast Neural Style Transfer

Das Verfahren des Fast Neural Style funktioniert ähnlich wie beim vorher vorgestellten Neural Style Transfer. Anstatt der Pixel des Ausgangsbildes werden jedoch die Gewichte eines Neuronalen Netzwerks optimiert. Der bestehende Code kann aus dem vorherigen Kapitel wiederverwendet werden. Lediglich die Initialisierung des PyTorch Optimizers muss angepasst werden.

```
1 model = TransformerNet(...)
2 optimizer = optim.Adam(model.parameters())
```

Code snippet 4.8: Optimierung der Gewichte des Netzwerks

4.3.1 COCO-Datensatz

Um das Neuronale Netzwerk zu trainieren, wird ein großer Bilddatensatz benötigt. In dieser Arbeit wird dafür der COCO-Datensatz der Firma Microsoft verwendet [Lin+14]. Es handelt sich um einen öffentlichen, frei verfügbaren Datensatz vieler verschiedener Bilder unterschiedlicher Kategorien. Während des Trainingsverlaufs werden zufällige Bilder aus dem COCO-Datensatz genutzt und das Loss (Style-, Content- und Total-Variation-Loss) über die Ausgaben, die das Netzwerk generiert, berechnet. Mit dem berechneten Loss ist PyTorch in der Lage, die Gradienten der Gewichte des Neuronalen Netzwerks zu berechnen und diese schrittweise zu optimieren.

Der COCO-Datensatz liegt in Form vieler verschiedener JPEG-Bilder vor. Um nicht alle Bilder gleichzeitig laden zu müssen, bietet PyTorch die Möglichkeit, einen Dataloader⁵ zu nutzen, der die Bilder in der gewünschten Batchgröße Schritt für Schritt lädt.

⁵Weitere Information zum PyTorch-Dataloader: https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

```
1  dataset = datasets.ImageFolder(
2      config['dataset_path'],
3      transform=transforms.Compose([
4          transforms.Resize(config['content_image_size']),
5          transforms.CenterCrop(config['content_image_size']),
6          transforms.ToTensor()
7      ])
8  )
9
10 dataloader = data.DataLoader(dataset, batch_size=config['batch_size'], shuffle=True)
```

Code snippet 4.9: Der PyTorch-Dataloader lädt Bilder in der gewünschten Batchgröße

Der Dataloader lädt alle Bilder in allen Unterordnern innerhalb des konfigurierten Pfades.

4.3.2 Netzwerkarchitektur

Die Architektur des Neuronalen Netzes spielt eine wichtige Rolle. Sie muss in der Lage sein, die Eigenschaften eines Stils möglichst gut zu erlernen und gleichzeitig performant bleiben. Daher wurde die in Kapitel 2.6.2 vorgestellte Netzwerkarchitektur des Image Transformer Networks implementiert. Dies geschieht in PyTorch in Form einer Klasse, welche von `Module` erbt.

Um in Kapitel 5 das Testen unterschiedlicher Netzwerkarchitekturen zu ermöglichen, wird die Klasse des Image Transformer Networks möglichst dynamisch implementiert. Einstellbar sind folgende Parameter, die im weiteren Verlauf erklärt werden:

- channel_multiplier
- bottleneck_size
- bottleneck_type
- final_activation_fn
- intermediate_activation_fn

4.3.3 Aktivierungsfunktionen

Für den dynamischen Aufbau des Netzwerks wird eine Funktion benötigt, die entsprechende PyTorch-Aktivierungsfunktionen zurückgibt. Hardtanh und Sigmoid kommen als finale Aktivierungsfunktionen des Netzwerks in Betracht, da sie Werte zwischen 0.0 und 1.0 zurückliefern. Das Ergebnis eignet sich für die Generierung von Bildern, welche in PyTorch ebenfalls Pixelwerte

zwischen 0.0 und 1.0 besitzen. Zu beachten ist, dass die Hardtanh-Funktion mit den Parametern `min_val=0.0` und `max_val=1.0` initialisiert wird, die den Wertebereich verschieben. Mit den Standardeinstellungen liegt der Wertebereich der Hardtanh-Funktion zwischen -1.0 und 1.0. Die Parameter `final_activation_fn` und `intermediate_activation_fn` werden auf die entsprechenden PyTorch-Aktivierungsfunktionen überführt.

4.3.4 ConvBlock

Des weiteren müssen die in Kapitel 3.3.1 konzipierten Layer-Blöcke implementiert werden. Der ConvBlock besteht aus den PyTorch-Layern `conv2d`, `InstanceNorm2d` und der gewählten Aktivierungsfunktion. `InstanceNorm2d` wurde aufgrund des Papers [UVL16] gewählt und dort näher in Bezug auf die Verwendung mit Style Transfer-Methoden erläutert.

Die Parameter `in_channels` und `out_channels` ergeben sich aus dem Parameter `channel_multiplier` (m) multipliziert mit einem festen Wert, der je nach Layer die Werte 1, 2 oder 4 beträgt.

4.3.5 ResidualBlock

Der ResidualBlock, vgl. 3.4, besteht aus zwei aufeinanderfolgenden `conv2d`-Layern. Wie beim vorherigen ConvBlock werden die Daten über `InstanceNorm2d` normalisiert. Abschließend wird die Aktivierungsfunktion angewendet. Die Besonderheit ist, dass die Eingangsdaten kopiert und danach mit den berechneten Daten der Convolutional-Layer addiert werden. Durch `conv2d` verkleinert sich die Dimension der Daten, vgl. 2.4, was durch den `ReflectionPad2d`-Layer wieder ausgeglichen wird. Eingangsdaten und berechnete Daten müssen die gleichen Dimensionen aufweisen, um addiert werden zu können. Die Parameter `in_channels` und `out_channels` ergeben sich aus `channel_multiplier` multipliziert mit 4.

4.3.6 UpSampleBlock

Der UpSampleBlock schaltet dem ConvBlock ein Upsampling-Verfahren vor, um das Bild um einen entsprechenden Faktor zu vergrößern. In dieser Arbeit wird Nearest Neighbor Interpolation verwendet.

4.3.7 TransformerNet

Neben den Layer-Blöcken wurde noch die eigentliche Netzwerkarchitektur implementiert. Die Implementierung nimmt die in 4.3.2 beschriebenen Parameter im Konstruktor entgegen, um

eine entsprechende Netzwerkarchitektur zu generieren. Neben den bereits beschriebenen Einstellungen gibt `bottleneck_size` die Anzahl der hintereinander geschalteten ResidualBlocks an. Der `bottleneck_type` ist für zukünftige Erweiterungen vorgesehen, damit der ResidualBlock durch einen anders aufgebauten Block ersetzt werden kann. Außerdem wurde ein anfängliches Padding eingefügt, um die ursprüngliche Bildgröße beizubehalten.

4.3.8 Weitere Implementierungen

Zusätzlich zu den vorgestellten Implementierungen wurden Hilfsskripte und Jupyter Notebooks zum Testen erstellt. Außerdem wurde eine Trainer-Klasse entworfen, die in der Lage ist, Einstellungen für Trainingsläufe aus einer Konfigurationsdatei zu lesen und somit ein aufeinanderfolgendes Training mehrerer Modelle zu ermöglichen. Auf diese Art kann man viele unterschiedliche Parametereinstellungen testen.

Zur Realisierung eines Prototyps wurde eine JSON-API mit dem Framework Flask⁶ erstellt. Die API stellt einen Endpunkt zur Verfügung, welcher den Style Transfer mit einem gewünschten Modell durchführt. Sie wurde in einen Docker⁷-Container überführt, um das Hosting auf unterschiedlichen Geräten zu erleichtern.

Um die Ergebnisse zu visualisieren, wurde eine Frontend-Applikation mit dem Framework Angular⁸ erstellt. Diese kann Bilder über die Webcam eines Handys oder eines Computers aufnehmen und sie an die Flask-API senden, die nachfolgend das aufgenommene Bild in einen vorher ausgewählten Stil überführt.

Skripte, Frontend-Applikation und JSON-API sind nicht Bestandteil dieser Arbeit und befinden sich in einem prototypischen Zustand. Der komplette Code zu diesem Projekt ist unter <https://github.com/christophstach/style-transfer-project> öffentlich einsehbar. Der Code des Frontends unter <https://github.com/christophstach/style-transfer-frontend>. Ein gehosteter Prototyp zum Testen des Style Transfers ist unter <https://stylized.christophstach.me/> verfügbar.

⁶Flask: <http://flask.pocoo.org/>

⁷Docker: <https://www.docker.com/>

⁸Angular: <https://angular.io/>

Kapitel 5

Tests und Experimente

Im Kapitel Tests und Experimente werden unterschiedliche Hyperparameter-Konfigurationen auf ihre Auswirkungen getestet. Außerdem wird mit unterschiedlichen Netzwerkarchitekturen die Performanz auf Geräten mit leistungssarmer Hardware gemessen.

5.1 Auswirkungen der Hyperparameter

Im Folgenden werden unterschiedliche Kombinationen der Gewichtungsparameter Content-Weight α , Style-Weight β und Total-Variation-Weight γ getestet. Dabei wird als Inhaltsbild eine eigene Abbildung der HTW-Berlin benutzt. Als Stilbild wird das Kunstwerk *The Starry Night* des Künstlers Vincent Van Gogh und *The Scream* von Edvard Munch verwendet. Die generierten Bilder entsprechen einer Größe von 768 * 768 Pixeln. Die restlichen Hyperparameter werden aus dem Kapitel Methodologie 3.2 übernommen.

5.1.1 Experiment 1: Starry Night

In diesem Experiment wird das Stilbild *The Starry Night* von Vincent Van Gogh verwendet.



Abbildung 5.1: HTW kombiniert mit *The Starry Night* [Nig89]

In der ersten Abbildungen werden die verschiedenen Stilgewichtungen $\beta = 10^6$, $\beta = 10^7$, $\beta = 10^8$ und $\beta = 10^9$ für *The Starry Night* getestet.



Abbildung 5.2: *The Starry Night* mit $\alpha = 1$, $\beta = 10^6 - 10^9$, $\gamma = 0$

In der zweiten Abbildungen werden die verschiedenen Total-Variation-Gewichtungen $\gamma = 10^{-6}$, $\gamma = 10^{-5}$, $\gamma = 10^{-4}$ und $\gamma = 10^{-3}$ für Starry Night getestet.



Abbildung 5.3: Starry Night mit $\alpha = 1$, $\beta = 10^8$, $\gamma = 10^{-6} - 10^{-3}$

Die Druckversion entspricht nicht der digitalen Qualität der Bilder.

5.1.2 Experiment 2: The Scream



Abbildung 5.4: HTW kombiniert mit *The Scream* [Høs93]

In der ersten Abbildungen werden die verschiedene Stilgewichtungen $\beta = 10^6$, $\beta = 10^7$, $\beta = 10^8$ und $\beta = 10^9$ für *The Scream* getestet.

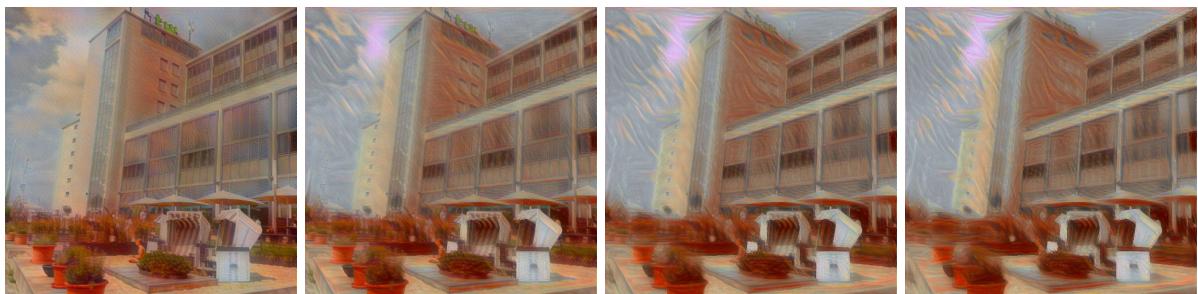


Abbildung 5.5: *The Scream* mit $\alpha = 1$, $\beta = 10^6 - 10^9$, $\gamma = 0$

In der zweiten Abbildungen werden die verschiedenen Total-Variation-Gewichtungen $\gamma = 10^{-6}$, $\gamma = 10^{-5}$, $\gamma = 10^{-4}$ und $\gamma = 10^{-3}$ für *The Scream* getestet.



Abbildung 5.6: *The Scream* mit $\alpha = 1$, $\beta = 10^7$, $\gamma = 10^{-6} - 10^{-3}$

Die Druckversion entspricht nicht der digitalen Qualität der Bilder.

5.1.3 Interpretation der Ergebnisse

Wie bei beiden Tests zu sehen ist, werden die Muster mit zunehmendem Style-Weight β auf dem Ausgangsbild immer stärker generiert. Mit $b = 10^5$ sind bei beiden Stilen die Muster des Gemäldes kaum noch zu erkennen. Lediglich wird das Ausgangsbild den Farben des Stilbildes angepasst. Ein optisch ansprechender Effekt ergibt sich bei beiden Stilen ab $\beta = 10^7$.

Mit zunehmendem Total-Variation-Weight γ fließen die Farben mehr ineinander und das Ausgangsbild wird verschwommener. Das ist besonders gut zu sehen bei *The Starry Night* mit $\gamma = 10^{-3}$ und *The Scream* mit $\gamma = 10^{-4}$.

5.2 Auswirkungen der Netzwerkarchitekturen

In diesem Experiment wurden drei unterschiedliche Bildgrößen auf verschiedenen Geräten auf Durchführbarkeit und Performanz getestet. Gemessen wird die Berechnungsgeschwindigkeit beim Forward-Pass durch das Netzwerk. Ein bereits erster Indikator für die Performanz eines Neuronalen Netzwerks ist die Anzahl der lernbaren Parameter, die bei der Trainingsphase optimiert werden. Erstellt wurden 16 unterschiedliche Netzwerke mit verschiedenen Kombinationen für m und s , vgl. B.1.

5.2.1 Verwendete Geräte

Bei der Berechnung des Forward-Pass macht es einen Unterschied, ob sie auf dem CPU oder dem GPU eines Geräts durchgeführt wird. Neuronale Netzwerke können auf einer GPU schneller als auf einem CPU berechnet werden. Das liegt daran, dass eine GPU besonders auf die Berechnung von Matrix-Operationen (wie sie auch bei grafischen Anwendungen benutzt werden) spezialisiert ist.

Die Daten werden beim Einsatz des GPUs in den Grafikspeicher geladen. Bei der Berechnung über den CPU wird der Arbeitsspeicher (RAM) des Geräts verwendet.

Gerät 1: Dell XPS 15 9550

Beim Dell XPS 15 9550 handelt es sich um einen handelsüblichen Laptop aus dem Jahr 2016.

Prozessor	Intel Core i7-6700HQ
Grafikkarte	NVIDIA® GeForce™ GTX 960M (2GB GDDR5)
Arbeitsspeicher	16GB DDR4
Festplatte	512GB SSD

Tabelle 5.1: Spezifikation: Dell XPS 15 9550

Gerät 2: Jetson TX2

Beim Jetson TX2 handelt es sich um ein leistungsarmes Gerät für die Realisierung von Algorithmen der künstlichen Intelligenz auf eingebetteten Systemen.

Prozessor	Dual-Core NVIDIA Denver 2 64-Bit CPU, Quad-Core ARM®Cortex®-A57 MPCore
Grafikkarte	256-core NVIDIA Maxwell™ GPU
Arbeitsspeicher	8GB 128-bit LPDDR4 Memory
Festplatte	32GB eMMC 5.1

Tabelle 5.2: Spezifikation: Jetson TX2

5.2.2 Durchführung der Experimente

Alle Netzwerke wurden mit einer Batch-Size von 24, Bildgrößen von 224 * 224 Pixeln und einer Learning-Rate von 10^{-3} trainiert. Es wurden 6 Epochen der Trainingsdaten aus dem Jahr 2017 des COCO-Datensatzes verwendet, welche 118287 Bilder enthalten. Insgesamt wurden die Modelle mit 709722 Bildern trainiert. Die Trainingszeit lag auf einem Server mit einem GeForce GTX 1080 Ti GPU pro Modell zwischen 5 und 10 Stunden.

Beim Training fiel auf, dass die generierten Bilder großflächige, schwarze oder weiße Artefakte enthielten. Ein Austausch der finalen Aktivierungsfunktion HardTanh, vgl. 2.3.2, mit der Sigmoid-Aktivierungsfunktion, vgl. 2.3.3, konnte das Problem lösen.

Abschließend wurden die unterschiedlichen Netzwerkarchitekturen auf ihre Performanz getestet. Tests, die wegen unzureichendem Arbeitsspeicher oder Grafikspeicher fehlschlugen, wurden als **nicht durchführbar** gekennzeichnet. Um die Tests durchzuführen, wurden Bilder der Größen 1920 * 1080 Pixel, 1024 * 786 Pixel und 640 * 480 Pixel jeweils 10 mal mit jedem Netzwerk berechnet, die Berechnungsgeschwindigkeit gemessen, und das arithmetische Mittel der 10 Durchläufe gebildet. Die Ergebnisse sind in den Anlagen B.2, B.3 und B.4 angefügt.

5.2.3 Interpretation der Ergebnisse

Die Performanz-Tests haben ergeben, dass der Jetson TX2 mit fast allen Netzwerkarchitekturen in der Lage war, den Style-Transfer durchzuführen. Für Bilder der Größe 1920 * 768 Pixel war es jedoch nicht möglich, Netzwerk 1 und Netzwerk 5 zu verwenden, vgl. B.2. Alle anderen Netzwerkarchitekturen konnten für Full HD Bilder in weniger als einer Sekunde berechnet werden.

Die Abbildungen B.1 und B.2 stellen die Loss-Kurven beim Trainingsverlauf aller Netzwerke dar. Die Loss-Kurven wurden mit einem laufenden arithmetischen Mittel über 100 Iterationen geflacht, um die Ergebnisse besser darstellen zu können. Für beide Stilbilder ergibt sich ein ähnlicher Trainingsverlauf. Die Netzwerkarchitekturen mit mehr lernbaren Parametern erreichen ein niedrigeres Endloss. Eine Ausnahme bilden die Netzwerke mit dem Parameter $m = 16$. Sie erreichen in allen Fällen das niedrigste Endloss. In den Abbildungen sind die Loss-Kurven dieser Netzwerke mit einer stärkeren Linie gekennzeichnet.

Des weiteren wurden die optischen Ergebnisse aller Netzwerkarchitekturen getestet. Die Ergebnisse sind in B.3 und B.4 dargestellt. Wieder zeigt sich, dass Netzwerke mit dem Parameter $m = 16$ optisch vielfältigere Bilder generieren als die anderen Netzwerke. Daraus wird geschlussfolgert, dass die optische Vielfalt der generierten Bilder mit dem Loss zusammenhängt.

5.2.4 Weitere Ergebnisse

Auf Basis der zuvor gewonnenen Erkenntnis wurden weitere Modelle mit unterschiedlichen Stilbildern trainiert. Für das Training wurde jeweils die Netzwerkarchitektur 2, $m = 16$ und $s = 5$, verwendet, die in den durchgeführten Tests am besten abschnitt. Die Abbildungen B.5, B.6 und B.7 zeigen Beispiele erfolgreich trainierter Stilmodelle.

Kapitel 6

Evaluation

Im Kapitel Evaluation werden die restlichen Kapitel dieser Arbeit beleuchtet und kritisch betrachtet. Dabei wird auf die Kapitel Grundlagen 2, Methodologie 3, Implementierung 4 und Tests und Experimente 5 eingegangen.

6.1 Grundlagen

Im Kapitel Grundlagen 2 werden die Grundlagen von Neuronalen Netzwerken erläutert. Besonders der Backpropagation-Algorithmus 2.5 und die Beschreibung sowie tieferes Auseinandersetzen mit Convolutions 2.4 waren für das Verständnis der Problemstellung hilfreich.

Anhand der Paper von Gatys [GEB15] und Johnson [JAL16] konnte die Problemstellung des Style Transfers nachvollzogen werden. Die in den Papern verwendete Loss-Funktion ist bei beiden die Gleiche. Jedoch werden unterschiedliche Notationen verwendet. Die Loss-Funktion konnte für das Trainieren eines Neuronalen Netzwerks erneut gebraucht werden. Das Paper [JAL16] geht jedoch nicht auf die genaue Architektur des verwendeten Netzwerks ein. Hierzu war es hilfreich, die offizielle Implementierung [Joh16] und die Referenz-Implementierungen im PyTorch-Repository [Jac18] zu analysieren.

Das Konzept von Total-Variation-Denoising wurde in den Referenz-Implementierungen eingebaut, ist dort jedoch unzureichend beschrieben. Eine detaillierte Beschreibung ist in den Papern [ROF92; EMS16] vorhanden.

6.2 Methodologie

Im Kapitel 3 wurde die Vorgehensweise der geplanten Implementierung beschrieben. Das bereits in [JAL16] dargestellte Konzept des Image Transformer Networks wird um die Parameter der Bottleneck-Size s und Channel-Multiplikator m erweitert. Dadurch können beliebig viele

Netzwerkarchitekturen dynamisch erstellt werden und es wurde die Grundlage für weitreichende Performanz-Tests geschaffen.

Das Konzept des ResidualBlocks ist dahingehend interessant, da es auch für andere Problemstellungen verwendet werden kann. Der Effekt des Vanishing Gradient kann mit ihm bei jeglicher Art von Convolutional-Neural-Network verringert werden und ein effizientes Training der Netzwerke erleichtern.

6.3 Implementierung

Bei der Implementierung der Netzwerkarchitektur erwies es sich als vorteilhaft, diese möglichst dynamisch aufzubauen. Die ursprünglich geplante, abschließende Aktivierungs-Funktion Hardtanh 2.3.2 konnte somit problemlos durch die Sigmoid-Aktivierungs-Funktion 2.3.3 ausgetauscht werden. Es wurde außerdem die Möglichkeit geplant, die im Bottleneck-Teil des Netzwerks verwendeten ResidualBlocks durch andere Arten von Blöcken auszutauschen.

Das in 3.2 entworfene Skript zur Durchführung des Neural-Style-Algorithmus konnte zum Training der Netzwerke nicht wiederverwendet werden. Hierzu wurde eine Trainer-Klasse entworfen, die weitreichende Konfigurationsmöglichkeiten bietet und das aufeinanderfolgende Training mehrerer Netzwerke erleichtert.

6.4 Tests und Experimente

Im Kapitel Tests und Experimente 5 wurden verschiedene Tests mit unterschiedlichen Hyperparametereinstellungen durchgeführt. Es wurde demonstriert, wie sich verschiedene Content-, Style- und Total-Variation-Loss-Gewichtungen zueinander verhalten. Style Transfer im Allgemeinen hat weitere Hyperparameter, welche bei den durchgeführten Tests nicht berücksichtigt wurden.

Außerdem wurden verschiedene Tests hinsichtlich der Netzwerkarchitektur und Performanz durchgeführt. Es wurde gezeigt, dass auch Netzwerkarchitekturen mit weniger lernbaren Parametern in der Lage sein können, Stile zu erlernen. Zudem wurde gezeigt, dass die gewählten Netzwerkarchitekturen größtenteils auf Geräten mit leistungssarmer Hardware zu berechnen sind.

Die im Laufe der Experimente trainierten Modelle wurden alle ohne die Verwendung des Total-Variation-Loss erstellt. Mehr Experimente könnten zukünftig die Qualität der Modelle verbessern, indem eine passende Gewichtung für das Total-Variation-Loss gefunden wird. Diese müsste für jeden Stil durch empirische Beobachtungen herausgefunden werden.

Kapitel 7

Fazit

Dieses Kapitel beinhaltet eine kurze Zusammenfassung der Abschlussarbeit. Das Thema wird kritisch betrachtet und es wird auf mögliche Probleme eingegangen. Letztlich wird ein Ausblick über zukünftige Entwicklungen und Möglichkeiten gegeben.

7.1 Zusammenfassung

In der Abschlussarbeit wurde sich intensiv mit Style Transfer Algorithmen beschäftigt und diese auf Geräten mit leistungssarmer Hardware getestet. Anfangs wurde die Theorie zu Neuronalen Netzwerken und Style Transfer analysiert. Es wurde eine Implementierung geplant und abschließend mit dem Framework PyTorch umgesetzt. Unterschiedliche Netzwerke wurden erfolgreich trainiert und Stile extrahiert. Verschiedene Hyperparametereinstellungen wurden getestet und die Performanz der Netzwerke gemessen.

7.2 Kritischer Rückblick

Besonders das Training und das Durchführen der Experimente muss kritisch betrachtet werden. Es wurden die entsprechenden Hyperparametereinstellungen ausführlich nur mit zwei Gemälden getestet. Fraglich ist ob, diese den gleichen Effekt bei anderen Gemälden erzielen. Deswegen wird davon ausgegangen, dass für jeden Stil die Hyperparametereinstellungen individuell getestet werden müssen.

Bei den gewählten Gemälden waren alle Netzwerkarchitekturen in der Lage, den Stil zu extrahieren. Daraus kann geschlussfolgert werden, dass möglicherweise sogar Netzwerkarchitekturen mit weniger lernbaren Parametern, als die hier vorgeschlagenen, in der Lage sein könnten, Stile aus Gemälden zu erlernen. Bei weiteren Experimenten wäre man unter Umständen in der Lage gewesen, noch performantere Netzwerkarchitekturen zu finden.

Die Auswahl der verwendeten Gemälde in dieser Arbeit geschah unter besonderer Berücksichtigung des Urheberschutzes. Ein kommerzieller Einsatz wäre nicht in jedem Fall möglich. Der Gebrauch von Style Transfer Methoden wirft besondere urheberrechtliche Bedenken auf. Urheberrechtlich geschützte Bilder können dazu verwendet werden, Stile zu extrahieren und auf neue Bilder zu übertragen. Die in Bezug auf Urheberrecht aufkommenden Fragen müssen für den produktiven Einsatz eines Software-Systems evaluiert werden.

7.3 Ausblick

Nachfolgend wird ein Ausblick auf Erweiterungen des Algorithmus gewährt. Es wird auf vorstellbare Funktionen und Einstellungen eingegangen.

7.3.1 Verwendung anderer Layer-Kombinationen

In dieser Arbeit wurden für die Berechnung des Style-Loss die Layer relu1_2, relu2_2, relu3_3, relu4_3 und für die Berechnung des Content-Loss der Layer relu3_3 des VGG16-Modells verwendet. Man kann ein anderes Loss-Network und andere Kombinationen von Layern für die Berechnung des Loss benutzen. Das hätte zur Folge, dass die generierten Stile sich unterscheiden würden. Im GitHub-Repository von Logan Engstrom [Eng16] wurde dies bereits implementiert.

7.3.2 Kombination aus mehreren Stilen

Vorstellbar ist eine Vermischung von mehreren Stilen. Hierzu müsste man das Style-Loss aus den Gram-Matrizen mehrerer Gemälde kombinieren. Im Paper [MH17] wurde dies und andere Erweiterungen beschrieben.

7.3.3 Superresolution

Neben Style Transfer kann eine ähnliche Methodik für einen Super-Resolution-Algorithmus verwendet werden. Dieser wird ebenfalls im Paper von Johnson et al. [JAL16] beschrieben. Es muss eine Netzwerkarchitektur verwendet werden, die in der Lage ist, Bilder auf eine höhere Auflösung zu skalieren. Die verwendete Loss-Funktion wäre das bereits vorgestellte Perceptual-Loss jedoch ohne den Style-Loss-Anteil.

7.3.4 Alternativen zum ResidualBlock

Hinsichtlich der Performanz können weitere Experimente mit kleineren Netzwerkarchitekturen durchgeführt werden. Außerdem kann der ResidualBlock durch eine andere Art von Block ersetzt werden. Die Paper [How+17] und [San+18] behandeln sogenannte MobileNets, welche besonders auf die Verwendung von Geräten mit leistungssarmer Hardware abzielen. Vorstellbar wäre es, die verwendeten Blöcke anstelle des ResidualBlocks zu benutzen und damit weitere Experimente durchzuführen.

7.3.5 Video Stylization

Im Paper [Gao+18] wird beschrieben, wie Style Transfer für Videos realisiert werden kann. Eine zusätzliche Loss-Funktion wird eingeführt, die ein Flickern zwischen den einzelnen Video-Frames verhindert.

7.3.6 Ausführung im Webbrowser

Ein neu aufkommendes Feld ist die Realisierung von Deep Learning im Webbrowser. Tensorflow.js ist ein Framework, dass Modelle direkt im Browser und somit auch direkt auf beispielsweise Mobiltelefonen oder anderen Geräten mit leistungssarmer Hardware ausführen kann. Ein weiteres Werkzeug ist ONNX.js. In PyTorch trainierte Modelle können in das ONNX-Format exportiert werden und mit ONNX.js direkt im Webbrowser ausgeführt werden. Zum Zeitpunkt der Erstellung dieser Arbeit befindet sich ONNX.js noch in einem frühen Entwicklungsstadium. Zukünftig ist ein Exportieren der in dieser Arbeit erstellten Modelle vorstellbar.

<https://www.tensorflow.org/js>
<https://github.com/microsoft/onnxjs>

Glossar

Activation Map Bezeichnet das Ergebnis der Anwendung eines Layers auf die Eingabedaten in einem CNN. 9, 12, 17, 22, 23

Adam Populärer Optimierungs Algorithmus [KB15]. 9, 24

COCO Großer Bilddatensatz von der Firma Microsoft [Lin+14]. 25

JPEG Joint Photographic Experts Group: Ein oft verwendetes Bildkomprimierungsverfahren. 25

MSE-Loss Das Mean-Squared-Error-Loss bildet den quadrierte gemittelten Fehler zwischen zwei Matrizen. 9, 12, 13, 22, 23

Receptive Field Ein CNN-Layer speichert Informationen mit seinen Filtern. Je nach Tiefe und Stride des CNN-Layers unterscheiden sich die Informationsmenge die ein CNN-Filter speichert. iv, 12

Tensor Als Tensor wird ein Scalar, Vektor oder eine Matrix beliebiger Dimension bezeichnet. 16, 17, 23

Vanishing Gradient Mit tiefen Neuronalen Netzwerken "verwischen" die Gradienten und das Netzwerk lässt sich somit schwerer trainieren, siehe Paper [He+15b]. 19, 36

VGG16 Vortrainierte Modellarchitektur trainiert auf dem ImageNet Datensatz. 21, 22, 38

Quellen- und Literaturverzeichnis

- [Jos89] Portrait of Joseph Roulin. *Vincent van Gogh*. 1889. URL: <https://www.moma.org/collection/works/79105> (besucht am 16.08.2019).
- [Nig89] The Starry Night. *Vincent van Gogh*. 1889. URL: <https://www.moma.org/collection/works/79802> (besucht am 16.08.2019).
- [Tre89] The Olive Trees. *Vincent van Gogh*. 1889. URL: <https://www.moma.org/collection/works/80013> (besucht am 16.08.2019).
- [Høs93] Edvard Munch (Photo: Nasjonalmuseet / Børre Høstland). *The Scream*. 1893. URL: <http://samling.nasjonalmuseet.no/en/object/NG.M.00939> (besucht am 16.08.2019).
- [Liq09] Still Life with Liqueur Bottle. *Pablo Picasso*. 1909. URL: <https://www.moma.org/collection/works/78986> (besucht am 16.08.2019).
- [LeC+89] Y. LeCun u. a. „Backpropagation Applied to Handwritten Zip Code Recognition“. In: *Neural Computation* 1.4 (1989), S. 541–551. DOI: 10.1162/neco.1989.1.4.541. eprint: <https://doi.org/10.1162/neco.1989.1.4.541>. URL: <https://doi.org/10.1162/neco.1989.1.4.541> (besucht am 16.08.2019).
- [LN89] Dong C. Liu und Jorge Nocedal. „On the limited memory BFGS method for large scale optimization“. In: *Mathematical Programming* 45.1 (Aug. 1989), S. 503–528. ISSN: 1436-4646. DOI: 10.1007/BF01589116. URL: <https://doi.org/10.1007/BF01589116> (besucht am 16.08.2019).
- [Gal90] S. I. Gallant. „Perceptron-based learning algorithms“. In: *IEEE Transactions on Neural Networks* 1.2 (Juni 1990), S. 179–191. ISSN: 1045-9227. DOI: 10.1109/72.80230.
- [ROF92] Leonid I. Rudin, Stanley Osher und Emad Fatemi. „Nonlinear total variation based noise removal algorithms“. In: *Physica D: Nonlinear Phenomena* 60.1 (1992), S. 259–268. ISSN: 0167-2789. DOI: 10.1016/0167-2789(92)90242-F. URL: <http://www.sciencedirect.com/science/article/pii/016727899290242F> (besucht am 16.08.2019).
- [LeC+98] Yann LeCun u. a. „Gradient-Based Learning Applied to Document Recognition“. In: *Proceedings of the IEEE*. Bd. 86. 11. 1998, S. 2278–2324. URL: <http://citeserx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665> (besucht am 16.08.2019).

- [Den+09] J. Deng u. a. „ImageNet: A large-scale hierarchical image database“. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Juni 2009, S. 248–255. DOI: [10.1109/CVPR.2009.5206848](https://doi.org/10.1109/CVPR.2009.5206848).
- [NH10] Vinod Nair und Geoffrey E. Hinton. „Rectified Linear Units Improve Restricted Boltzmann Machines“. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, 2010, S. 807–814. ISBN: 978-1-60558-907-7. URL: <http://www.cs.toronto.edu/~fritz/absps/reluICML.pdf> (besucht am 16.08.2019).
- [CKF11] R. Collobert, K. Kavukcuoglu und C. Farabet. „Torch7: A Matlab-like Environment for Machine Learning“. In: *BigLearn, NIPS Workshop*. 2011.
- [Col+11] Ronan Collobert u. a. „Natural Language Processing (Almost) from Scratch“. In: *J. Mach. Learn. Res.* 12 (Nov. 2011), S. 2493–2537. ISSN: 1532-4435. URL: <http://dl.acm.org/citation.cfm?id=1953048.2078186> (besucht am 16.08.2019).
- [Lin+14] Tsung-Yi Lin u. a. „Microsoft COCO: Common Objects in Context“. In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: <http://arxiv.org/abs/1405.0312> (besucht am 16.08.2019).
- [SZ14] Karen Simonyan und Andrew Zisserman. „Very Deep Convolutional Networks for Large-Scale Image Recognition“. In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: <http://arxiv.org/abs/1409.1556> (besucht am 16.08.2019).
- [Com15] Wikimedia Commons. *Typical CNN architecture*. 2015. URL: https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png (besucht am 16.08.2019).
- [GEB15] Leon A. Gatys, Alexander S. Ecker und Matthias Bethge. „A Neural Algorithm of Artistic Style“. In: *CoRR* abs/1508.06576 (2015). arXiv: 1508.06576. URL: <http://arxiv.org/abs/1508.06576> (besucht am 16.08.2019).
- [He+15a] Kaiming He u. a. 2015. URL: <https://arxiv.org/abs/1512.03385> (besucht am 16.08.2019).
- [He+15b] Kaiming He u. a. „Deep Residual Learning for Image Recognition“. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: <http://arxiv.org/abs/1512.03385> (besucht am 16.08.2019).
- [Joh15] Justin Johnson. *neural-style*. <https://github.com/jcjohnson/neural-style>. 2015. (Besucht am 16.08.2019).
- [KB15] Diederik Kingma und Jimmy Ba. „Adam: a method for stochastic optimization (2014)“. In: *arXiv preprint arXiv:1412.6980* 15 (2015).
- [Xu+15] Bing Xu u. a. „Empirical Evaluation of Rectified Activations in Convolutional Network“. In: *CoRR* abs/1505.00853 (2015). arXiv: 1505.00853. URL: <http://arxiv.org/abs/1505.00853> (besucht am 16.08.2019).

- [Eng16] Logan Engstrom. *Fast Style Transfer*. <https://github.com/lengstrom/fast-style-transfer/>. commit c77c028fe4412ce0bbb0e9f281a5970ab90fc7a5. 2016. (Besucht am 16.08.2019).
- [EMS16] Vania V. Estrela, Hermes Aguiar Magalhaes und Osamu Saotome. „Total Variation Applications in Computer Vision“. In: *CoRR* abs/1603.09599 (2016). arXiv: 1603.09599. URL: <http://arxiv.org/abs/1603.09599> (besucht am 16.08.2019).
- [GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016. (Besucht am 16.08.2019).
- [Joh16] Justin Johnson. *fast-neural-style*. <https://github.com/jcjohnson/fast-neural-style>. 2016. (Besucht am 16.08.2019).
- [JAL16] Justin Johnson, Alexandre Alahi und Fei-Fei Li. „Perceptual Losses for Real-Time Style Transfer and Super-Resolution“. In: *CoRR* abs/1603.08155 (2016). arXiv: 1603.08155. URL: <http://arxiv.org/abs/1603.08155> (besucht am 16.08.2019).
- [Jus16] Li Fei-Fei Justin Johnson Alexandre Alahi. 2016. URL: <https://www.neural-networks.io/en/single-layer/perceptron.php> (besucht am 16.08.2019).
- [ODO16] Augustus Odena, Vincent Dumoulin und Chris Olah. „Deconvolution and Checkerboard Artifacts“. In: *Distill* (2016). DOI: 10.23915/distill.00003. URL: <http://distill.pub/2016/deconv-checkerboard> (besucht am 16.08.2019).
- [UVL16] Dmitry Ulyanov, Andrea Vedaldi und Victor S. Lempitsky. „Instance Normalization: The Missing Ingredient for Fast Stylization“. In: *CoRR* abs/1607.08022 (2016). arXiv: 1607.08022. URL: <http://arxiv.org/abs/1607.08022> (besucht am 16.08.2019).
- [Vel16] Petar Veličković. 2016. URL: <https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution> (besucht am 16.08.2019).
- [Fer17] Max Ferguson. 2017. URL: https://www.researchgate.net/figure/Fig-A1-The-standard-VGG-16-network-architecture-as-proposed-in-32-Note-that-only_fig3_322512435 (besucht am 16.08.2019).
- [Hie17] Dang Ha The Hien. 2017. URL: <https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807> (besucht am 16.08.2019).
- [How+17] Andrew G. Howard u. a. „MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications“. In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: <http://arxiv.org/abs/1704.04861> (besucht am 16.08.2019).
- [Lee17] Ceshine Lee. *Pytorch Implementation of Perceptual Losses for Real-Time Style Transfer*. <https://towardsdatascience.com/8d608e2e9902>. Juli 2017. (Besucht am 16.08.2019).

- [MH17] Noah Makow und Pablo Hernandez. „Exploring Style Transfer: Extensions to Neural Style Transfer“. In: (2017). URL: <http://cs231n.stanford.edu/reports/2017/pdfs/428.pdf> (besucht am 16.08.2019).
- [Art18] Multicolored Abstract Artwork. *Sharon McCutcheon*. 2018. URL: <https://www.pexels.com/photo/abstract-abstract-painting-art-artistic-1149019> (besucht am 16.08.2019).
- [Gao+18] Chang Gao u. a. „ReCoNet: Real-time Coherent Video Style Transfer Network“. In: *CoRR* abs/1807.01197 (2018). arXiv: 1807.01197. URL: <http://arxiv.org/abs/1807.01197> (besucht am 16.08.2019).
- [Jac18] Alexis Jacq. *Neural Transfer using PyTorch*. https://pytorch.org/tutorials/advanced/neural_style_tutorial.html. 2018. (Besucht am 16.08.2019).
- [Nwa+18] Chigozie Nwankpa u. a. „Activation Functions: Comparison of trends in Practice and Research for Deep Learning“. In: *CoRR* abs/1811.03378 (2018). arXiv: 1811.03378. URL: <http://arxiv.org/abs/1811.03378> (besucht am 16.08.2019).
- [PyT18] PyTorch. *Example: fast-neural-style*. https://github.com/pytorch/examples/tree/master/fast_neural_style. 2018. (Besucht am 16.08.2019).
- [San+18] Mark Sandler u. a. „Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation“. In: *CoRR* abs/1801.04381 (2018). arXiv: 1801.04381. URL: <http://arxiv.org/abs/1801.04381> (besucht am 16.08.2019).
- [Bac19] Crystal Glass On A Colorful Background. *Steve Johnson*. 2019. URL: <https://www.pexels.com/photo/crystal-glass-on-a-colorful-background-2179374> (besucht am 16.08.2019).
- [Kar19a] Andrej Karpathy. 2019. URL: <https://cs231n.github.io/neural-networks-1> (besucht am 16.08.2019).
- [Kar19b] Andrej Karpathy. *Module 1: Neural Networks: Backpropagation, Intuitions*. <https://cs231n.github.io/optimization-2>. 2019. (Besucht am 16.08.2019).
- [PyT19a] PyTorch. *Hardtanh activation function*. 2019. URL: https://pytorch.org/docs/stable/_images/Hardtanh.png (besucht am 16.08.2019).
- [PyT19b] PyTorch. *PReLU activation function*. 2019. URL: https://pytorch.org/docs/stable/_images/PReLU.png (besucht am 16.08.2019).
- [PyT19c] PyTorch. *ReLU activation function*. 2019. URL: https://pytorch.org/docs/stable/_images/ReLU.png (besucht am 16.08.2019).
- [PyT19d] PyTorch. *Sigmoid activation function*. 2019. URL: https://pytorch.org/docs/stable/_images/Sigmoid.png (besucht am 16.08.2019).

- [TP19] Teal und Black Abstract Painting. *Anni Roenkae*. 2019. URL: <https://www.pexels.com/photo/teal-and-black-abstract-painting-2559624> (besucht am 16.08.2019).

Anhang A

A.1 Notebook Neural Style Transfer

```
1  from tqdm import tqdm_notebook
2  import matplotlib.pyplot as plt
3  import datetime
4
5  import torch
6  import torch.optim as optim
7  import torch.nn as nn
8
9  import torchvision.models as models
10
11 from csfnst.utils import load_image, plot_image_tensor, save_image_tensor
12 from csfnst.utils import rename_network_layers, replace_network_layers, get_criterion
13 from csfnst.losses import PerceptualLoss
14
15
16 output_image_file = '../images/output/htw-test.jpg'
17 style_image_file = 'license-checked/the_scream.jpg'
18 content_image_file = '../images/content/htw-768x768.jpg'
19
20 use_lbfgs = True
21 force_cpu = True
22 use_random_noise = False
23 content_image_size = 128
24 style_image_size = 768
25 epochs = 10
26
27 device = torch.device('cuda' if torch.cuda.is_available() and not force_cpu else 'cpu')
28 content_image = load_image(content_image_file, size=content_image_size,
29    ↳ normalize=False).to(device)
30 style_image = load_image(f'../images/style/{style_image_file}', size=style_image_size,
31    ↳ normalize=False).to(device)
32
33 if use_random_noise:
34     output_image = torch.rand(content_image.shape[0], content_image.shape[1],
35        ↳ content_image.shape[2]).to(device)
36 else:
```

```
34     output_image = content_image.clone().to(device)
35
36 config = {
37     'loss_network': 'vgg16',
38     'content_weight': 1,
39     'style_weight': 1e8,
40     'total_variation_weight': 1e-5,
41     'style_image': style_image_file,
42     'style_image_size': style_image_size,
43     'content_layers': ['relu3_3'],
44     'style_layers': ['relu1_2', 'relu2_2', 'relu3_3', 'relu4_3']
45 }
46
47 criterion = get_criterion(config, device='cpu' if force_cpu else 'cuda')
48 optimizer = optim.LBFGS([output_image]) if use_lbfgs else optim.Adam([output_image],
49                         lr=1e-1)
50
51 content_image.unsqueeze_(0)
52 output_image.unsqueeze_(0)
53 output_image.requires_grad_()
54
55 content_loss_history = []
56 style_loss_history = []
57 total_variation_loss_history = []
58 loss_history = []
59
60 progress_bar = tqdm_notebook(range(epochs))
61
62 if use_lbfgs:
63     for epoch in progress_bar:
64         def closure():
65             output_image.data.clamp_(0, 1)
66             optimizer.zero_grad()
67
68             loss = criterion(output_image, content_image)
69             loss.backward()
70
71             content_loss_history.append(criterion.content_loss_val)
72             style_loss_history.append(criterion.style_loss_val)
73             total_variation_loss_history.append(criterion.total_variation_loss_val)
74             loss_history.append(criterion.loss_val)
75
76             progress_bar.set_description(f'Loss: {loss.item():.2f}')
77
78         return loss
79
80         optimizer.step(closure)
81     else:
```

```

82     for epoch in progress_bar:
83         output_image.data.clamp_(0, 1)
84         optimizer.zero_grad()
85
86         loss = criterion(output_image, content_image)
87         loss.backward()
88
89         content_loss_history.append(criterion.content_loss_val)
90         style_loss_history.append(criterion.style_loss_val)
91         total_variation_loss_history.append(criterion.total_variation_loss_val)
92         loss_history.append(criterion.loss_val)
93
94         progress_bar.set_description(f'Loss: {loss.item():.2f}')
95
96         optimizer.step()
97
98
99     content_image.squeeze_()
100
101    output_image.detach_()
102    output_image.squeeze_()
103    output_image.data.clamp_(0, 1)
104
105    fig, axes = plt.subplots(1, 3)
106    fig.set_size_inches(18, 20)
107
108    plot_image_tensor(content_image, ax=axes[0])
109    plot_image_tensor(style_image, ax=axes[1])
110    plot_image_tensor(output_image, ax=axes[2])
111
112    save_image_tensor(output_image, output_image_file)
113
114
115    fig, axes = plt.subplots(1, 1, figsize=(18, 20))
116    axes.plot(content_loss_history, label='Content Loss')
117    axes.plot(style_loss_history, label='Style Loss')
118    axes.plot(total_variation_loss_history, label='Total Variation Loss')
119    axes.plot(loss_history, label='Loss')
120    plt.legend()
121    plt.show()

```

Code snippet A.1: Notebook zur Durchführung des Neural Style Transfer Algorithmus

A.2 Script Activation-Functions

```

1  def get_activation_fn(name):
2      activation_fn_map = {

```

```
3     'ELU': lambda: nn.ELU(),
4     'ReLU': lambda: nn.ReLU(),
5     'RReLU': lambda: nn.RReLU(),
6     'PReLU': lambda: nn.PReLU(),
7     'SELU': lambda: nn.SELU(),
8     'CELU': lambda: nn.CELU(),
9     'ReLU6': lambda: nn.ReLU6(),
10    'Hardtanh': lambda: nn.Hardtanh(min_val=0.0, max_val=1.0),
11    'Sigmoid': lambda: nn.Sigmoid(),
12    'None': lambda: None
13 }
14
15 return activation_fn_map[name]()
```

Code snippet A.2: Activation-Functions

A.3 Script ConvBlock

```
1 class ConvBlock(nn.Module):
2     def __init__(self, in_channels, out_channels, kernel_size=3, stride=1,
3                  activation_fn='None'):
4         super(ConvBlock, self).__init__()
5
6         self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride)
7         self.norm = nn.InstanceNorm2d(out_channels, affine=True)
8         self.activation_fn = get_activation_fn(activation_fn)
9
10    def forward(self, x):
11        x = self.norm(self.conv(x))
12        x = self.activation_fn(x) if self.activation_fn else x
13
14    return x
```

Code snippet A.3: ConvBlock

A.4 Script ResidualBlock

```
1 class ResidualBlock(nn.Module):
2     def __init__(self, in_channels, out_channels, inner_channels=None, kernel_size=3,
3                  activation_fn='None'):
4         super(ResidualBlock, self).__init__()
5
6         inner_channels = inner_channels if inner_channels else in_channels
7
8         self.conv1 = nn.Conv2d(in_channels, inner_channels, kernel_size)
```

```
8     self.conv2 = nn.Conv2d(inner_channels, out_channels, kernel_size)
9
10    self.pad1 = nn.ReflectionPad2d(padding=kernel_size // 2)
11    self.pad2 = nn.ReflectionPad2d(padding=kernel_size // 2)
12
13    self.norm1 = nn.InstanceNorm2d(inner_channels, affine=True)
14    self.norm2 = nn.InstanceNorm2d(out_channels, affine=True)
15
16    self.activation_fn = get_activation_fn(activation_fn)
17
18    def forward(self, x):
19        identity = x
20
21        x = self.norm1(self.conv1(self.pad1(x)))
22        x = self.activation_fn(x) if self.activation_fn else x
23        x = self.norm2(self.conv2(self.pad2(x)))
24
25        return x + identity
```

Code snippet A.4: ResidualBlock

A.5 Script UpsampleBlock

```
1  class UpSampleBlock(nn.Module):
2      def __init__(self, in_channels, out_channels, kernel_size=3, stride=1,
3                   scale_factor=None,
4                   activation_fn='None'):
5          super(UpSampleBlock, self).__init__()
6
7          self.scale_factor = scale_factor
8
9          self.conv = ConvBlock(
10              in_channels=in_channels,
11              out_channels=out_channels,
12              kernel_size=kernel_size,
13              stride=stride,
14              activation_fn=activation_fn
15          )
16
17      def forward(self, x):
18          if self.scale_factor:
19              x = F.interpolate(x, mode='nearest', scale_factor=self.scale_factor)
20
21      return self.conv(x)
```

Code snippet A.5: UpsampleBlock

A.6 Script TransformerNet

```

1  class TransformerNet(nn.Module):
2      def __init__(self, channel_multiplier=32, bottleneck_size=5,
3          ↪ bottleneck_type=BottleneckType.RESIDUAL_BLOCK,
4              expansion_factor=6, final_activation_fn='None',
5                  ↪ intermediate_activation_fn='None'):
6          super(TransformerNet, self).__init__()
7
8          self.pad = nn.ReflectionPad2d(padding=20)
9
10         self.down1 = ConvBlock(3, channel_multiplier, kernel_size=9, stride=1,
11             ↪ activation_fn=intermediate_activation_fn)
12         self.down2 = ConvBlock(channel_multiplier, channel_multiplier * 2, kernel_size=5,
13             ↪ stride=2,
14                 activation_fn=intermediate_activation_fn)
15
16         self.down3 = ConvBlock(channel_multiplier * 2, channel_multiplier * 4,
17             ↪ kernel_size=5, stride=2,
18                 activation_fn=intermediate_activation_fn)
19
20         if bottleneck_type == BottleneckType.RESIDUAL_BLOCK:
21             self.bottleneck = nn.Sequential([
22                 ResidualBlock(channel_multiplier * 4, channel_multiplier * 4,
23                     ↪ activation_fn=intermediate_activation_fn)
24                 for _ in range(bottleneck_size)
25             ])
26         else:
27             raise ValueError('Wrong value for bottleneck_type')
28
29         self.up1 = UpSampleBlock(channel_multiplier * 4, channel_multiplier * 2,
30             ↪ kernel_size=5, scale_factor=2,
31                 activation_fn=intermediate_activation_fn)
32         self.up2 = UpSampleBlock(channel_multiplier * 2, channel_multiplier, kernel_size=5,
33             ↪ scale_factor=2,
34                 activation_fn=intermediate_activation_fn)
35         self.up3 = ConvBlock(channel_multiplier, 3, kernel_size=9,
36             ↪ activation_fn=final_activation_fn)
37
38     def forward(self, x):
39         x = self.pad(x)
40         x = self.down3(self.down2(self.down1(x)))
41         x = self.bottleneck(x)
42         return self.up3(self.up2(self.up1(x)))

```

Code snippet A.6: TransformerNet

Anhang B

B.1 Netzwerkarchitekturen

Name	Bottleneck Size s	Multiplikator m	Anzahl lernbarer Parameter
Netzwerk 1	5	32	2.007.881
Netzwerk 2	5	16	507.305
Netzwerk 3	5	8	129.497
Netzwerk 4	5	4	33.713
Netzwerk 5	4	32	1.712.073
Netzwerk 6	4	16	433.129
Netzwerk 7	4	8	110.841
Netzwerk 8	4	4	28.993
Netzwerk 9	3	32	1.416.265
Netzwerk 10	3	16	358.953
Netzwerk 11	3	8	92.185
Netzwerk 12	3	4	24.273
Netzwerk 13	2	32	1.120.457
Netzwerk 14	2	16	284.777
Netzwerk 15	2	8	73.529
Netzwerk 16	2	4	19.553

Tabelle B.1: Unterschiedliche Netzwerkgrößen und ihre Parameter

B.2 Performanz-Test mit 1920 * 1080 (Full HD) Pixel Bildern

Name	XPS - CPU	XPS - GPU	Jetson - CPU	Jetson - GPU
Network 1	9,53208	nicht durchführbar	nicht durchführbar	nicht durchführbar
Network 2	4,10052	nicht durchführbar	nicht durchführbar	0,86413
Network 3	1,77594	nicht durchführbar	nicht durchführbar	0,48424
Network 4	0,89486	0,04353	nicht durchführbar	0,35886
Network 5	8,81463	nicht durchführbar	nicht durchführbar	nicht durchführbar
Network 6	3,79047	nicht durchführbar	nicht durchführbar	0,70640
Network 7	1,66949	nicht durchführbar	nicht durchführbar	0,39689
Network 8	0,83914	0,01236	nicht durchführbar	0,30227
Network 9	8,10067	nicht durchführbar	nicht durchführbar	0,67149
Network 10	3,49074	nicht durchführbar	nicht durchführbar	0,51339
Network 11	1,52498	nicht durchführbar	nicht durchführbar	0,30213
Network 12	0,79769	0,00532	nicht durchführbar	0,24884
Network 13	7,37850	nicht durchführbar	nicht durchführbar	0,37438
Network 14	3,18989	nicht durchführbar	nicht durchführbar	0,25658
Network 15	1,46334	nicht durchführbar	nicht durchführbar	0,21752
Network 16	0,75117	0,00454	nicht durchführbar	0,19545

Tabelle B.2: Berechnungsgeschwindigkeit in Sekunden für Bilder der Größe 1920 * 1080 Pixel

B.3 Performanz-Test mit 1024 * 768 Pixel Bildern

Name	XPS - CPU	XPS - GPU	Jetson - CPU	Jetson - GPU
Network 1	3,35983	nicht durchführbar	nicht durchführbar	0,73296
Network 2	1,40631	0,04695	nicht durchführbar	0,38569
Network 3	0,63503	0,02816	28,13781	0,31154
Network 4	0,32738	0,01726	9,17026	0,27334
Network 5	3,17060	nicht durchführbar	nicht durchführbar	0,53276
Network 6	1,31797	0,01821	nicht durchführbar	0,31762
Network 7	0,59221	0,00632	26,26611	0,30870
Network 8	0,30782	0,00552	8,74099	0,22172
Network 9	2,90782	nicht durchführbar	nicht durchführbar	0,34536
Network 10	1,22248	0,01569	nicht durchführbar	0,25033
Network 11	0,55174	0,00525	24,38772	0,23672
Network 12	0,28889	0,00493	8,26189	0,20012
Network 13	2,69121	nicht durchführbar	nicht durchführbar	0,23276
Network 14	1,14871	0,01272	nicht durchführbar	0,18833
Network 15	0,51133	0,00468	22,53269	0,17650
Network 16	0,27009	0,00436	7,84484	0,17443

Tabelle B.3: Berechnungsgeschwindigkeit in Sekunden für Bilder der Größe 1024 * 768 Pixel

B.4 Performanz-Test mit 640 * 480 Pixel Bildern

Name	XPS - CPU	XPS - GPU	Jetson - CPU	Jetson - GPU
Network 1	1,33538	0,05155	nicht durchführbar	0,50566
Network 2	0,53889	0,02326	36,90058	0,25883
Network 3	0,25448	0,01054	10,93982	0,24857
Network 4	0,13798	0,00717	3,70690	0,23639
Network 5	1,23604	0,01890	nicht durchführbar	0,30614
Network 6	0,50666	0,00579	34,34460	0,27297
Network 7	0,23769	0,00571	10,35927	0,21817
Network 8	0,12903	0,00536	3,52992	0,32184
Network 9	1,13755	0,01991	nicht durchführbar	0,24380
Network 10	0,46090	0,00502	31,71307	0,19767
Network 11	0,22101	0,00480	9,62866	0,24432
Network 12	0,12269	0,00458	3,37524	0,17682
Network 13	1,05401	0,01383	nicht durchführbar	0,17912
Network 14	0,43314	0,00441	29,12002	0,16645
Network 15	0,20557	0,00414	8,90912	0,16807
Network 16	0,11306	0,00395	3,14229	0,17277

Tabelle B.4: Berechnungsgeschwindigkeit in Sekunden für Bilder der Größe 640 * 480 Pixel

B.5 Loss: The Starry Night

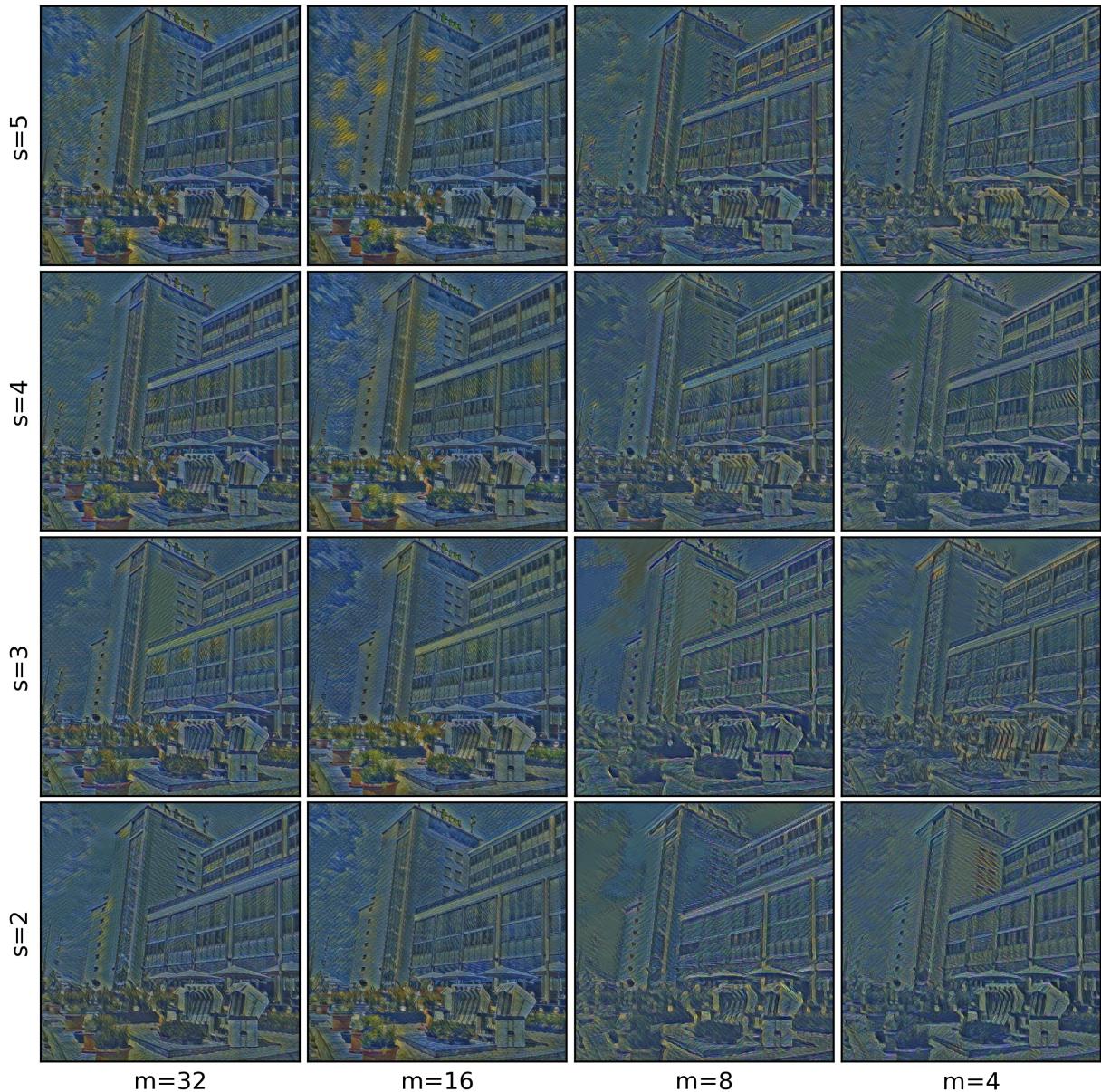


Abbildung B.1: Loss: The Starry Night

Die Druckversion entspricht nicht der digitalen Qualität der Bilder.

B.6 Loss: The Scream

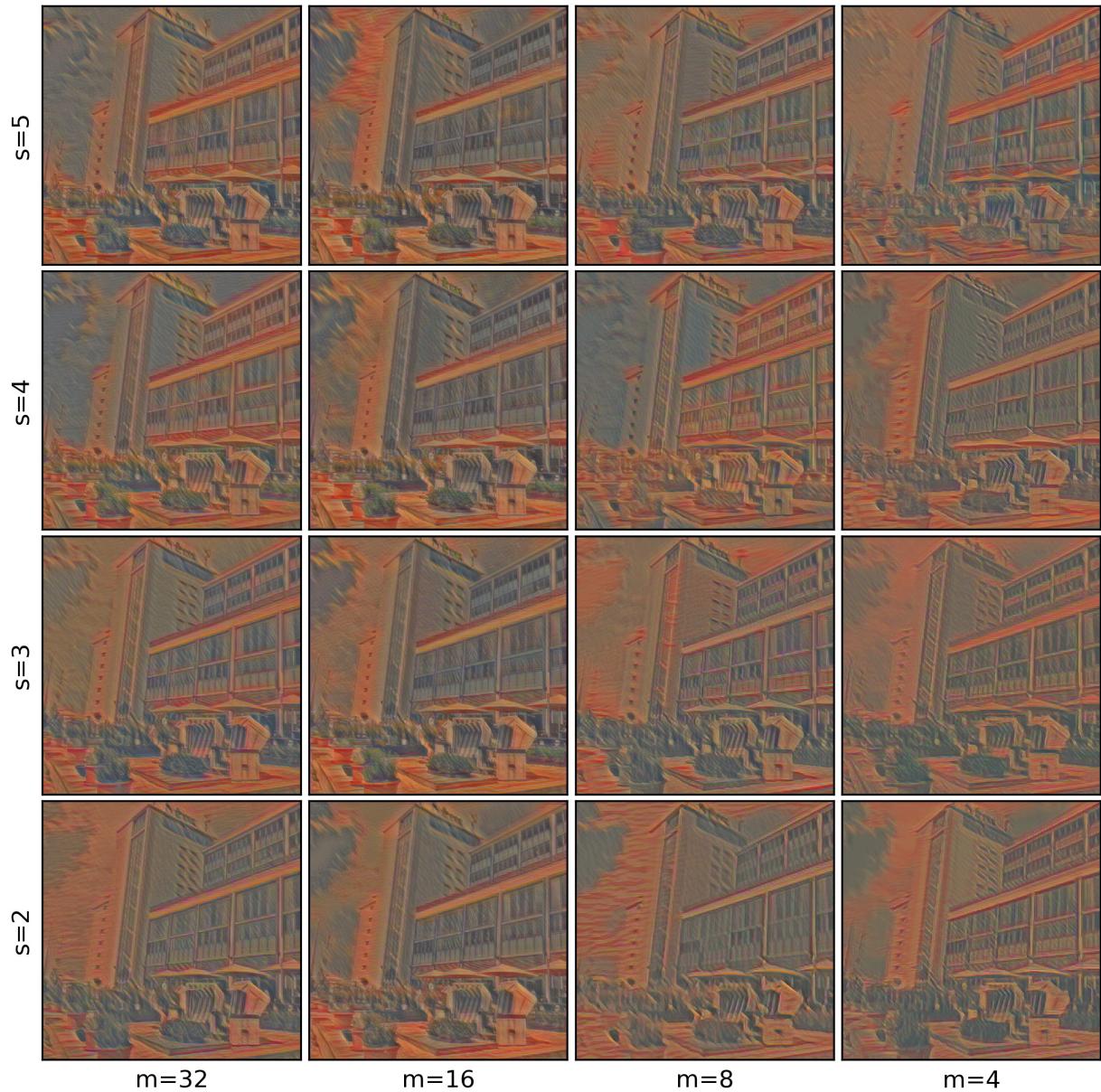


Abbildung B.2: Loss: The Scream

Die Druckversion entspricht nicht der digitalen Qualität der Bilder.

B.7 Ergebnisse: The Starry Night

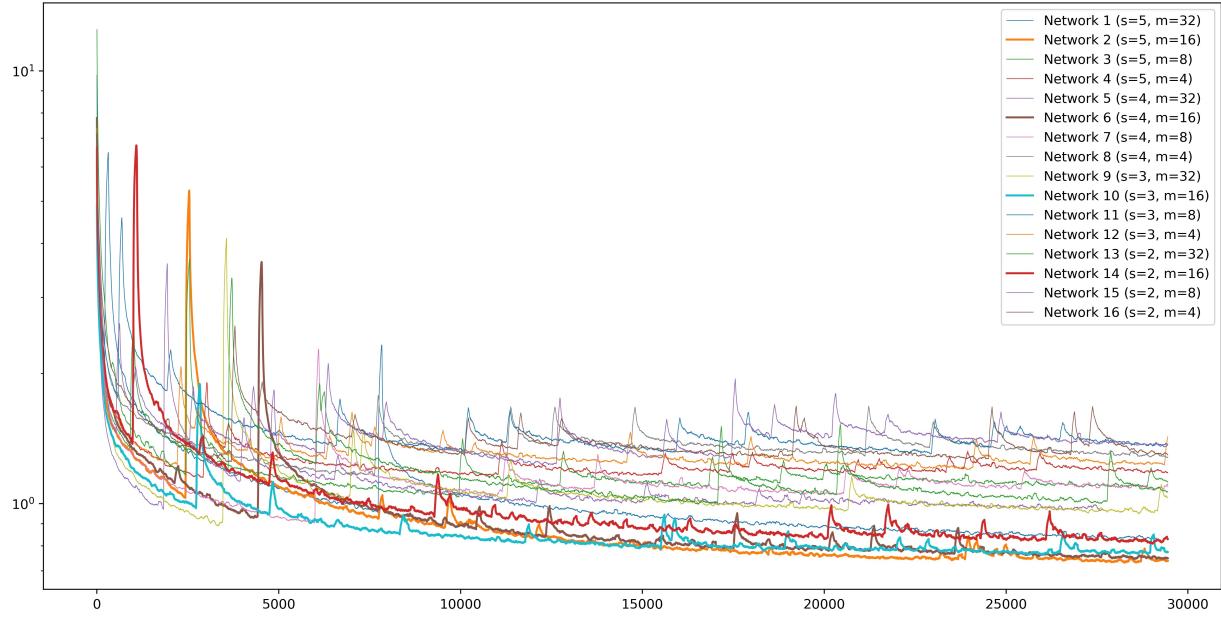


Abbildung B.3: Ergebnisse: The Starry Night

B.8 Ergebnisse: The Scream

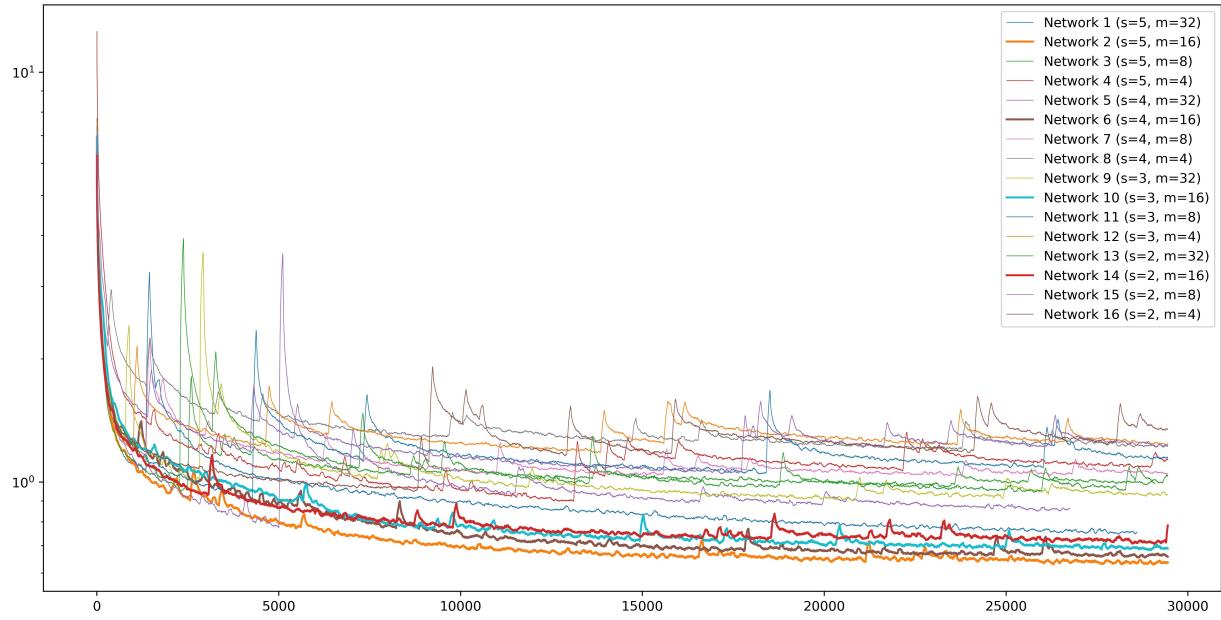


Abbildung B.4: Ergebnisse: The Scream

B.9 Trainierte Modelle

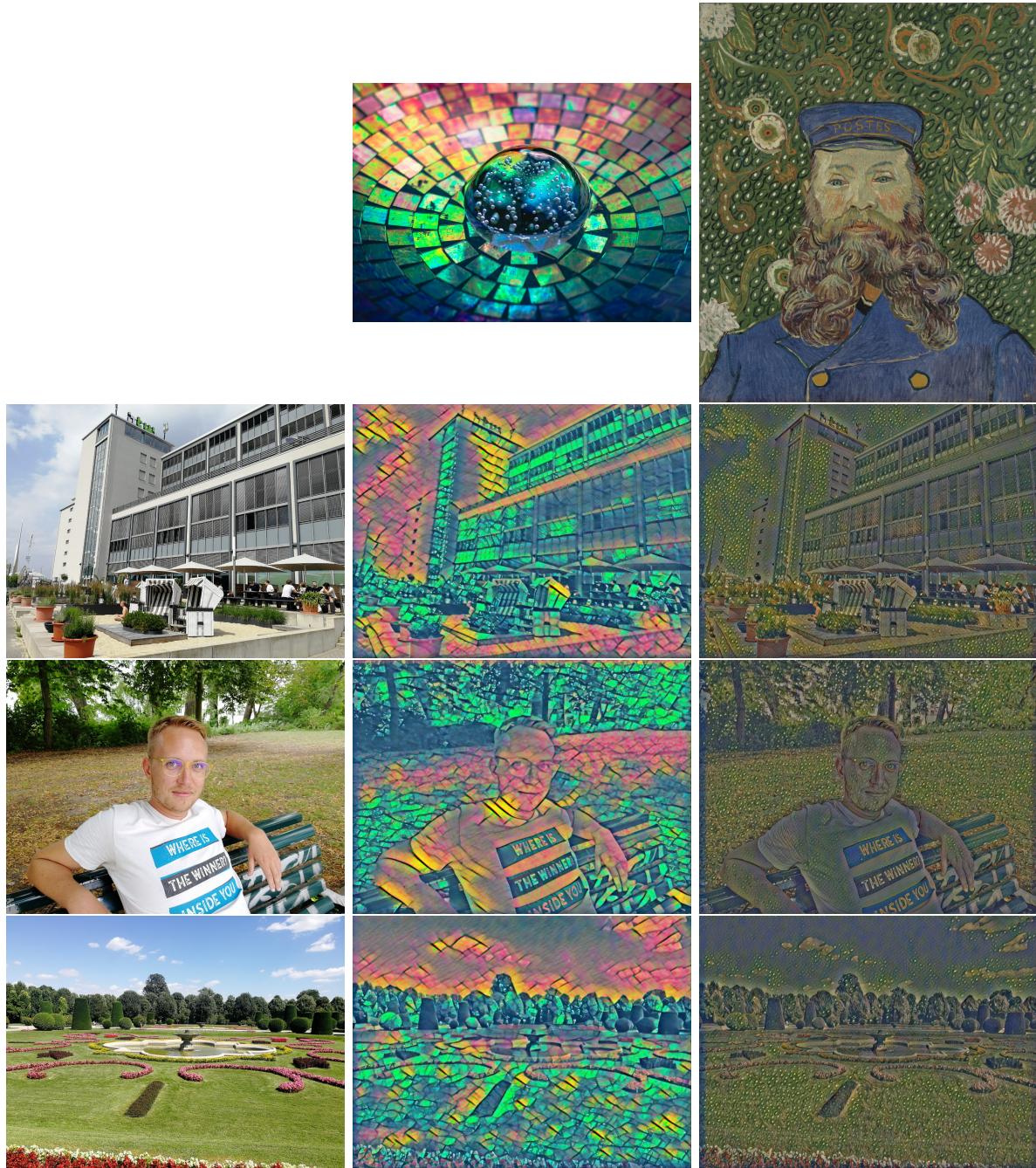


Abbildung B.5: Eigene Abbildungen kombiniert mit verschiedenen Stilen: [Bac19], [Jos89]

Die Druckversion entspricht nicht der digitalen Qualität der Bilder.

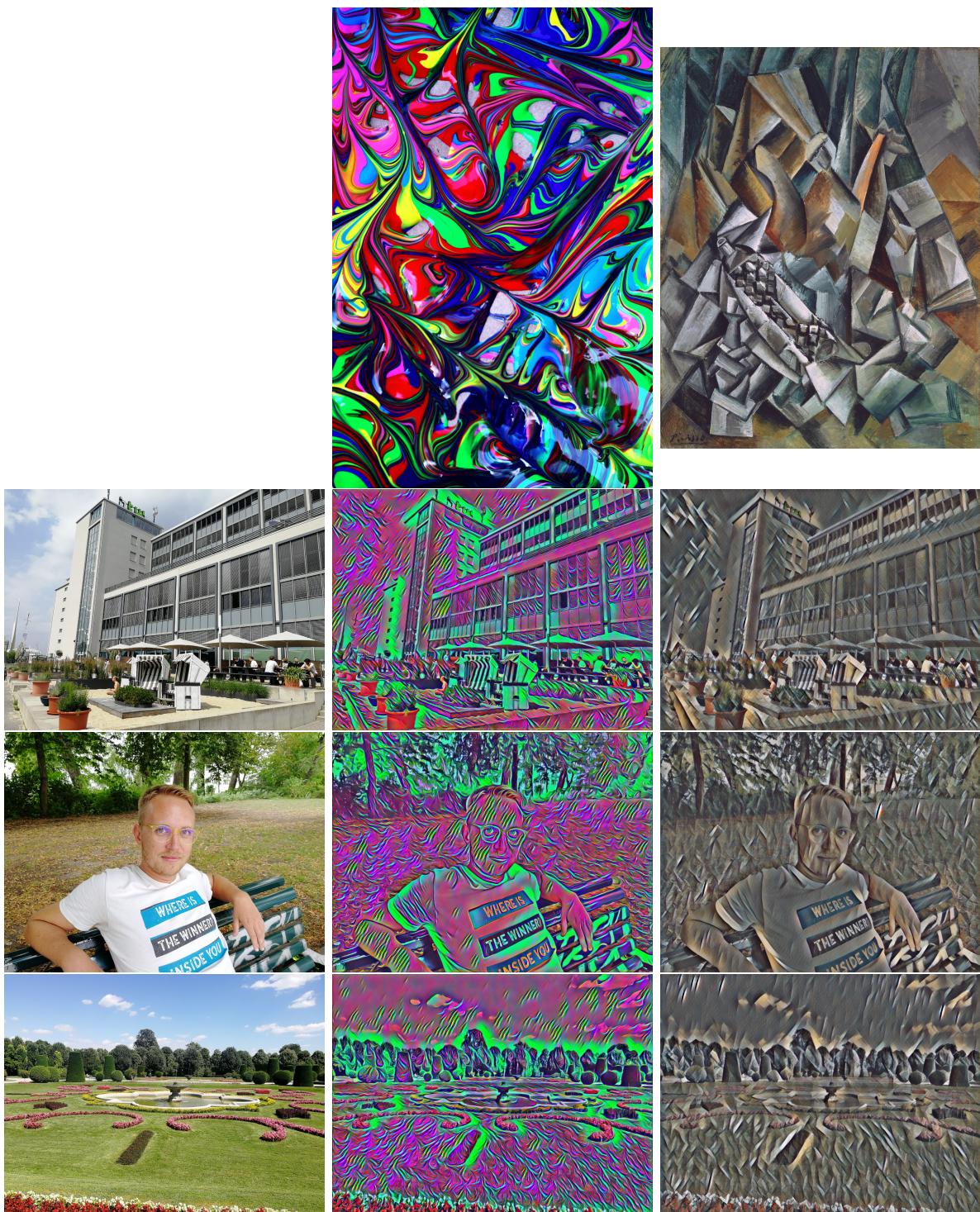


Abbildung B.6: Eigene Abbildungen kombiniert mit verschiedenen Stilen: [Art18], [Liq09]

Die Druckversion entspricht nicht der digitalen Qualität der Bilder.

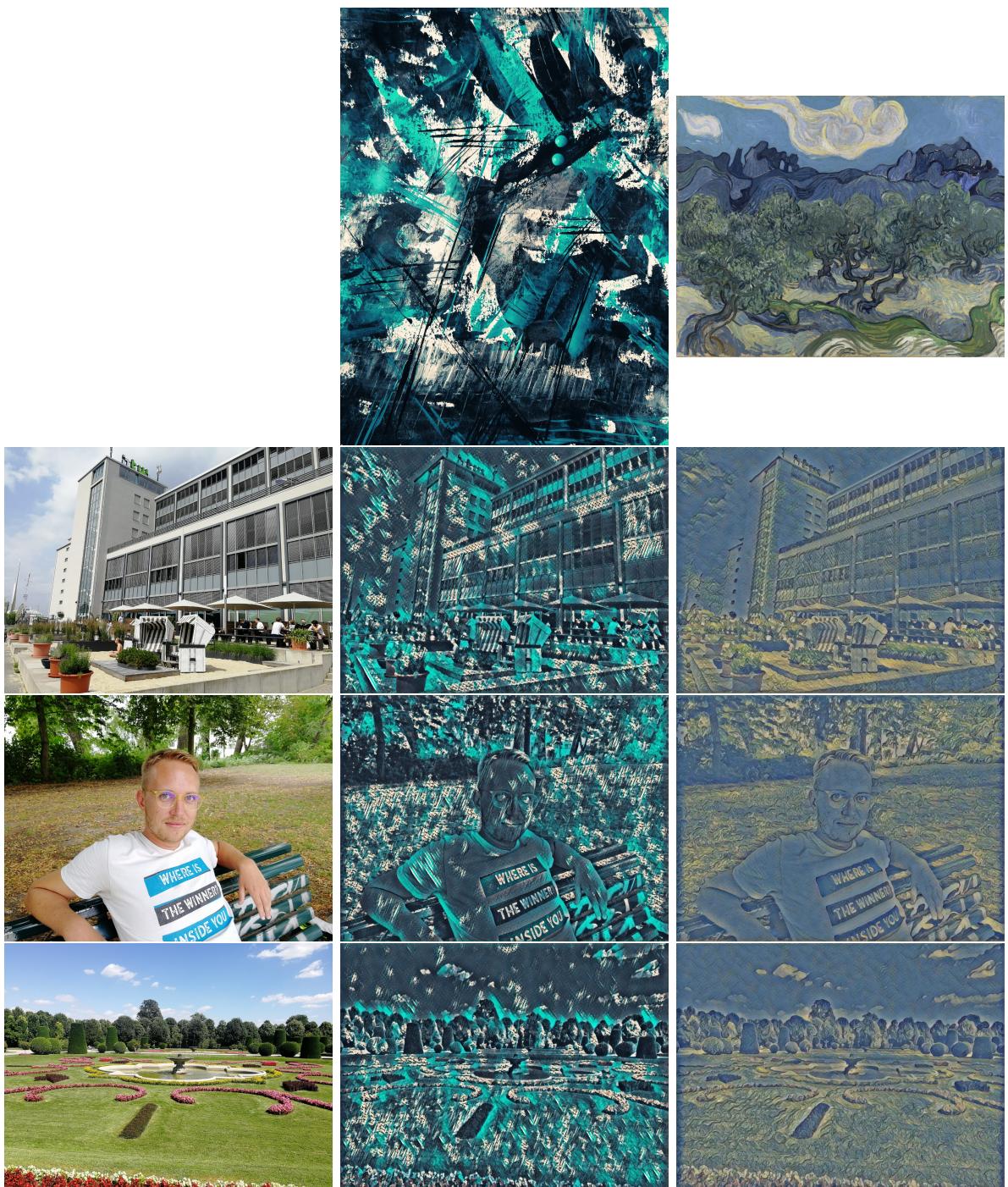


Abbildung B.7: Eigene Abbildungen kombiniert mit verschiedenen Stilen: [TP19], [Tre89]

Die Druckversion entspricht nicht der digitalen Qualität der Bilder.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 20.08.2019

Christoph Stach