Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Untersuchung von Image Colorization Methoden anhand Convolutional Neuronal Networks

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:	M.Sc. Patrick Baumann

Eingereicht von:	Adrian Saiz Ferri
Immatrikulationsnummer:	s0554249
Eingereicht am:	XX.XX.2020

Abstract

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

TODO

1.2 Zielsetzung

TODO

1.3 Vorgehensweise und Aufbau der Arbeit

TODO

Kapitel 2

Grundlagen

Dieses Kapitel verschafft einen Überblick über die benötigten theoretische Grundlagen, um die Methoden dieser Arbeit zu verstehen. Als erstes wird eine Einführung in Neuronale Netzwerke gegeben, anschließend werden einzelne Bestandsteile und Varianten von Neuronalen Netzwerken erklärt. Als nächstes wird der "Lab-Farbraum" kurz erklärt. Abschließend wird einen Überblick über verwandte Arbeiten gegeben.

2.1 Neuronale Netze

Künstliche Neuronale Netze sind inspiriert durch das Menschliche Gehirn und werden für Künstliche Intelligenz und Maschinelles Lernen angewendet. Sie werden für überwachtes und unüberwachtes lernen verwendet. In der vorliegende Arbeit werden nur Methoden des überwachtes lernen angewendet. Bei überwachtes lernen sind die Datensätze gelabelt sodass den Output von dem Neuronales Netz mit den richtigen Ergebnissen verglichen werden kann.

Neuronale Netze bestehen aus Neuronen oder auch "Units" genannt, die Schichtenweise in "Layers" (Schichten) angeordnet sind. Beginnend mit der Eingabeschicht (Input Layer) fließen Informationen über eine oder mehrere Zwischenschichten (Hidden Layer) bis hin zur Ausgabeschicht (Output Layer). Dabei ist der Output des einen Neurons der Input des nächsten. [Moe18]

2.1.1 Feedforward Neural Network

Das Ziel von einem Feedforward Neural Network ist die Annäherung an irgendeine Funktion f^*. Ein Feedforward Neural Network definiert eine Abbildung $y = f(x; W)$ wobei x den Input ist und W die lernbare Parameter sind (auch Gewichte genannt). [GBC16, S. 164-223]

Diese Netzwerkarchitektur heißt "feedforward" weil der Informationsfluss von der Input Layer über die Hidden Layers bis zur Output Layer in einer Richtung weitergereicht wird.

Feedforward Neural Networks werden als eine Kette von Funktionen repräsentiert. Als Beispiel, kann man die Funktionen $f^{(1)}, f^{(2)}, f^{(3)}$ in Form einer Kette verbinden um $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$ zu bekommen. Diese Kettenstrukturen sind die am häufigsten genutzte Struktur bei Neuronale Netzwerke. In diesem Fall, $f^{(1)}$ ist das erste Layer, $f^{(2)}$ das zweite und $f^{(3)}$ der Output Layer von diesem Netzwerk. Die Länge dieser Kette definiert die Tiefe von einem Netzwerk. Je tiefer ein Netzwerk ist desto mehr lernbare Parameter hat es und somit eine erhöhte Rechenleistung braucht um trainiert zu werden. In der Praxis werden die Netzwerke sehr tief, daher der Begriff Deep Learning.

Während dem Training werden die Gewichte von $f(x)$ verstellt, um $f^*(x)$ zu erhalten. Jedes Trainingsbeispiel x ist mit einem Label $y = f^*(x)$ versehen. Die Trainingsbeispiele legen genau fest, was der Output Layer generieren soll. Der Output Layer soll Werte generieren, die nah an y liegen. Das Verhalten von den Hidden Layers wird nicht durch die Trainingsbeispiele festgelegt, sondern der Lernalgorithmus soll definieren, wie diese Layers verwendet werden, um die beste Annäherung von $f^*(x)$ zu generieren.

2.1.2 Fully-connected Neural Network

Fully-connected Neural Networks sind die am häufigsten vorkommende Art von Neuronale Netze. In dieser Netzwerkarchitektur sind alle Neuronen von einem Layer mit alle Neuronen von der vorherige und nächsten Layer verbunden. Neuronen in dem gleichen Layer sind aber nicht miteinander verbunden. [Fei17a]

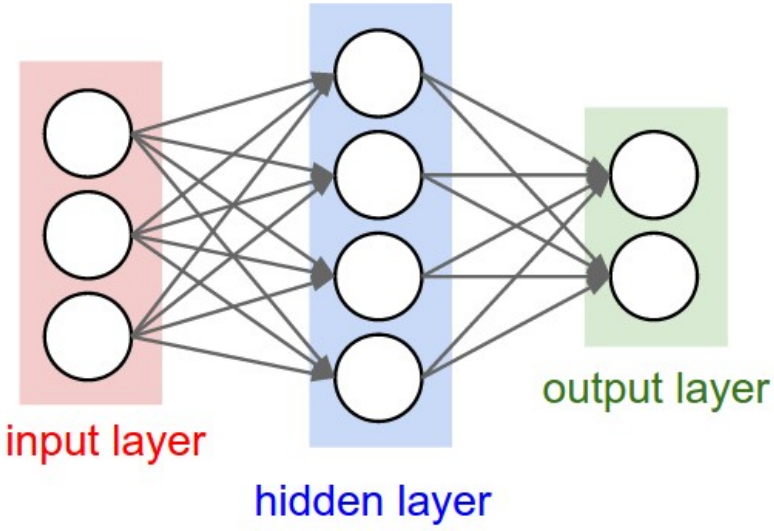

Abbildung 2.1: Fully-connected Neural Network mit 2 Layers (ein Hidden Layer mit 4 Neuronen) und ein Output Layer mit 2 Neuronen [Fei20a]

Eine der wichtigsten Gründe für die Anordnung von Neuronale Netze in Layers ist dass so eine Struktur anhand von Matrix Multiplikationen berechnet werden kann. Das obere Bild 2.1 stellt ein Netzwerk mit 3 Inputs x, eine Hidden Layer mit 4 Neuronen und eine Output Layer mit 2 Neuronen dar. Die Kreisen repräsentieren die Neuronen und einem Bias Wert b, die Pfeilen stellen die Gewichte w dar.

$$f(x) = w * x + b \tag{2.1}$$

Nach jeden Hidden Layer läuft den Output durch eine Aktivierungsfunktion σ die in 2.1.3 erklärt wird. Daraus wird die vorherige Formel um σ erweitert:

$$f(x) = \sigma(w * x + b) \tag{2.2}$$

Forward Pass

Den Forward Pass von einem Neuronalem Netz wird anhand von Matrizen Multiplikationen berechnet. Um das zu veranschaulichen wird es anhand eines Beispiels erklärt.

Ausgehend von einem Netzwerk mit 3 Inputs, eine Hidden Layer mit 2 Neuronen und einem Output Neuron, ergeben sich folgende Beispielwerte:

$$X = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \quad W = \begin{pmatrix} 10 & 20 \\ -20 & -40 \\ 20 & 0 \\ -40 & 0 \end{pmatrix} \quad W_{out} = \begin{pmatrix} 20 \\ 40 \\ -40 \end{pmatrix} \tag{2.3}$$

X sind die Inputs, W die Gewichte von dem Hidden Layer und W_{out} die Gewichte von dem Output Layer. Die erste Spalte in dem Input X und die erste Zeile in beide Gewichtsmatrizen W und W_{out} sind die Werte für den Bias. Diese Anordnung von den Bias Werte ermöglicht die Berechnung durch eine einzigen Matrix Multiplikation. Als Aktivierungsfunktion wird ReLU [NH10] verwendet:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{2.4}$$

Im ersten Schritt durchläuft den Input durch das Hidden Layer $f(X \times W)$:

$$f\left(\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 10 & 20 \\ -20 & -40 \\ 20 & 0 \\ -40 & 0 \end{pmatrix}\right) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{2.5}$$

Im zweiten Schritt wird den Output von der vorherige Multiplikation mal die Gewichte von dem Output Layer multipliziert:

$$f\left(\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 20 \\ 40 \\ -40 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{2.6}$$

2.1.3 Aktivierungsfunktionen

Eine Aktivierungsfunktion definiert die Aktivierungsrate von einem Neuron. Es gibt verschiedene Aktivierungsfunktionen:

ReLU

Die Rectified Linear Unit konvertiert alle negative Werte in 0 und alle positive Werte behalten deren Identität. Diese Aktivierungsfunktion wurde für die Netzwerke in dieser Arbeit benutzt da es Vorteile gegenüber Sigmoid zeigt. Einer der Vorteile ist dass die Mathematische Auswertung der Funktion unkompliziert ist. Außerdem beschleunigt es die Konvergenz von Stochastisches Gradientenabstiegsverfahren im Vergleich zu Sigmoid. ReLU ist definiert als:

$$f(x) = max(0, x) \tag{2.7}$$

wobei x einem Input ist.
Neuronen die ReLU als Aktivierungsfunktion verwenden, können während des Trainings "sterben". Zum Beispiel, wenn der Gradient in einem Neuron zu groß ist, kann dieser zu einem update der Gewichte führen, wo das Neuron nie wieder aktiviert werden kann. Mit einer korrekter Einstellung der Lernrate kann das vermieden werden. [Fei17a]

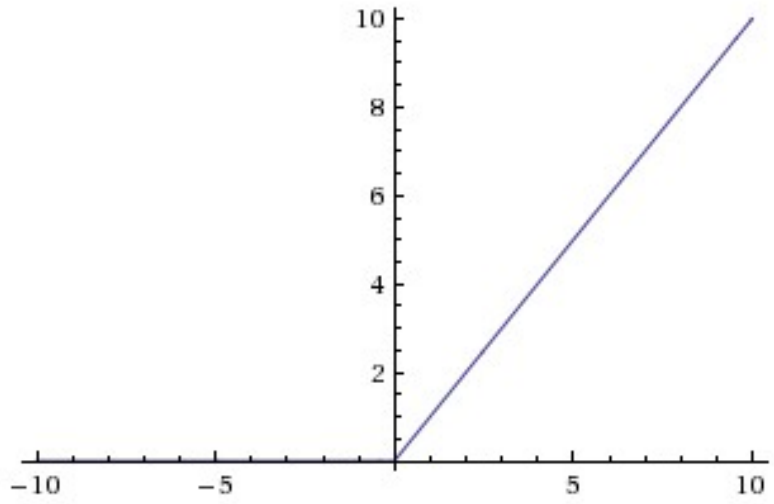

Abbildung 2.2: Rectified Linear Unit (ReLU) [Fei20b]

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) sind eine besondere Form von künstliche neuronale Netzwerke, das speziell für die Verarbeitung von Bilddaten, unter anderen, vorgesehen ist

[Dip19].
Im gegensatz zu traditionelle neuronale Netzwerke, die ein Vektor als Input nehmen, nehmen Convolutional Neural Networks ein 3D Volumen als Input ($W \times H \times C$, wobei W die Breite, H die Höhe und C die Farbkanäle sind).

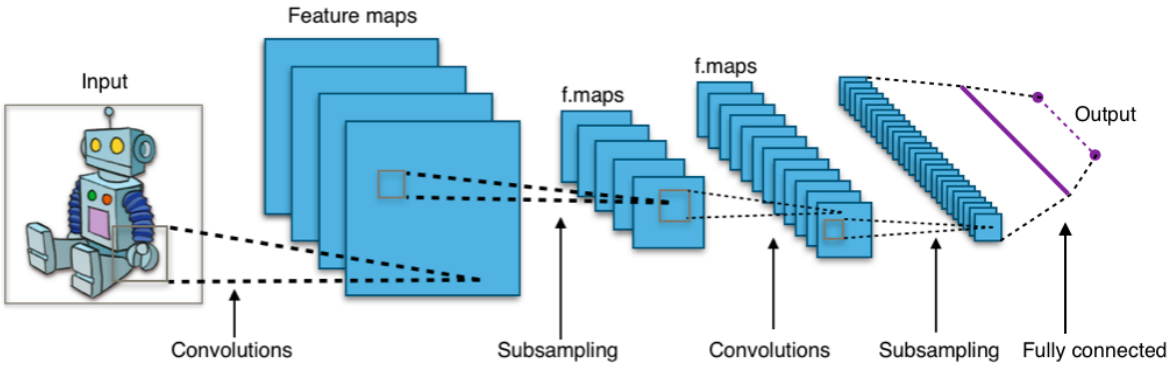

Abbildung 2.3: Typische Struktur von einem Convolutional Neural Network [Com15]

CNNs bestehen in der Regel aus 2 Formen von Layers, die Convolutional Layer und die Pooling Layer.

Die Convolutional Layer besteht aus mehrere hintereinander geschaltete 3 Dimensionale Filter, auch Kernel genannt ($W \times H \times D$, wobei D die Tiefe der Feature maps darstellt), die während den Forward pass, über dem Bild mit einer festgelegten Schrittweite (Stride), geschoben werden. Mit den sogenannten Padding wird der Verhalten an den Rändern festgelegt. An jeder stelle wird eine Matrix Multiplikation zwischen den Filter und die aktuelle Position auf dem Bild durchgeführt. Als Output wird eine 2 Dimensionalen Feature maps generiert. Die Größe dieser Feature map ist abhängig von der Größe des Filters, dem Padding und vor allem dem Stride. Ein Stride von 2 bei einer Filter Größe von 2×2 führt beispielsweise pro Filter zu einer Halbierung der Größe der Feature map im Vergleich zum Input Volumen [Bec19]. Ein Stride von 1 bei einem 3×3 Filter mit Padding 1 führt zu einer Feature map mit der gleiche Größe wie der Input Volumen.

Die Filter erkennen in den ersten Ebenen einfache Strukturen wie Linien, Farben oder Kanten. In den nächsten Ebenen lernt das CNN Kombinationen aus diesen Strukturen wie Formen oder Kurven. In den tieferen Layers werden komplexere Strukturen und Objekte identifiziert.

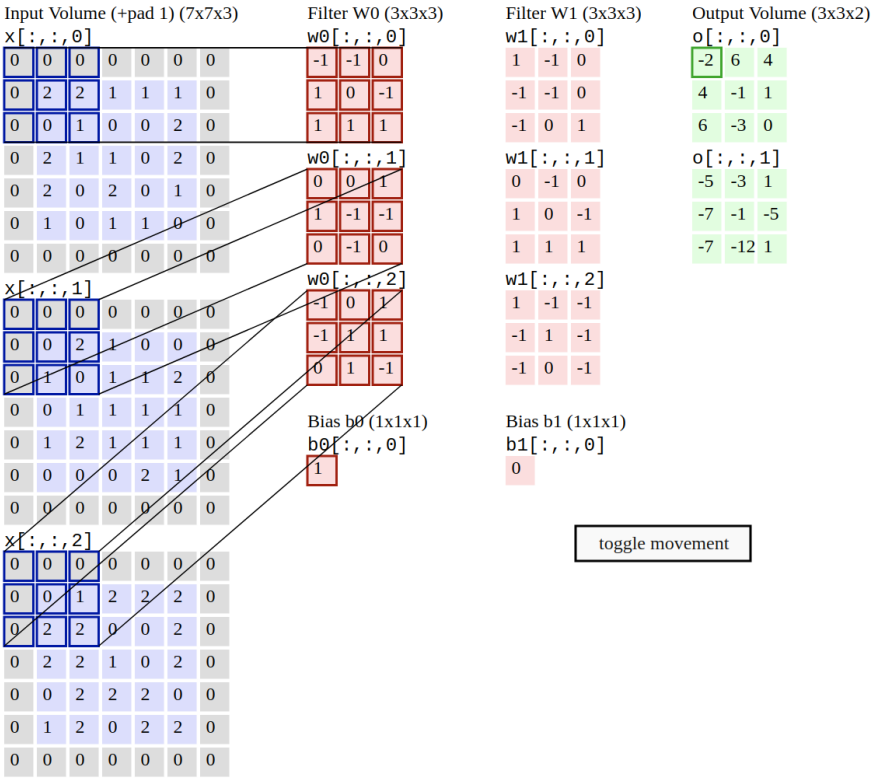

Abbildung 2.4: Beispiel Forward pass von einem Convolutional Layer mit einem $7 \times 7 \times 3$ Input Volumen, zwei $3 \times 3 \times 3$ Filter, Padding 1 und Stride 2. [Fei20c]

Die Pooling Layer dient zur Reduktion der Dimensionen von einem Input Volumen und somit die Parameter von dem Netzwerk. Es gibt verschiedene Pooling Operationen die angewendet werden können, wie zum Beispiel Maximum Pooling, Minimum Pooling, oder Average Pooling. Im Rahmen dieser Arbeit wird Maximum Pooling (oder Max Pooling) verwendet.

Ein Max Pooling Layer aggregiert die Aktivierungsmatrizen von Convolutional Layers, in dem nur die höchste Zahl von einem Filter weitergegeben wird. Zum Beispiel, bei einem 2×2 Filter wird aus 4 Zahlen nur 1 Zahl weitergegeben. Damit wird einer Reduktion der Dimensionen erreicht.

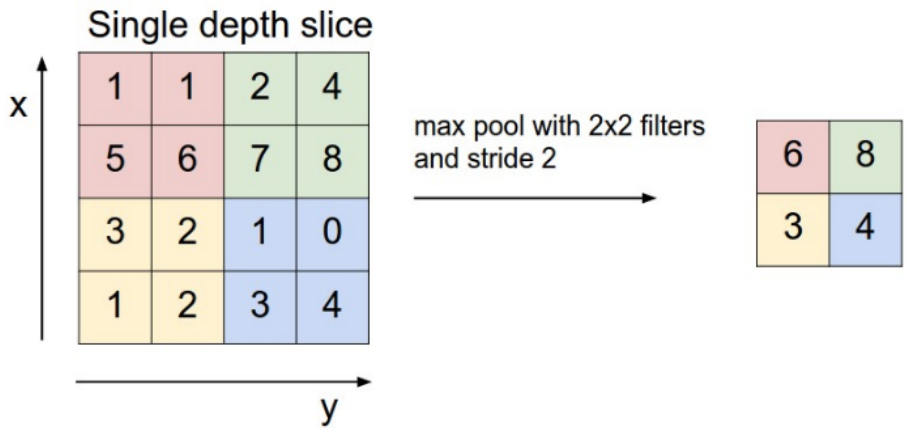

Abbildung 2.5: Max pooling Operation mit 2×2 Filtern und Stride 2 [Fei20c]

2.2.1 Loss Functions

Die Loss Function (Kostenfunktion) dient zur Feststellung der Fehler (Loss) zwischen dem Output von einem Model und die Zielwerte. Das Ziel von Neuronalen Netzen ist es den Loss zu minimieren. Wenn der Loss gleich Null ist, heißt dass $y = \hat{y}$. Es gibt verschiedene Arten von Loss Functions. Im Rahmen dieser Arbeit werden Loss Functions bezogen auf Regressions– und Klassifizierungsprobleme behandelt.

Mean Square Error Loss

Mean Square Error (MSE) Loss misst der mittlere quadratische Fehler und ist definiert als:

$$J = \frac{1}{N} \sum (y - \hat{y})^2 \tag{2.8}$$

wobei, J der Loss ist, N die Anzahl der Klassen, y die Korrekte Klasse (Ground Truth) und $\hat{y}$ die vorhergesagte Klasse ist.

Cross Entropy Loss

Der Cross Entropy Loss wird bei Klassifizierungsprobleme verwendet. Es gibt 2 Arten, Binär und Multiclass Cross Entropy Loss. Bei der Multiclass Cross Entropy Loss wird ein Vektor mit einer Wahrscheinlichkeitsverteilung $x \in [0, 1]$ ausgewertet, wenn die Korrekte

Klasse eine 1 hat ist der Loss 0. Je weniger Wahrscheinlichkeit die Korrekte Klasse besitzt desto höher der Loss sein wird. Der Multiclass Cross Entropy Loss ist definiert als:

$$J = -\frac{1}{N}\Big(\sum_{i=0}^{N} y_i * \log(\hat{y}_i)\Big) \tag{2.9}$$

wobei, J der Loss ist, N die Anzahl der Klassen, y die Korrekte Klasse (Ground Truth) und $\hat{y}$ die vorhergesagte Klasse ist.

Weighted Cross Entropy Loss

Bei der Weighted Cross Entropy Loss werden die Klassen gewichtet bevor der Loss berechnet wird. Das ist zum Beispiel nützlich um Klassen mit einer niedrige Wahrscheinlichkeit zu bevorzugen.

2.2.2 Optimierungsalgorithmen

Das Ziel von Optimierungsalgorithmen ist eine Kombination von Gewichte W zu finden, dass die Loss Function minimiert. Es gibt verschiedene relevante Optimierungsalgorithmen. In der vorliegende Arbeit werden Gradient Descent und Adam verwendet.

Gradient Descent

Gradient Descent (Gradientenabstiegsverfahren) ist ein iteratives Verfahren, um bei einer Funktion das Minimum (oder das Maximum) zu finden. Mit Hilfe von partielle Ableitungen kann der Gradient von einer Funktion berechnet werden. Ein Gradient ist, in dem Fall von Neuronale Netze, ein Vektor der zum höchsten Punkt der Loss Function zeigt. Wird der negative Gradient genommen, zeigt dieser zum tiefsten Punkt. Bei jeder Kombination von Gewichte wird der Gradient berechnet und mal eine bestimmte Lernrate α multipliziert, anschließend werden alle Gewichte aktualisiert. Die Lernrate definiert die Größe der Schritte in Richtung Minimum. Die Update Regel für die Gewichte ist definiert als:

$$w_{x+1} = w_x - \alpha * \nabla J(w_x) \tag{2.10}$$

wobei, w_{x+1} die aktualisierte Gewichte sind, w_x die vorherige Gewichte, α die Lernrate und $\nabla J(w_x)$ der Gradient ist. Die Update Regel für den Bias sieht identisch aus.

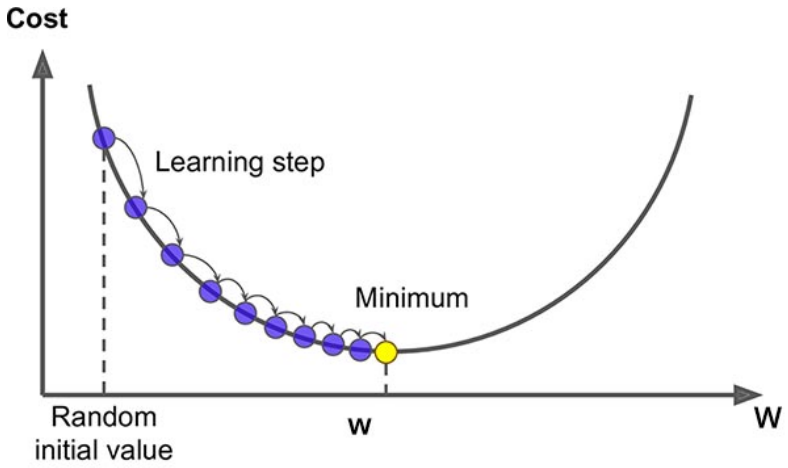

Abbildung 2.6: Gradient descent visualisiert [Bha18]

Es gibt verschiedene Arten von Gradient Descent, Mini-Batch Gradient Descent und Stochastisches Gradient Descent. Bei den normalen Gradient Descent werden die Gradienten im Bezug zu den gesamten Datensatz berechnet und damit das Update durchgeführt. Mini-Batch Gradient Descent berechnet die Gradienten im Bezug zu einen kleinen Teil von dem Datensatz und führt einen Update für alle Parameter durch. Letztendlich, Stochastisches Gradient Descent berechnet den Gradient bezogen auf einem einzigen Element von dem Datensatz und führt einen Update für alle Parameter durch.
Um die Konvergenz Richtung Minimum zu beschleunigen, wurde das Gradient Descent mit Momentum entwickelt. Bei diesem Ansatz wird einen Geschwindigkeitsparameter zu der Update Regel hinzugefügt die alle vorherige Updates akkumuliert. Das ermöglicht die schnellere Konvergenz mit jedem Schritt. Die neue Update regel ist definiert als:

$$\begin{aligned} v_{t+1} &= \rho v_t - \alpha * \nabla J(w_x) \\ w_{x+1} &= w_x + v_{t+1} \end{aligned} \tag{2.11}$$

wobei v_{t+1} der nächste Geschwindigkeitsparameter ist, v_t der aktuelle Geschwindigkeitsparameter und ρ ein Reibungsparameter (typisch 0.9) zu Regulierung ist.

Adam

Adam steht für "Adaptive Moment Estimation Algorithm" und ist ein Optimierungsalgorithmus der eine angepasste Lernrate für die verschiedene Parameter berechnet [KB14]. Adam ist der bevorzugte Optimierungsalgorithmus für die vorliegende Arbeit. Adam kombiniert die Ansätze von AdaGrad [DHS11] und RMSProp. AdaGrad ist eine verbesserte Version von Gradient Descent dass eine angepasste Lernrate für die verschiedene Parameter einführt.

2.2.3 Backpropagation

Neuronale Netze lernen in dem das Loss minimiert wird. Wie in der vorherigen Sektionen erläutert, bestimmt die Loss Function, die Fehlerrate von einem Neuronales Netz. Dieses Loss kann mit Hilfe von einem Optimierungsalgorithmus reduziert werden. Backpropagation ermöglicht es, einer effizienten Berechnung der Gradienten in einem neuronalen Netzwerk [15]. Mit Hilfe der Kettenregel kann eine Komplexe Loss Function in kleinere Unterfunktionen zerlegt werden um Lokal die Ableitung zu berechnen. Das ermöglicht eine unkomplizierte Berechnung des Gradienten.

Als Beispiel kann die folgende Sigmoid Funktion in Unterfunktionen zerlegt werden:

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}} \tag{2.12}$$

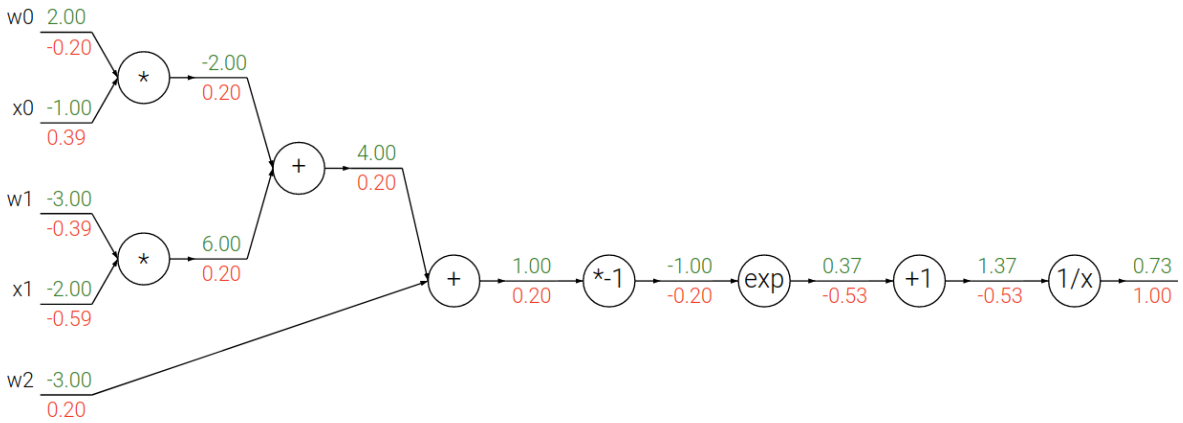

Abbildung 2.7: Backpropagation Beispiel anhand einer 2D Neuron mit der Aktivierungsfunktion Sigmoid [Fei20d]

Bei dem oberen Bild sind $[w_0, w_1, w_2]$ die Gewichte und $[x_0, x_1]$ die Inputs des Neurons. Um es unkompliziert zu halten, kann die obere Funktion als irgendeine Funktion, die Inputs (w, x) entgegennimmt und eine einzelne Zahl als Output hat, visualisiert werden. Die Grüne Zahlen repräsentieren die Ergebnissen aus dem Forward Pass und die Rote Zahlen der zurück propagierte Loss. Jeder Knoten ist fähig ein Output und der lokale Gradient von dem Output im Bezug auf den Input zu berechnen, ohne die komplette Funktion kennen zu müssen [Fei17b].

2.2.4 Transposed Convolution

Im gegensatz zu einem Pooling Layer, ermöglicht einer Transposed Convolutional Layer die Dimensionen von einem Volumen zu vergrößern. Die funktionsweise einer Transposed Convolution wird anhand von einem Beispiel erklärt.
Ausgehend von einer 2×2 Input Matrix die auf 3×3 vergrößert werden soll, ein 2×2 Filter, Null Padding und Stride 1, ergibt sich den Folgenden Output.

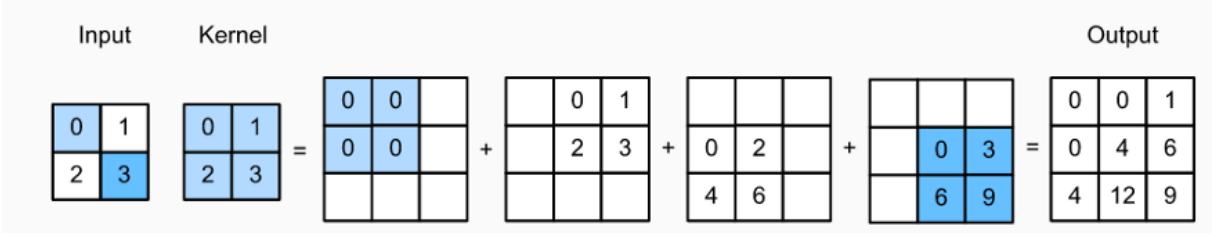

Abbildung 2.8: Die Komplette Transposed Convolution Operation [Zha20]

Jeder Zahl in der Input Matrix wird mit jeder Zahl in den Filter multipliziert. Daraus ergibt sich an jeder Position in der Input Matrix einer 2×2 Matrix. Die überlappende Zahlen auf der Output Matrix werden addiert. Daraus ergibt sich einer 3×3 Output Matrix.

2.2.5 Autoencoder

Ein Autoencoder ist ein neuronales Netz, welches versucht die Eingangsinformationen zu komprimieren und mit den reduzierten Informationen im Ausgang wieder korrekt nachzubilden [Ngu20]. Die Komprimierung und Rekonstruktion der Input läuft in zwei Schritten ab, weshalb das Netz in zwei Teile betrachtet werden kann.

Encoder

Der Encoder reduziert die Dimensionen von einem Input und somit werden die wichtigsten Features in einer reduzierter Dimension komprimiert. In einem neuronalen Netz wird diese Komprimierung durch die Hidden Layers erreicht. Der Encoder ist definiert als:

$$h = f(x) \tag{2.13}$$

wobei x einem Input ist, f der Encoder und h die Kodierte Features von x.

Decoder

Der Decoder ist zuständig für die Rekonstruktion von x anhand h und ist definiert als:

$$\hat{x} = g(h) \tag{2.14}$$

wobei, $\hat{x}$ der rekonstruierte Input ist, g der Decoder und h die Kodierten Features sind.

In der vorliegende Arbeit wird einer Variante von Autoencoders, basierend auf Convolutional Neural Networks, eingesetzt.

2.3 *Lab*-Farbraum

Der *Lab*-Farbraum (auch CIELAB-Farbraum genannt) ist ein Farbraum definiert bei der Internationale Beleuchtungskommission (CIE) in 1976. Farben werden mit drei Werte beschrieben. "L" (Lightness) definiert die Helligkeit. Die Werte liegen zwischen 0 und 100. "a" gibt die Farbart und Farbintensität zwischen Grün und Rot und "b" gibt die Farbart und Farbintensität zwischen Blau und Gelb. Die Werte für "a" und "b" liegen zwischen -128 und 127.
In der vorliegende Arbeit wird den *Lab*-Farbraum verwendet, da es unkompliziert ist, den "L" Kanal von beide Farbkanäle "a" und "b" zu trennen. Außerdem bildet den *Lab*-Farbraum das Menschliche Sehvermögen besser als den RGB-Farbraum[1] ab.

[1]RGB steht für Red, Green und Blue, die 3 Farbkanäle des Farbraums

2.4 Verwandte Arbeiten

Vor der Erstellung dieser Arbeit wurden Methoden von Image Colorization bereits untersucht. Frühere Methoden waren stark an Menschliches Input gebunden. Der Methode von Levin et al. verwendet Farbstiche auf dem Graustufenbild die Automatisch von einem Algorithmus über das gesamte Bild propagiert werden [LLW04].

Der Fokus dieser Arbeit ist auf volle automatische Image Colorization Methoden gesetzt. Konservative Methoden die Convolutional Neural Networks verwenden, versuchen die Farben von dem Originalen Bild wiederherzustellen in dem die Loss Function die Distanz der vorhergesagte Farbe und die Reale Farbe berechnet. Diese Methoden liefern in der Regel entsättigte und blasse Bilder wie bei [Özb19]. Einer der Gründe für diese Ergebnissen ist dass die Modelle nicht richtig lernen. Zum Beispiel, Äpfel können verschiedene Farben einnehmen, wie Rot oder Grün. Wenn das Netzwerk mit einem MSE Loss trainiert wird, wird der Output bei einem Apfel eine Farbe zwischen Rot und Grün sein, was ein entsättigtes Bild erzeugen wird.

Aus diesem Grund betrachtet die vorliegende Arbeit das Problem von Image Colorization als ein Multimodales Problem, da gleiche Objekte verschiedene Farbe einnehmen können. Die vorliegende Arbeit orientiert sich auf die Methoden von Zhang et al. und Billaut et al., die ähnliche Ansätze für Image Colorization vorschlagen. Sie betrachten das Problem als ein Klassifizierungsproblem und trainieren eine CNN mit der Weighted Cross Entropy Loss. Der Output von dem Netzwerk ist eine Wahrscheinlichkeitsverteilung über die mögliche Bins, die im Kapitel 3.3 erläutert werden, für jeden Pixel. Außerdem wird der Loss während des Trainings gewichtet um seltene Farben zu bevorzugen.

Beide Ansätze verwenden “Color Bins” die es unkompliziert ermöglichen das CNN mit der Weighted Cross Entropy Loss zu Trainieren. Es wird im nächsten Kapitel weiter auf diese Methode eingegangen.

Kapitel 3

Konzeption

Dieses Kapitel beinhaltet alle Schritte für die Konzeption der angewandte Methoden. Außerdem wird der Datensatz und die verwendeten Tools präsentiert.

3.1 Image Colorization als Multimodales Problem

Konventionelle automatische Methoden zielen darauf ab, die Farben für ein generiertes Bild so nah wie möglich an das Originale Bild vorherzusagen. Diese Methoden verwenden ein MSE Loss der Vorhersagen die Weit von den Originalen Farbwerte entfernt liegen, stärker bestraft, als Farbwerte die dichter an den Originalen Farbwerte liegen. Das führt, wie bei 2.4 beschrieben, zu entsättigte Bilder. Die Gründe für diese Ergebnisse ist dass verschiedene Objekte, verschiedene Farben einnehmen können. Aus diesem Grund, behandelt die vorliegende Arbeit das Problem als ein Multimodales Problem.

3.2 Farbraum

Der Standard Farbraum der Bilder für die Methoden dieser Arbeit ist der RGB-Farbraum. Bei diesem Farbraum lässt sich schwer das Graustufen Bild von den Farbkanäle zu trennen, daher wird der Lab-Farbraum verwendet.

Bei den Lab-Farbraum können ohne Probleme die Farbkanäle “ab” von den Belichtungskanal “L” getrennt werden. Der Belichtungskanal “L” enthält das Graustufenbild die in den

CNN eingespeist wird. Das generierte Bild wird für die Darstellung von dem Lab-Farbraum in den RGB-Farbraum konvertiert.

Abbildung 3.1: Originales Bild oben links, den Belichtungskanal “L” oben rechts, unten links den Farbkanal “a” und unten rechts den Farbkanal “b”.

3.3 Binning

Binning ist eine Technik, die für die Bildverarbeitung verwendet wird. Binning wird, in dem Kontext von Image Colorization, als Eingruppierung von naheliegenden Farben definiert. Die Farben werden in gleich große Intervalle aufgeteilt. Diese Intervalle bezeichnet man im Englischen als “Bins”. Jedes dieser Intervalle wird durch einen Bin Index repräsentiert, somit reduziert sich die Anzahl der Klassen die vorhergesagt werden können.

Als Beispiel für die Veranschaulichung wird der normalisierte Lab-Farbraum in 36 gleich große Bins unterteilt. Da die Farbinformationen in den “ab” Farbkanäle kodiert sind,

werden nur diese 2 Farbkanäle in Bins klassifiziert. Auf dem Bild 3.2 ist der Farbkanal "a" auf der x-Achse und der "b" Farbkanal auf der y-Achse abgebildet. Die Vierecke repräsentieren die Bins. Die obere Zahl in den Bins symbolisiert die xy Koordinaten auf dem Grid, die untere unterstrichene Zahl symbolisiert den Bin Index. Die xy Koordinaten Grid sind Bedeutsam für die Berechnung der Bins.

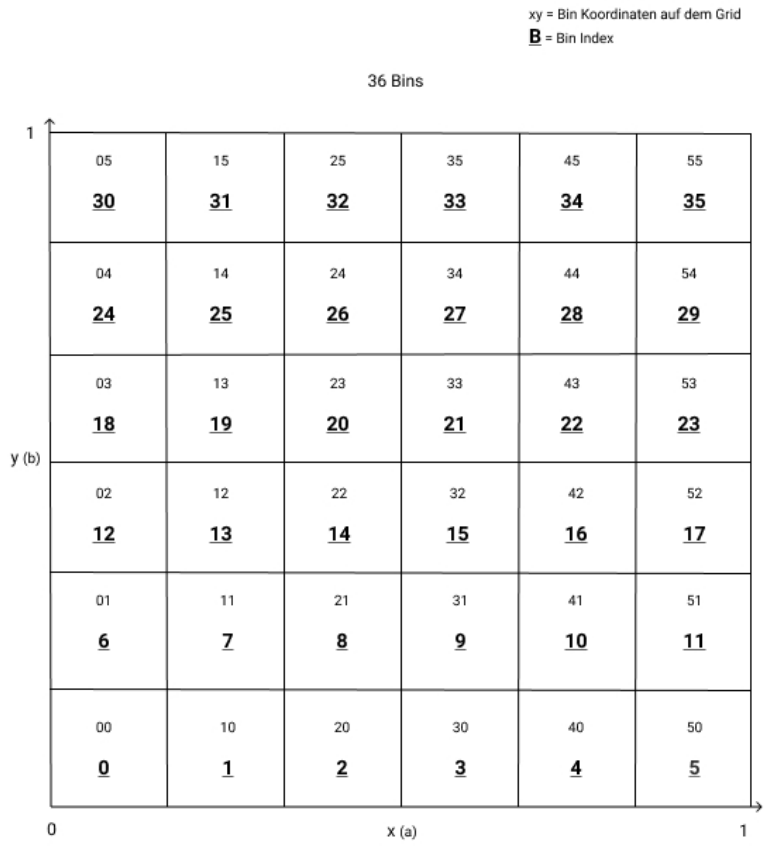

Abbildung 3.2: Grid mit 36 bins. Die x-Achse bildet die Werte von dem Farbkanal "a" und die y-Achse die Werte von den Farbkanal "b" ab.

Für die Methoden dieser Arbeit wurden nur symmetrische Grids verwendet, so ergibt zum Beispiel ein 6×6 Grid 36 Bins und ein 18×18 Grid 324 Bins.

3.3.1 Umwandlung von Bin zu Farbe

Die Umwandlung von Bin zu Farbe muss für die Ergebnisse von dem Netzwerk ebenfalls vorgegeben sein. Für dieses Problem werden vor dem Training die Farben von jedem Pixel aus den Trainingsbildern in Bins klassifiziert. Diese Klassifikation verläuft separat für

jeden Farbkanal. Anschließend wird der Modus und den Durchschnitt pro Farbkanal der unter jedem Bin klassifizierten Farben ausgerechnet. Zuletzt werden die Werte in zwei separate Dateien gespeichert.

3.4 Netzwerkarchitektur

Die Netzwerkarchitektur ist ein wichtiger Faktor dass u.a. die Ergebnisse beeinträchtigen kann. Wichtig um die Methoden zu vergleichen ist ein leichtes Netzwerk der wenige Parameter besitzt, schnell zu trainieren ist und gute Ergebnisse liefert. Da mit Bildern gearbeitet wird, eignen sich Convolutional Neural Networks besonders gut.

Das Ziel von dem Netzwerk ist es, ein Graustufenbild als Input zu bekommen und eine Wahrscheinlichkeitsverteilung über alle Bins per Pixel vorherzusagen. Das Output Volumen hat die Dimensionen $W_{Input} \times H_{Input} \times 1$, wobei W und H die Breite und Höhe von dem Bild sind, mit einer Wahrscheinlichkeitsverteilung über alle Bins bei jedem Pixel. Dieser Ansatz ist auch bei Image Segmentation Probleme genutzt, wo ein Bild in das Netzwerk einspeist wird und als Output, wird ein Segmentation map, mit eine Klasse per Pixel, erzeugt. In der Regel hat jeder Klasse eine bestimmte Farbe und dadurch werden Objekte klassifiziert und getrennt. In dem Fall von Image Colorization bekommt jeder Pixel in dem Output Volumen eine Wahrscheinlichkeitsverteilung über alle Bins die in einer Farbe umgewandelt wird.

Die Methoden von Zhang et al. [ZIE16] verwenden einen CNN das einen Graustufenbild als Input entgegen nimmt und ein Volumen mit eine Wahrscheinlichkeitsverteilung über alle Bins per Pixel generiert. Diese Netzwerkarchitektur besteht aus Blöcke mit jeweils 2 oder 3 Convolutional und ReLU Layers, gefolgt von einem Batch Normalisation Layer. Batch Normalisation ist eine Regularisierungstechnik die die Werte in einem Hidden Layer normalisiert, bevor sie in den nächsten Layer weitergereicht werden. Das Netzwerk hat keine Pooling Layers, alle Änderungen in der Auflösung werden durch Downsampling oder Upsampling zwischen Blöcke erreicht.
Dieser Netzwerkarchitektur ist sehr schwer für die GPU[1], was die Batch Größe und Trainingszeit stark beeinflusst.

[1]Graphics Processing Unit

Aus diesem Grund richtet sich die Netzwerkarchitektur dieser Arbeit nach der Netzwerkarchitektur von Billaut et al. [BRT18]. Sie verwenden eine angepasste Version von einem U-net Convolutional Neural Network [RFB15].

3.4.1 U-net

Ein U-net wird häufig bei Image Segmentation angewendet und ist einer Art Autoencoder mit Skip Connections. Im vergleich zu konventionelle Autoencoder, können der Encoder und Decoder nicht getrennt voneinander verwendet werden und bei dem Decoder werden Transposed Convolutions, als Upsampling Methode, verwendet. Das U-net verfügt außerdem über Skip Connections die ermöglichen fein-granuläre Details in dem Output Volumen zu wiederherstellen und helfen mit dem Vanishing Gradient Problem in Backpropagation. Skip Connections konkatenieren bestimmte Layers von dem Encoder mit Layers von dem Decoder, mit der gleichen Dimensionen.

Ein U-net besteht, wie einem Autoencoder, aus einem Encoder und Decoder Teil. Der Encoder besteht aus sogenannten “ConvBlocks”. ConvBlocks bestehen aus 2 Convolutional Layers gefolgt von ReLU und Batch Normalisation. Die ConvBlocks werden gefolgt von einem Pooling Layer die die Dimensionen von dem Volumen verringern. Der Decoder besteht aus ConvBlocks gefolgt von Transposed Convolutions die die Dimensionen von dem Volumen wieder vergrößern. Das letzte Layer ist eine 1×1 Convolutional Layer die das Output Volumen generiert. Skip Connections konkatenieren ConvBlocks aus dem Encoder mit den Transposed Convolutions aus dem Decoder, die die gleiche Dimensionen haben.

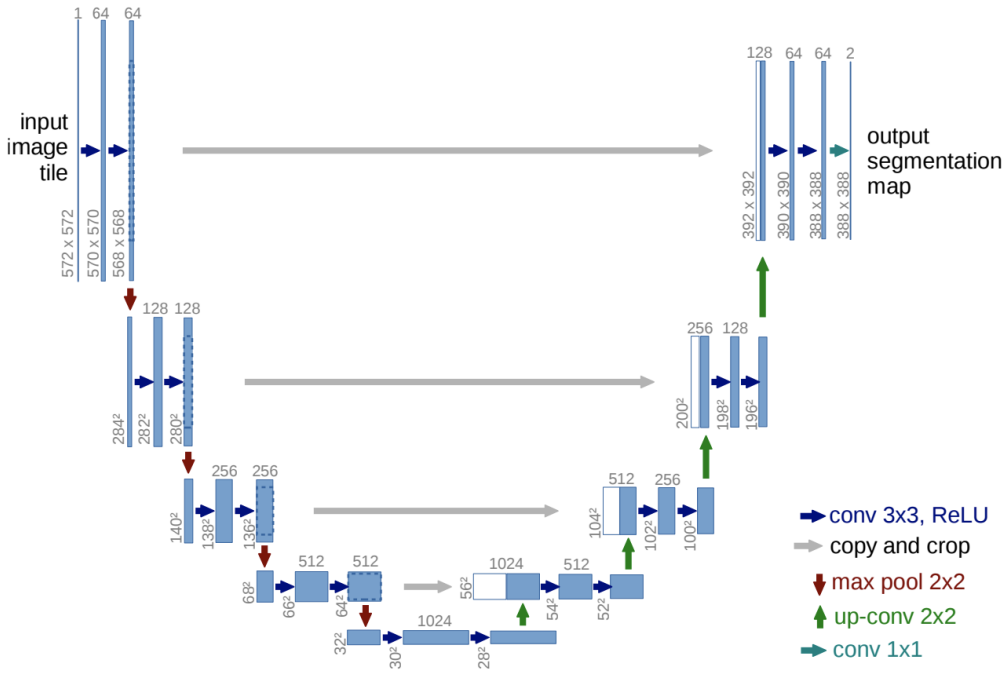

Abbildung 3.3: U-net Architektur (Beispiel für 32×32 Pixels in der niedrigste Auflösung). Jeder blaue Box entspricht ein multi-Kanal Feature Map. Die Tiefe der Feature Maps ist gekennzeichnet durch die Zahl über die Box. Die Breite und Höhe ist durch die Zahl unten links erkennbar. Die weiße Boxen repräsentieren die kopierte Feature Maps. Die Pfeile bestimmen die verschiedene Operationen. [RFB15]

3.5 Datensätze

Um das Netzwerk zu trainieren werden bedeutsame Bilder gebraucht. Ein Vorteil von Image Colorization ist, dass jedes Bild für das Trainieren verwendet werden kann, da nur die Graustufen Version davon gebraucht wird.

Um die Methode zu prüfen werden 3 Datensätze benutzt, die verschiedene Auflösungen und Themen beinhalten.

Als erstes wird ein Spiel–Datensatz von 32×32 Bilder generiert. Dieser setzt sich aus 3 geometrischen Objekten und 3 Farben pro Objekt zusammen. Die geometrischen Objekte sind ein Rechteck, ein Kreis und ein Dreieck pro Bild, die jeweils in eine der 3 Farben eingefärbt sind. Die Bilder haben einen einheitlichen schwarzen Hintergrund.

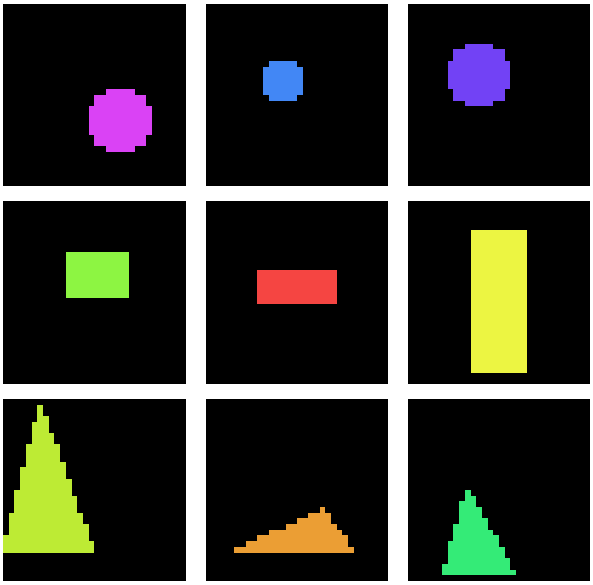

Abbildung 3.4: Beispiel von Trainingsbildern mit einer der möglichen Farbe pro Klasse

Der Datensatz besteht aus 2664 Bilder für das Training und 324 Bilder für das Testen. Dieser Datensatz dient als Beweis für die Methode und hilft bei der Entwicklung und Optimierung des Netzwerks.

Als zweites werden einige Klassen des CIFAR-100[2] Datensatzes verwendet. CIFAR-100 ist ein öffentlich verfügbares Datensatz der von Krizhevsky et al. erstellt wurde. Der Datensatz setzt sich aus 100 Klassen mit jeweils 600 Bildern pro Klasse, außerdem sind die 100 Klassen in 20 Superklassen grupiert. Der Datensatz wird in 50000 Bilder für das Training und 10000 Bilder für das Testen aufgeteilt. Einige Beispiele für Klassen sind z.B. *apples*, *palm* oder *bee*. Die Bildern haben ebenfalls einer Auflösung von 32×32.

Da ein Model von Null auf zu trainieren, die jedes Objekt auf dem CIFAR-100 Datensatz richtig erkennt und einfärbt, sehr aufwendig wäre, werden nur bestimmte Klassen für das Training verwendet. Diese Klassen sind: *apples*, *sunflower*, *cloud*, *oak_tree*, *mountain*, *forest*, *palm_tree* und *plain*. Alle Klassen haben Gemeinsamkeiten, was das erlernen von Merkmale erleichtert im Vergleich zu einem sehr allgemeinen Datensatz.

Abschließend wird ein komplexeres und hochauflösendes Datensatz verwendet, dass Naturbezogene Bilder enthält. Ziel damit ist, ein Netzwerk zu trainieren das Bilder aus Natur

[2]https://www.cs.toronto.edu/ kriz/cifar.html

einfärben kann. Dieser besteht 3 Datensätze aus Kaggle[3] und GitHub[4]. Das erste ist das "Landscape Pictures"[5] Datensatz von Arnaud Rougetet, von diesem Datensatz werden alle Bilder verwendet. Das zweite Datensatz ist das "Landscape Classification"[6] von Huseyb Guliyev, hiervon werden nur die Klassen *forest*, *glacier*, *mountain* und *sea* verwendet. Das letzte Datensatz ist das "Landscapes dataset" [7] von ml5js auf GitHub. Von diesem Datensatz wurden die Klassen *field*, *forest*, *lake*, *mountain* und *road* verwendet.
Das fertige Datensatz besteht aus 8 Klassen und hat insgesamt 13363 Bilder, 10945 für das Training und 2409 für das Testen. Die 8 Klassen sind: *field*, *forest*, *glacier*, *lake*, *mountain*, *road*, *sea* und "ohne Kategorie". Die Klasse "ohne Kategorie" beinhaltet die Bilder aus dem "Landscape Pictures" Datensatz. Die Klassen sind für das Training und die Methode nicht relevant, da jedes Pixel in Bins klassifiziert wird.

3.6 Data Preprocessing und Argumentation

Für das optimale Training und Ergebnisse werden die Bilder vorverarbeitet. Außerdem werden Techniken von Image Argumentation angewendet. Das Spiel–Datensatz und die 8 Klassen von CIFAR-100 werden für das Training mit einer Wahrscheinlichkeit von 50% horizontal gespiegelt.
Da das hochauflösendes Datensatz, Bilder mit verschiedene Auflösungen beinhaltet, werden alle zufällig auf 128×128 angepasst. Das reduziert die Trainingszeit und die Komplexität von dem Datensatz. Für das Training werden die Bilder von diesem Datensatz gleichermaßen mit einer Wahrscheinlichkeit von 50% horizontal gespiegelt. Für die Evaluierung werden die Bilder ebenfalls auf 128×128 angepasst.

3.7 Tools

Um die Methode zu realisieren werden einige Tools angewendet. Für die Implementierung wird das Framework PyTorch[8] angewendet. PyTorch ist ein Open-Source Framework basierend auf Python für Machine Learning und Deep Learning. Es wurde von das Facebook

[3] `https://www.kaggle.com/`
[4] `https://github.com/`
[5] https://www.kaggle.com/arnaud58/landscape-pictures
[6] https://www.kaggle.com/huseynguliyev/landscape-classification?
[7] https://github.com/ml5js/ml5-data-and-models/tree/master/datasets/images/landscapes
[8] https://pytorch.org/

AI Research Team entwickelt und erschien im Jahr 2016. Zum Zeitpunkt der Verfassung dieser Arbeit ist die Version 1.6.0 die aktuellste. Für die Farbraum Konvertierung wird das "scikit-image" Bibliothek eingesetzt.
Das Trainieren von den Modellen wird, wegen den hohen Rechenaufwands, auf zwei verschiedene Plattformen durchgeführt. Die Modelle mit den 32×32 Bildern werden auf Google Colab[9] trainiert. Google Colab ist eine Plattform von Google, die es ermöglicht, Experimente im Browser mit einer Hochleistungsgrafikkarte umzusetzen. Für das Modell mit den 128×128 Bildern wird das Curious Containers (CC) Framework[10] benutzt. Curious Containers ermöglicht eine gleichzeitige Durchführung von verschiedene Experimente in einem Cluster von Hochleistungsrechner.

[9]https://colab.research.google.com/
[10]https://www.curious-containers.cc/

Kapitel 4

Implementierung

In diesem Kapitel wird die Implementierung der Methode näher erläutert. Bei der Implementierung werden alle Konzepte aus dem Kapitel Konzeption angewendet.

4.1 Binning

Das Binning ist eine wichtige Komponente der Methode, dass mit Hilfe der "numpy" Bibliothek implementiert wurde. Als erstes wird anhand der Anzahl von Bins (n_bins) die Breite (B) und Höhe (H) des Grids, das auf der Abbildung 3.2 zu sehen ist, mit Hilfe der Wurzel berechnet. Es wird angenommen dass $B = H$. Nachdem die Breite des Grids berechnet wurde, wird der Intervall von $[0, 1]$ in B gleich große Intervalle aufgeteilt. Als nächstes wird mit Hilfe der *digitize* Funktion, der Intervall Index der jeweiligen a, b Farbkanal Wert von jedem Pixel kalkuliert. Abschließend werden beide Indices zu einem Bin Index auf dem Grid umgewandelt. Der Output ist ein kodiertes Bild mit einem Bin Index per Pixel.

```
# a, b sind die Koordinaten der Farbkanäle auf dem Grid
def calculate_bin(a, b, width):
  return (width * b) + a

def encode_bins(ab_image, n_bins):
  B = np.sqrt(n_bins).astype(int)

  # Intervall in gleich große Intervalle aufteilen
  interval = np.linspace(0, 1, B+1)

  # Indices für jeweils a, b Kanäle berechnen
  indices = np.digitize(ab_image, interval) - 1

  # Bin Index berechnen
  bins = np.vectorize(calculate_bin)(indices[:,:,0], indices[:,:,1], B)

  return bins
```

Code snippet 4.1: Binning eines normalisierten Lab Bildes

4.1.1 Umwandlung von Bin zu Farbe

Für die Umwandlung werden vor dem Training alle Farben von den Trainingsbilder in Bins klassifiziert. Es wird ein Python Dictionary mit Bin Indices als Key und einem 2-Dimensionalen Array mit einem Array pro Farbkanal als Value erzeugt.

```
bin_colors = {i: [[], []] for i in range(n_bins)}
```

Code snippet 4.2: Leeres Dictionary Erzeugung für n_bins

Anschließend werden die Farben von jedem Farbkanal in den jeweiligen Bin Array zugeordnet. Als letztes wird den Durchschnitt pro Bin Index und Farbkanal berechnet. Der Dictionary beinhaltet abschließend ein Array mit 2 einzelne Werte für den jeweiligen Farbkanal pro Bin Index. Somit kann unkompliziert ein Bin in eine Farbe umgewandelt werden.

```
1       a_color = bin_colors[bin_index][0]
2       b_color = bin_colors[bin_index][1]
```

Code snippet 4.3: Bin zu Farbe Umwandlung

Der gleiche Vorgang wird für den Modus angewendet. Um verschiedene Farben erzeugen zu können werden den Durchschnitt und der Modus mit einem Temperatur Wert interpoliert.

```
1       a_distance = a_mode - a_mean
2       b_distance = b_mode - b_mean
3
4       a = a_mode - (a_distance * T)
5       b = b_mode - (b_distance * T)
```

Code snippet 4.4: Bin zu Farbe Berechnung mit einem Temperaturwert

4.2 Datensätze

Die Datensätze werden mit Hilfe der "torchvision" Bibliothek von PyTorch importiert und transformiert. Für das Importieren wurde eine "ImageFolder" implementiert, der die Bilder importiert, transformiert, normalisiert und in Bins kodiert. Der Output von den "ImageFolder" sind das Graustufenbild, das Bild mit den "ab" Farbkanäle und das in Bins kodierte Bild.

Mit Hilfe eines "DataLoaders" werden die Datensätze in Batches aufgeteilt und gemischt.

4.3 Netzwerkarchitektur

Für dieser Arbeit wurden 2 U-Nets mit verschiedene Größen verwendet. Ein U-Net für 32×32 Bilder und ein U-Net für 128×128 Bilder. Außerdem wurde das U-Net für 32×32 Bilder angepasst damit es mit ein MSE Loss verwendet werden kann. Bei der Anpassung wurde der Output Volumen zu $W_{Input} \times H_{Input} \times 2$ geändert, wobei das Netzwerk direkt die Werte für die "ab" Farbkanäle vorhersagt. Bei der Verwendung von einem MSE Loss wird Binning nicht verwendet.

4.3.1 ConvBlock

Das Kernstück eines U-Nets ist der sogenannte ConvBlock. Ein ConvBlock beinhaltet zwei hintereinander geschaltete Blöcke, die wiederrum aus einer Convolutional Layer mit 3×3 Filtern und Stride 1, gefolgt von ReLU und Batch Normalisation bestehen. Die Convolutional Layers verwenden Padding um die Dimensionen von dem Input nicht zu verändern. Die Layers wurden mit den PyTorch Klassen *Conv2d*, *BatchNorm2d* und die Funktion *relu* implementiert. Das ConvBlock wird mit einer Klasse implementiert die wiederrum von der Oberklasse *torch.nn.Module* erbt.

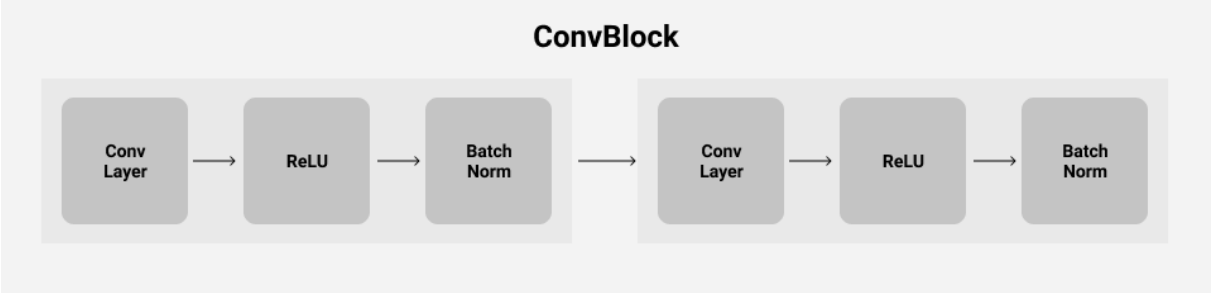

Abbildung 4.1: ConvBlock

4.3.2 U-Net

Das U-Net wird mit den Spezifikationen aus 3.4.1 implementiert. Dies geschieht mit einer Klasse, die von der Oberklasse *torch.nn.Module* erbt. Das U-Net verwendet neben den ConvBlocks, Max Pooling Layers mit einem 2×2 Filter und Stride 2, um die Dimensionen in den Encoder zu halbieren. Die kleinste Größe der Feature Maps bei dem Encoder ist 8×8. Der Decoder verwendet ebenfalls neben den ConvBlocks, Transposed Convolutions mit 3×3 Filtern, Stride 2 und Padding 1, die die Größe der Feature Maps verdoppeln. Der letzte Layer ist ein Convolutional Layer mit 1×1 Filter, Stride 1 und Padding 0, der die Wahrscheinlichkeitsverteilung pro Pixel generiert. Alle Transposed Convolutions werden mit den Outputs der ConvBlocks mit den gleichen Dimensionen aus dem Encoder konkateniert. Es werden zwei verschieden große U-Nets implementiert, eins für 32×32 Bilder und eins für 128×128 Bilder.
Die Max Pooling Layers wurden mit der *MaxPool2d* Klasse und die Transposed Convolutions mit der *ConvTranspose2d* implementiert.

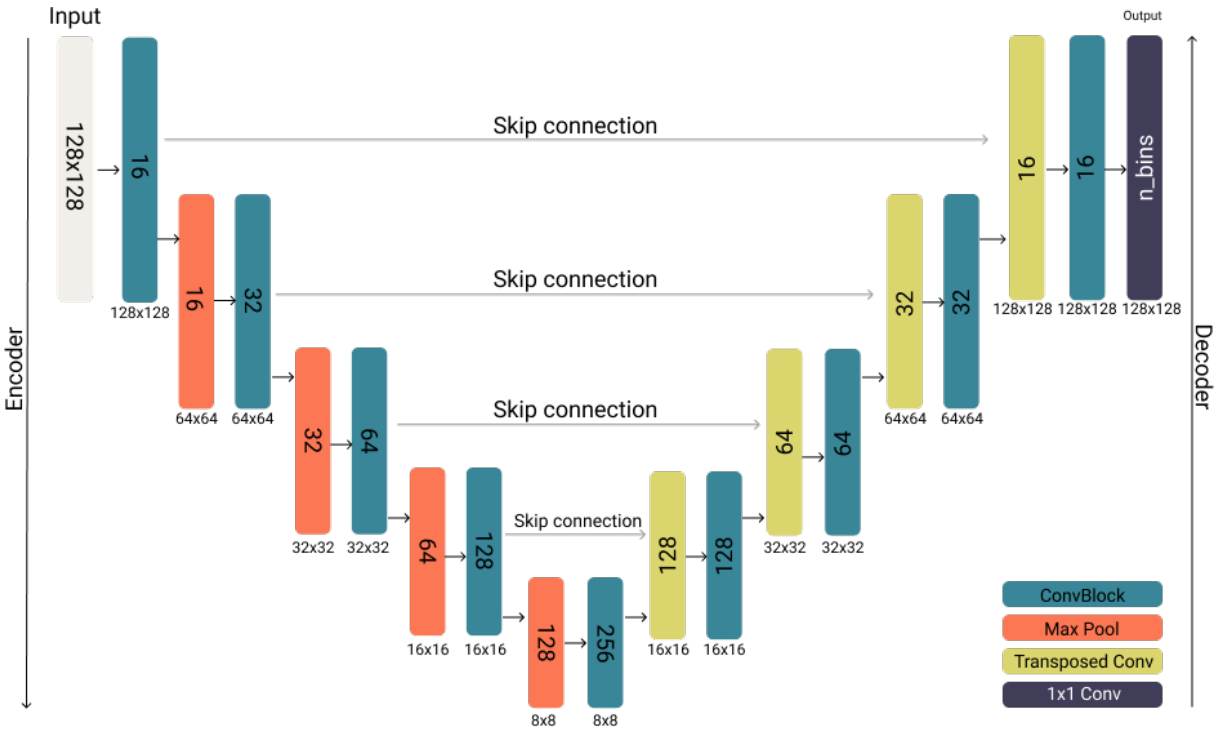

Abbildung 4.2: U-Net Architektur für 128 × 128 Input Bilder

Kapitel 5

Tests und Experimente

In diesem Kapitel werden Experimente mit den drei Datensätze durchgeführt. Ebenfalls werden verschiedene Hyperparameter getestet.

5.1 Spiel-Datensatz Experimente

Als erstes wird die Methode auf dem Spiel-Datensatz angewendet. Hiermit soll geprüft werden ob die Methode die erwartete Ergebnisse liefert. Das U-Net wird für 100 Epochen mit Adam, eine Lernrate von 0.001 und die Cross Entropy Loss Funktion trainiert. Dieser Einstellung erwies die besten Ergebnisse. Außerdem wurde ein Experiment mit 36 und ein mit 324 Bins durchgeführt, was kein Einfluss auf die Ergebnisse vorwies. Es kann davon ausgegangen werden, dass bei der niedrige Anzahl an möglichen Farben, ein Unterschied bei 36 und 324 Bins nicht zu erkennen ist. Die unteren Ergebnissen wurden mit 324 Bins erstellt.

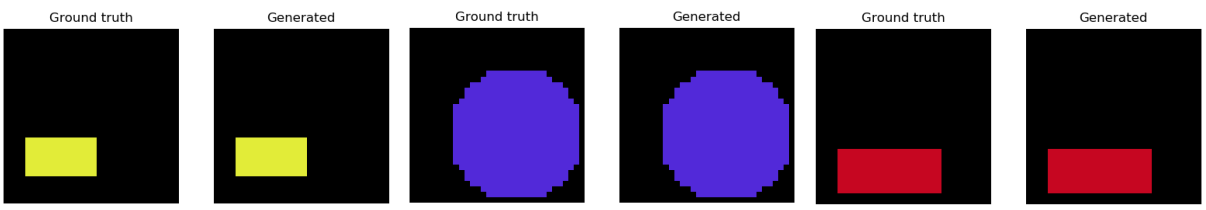

Abbildung 5.1: Beispiele von sehr gute Ergebnisse aus dem Spiel-Datensatz

Bei den oberen Ergebnisse wurden alle Pixeln richtig klassifiziert, was bei der Größe des Datensatzes oft zu overfitting deutet. Die unteren Ergebnissen zeigen dass das Model generalisiert und nicht overfitet hat.

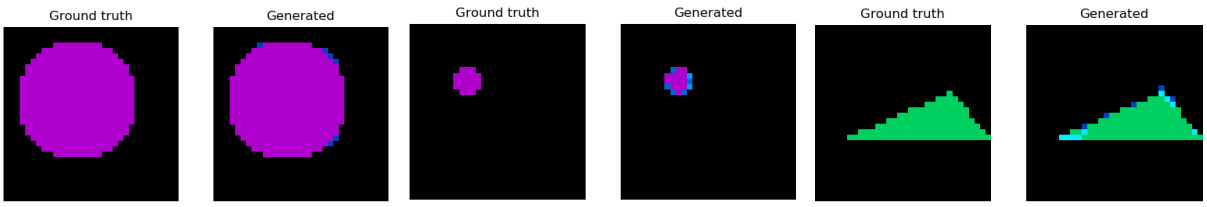

Abbildung 5.2: Beispiele von generalisierte Ergebnisse

Das Model hat bei einige Ergebnisse, Schwierigkeiten die Pixeln am Rand der geometrische Formen, richtig zu klassifizieren. Dies tritt speziell auf die Kreise und Dreiecke wo die Ränder nicht glatte Linien sind.
Die Ergebnisse bestätigen dass das Binning und die Methode funktionieren. Anschließend wurden Experimente auf komplexere Bilder von dem Subset von CIFAR-100 durchgeführt.

5.2 CIFAR-100 Subset Experimente

Das Model wurde auf 12 Klassen von CIFAR-100 über 100 Epochen mit Adam, eine Lernrate von 0.001 und die Cross Entropy Loss Funktion trainiert.

Kapitel 6

Evaluation

6.1 Vergleich der Modelle

TODO

Kapitel 7

Fazit

TODO

7.1 Zusammenfassung

TODO

7.2 Kritischer Rückblick

TODO (Reflexion und Bewertung der Zielsetzung gegenüber erreichtem Ergebnis)

7.3 Ausblick

TODO

Abbildungsverzeichnis

Tabellenverzeichnis

Source Code Content

Glossar

Bin Behälter. 14, 17

CIE Internationale Beleuchtungskommission. 14

CNN Convolutional Neural Network (Gefaltetes Neuronales Netzwerk). 6, 7, 19, IV

Grid Raster. 17, 18, 25, I

Layer Schicht. 2, 7, 8, I

Loss Function Kostenfunktion. 9, 10, 12, 14

Stride Schrittweite einer Faltung bei einem CNN. 7, 8, I

Literaturverzeichnis

[DHS11] John Duchi, Elad Hazan und Yoram Singer. „Adaptive subgradient methods for online learning and stochastic optimization“. In: *Journal of Machine Learning Research* 12.Jul (2011), S. 2121–2159.

[GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[LLW04] Anat Levin, Dani Lischinski und Yair Weiss. „Colorization Using Optimization“. In: *ACM Trans. Graph.* 23.3 (Aug. 2004), S. 689–694. ISSN: 0730-0301. DOI: `10.1145/1015706.1015780`. URL: `https://doi.org/10.1145/1015706.1015780`.

[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: 9781605589077. (Besucht am 27. 07. 2020).

[Zha20] Aston Zhang u. a. *Dive into Deep Learning*. `https://d2l.ai`. 2020.

Onlinereferenzen

[15] *Backpropagation*. 2015. URL: http://www.inztitut.de/blog/glossar/backpropagation/ (besucht am 01. 08. 2020).

[Bec19] Roland Becker. *Convolutional Neural Networks – Aufbau, Funktion und Anwendungsgebiete*. 2019. URL: https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsgebiete-1691/ (besucht am 01. 08. 2020).

[Dip19] Nico Litzel Dipl.-Ing. (FH) Stefan Luber. *Was ist ein Convolutional Neural Network?* 2019. URL: https://www.bigdata-insider.de/was-ist-ein-convolutional-neural-network-a-801246/ (besucht am 01. 08. 2020).

[Fei17a] Serena Yeung Fei-Fei Li Justin Johnson. *Neural Networks 1*. 2017. URL: https://cs231n.github.io/neural-networks-1/ (besucht am 12. 07. 2020).

[Fei17b] Serena Yeung Fei-Fei Li Justin Johnson. *Optimization 2*. 2017. URL: https://cs231n.github.io/optimization-2/ (besucht am 01. 08. 2020).

[Moe18] Julian Moeser. *Funktionsweise und Aufbau künstlicher neuronaler Netze*. 2018. URL: https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/ (besucht am 10. 07. 2020).

[Ngu20] Hoang Tu Nguyen. *Einführung in die Welt der Autoencoder*. 2020. URL: %5Curl%7Bhttps://data-science-blog.com/blog/2020/04/01/einfuhrung-in-die-welt-der-autoencoder/%7D (besucht am 01. 08. 2020).

Bildreferenzen

[Bha18] Saugat Bhattarai. *What is gradient descent in machine learning?* 2018. URL: https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/ (besucht am 01. 08. 2020).

[Com15] Wikimedia Commons. *Typical CNN architecture*. 2015. URL: https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png (besucht am 03. 04. 2018).

[Fei20a] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: https://cs231n.github.io/neural-networks-1/ (besucht am 10. 07. 2020).

[Fei20b] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: https://cs231n.github.io/neural-networks-1/ (besucht am 12. 07. 2020).

[Fei20c] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: https://cs231n.github.io/convolutional-networks/ (besucht am 01. 08. 2020).

[Fei20d] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: https://cs231n.github.io/optimization-2/ (besucht am 01. 08. 2020).

Anhang A

TODO

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 XX.XX.2018

Adrian Saiz Ferri