



**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: 03.01.2021

Danksagung

Ich bedanke mich bei allen Personen die mich während meines Studiums und besonders bei der Erstellung dieser Arbeit unterstützt haben. Ich bedanke mich bei meinen Betreuern Frau Prof. Dr. Christin Schmidt und Herrn Patrick Baumann für die Unterstützung während der Arbeit und die Bereitstellung von technischen Ressourcen.

Abstract

Die vorliegende Arbeit beschäftigt sich mit der Einfärbung von Graustufenbildern durch Convolutional Neuronal Networks. Es werden Methoden von Image Colorization untersucht, die das Problem als ein multimodales Problem behandeln. Außerdem wird eine Methode implementiert, um vergleichen zu können. Um das Modell zu trainieren wird die Graustufenversion des jeweiligen Farbbildes verwendet. Die Graustufenbilder werden in das Netzwerk eingespeist und daraus werden die Farbkanäle erzeugt. Am Ende wird das Graustufenbild mit den Farbkanälen konkateniert, um das Farbbild zu generieren.

Zunächst werden mehrere Tests mit verschiedenen Hyperparametern durchgeführt. Anschließend werden die Tests ausgewertet, die Ergebnisse untersucht und mit anderen Methoden verglichen.

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	1
1.3 Vorgehensweise und Aufbau der Arbeit	2
2 Grundlagen	3
2.1 Neuronale Netze	3
2.1.1 Feedforward Neural Network	4
2.1.2 Fully-connected Neural Network	4
2.1.3 Aktivierungsfunktionen	7
2.1.4 Loss Functions	10
2.1.5 Optimierungsalgorithmen	11
2.1.6 Backpropagation	13
2.2 Convolutional Neural Networks	14
2.3 Transposed Convolution	16
2.4 Autoencoder	17
2.5 <i>Lab</i> -Farbraum	18
2.6 Image Colorization Methoden	18
2.7 Verwandte Arbeiten	19
3 Konzeption	20
3.1 Image Colorization als Multimodales Problem	20
3.2 Farbraum	21
3.3 Binning	22
3.4 Netzwerkarchitektur	23
3.4.1 U-net	24
3.4.2 Aufbau eines U-nets	25
3.5 Datensätze	26
3.6 Image Preprocessing und Augmentation	28

3.7 Tools	29
4 Implementierung	30
4.1 Binning	30
4.2 Datensätze	32
4.3 Netzwerkarchitektur	32
4.3.1 ConvBlock	33
4.3.2 U-Net	33
5 Test	35
5.1 Spiel-Datensatz Training	35
5.2 CIFAR-100 Subset Training	36
5.3 Landscape Datensatz Training	39
6 Evaluation	41
6.1 Evaluationsmetrik	41
6.2 Evaluation des Spiel-Datensatzes	41
6.3 Evaluation des CIFAR-100 Subsets	42
6.4 Evaluation des Landscape Datensatzes	44
7 Fazit	46
7.1 Zusammenfassung	46
7.2 Kritischer Rückblick	46
7.3 Ausblick	47
Abbildungsverzeichnis	I
Source Code Content	IV
Glossar	V
Abkürzungsverzeichnis	VI
Literaturverzeichnis	VII
Onlinereferenzen	VIII
Bildreferenzen	IX

Anhang A	XI
A.1 Ergebnisse aus dem Landscape Datensatz	XI
A.2 Notebook CIFAR-100 Subset Colorization	XIII
A.3 Skripte Lanscape Datensatz Colorization	XXIII
A.3.1 Image augmentation Skript	XXIII
A.3.2 Modell Skript	XXV
A.3.3 Utils Skript	XXVIII
A.3.4 Train Skript	XXXII
A.3.5 Validation Skript	XXXIII
A.3.6 Main Skript	XXXIV
Eigenständigkeitserklärung	XL

Kapitel 1

Einleitung

Das Kapitel Einleitung verschafft einen Überblick über die Motivation, die angestrebte Zielsetzung, das genaue Vorgehen und den Aufbau der Arbeit.

1.1 Motivation

Jeder hat sich sicherlich gefragt, vor allem wenn es um Familienbilder geht, wie ein Graustufenbild in Farbe aussehen würde. Es wäre faszinierend zu sehen wie die Welt von damals in Farbe aussehen würde, etwas dass mich seit jeher fasziniert hat. Ein Graustufenbild kann interaktiv von einem Menschen gefärbt werden, sodass Farben möglichst akkurat vergeben werden. Wenn jedoch mehrere Tausend Bilder zu bearbeiten sind, würde das einige Zeit in Anspruch nehmen. Dieses Problem kann mit Deep Learning gelöst werden, indem ein Algorithmus selbstständig und möglichst realistisch ein Graustufenbild färbt. Der Prozess der Einfärbung eines Bildes ist ein aktives Forschungsgebiet im Deep Learning. Es gibt bereits Methoden, basierend auf Convolutional Neuronal Networks, die sehr realistische Ergebnisse liefern.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die bestehenden Methoden zu untersuchen und zu vergleichen. Es wird eine Methode implementiert, um die Ergebnisse mit den bestehenden Methoden zu vergleichen. Der Fokus wird auf Methoden, die das Problem von Image Colorization

als ein Multimodales Problem behandeln, gelegt und implementiert auf Grund dessen Klassifikationsmethoden.

1.3 Vorgehensweise und Aufbau der Arbeit

Die vorliegende Arbeit lässt sich in fünf Hauptkapitel aufteilen. Zu Beginn wird eine ausführliche Erläuterung der Grundlagen gegeben, um die Methoden und Techniken der Arbeit zu verstehen. Anschließend werden die Konzepte, Techniken und Methoden präsentiert. Um auf die bevorzugte Methode aufbauen zu können werden einige Datensätze gebraucht, diese werden bei den Konzepten präsentiert. Nachdem auch diese erläutert wurden, wird die Implementierung erklärt. Darauf folgend werden zahlreiche Tests mit verschiedenen Hyperparametern durchgeführt. Abschließend werden die Tests evaluiert und mit anderen Methoden verglichen.

Kapitel 2

Grundlagen

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

2.1 Neuronale Netze

Künstliche Neuronale Netze sind vom menschlichen Gehirn inspiriert und werden für Künstliche Intelligenz und Maschinelles Lernen angewendet. Genutzt werden sie für überwachtes und unüberwachtes lernen. In der vorliegenden Arbeit werden nur Methoden des überwachten lernens angewendet. Beim überwachten lernen sind die Datensätze gelabelt, sodass der Output des Neuronalen Netzes mit den richtigen Ergebnissen verglichen werden kann.

Neuronale Netze bestehen aus Neuronen, auch “Units” genannt, die schichtweise in “Layers” (Schichten) angeordnet sind. Beginnend mit der Eingabeschicht (Input Layer) fließen Informationen über eine oder mehrere Zwischenschichten (Hidden Layers) 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 eine Funktion f^* . Ein Feedforward Neural Network definiert eine Abbildung $y = f(x; W)$ wobei x der Input ist und W die lernbaren Parameter sind (auch Gewichte genannt). [GBC16, S. 164-223]

Diese Netzwerkarchitektur trägt den Namen “feedforward” weil der Informationsfluss von dem Input Layer, über die Hidden Layers bis zum Output Layer in eine Richtung weitergereicht wird.

Feedforward Neural Networks werden als eine Kette von Funktionen dargestellt. So, 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 Neuronalen Netzwerken. Im genannten Beispiel ist $f^{(1)}$ die erste Layer, $f^{(2)}$ die zweite und $f^{(3)}$ die Output Layer von diesem Netzwerk. Die Länge dieser Kette definiert die Tiefe des Netzwerks. Je tiefer ein Netzwerk ist, desto mehr erlernbare Parameter besitzt es und braucht somit eine erhöhte Rechenleistung, um trainiert zu werden. In der Praxis sind 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 geben genau vor, was die Output Layer generieren soll. Die Output Layer soll Werte generieren, die nah an y liegen. Das Verhalten der Hidden Layers wird nicht durch die Trainingsbeispiele festgelegt, sondern der Lernalgorithmus soll selbst definieren, wie diese Layers verwendet werden, um die beste Annäherung von $f^*(x)$ zu erzielen.

2.1.2 Fully-connected Neural Network

Fully-connected Neural Networks sind die am häufigsten vorkommende Art von Neuronalen Netzen. In dieser Netzwerkarchitektur sind alle Neuronen eines Layers mit allen Neuronen des vorherigen und des nächsten Layers verbunden. Neuronen die sich im selben Layer befinden, sind jedoch nicht miteinander verbunden. [Fei17a]

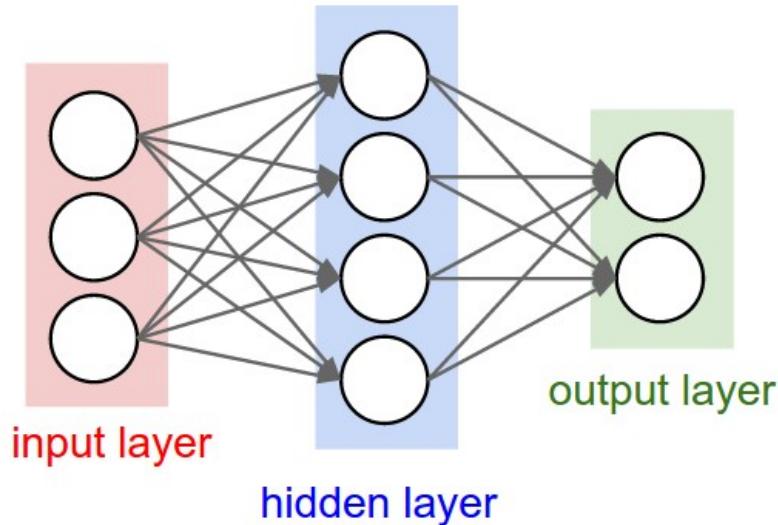


Abbildung 2.1: Fully-connected Neural Network mit 2 Layers (eine Hidden Layer mit 4 Neuronen und eine Output Layer mit 2 Neuronen) [Fei20a]

Einer der wichtigsten Gründe für die Anordnung von Neuronalen Netzen in Layers ist, dass so eine Struktur anhand von Matrix Multiplikationen berechnet werden kann. Die Abbildung 2.1 stellt ein Netzwerk mit 3 Inputs x , einer Hidden Layer mit 4 Neuronen und einer Output Layer mit 2 Neuronen dar. Die Kreise repräsentieren die Neuronen und einen Bias Wert b , die Pfeile stellen die Gewichte w dar.

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

Nach jedem Hidden Layer läuft der Output durch eine Aktivierungsfunktion σ die unter Kapitel 2.1.3 erklärt wird. Dadurch wird die vorherige Formel um σ erweitert:

$$f(x) = \sigma(w * x + b) \quad (2.2)$$

Forward Pass

Der Forward Pass von einem Neuronalen Netz wird anhand von Matrizen Multiplikationen berechnet. Um dies zu veranschaulichen wird es anhand eines Beispiels erklärt.

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

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

X sind die Inputs, W die Gewichte des Hidden Layers und W_{out} die Gewichte des Output Layers. Die erste Spalte aus dem Input X und die ersten Zeilen aus beide Gewichtsmatrizen W und W_{out} sind die Werte für den Bias. Diese Anordnung des Bias Wertes ermöglicht die Berechnung durch eine einzige Matrix Multiplikation. Als Aktivierungsfunktion wird ReLU [NH10] verwendet:

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

Im ersten Schritt durchläuft der Input die Hidden Layer $f(X \times W)$:

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

Im zweiten Schritt wird der Output von der vorherigen Multiplikation mit den Gewichten des Output Layers multipliziert:

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

2.1.3 Aktivierungsfunktionen

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

Sigmoid

Sigmoid ist eine nicht lineare Funktion welche die Werte in einem Wertebereich von $[0, 1]$ bringt. Große negative Werte werden annähernd 0 und große positive Werte werden annähernd 1. Sigmoid hat einige Nachteile, so neigt es dazu den Gradienten verschwinden zu lassen und die Outputs sind nicht Null zentriert. Sigmoid wird definiert als:

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

wobei x ein Input ist.

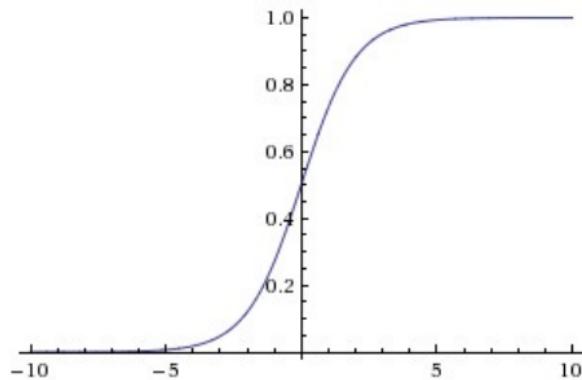


Abbildung 2.2: Sigmoid Aktivierungsfunktion [Fei20b]

Tanh

Die Tanh Aktivierungsfunktion bringt Werte in einen Wertebereich von $[-1, 1]$. Es ist eine skalierte Sigmoid (σ) Funktion und ist definiert als:

$$\tanh(x) = 2\sigma(2x) - 1 \quad (2.8)$$

wobei x ein Input ist. Die Nachteile von Tanh ähneln den von Sigmoid, wobei jedoch der Output Null zentriert ist.

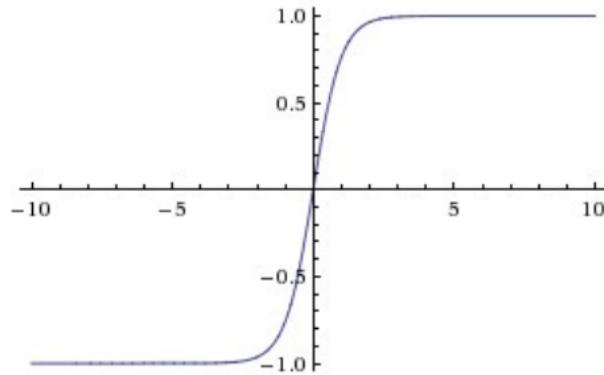


Abbildung 2.3: Tanh Aktivierungsfunktion [Fei20b]

ReLU

Die Rectified Linear Unit konvertiert alle negativen Werte zu 0 und alle positiven Werte behalten ihre Identität. Diese Aktivierungsfunktion wurde für die Netzwerke in dieser Arbeit verwendet, da sie Vorteile gegenüber Sigmoid und Tanh zeigt. Einer der Vorteile ist, dass die mathematische Auswertung der Funktion unkompliziert ist. Außerdem beschleunigt sie die Konvergenz des Stochastischen Gradientenabstiegsverfahrens im Vergleich zu Sigmoid. ReLU ist definiert als:

$$f(x) = \max(0, x) \quad (2.9)$$

wobei x ein 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, wodurch das Neuron nie wieder aktiviert werden kann. Mit einer korrekten Einstellung der Lernrate kann das vermieden werden. [Fei17a]

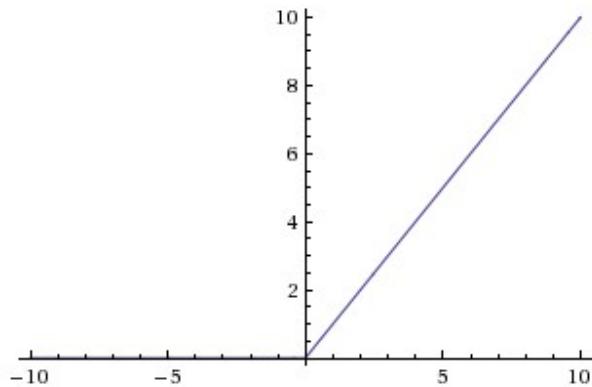


Abbildung 2.4: Rectified Linear Unit (ReLU) [Fei20b]

Leaky ReLU

Leaky ReLU ist eine Variante der ReLU Aktivierungsfunktion, die versucht das Problem der "sterbenden" Neuronen zu minimieren. Anstatt alle negativen Werte zu Null zu konvertieren, werden die Werte mit einer Konstanten multipliziert. Die Funktion wird zu,

$$f(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x) \quad (2.10)$$

wobei x ein Input ist und α eine Konstante mit geringerem Wert ist, zum Beispiel 0.001.

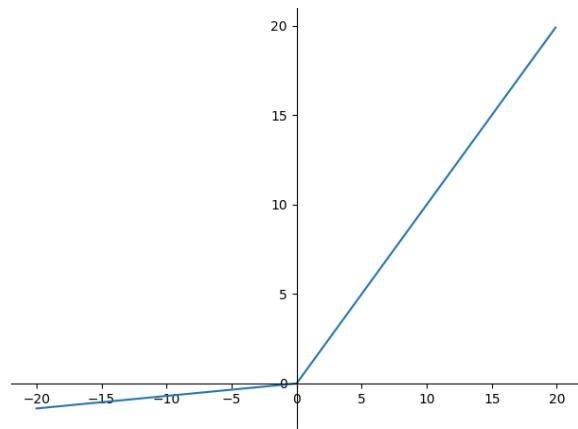


Abbildung 2.5: Leaky ReLU [ccs20]

2.1.4 Loss Functions

Die Loss Function (Kostenfunktion) dient zur Feststellung der Fehler (Loss) zwischen dem Output von einem Modell und dem vorgesehenen Zielwerten. 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 den mittleren quadratischen Fehler und ist definiert als:

$$J = \frac{1}{N} \sum (y - \hat{y})^2 \quad (2.11)$$

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 Klassifizierungsproblemen verwendet. Es wird unterschieden zwischen, Binär und Multiclass Cross Entropy Loss. Bei dem Multiclass Cross Entropy Loss wird ein Vektor mit einer Wahrscheinlichkeitsverteilung $x \in [0, 1]$ ausgewertet, wenn die korrekte Klasse eine 1 besitzt ist der Loss 0. Dabei gilt: je weniger Wahrscheinlichkeit die korrekte Klasse besitzt, desto höher wird der Loss sein. Der Multiclass Cross Entropy Loss ist definiert als:

$$J = -\frac{1}{N} \left(\sum_{i=0}^N y_i * \log(\hat{y}_i) \right) \quad (2.12)$$

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

Weighted Cross Entropy Loss

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

2.1.5 Optimierungsalgorithmen

Das Ziel von Optimierungsalgorithmen ist eine Kombination von Gewichten W zu finden, die die Loss Function minimieren. Es gibt diverse relevante Optimierungsalgorithmen. In der vorliegenden Arbeit werden Gradient Descent, Adam und RMSProp verwendet.

Gradient Descent

Gradient Descent (Gradientenabstiegsverfahren) ist ein iteratives Verfahren, um bei einer Funktion das Minimum (oder das Maximum) zu finden. Mit Hilfe von partiellen Ableitungen kann der Gradient von einer Funktion berechnet werden. Ein Gradient ist, im Fall von Neuronalen Netzen, ein Vektor, der zum höchsten Punkt der Loss Function zeigt. Wird der negative Gradient genommen, zeigt dieser zum tiefsten Punkt. Bei jeder Kombination von Gewichten wird der Gradient berechnet und mit einer bestimmten 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:

$$w_{x+1} = w_x - \alpha * \nabla J(w_x) \quad (2.13)$$

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

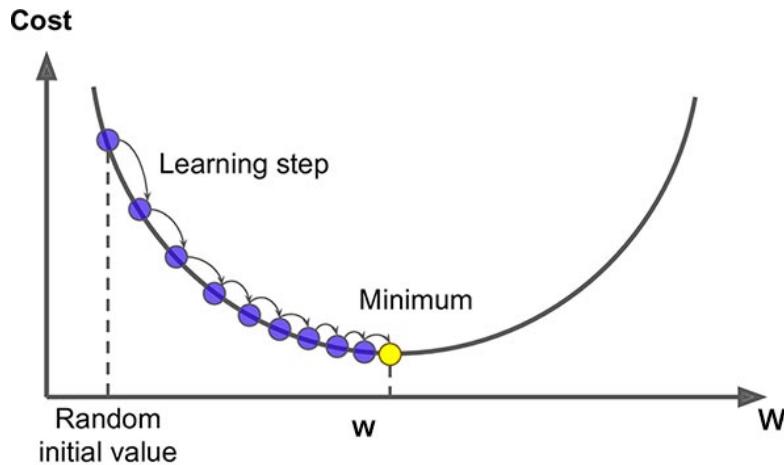


Abbildung 2.6: Gradient descent visualisiert [Bha18]

Es gibt verschiedene Arten von Gradient Descent: Gradient Descent, Mini-Batch Gradient Descent und Stochastic Gradient Descent. Beim Gradient Descent werden die Gradienten im Bezug zu dem gesamten Datensatz berechnet und damit das Update durchgeführt. Der Mini-Batch Gradient Descent berechnet die Gradienten im Bezug zu einem kleinen Teil des Datensatzes und führt ein Update für alle Parameter durch. Der Stochastic Gradient Descent berechnet den Gradient bezogen auf ein einzelnes Element des Datensatzes und führt einen Update für alle Parameter durch.

Um die Konvergenz Richtung Minimum zu beschleunigen wurde der Gradient Descent mit Momentum entwickelt. Bei diesem Ansatz wird ein Geschwindigkeitsparameter zu der Update Regel hinzugefügt, der alle vorherigen Updates akkumuliert. Das ermöglicht die schnellere Konvergenz mit jedem Schritt. Die neue Update Regel ist definiert als:

$$\begin{aligned} v_{t+1} &= \rho v_t - \alpha * \nabla J(w_x) \\ w_{x+1} &= w_x + v_{t+1} \end{aligned} \tag{2.14}$$

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

Adam

Adam steht für “Adaptive Moment Estimation Algorithm” und ist ein Optimierungsalgorithmus, der eine angepasste Lernrate für die verschiedenen Parameter berechnet [KB14]

zusätzlich zu der Speicherung des exponentiell abnehmenden Mittelwerts vorangegangener Gradienten. 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, der eine angepasste Lernrate für die verschiedenen Parameter einführt.

2.1.6 Backpropagation

Neuronale Netze lernen indem der Loss minimiert wird. Wie in der vorherigen Sektionen erläutert, bestimmt die Loss Function die Fehlerrate von einem Neuronalen Netz. Dieser Loss kann mit Hilfe von einem Optimierungsalgorithmus reduziert werden. Backpropagation ermöglicht eine effiziente 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 wird die folgende Sigmoid Funktion in Unterfunktionen zerlegt:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} \quad (2.15)$$

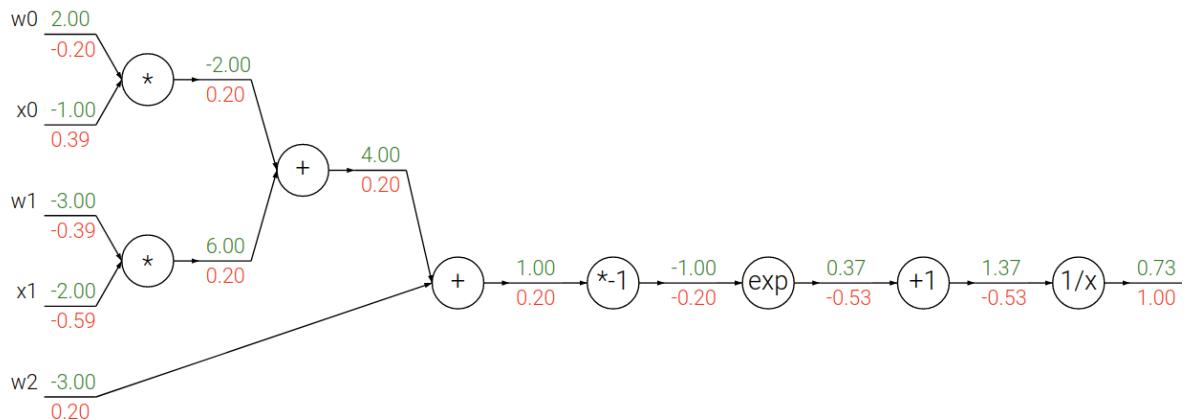


Abbildung 2.7: Backpropagation Beispiel anhand eines 2D Neurons mit der Aktivierungsfunktion Sigmoid [Fei20c]

Auf der Abbildung 2.7 stellen $[w_0, w_1, w_2]$ die Gewichte und $[x_0, x_1]$ die Inputs des Neurons dar. Um es unkompliziert zu halten wird die obere Funktion als eine beliebige Funktion, die

Inputs (w, x) entgegennimmt und eine einzelne Zahl als Output hat, visualisiert. Die grünen Zahlen repräsentieren die Ergebnisse aus dem Forward Pass und die roten Zahlen den zurück propagierten Loss. Jeder Knoten ist fähig ein Output und den lokalen Gradienten des Outputs im Bezug auf den Input zu berechnen, ohne die komplette Funktion kennen zu müssen [Fei17b].

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) sind eine besondere Form von künstlichen neuronalen Netzwerken, die unter anderem speziell für die Verarbeitung von Bilddaten vorgesehen sind [Dip19].

Im Gegensatz zu traditionellen neuronalen Netzwerken, die einen Vektor als Input nehmen, nehmen Convolutional Neural Networks ein 3D Volumen als Input ($W \times H \times C$, hierbei ist W die Breite, H die Höhe und C sind die Farbkanäle).

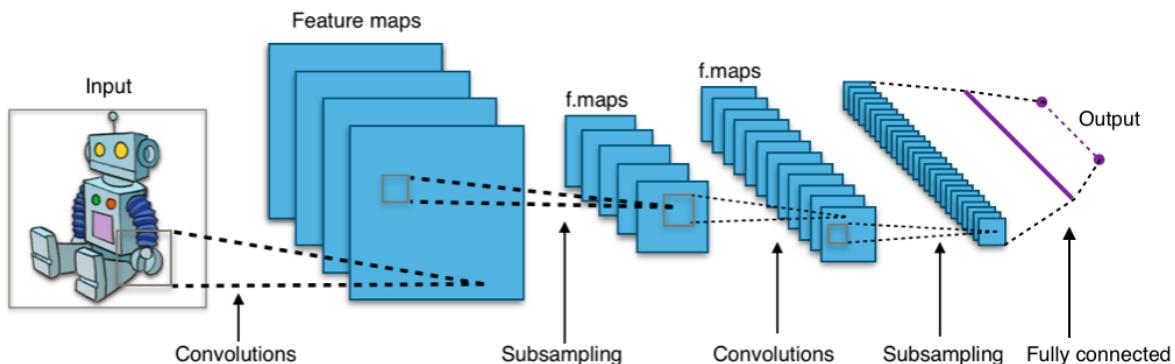


Abbildung 2.8: Typische Struktur von einem Convolutional Neural Network [Com15]

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

Die Convolutional Layer besteht aus mehreren hintereinander geschalteten 3 dimensionalen Filtern, auch Kernel genannt ($W \times H \times D$, wobei D die Tiefe der Feature Maps darstellt), die während dem Forward pass mit einer festgelegten Schrittweite (Stride), über das Bild geschoben werden. Mit dem sogenannten Padding wird das Verhalten an den Rändern festgelegt. An jeder Stelle wird eine Matrix Multiplikation zwischen dem Filter und der aktuellen Position auf dem Bild durchgeführt. Als Output wird eine 2 dimensionale Feature Map 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 gleichen Größe wie dem Input Volumen.

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

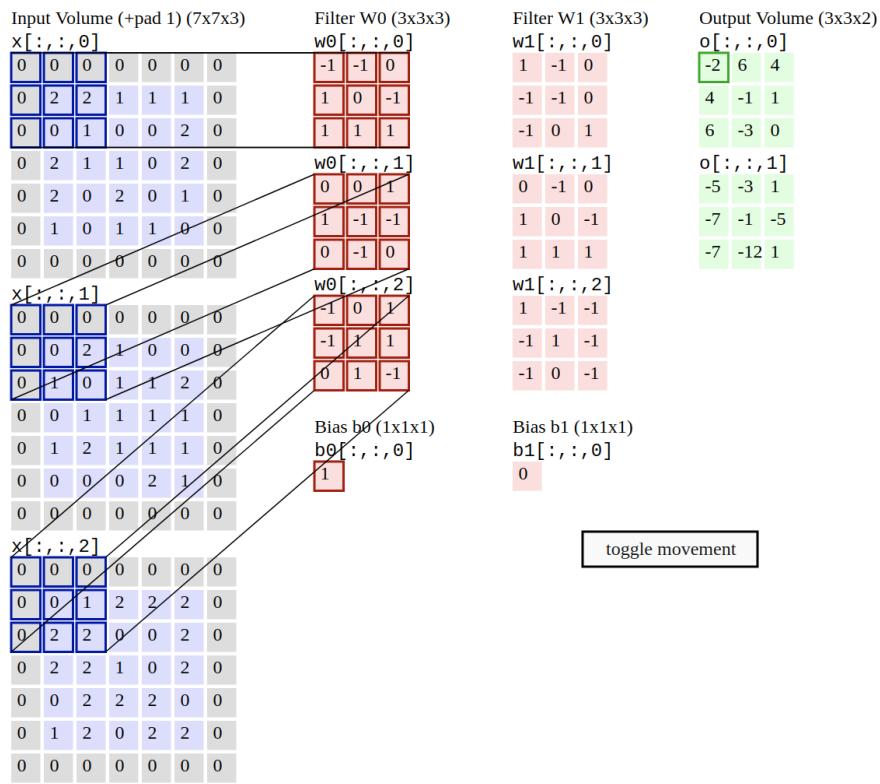


Abbildung 2.9: Beispiel eines Forward pass von einer Convolutional Layer mit einem $7 \times 7 \times 3$ Input Volumen, zwei $3 \times 3 \times 3$ Filtern, Padding 1 und Stride 2. [Fei20d]

Die Pooling Layer dient zur Reduktion der Dimensionen von einem Input Volumen und somit den Parametern vom 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 (auch Max Pooling

genannt) verwendet.

Eine Max Pooling Layer aggregiert die Aktivierungsmatrizen von Convolutional Layers in dem nur die höchste Zahl eines Filters weitergegeben wird. So wird bei einem 2×2 Filter von 4 Zahlen nur eine Zahl weitergegeben. Damit wird einer Reduktion der Dimensionen erreicht.

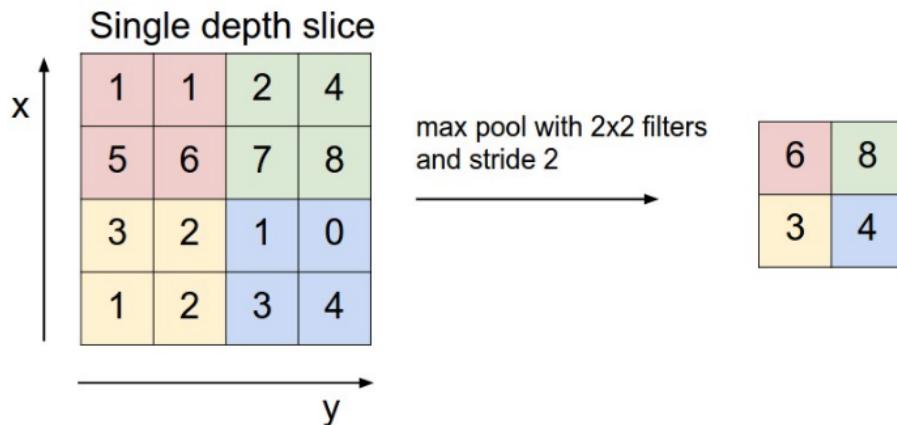


Abbildung 2.10: Max pooling Operation mit 2×2 Filtern und Stride 2 [Fei20d]

2.3 Transposed Convolution

Im Gegensatz zu einer Pooling Layer ermöglicht eine 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 der folgenden Output.

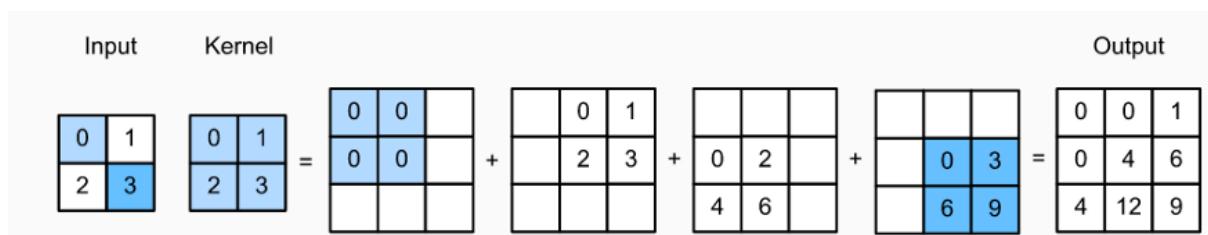


Abbildung 2.11: Die komplette Transposed Convolution Operation [Zha20]

Jede Zahl in der Input Matrix wird mit jeder Zahl in den Filtern multipliziert. Daraus ergibt sich an jeder Position in der Input Matrix einer 2×2 Matrix. Die sich überlappenden

Zahlen auf der Output Matrix werden addiert. Daraus ergibt sich eine 3×3 Output Matrix.

2.4 Autoencoder

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

Encoder

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

$$h = f(x) \quad (2.16)$$

wobei x ein Input ist, f der Encoder und h die Kodierten Features von x .

Decoder

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

$$\hat{x} = g(h) \quad (2.17)$$

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

In der vorliegenden Arbeit wird eine Variante von Autoencoder, basierend auf Convolutional Neural Networks, eingesetzt.

2.5 Lab-Farbraum

Der *Lab*-Farbraum (auch CIELAB-Farbraum genannt) ist ein 1976 von der internationalen Beleuchtungskommission (CIE) definierter Farbraum. Farben werden mit drei Werten beschrieben. „*L*“ (Lightness) definiert die Helligkeit. Die Werte liegen zwischen 0 und 100. „*a*“ gibt die Farbart und Farbintensität zwischen Grün und Rot an und „*b*“ gibt die Farbart und Farbintensität zwischen Blau und Gelb wieder. Die Werte für „*a*“ und „*b*“ liegen zwischen -128 und 127.

In der vorliegenden Arbeit wird der *Lab*-Farbraum verwendet, da es unkompliziert ist, den „*L*“ Kanal von beiden Farbkanälen „*a*“ und „*b*“ zu trennen. Außerdem bildet der *Lab*-Farbraum das menschliche Sehvermögen besser ab, als der RGB-Farbraum¹.

2.6 Image Colorization Methoden

Der Prozess von Image Colorization kann manuell oder automatisch erfolgen. Zu den manuellen Methoden zählen die analoge Einfärbung eines Bildes durch einen Menschen bis hin zur digitalen Bearbeitung mit einem dafür vorgesehenem Programm. Für die automatischen Methoden wird öfters menschlicher Input benötigt, um zum Beispiel Bereiche von einem Graustufenbild mit Farbstichen zu markieren, die dann automatisch von einem Algorithmus über das Bild propagiert werden. Aktuelle automatische Methoden nutzen die Vorteile von Convolutional Neural Networks, um diesen Prozess effizienter und performanter zu gestalten.

Um ein CNN, der Bilder automatisch einfärbt, zu trainieren, werden das Original Bild und das Graustufenbild benötigt. Das Graustufenbild wird in den CNN eingespeist und dieser wird versuchen die Farbkanal Werte vorherzusagen. Anschließend werden die Werte von jedem Pixel mit jedem Pixel aus dem Original Bild verglichen. Dieser Prozess wird iterativ wiederholt, bis die erzeugten Werten einen niedrigen Loss Wert haben. In den nächsten Kapiteln wird dieser Vorgehensweise näher erläutert.

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

2.7 Verwandte Arbeiten

Vor der Erstellung dieser Arbeit wurden zahlreiche automatische Methoden von Image Colorization bereits untersucht. Frühere Methoden waren stark an menschliches Input gebunden. Die 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 voll automatische Image Colorization Methoden gesetzt. Konservative Methoden, die Convolutional Neural Networks verwenden, versuchen die Farben von dem originalen Bild wiederherzustellen, indem die Loss Function die Distanz der vorhergesagten Farben zu den realen Farben berechnet. Diese Methoden liefern in der Regel entsättigte und blasses Bilder wie bei [Özb19]. Einer der Gründe für diese Ergebnisse ist, dass die Modelle nicht richtig lernen. So können beispielsweise Äpfel verschiedene Farben wie Rot oder Grün haben. Wenn das Netzwerk mit einem MSE Loss trainiert wird und der Datensatz die gleiche oder annähernd gleiche Anzahl an grünen und roten Äpfeln hat, 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 Farben einnehmen können. Die vorliegende Arbeit orientiert sich an den 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 ein CNN mit der Cross Entropy und Weighted Cross Entropy Loss. Der Output des Netzwerks ist eine Wahrscheinlichkeitsverteilung über die möglichen Bins für jeden Pixel, die im Kapitel 3.3 erläutert werden.

Beide Ansätze verwenden “Color Bins” die es unkompliziert ermöglichen die Farben von jedem Pixel zu klassifizieren. Es wird im nächsten Kapitel weiter auf diese Methode eingegangen.

Kapitel 3

Konzeption

Dieses Kapitel beschreibt alle notwendigen Schritte für die Konzeption der angewandten Methode, außerdem werden die Datensätze 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, das Vorhersagen, die weit von den originalen Farbwerten entfernt liegen, stärker bestraft, als Farbwerte die dichter an den originalen Farbwerten liegen. Das führt, wie bei 2.7 beschrieben zu entsättigten Bildern. Die Gründe für diese Ergebnisse lassen sich dadurch erklären dass verschiedene Objekte verschiedene Farben besitzen können.

Wenn die multimodalität des Problems berücksichtigt wird, werden Ergebnisse die nicht dem Original entsprechen, ebenfalls als valide Bilder erkannt. Das wird erreicht indem jedes Pixel klassifiziert wird, anstatt die Distanz zwischen dem Original und dem vorhergesagten zu berechnen.

Die gewählte Methode dieser Arbeit berücksichtigt die multimodalität von Image Colorization und behandelt das Problem als Klassifikation.

3.2 Farbraum

Der Standard Farbraum der Bilder für die Methode dieser Arbeit ist der RGB¹-Farbraum. Der RGB-Farbraum hat einige Nachteile, die es kompliziert machen mit diesem Farbraum zu arbeiten. Einer der Nachteile ist, dass das Graustufenbild sich schwer von den Farbkanälen trennen lässt. Ein anderer Nachteil ist, dass die Farbinformationen in drei Farbkanäle kodiert sind, was die Komplexität eines Modells erhöht. Aus diesem Grund werden die Bilder für das Preprocessing und die Methode in den Lab-Farbraum konvertiert. Für die Darstellung der Ergebnisse werden die Bilder von dem Lab-Farbraum in den RGB-Farbraum umgewandelt.

Im Lab-Farbraum können die Farbkanäle “ab” problemlos vom Belichtungskanal “L” getrennt werden. Der Belichtungskanal “L” enthält das Graustufenbild, das in den CNN eingespeist wird.



Abbildung 3.1: Original Bild in RGB oben links, Belichtungskanal “L” oben rechts, Farbkanal “a” unten links und Farbkanal “b” unten rechts.

¹Rot, Grün, Blau

3.3 Binning

Binning ist eine Technik, die für die Bildverarbeitung verwendet wird. Binning wird, im 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” Farbkanälen kodiert sind, werden nur diese 2 Farbkanäle in Bins klassifiziert. Auf der Abbildung 3.2 ist der Farbkanal “a” auf der x-Achse und der Farbkanal “b” auf der y-Achse abgebildet. Die Quadrate 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 sind Bedeutsam für die Berechnung der Bins.

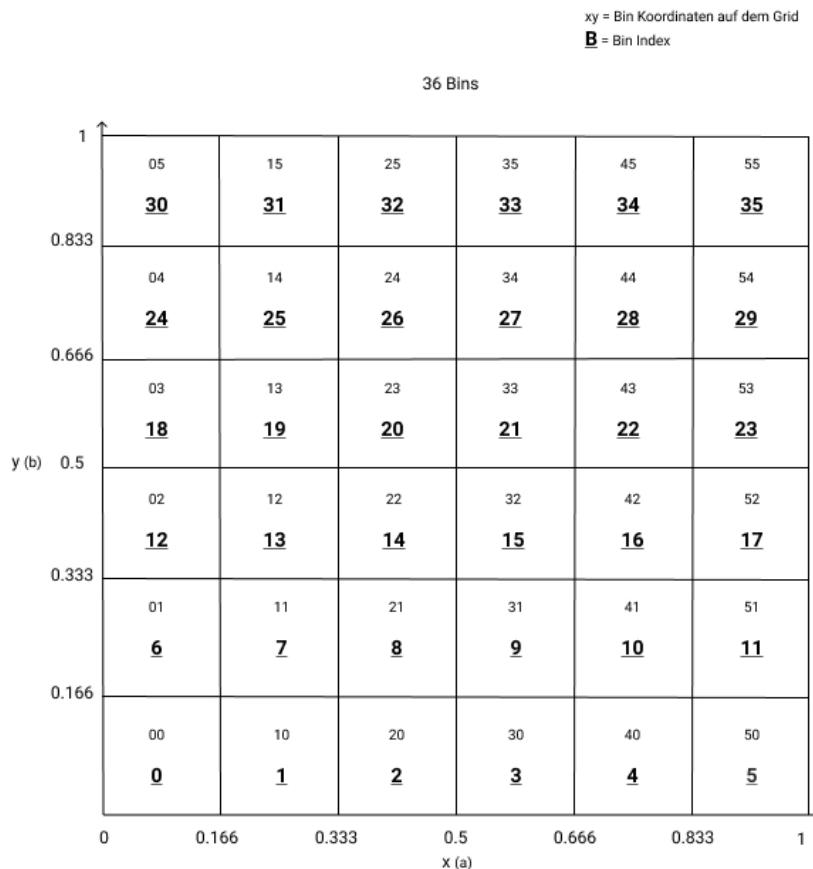


Abbildung 3.2: Grid mit 36 bins. Die x-Achse bildet die Werte des Farbkanals “a” und die y-Achse die Werte des Farbkanals “b” ab. (Eigene Darstellung)

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.

Die Umwandlung von Bin zu Farbe muss für die Ergebnisse des Netzwerks ebenfalls vorgegeben sein. Für dieses Problem werden vor dem Training die Farben von jedem Pixel aus den Trainingsbildern in Bins klassifiziert. Jeder Bin wird durch eine Liste mit je zwei Listen, für den jeweiligen Farbkanal, repräsentiert. Jeder Pixelwert wird zur entsprechenden Bin Liste hinzugefügt. Anschließend wird der Modus und der Durchschnitt jedes Farbkanals, der unter jedem Bin klassifizierten Farben, ausgerechnet. Am Ende beinhaltet jeder Bin eine Liste mit je zwei Werten, auf dem Nulltem Index der Wert für den Farbkanal “a” und auf dem ersten Index der Wert für den Farbkanal “b”. Es werden zwei separate Dateien erzeugt, eins für den Modus und eins für den Durchschnitt.

Für die Umwandlung von Bin zu Farbe werden die jeweiligen Werte für jeden Farbkanal eines Bins von der jeweiligen Datei abgeguckt. Es kann zwischen dem Durchschnitt und dem Modus ausgewählt werden. Der Durchschnitt verhält sich ähnlich wie ein Modell trainiert mit dem MSE Loss und der Modus würde Ergebnisse mit einem rötlichen Farbton liefern. Als Lösung für dieses Problem schlägt Zhang et al. den “annealed mean” vor [ZIE16]. Der “annealed mean” versucht einen Kompromiss zwischen dem Durchschnitt und dem Modus zu finden. Dieser Kompromiss wird durch einen Temperatur Parameter (T) reguliert. In der vorliegenden Arbeit wird eine von den “annealed mean” inspirierten Methoden implementiert. Die Methode wird wie folgt implementiert:

$$\begin{aligned} D &= K_{mode} - K_{mean} \\ \hat{y}_K &= K_{mode} - (D * T) \end{aligned} \tag{3.1}$$

wobei K ein Farbkanal ist, D die Distanz zwischen Durchschnitt und Modus, T ein Temperaturwert zwischen 0 und 1 und \hat{y}_k die endgültige Farbe für den Farbkanal K ist. Ein Temperaturwert von 1 würde den Durchschnitt ergeben, wohingegen ein Temperaturwert von 0, den Modus nicht verändert.

3.4 Netzwerkarchitektur

Die Netzwerkarchitektur ist ein wichtiger Faktor der u.a. die Ergebnisse beeinflusst. Um die Methoden zu vergleichen ist es wichtig ein leichtes Convolutional Neural Network,

das wenige Parameter besitzt, schnell zu trainieren ist und gute Ergebnisse liefert, zu verwenden.

Das Ziel von dem Netzwerk ist es, ein Graustufenbild als Input zu bekommen und eine Wahrscheinlichkeitsverteilung für alle Bins per Pixel vorherzusagen. Das Output Volumen hat die Dimensionen $W_{Input} \times H_{Input} \times n_bins$, wobei W und H die Breite und Höhe des Bildes sind und n_bins eine Wahrscheinlichkeitsverteilung für alle Bins pro Pixel ist. Dieser Ansatz wird auch bei Image Segmentation Problemen genutzt, wobei ein Bild in das Netzwerk eingespeist wird und eine Segmentation map, mit einer Klasse per Pixel, als Output erzeugt wird. In der Regel wird jeder Klasse eine bestimmte Farbe zugeordnet, was die Objekte klassifiziert und trennt. In dem Fall von Image Colorization bekommt jedes Pixel in dem Output Volumen eine Wahrscheinlichkeitsverteilung für alle Bins, die in eine Farbe umgewandelt wird.

Die Methoden von Zhang et al. [ZIE16] verwenden ein CNN, das ein Graustufenbild als Input entgegennimmt und ein Volumen mit einer Wahrscheinlichkeitsverteilung für alle Bins per Pixel generiert. Diese Netzwerkarchitektur besteht aus Blöcken mit jeweils 2 oder 3 Convolutional und ReLU Layers, gefolgt von einer Batch Normalization Layer. Batch Normalization ist eine Regularisierungstechnik, die die Werte in einer Hidden Layer normalisiert, bevor sie in die nächste Layer weitergereicht werden. Das Netzwerk hat keine Pooling Layers, alle Änderungen in der Auflösung werden durch Downsampling oder Upsampling zwischen Blöcken erreicht.

Diese Netzwerkarchitektur ist sehr herausfordernd für die GPU², was die Batch Größe und Trainingszeit stark beeinflusst.

Aus diesem Grund orientiert sich die Netzwerkarchitektur dieser Arbeit an 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

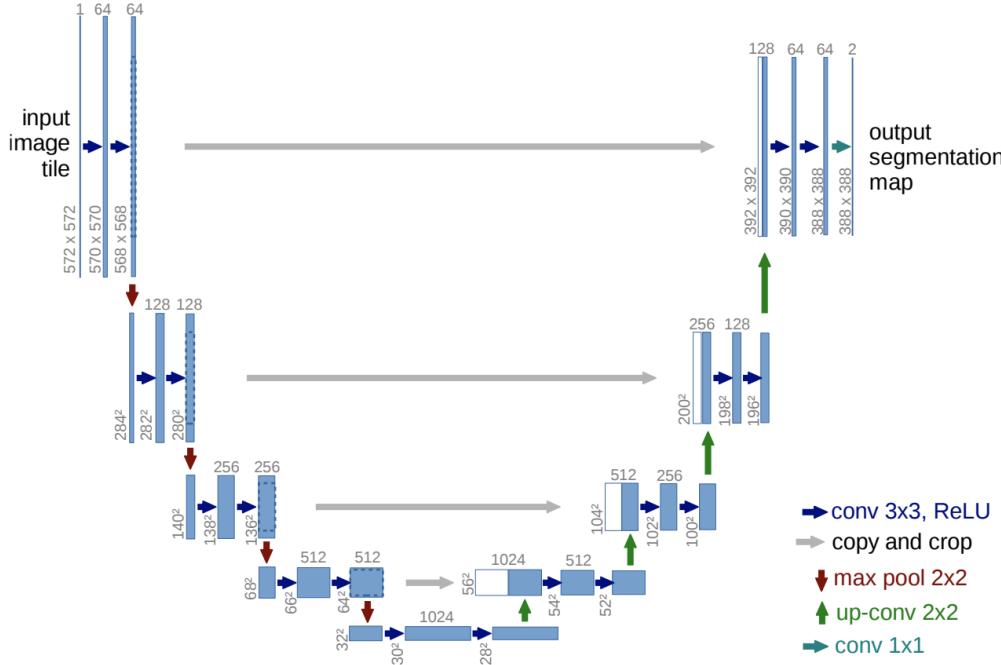
Ein U-net ist ein Autoencoder mit Skip Connections und Transposed Convolutions als Upsampling Methode das bei Image Segmentation angewendet wird. Im Vergleich zu konventionellen Autoencoder können der Encoder und Decoder nicht getrennt voneinander

²Graphics Processing Unit

verwendet werden. Die Skip Connections ermöglichen fein-granuläre Details in dem Output Volumen wiederherzustellen und helfen mit dem Vanishing Gradient Problem während Backpropagation. Skip Connections konkatenieren bestimmte Layers aus dem Encoder mit Layers aus dem Decoder, mit den gleichen Dimensionen.

3.4.2 Aufbau eines U-nets

Ein U-net besteht, wie ein Autoencoder, aus einem Encoder und Decoder Teil. Der Encoder besteht aus sogenannten “ConvBlocks”. ConvBlocks bestehen aus 2 Convolutional Layers gefolgt von Batch Normalization und ReLU. Den ConvBlocks folgt eine Pooling Layer, die die Dimensionen des Volumens verringert. Der Decoder besteht aus ConvBlocks gefolgt von Transposed Convolutions, die die Dimensionen von dem Volumen wieder vergrößern. Die letzte Layer ist ein 1×1 Convolutional Layer, die das Output Volumen generiert. Skip Connections konkatenieren ConvBlocks aus dem Encoder mit Transposed Convolutions aus dem Decoder mit der gleichen Dimensionen.



kann unter folgendem Link³ gefunden werden.

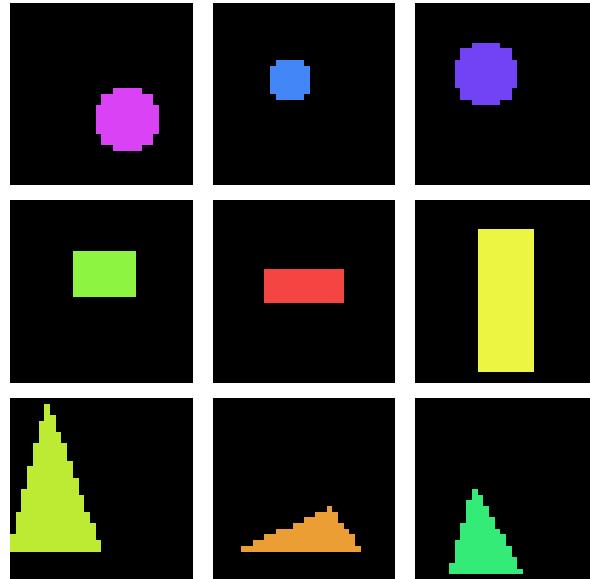


Abbildung 3.4: Beispiel von Trainingsbildern mit einer der möglichen Farben pro Klasse

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

Als zweites werden 12 Klassen des CIFAR-100⁴ Datensatzes verwendet. CIFAR-100 ist ein öffentlich verfügbarer Datensatz, der von Krizhevsky et al. erstellt wurde. Der Datensatz setzt sich aus 100 Klassen mit jeweils 600 Bildern pro Klasse zusammen, außerdem sind die 100 Klassen in 20 Superklassen gruppiert. Der Datensatz wird in 50000 Bildern für das Training und 10000 Bildern für die Validierung aufgeteilt. Einige Beispiele für Klassen sind *apples*, *palm* oder *bee*. Die Bilder haben ebenfalls eine Auflösung von 32×32 .

Da es äußerst aufwändig wäre ein Modell, dass jedes Objekt aus dem CIFAR-100 Datensatz richtig erkennt und einfärbt, von Null auf zu trainieren, werden nur bestimmte Klassen für das Training verwendet. Diese Klassen sind: *apples*, *sunflower*, *rose*, *cloud*, *maple_tree*, *oak_tree*, *pine_tree*, *willow_tree*, *palm_tree*, *mountain*, *forest* und *sea*. Alle Klassen haben Gemeinsamkeiten, was das Erlernen von Merkmalen erleichtert im Gegensatz zu einem

³<https://drive.google.com/file/d/197egEFSMLVq4cspScwdhp1sIk51YWT1U/view?usp=sharing>

⁴CIFAR-100: <https://www.cs.toronto.edu/~kriz/cifar.html>

sehr allgemeinen Datensatz.

Abschließend wird ein komplexerer und hochauflösender Datensatz verwendet, der Naturbezogene Bilder enthält. Ziel ist es, ein Netzwerk zu trainieren, dass Bilder aus der Natur einfärben kann. Dieses besteht aus 3 Datensätzen von Kaggle⁵ und GitHub⁶. Der erste ist der “Landscape Pictures”⁷ Datensatz von Arnaud Rougetet, von diesem Datensatz werden alle Bilder verwendet. Der zweite Datensatz ist “Landscape Classification”⁸ von Huseyb Guliyev, hiervon werden nur die Klassen *forest*, *glacier*, *mountain* und *sea* verwendet. Der letzte Datensatz ist der “Landscapes dataset”⁹ von ml5js auf GitHub. Von diesem Datensatz wurden die Klassen *field*, *forest*, *lake*, *mountain* und *road* verwendet. Des Weiteren werden alle Bilder entfernt, die keine öffentliche Lizenz haben.

Der komplette Datensatz besteht aus 8 Klassen und hat insgesamt 12479 Bilder, 10120 für das Training und 2359 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. Der finale Datensatz wird für den Rest der Arbeit “Landscape Datensatz” genannt und kann unter folgendem Link¹⁰ gefunden werden.

3.6 Image Preprocessing und Augmentation

Für das optimale Training und die besten Ergebnisse werden die Bilder vorverarbeitet. Außerdem werden Techniken von Image Augmentation angewendet. Der Spiel-Datensatz und die 12 Klassen von CIFAR-100 werden für das Training mit einer Wahrscheinlichkeit von 50% horizontal gespiegelt.

Da der Landscape Datensatz Bilder mit verschiedenen Auflösungen beinhaltet, werden alle Bilder auf 128×128 angepasst. Das reduziert die Trainingszeit und die Komplexität des Datensatzes. Für das Training werden pro Bild 4 zusätzliche augmentierte Bilder generiert. Zunächst werden die Bilder zufälligerweise um ± 30 Grad rotiert, anschließend wird die Größe der Bilder auf einen Wert zwischen 0 und 30% geändert. Nachdem dieser

⁵Kaggle: <https://www.kaggle.com/>

⁶Github: <https://github.com/>

⁷<https://www.kaggle.com/arnaud58/landscape-pictures>

⁸<https://www.kaggle.com/huseynguliyev/landscape-classification>

⁹<https://github.com/ml5js/ml5-data-and-models/tree/master/datasets/images/landscapes>

¹⁰<https://drive.google.com/file/d/1o73BaE-CYnmtEQgwN427znk1yizrwEsy/view?usp=sharing>

Schritt abgeschlossen ist, werden die Bilder horizontal gespiegelt und abschließend nochmal Vertikal gespiegelt. Nach der Image Augmentation besteht der Trainings Datensatz aus 50600 Bildern.

3.7 Tools

Um die Methode zu realisieren werden einige Tools genutzt. Für die Implementierung wird das Framework PyTorch¹¹ verwendet. PyTorch ist ein Open-Source Framework basierend auf Python für Machine Learning und Deep Learning. Es wurde vom Facebook 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 die “scikit-image” Bibliothek eingesetzt und für die Image Augmentation wurde die Bibliothek “imgaug” ausgewählt. Des Weiteren werden Hilfsbibliotheken wie “numpy” und “matplotlib” angewendet.

Das Trainieren von den Modellen wird, aufgrund des hohen Rechenaufwands, auf zwei verschiedenen Plattformen durchgeführt. Die Modelle mit den 32×32 Bildern werden auf Google Colab¹² trainiert. Google Colab ist eine Plattform von Google, die es ermöglicht Experimente im Browser mit einer Hochleistungsgrafikkarte (Nvidia Tesla P100) kostenlos umzusetzen. Für das Modell mit den 128×128 Bildern wird das Curious Containers (CC) Framework¹³ benutzt. Curious Containers ermöglicht eine gleichzeitige Durchführung von verschiedenen Experimenten in einem Cluster von Hochleistungsrechnern.

¹¹<https://pytorch.org/>

¹²<https://colab.research.google.com/>

¹³<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. Die Implementierung der Methode kann im Anhang oder auf dem folgenden GitHub Repository¹ gefunden werden.

4.1 Binning

Das Binning wurde mit Hilfe der “numpy” Bibliothek für normalisierte Lab-Bilder implementiert. Als erstes wird mit Hilfe der Wurzel anhand der Anzahl von Bins (n_bins) die Breite (W) und Höhe (H) des Grids, das auf der Abbildung 3.2 zu sehen ist, berechnet. Es wird angenommen dass $W = H$ ist. Nachdem die Breite des Grids berechnet wurde, wird der Intervall von $[0, 1]$ in W 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.

¹<https://github.com/SaizFerri/colorization-pytorch-unet>

```

1      # a, b sind die Koordinaten der Farbkanäle auf dem Grid
2      def calculate_bin(a, b, width):
3          return (width * b) + a
4
5      def encode_bins(ab_image, n_bins):
6          W = np.sqrt(n_bins).astype(int)
7
8          # Intervall in gleich große Intervalle aufteilen
9          interval = np.linspace(0, 1, W+1)
10
11         # Indices für jeweils a, b Kanäle berechnen
12         indices = np.digitize(ab_image, interval) - 1
13
14         # Bin Index berechnen
15         bins = np.vectorize(calculate_bin)(indices[:, :, 0], indices[:, :, 1], W)
16
17     return bins

```

Code snippet 4.1: Binning eines normalisierten Lab Bildes

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

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

Code snippet 4.2: Leere Dictionary Erzeugung für n_bins

Anschließend werden die Farben von jedem Farbkanal zu dem jeweiligen Bin Array zugeordnet. Als letztes wird der Durchschnitt pro Bin Index und Farbkanal berechnet. Das Dictionary beinhaltet abschließend ein Array mit 2 einzelnen Werten 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 einen Kompromiss zwischen Durchschnitt und Modus zu finden, werden der 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 ein “ImageFolder” implementiert, der die Bilder importiert, transformiert, normalisiert und die einzelnen Pixel in Bins klassifiziert. Der Output der “ImageFolder” sind das Graustufenbild, das Bild mit den “ab” Farbkanälen und das in Bins kodierte Bild.

Mit Hilfe eines “DataLoaders” werden die Datensätze in Batches aufgeteilt und zufällig gemischt.

4.3 Netzwerkarchitektur

Für diese Arbeit wurden 2 U-Nets mit verschiedenen Größen verwendet. Ein U-Net für 32×32 Input Bilder und ein U-Net für 128×128 Input Bilder. Außerdem wurde das U-Net für 32×32 Bilder angepasst, damit es mit einem MSE Loss verwendet werden kann. Bei der Anpassung wurde das 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 das Binning nicht angewendet.

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 Normalization bestehen. Die Convolutional Layers verwenden Padding, um die Dimensionen des Inputs nicht zu verändern. Die Layers wurden mit den PyTorch Klassen *Conv2d*, *BatchNorm2d* und der Funktion *relu* implementiert. Der ConvBlock wird mit einer Klasse implementiert, die wiederum von der Oberklasse *torch.nn.Module* erbt.

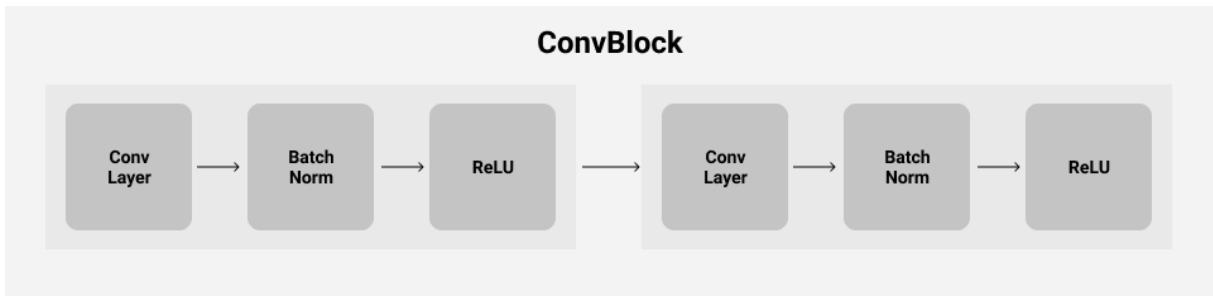


Abbildung 4.1: ConvBlock (Eigene Darstellung)

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. Die letzte Layer ist eine Convolutional Layer mit 1×1 Filter, Stride 1 und Padding 0, die 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.

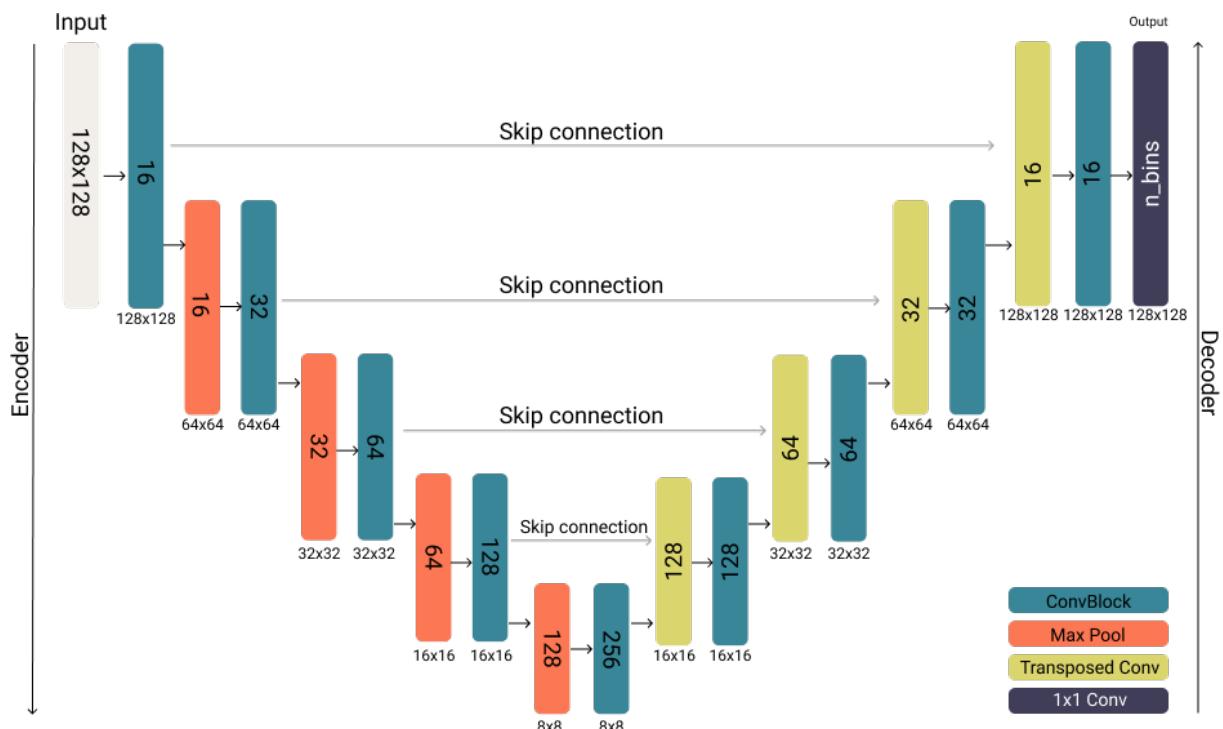


Abbildung 4.2: U-Net Architektur für 128×128 Input Bilder (Eigene Darstellung)

Kapitel 5

Test

In diesem Kapitel werden Tests mit den drei Datensätzen durchgeführt. Darüber hinaus werden verschiedene Hyperparameter getestet.

5.1 Spiel-Datensatz Training

Als erstes wird die Methode auf dem Spiel-Datensatz angewendet. Hierbei soll geprüft werden, ob die Methode die erwarteten Ergebnisse liefert. Das U-Net wird für 100 Epochen mit Adam, einer Lernrate von 0.001 und der Cross Entropy Loss Function trainiert. Diese Einstellung zeigte die besten Ergebnisse. Zeitgleich wurden Tests mit 36 und mit 324 Bins durchgeführt, die jedoch keinen Unterschied bei den Ergebnissen vorwiesen. Es kann davon ausgegangen werden, dass bei der niedrigen Anzahl an möglichen Farben, ein Unterschied zwischen 36 und 324 Bins nicht zu erkennen ist. Die unteren Ergebnisse wurden mit 324 Bins erstellt.

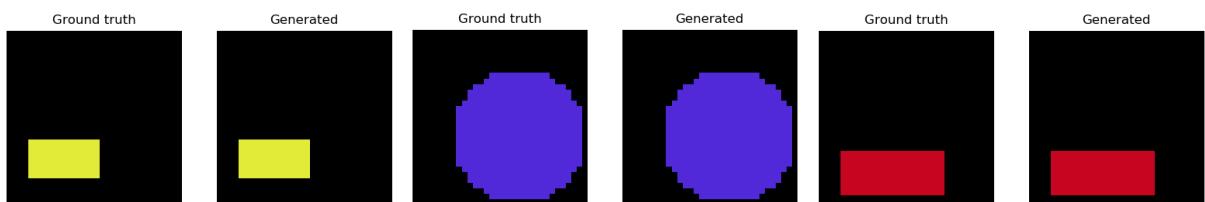


Abbildung 5.1: Beispiele von sehr guten Ergebnissen aus dem Spiel-Datensatz

Bei den oberen Ergebnissen wurden alle Pixel richtig klassifiziert, was bei der Größe des Datensatzes oft auf Overfitting hindeutet. Die folgenden Ergebnissen zeigen dass das Modell generalisiert und nicht overfitted hat.

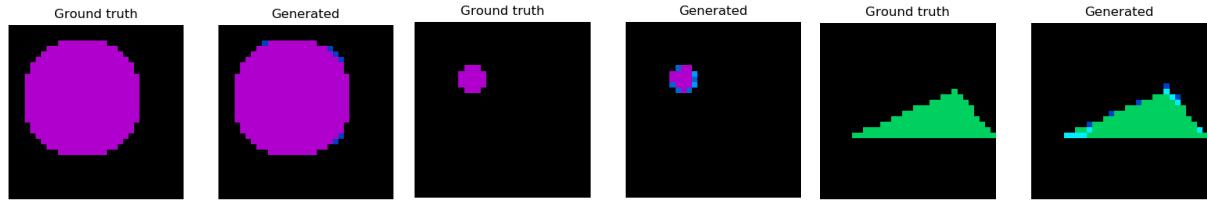


Abbildung 5.2: Beispiele von generalisierten Ergebnissen

Das Modell zeigte bei einigen Ergebnissen, Schwierigkeiten die Pixeln am Rand der geometrischen Formen richtig zu klassifizieren. Dies trifft besonders auf die Kreise und Dreiecke zu, wo die Ränder nicht aus glatten Linien bestehen. Des Weiteren wurden die Farben mittels des Durchschnitts für jeden möglichen Bin, für alle Farben von jedem Trainingsbild rekonstruiert. Ein Unterschied zwischen dem Modus und dem Durchschnitt konnte nicht erkannt werden, da beide Werte gleich waren.

Die Ergebnisse bestätigen, dass das Binning und die Methode funktionieren. Anschließend wurden Tests mit komplexeren Bildern des Subsets von CIFAR-100 durchgeführt.

5.2 CIFAR-100 Subset Training

Das Modell wurde auf 12 Klassen von CIFAR-100 über 100 Epochen mit Adam, einer Lernrate von 0.001 und der Cross Entropy Loss Function trainiert. Diese Einstellungen stellten sich als die beste Kombination von Hyperparametern heraus, jedoch wurde das Training vor Vollendung von 50 Epochen unterbrochen, um Overfitting zu verhindern.

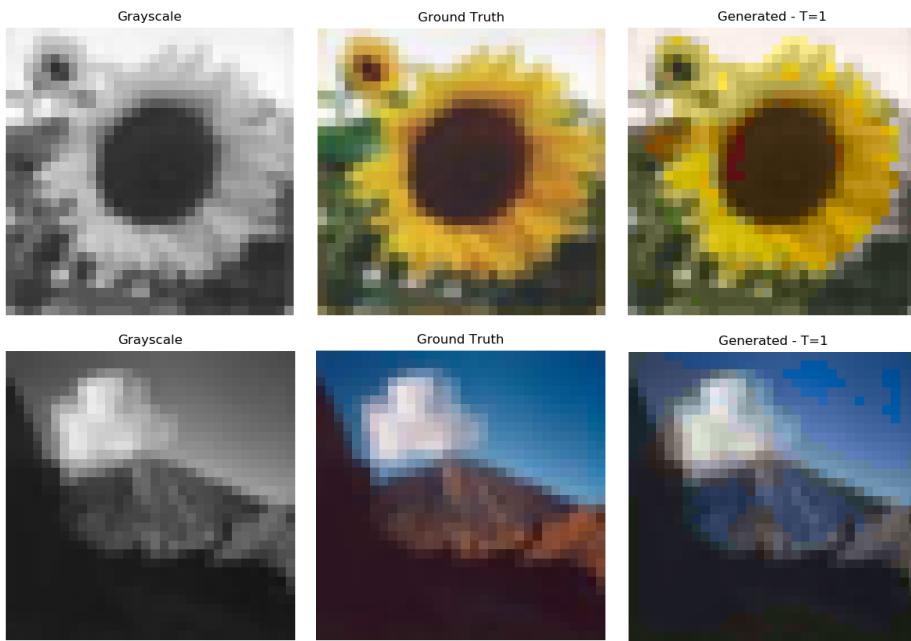


Abbildung 5.3: Beispiele von guten Ergebnissen aus dem Subset von CIFAR-100 mit 324 Bins. Die erste Spalte beinhaltet das Graustufenbild, die zweite Spalte beinhaltet das Original Bild und die letzte Spalte stellt das generierte Bild dar. Das generierte Bild wurde mit einer Temperatur von 1 erzeugt, was bedeutet, dass die rekonstruierten Farben den Durchschnitt aus jedem Bin repräsentieren.

Die Tests mit diesem Datensatz haben gezeigt, dass die Anzahl der Bins bei der Auswahl an möglichen Farben, die Ergebnisse beeinträchtigen. Eine Erhöhung der Trainingszeit zwischen 36 und 324 Bins war nicht zu erkennen.

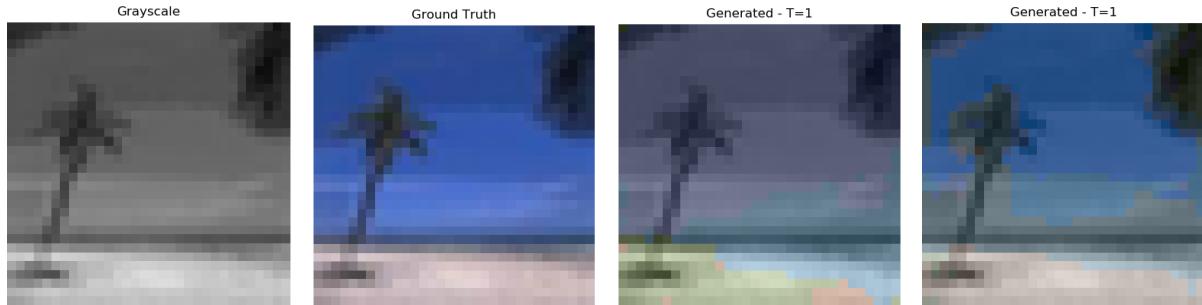


Abbildung 5.4: Einfluss der Anzahl der Bins auf die Ergebnisse. Das zweite Bild ist das original, das dritte Bild wurde mit 36 Bins und einem Temperaturwert von 1 generiert und das vierte mit 324 Bins und ebenfalls mit einem Temperaturwert von 1.

Um das Overfitting zu verhindern wurden zahlreiche Experimente mit verschiedenen Optimierern, Aktivierungsfunktionen und Lernraten durchgeführt. Ein Austausch von ReLU durch Tanh zeigte ein stabileres Trainingsverhalten, aber eine Verschlechterung der Validation Loss. Leaky ReLU zeigte ein ähnliches Verhalten wie ReLU, aber keine Verbesserung der Ergebnisse. Die Verwendung von RMSprop anstelle von Adam zeigte eine langsamere Konvergenz Richtung Minimum. Abschließend wurde die Anzahl der Filter in den Convolutional Layers halbiert, was eine positive Wirkung auf das Training hatte.

Um die Performance der Klassifikation gegenüber der Regression zu messen, wurde ein Modell mit der MSE Loss Function trainiert. Dieses Modell wurde ebenfalls mit den gleichen Parametern wie das Klassifikationsmodell trainiert und hat vergleichbare Ergebnisse erreicht. Einige Ergebnisse zeigten blasse Stellen im Vergleich zu dem Klassifikationsmodell, das leuchtende Farben an den gleichen Stellen gezeigt hat.

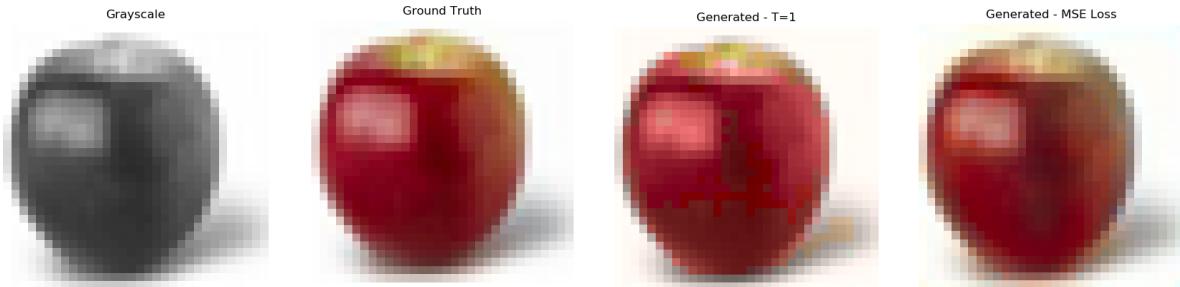


Abbildung 5.5: Vergleich von Klassifikation mit Binning gegenüber Regression. Das zweite Bild wurde mit 324 Bins und dem Cross Entropy Loss generiert. Das dritte Bild wurde ohne Binning und mit einem MSE Loss generiert.

5.3 Landscape Datensatz Training

Dieser Datensatz wurde anhand der Hyperparameter Optimierung des CIFAR-100 Subsets trainiert. Es wurde das größere Modell für 128×128 Input Bildern angewendet. Das Modell wurde ebenfalls für 36 und 324 Bins, mit dem Adam Optimizer, eine Lernrate von 0.001 und der Cross Entropy Loss Function für 60 Epochen trainiert. Das Modell mit dem besten Validation Loss wurde gespeichert, um die Ergebnisse zu evaluieren. Außerdem wurde der Einfluss der Temperaturwerte und die Anzahl der Bins auf die Ergebnisse gemessen.

Das Modell tendierte bei diesem Datensatz ebenfalls zum Overfitting. Um das zu verhindern wurden die gleichen Techniken wie bei dem Subset von CIFAR-100 angewendet, was keine besseren Ergebnisse liefert hat. Eine Halbierung der Anzahl der Filter bei den Convolutional Layers führte zu einer Verschlechterung des Validation Loss und half nicht bei Overfitting. Eine Änderung des Optimierers zu RMSprop und der Ersatz von ReLU durch Tanh zeigten ein stabileres Training, aber keine Verbesserung des Validation Loss. Das endgültige Modell wurde trainiert, bis der Validation Loss wieder stieg.

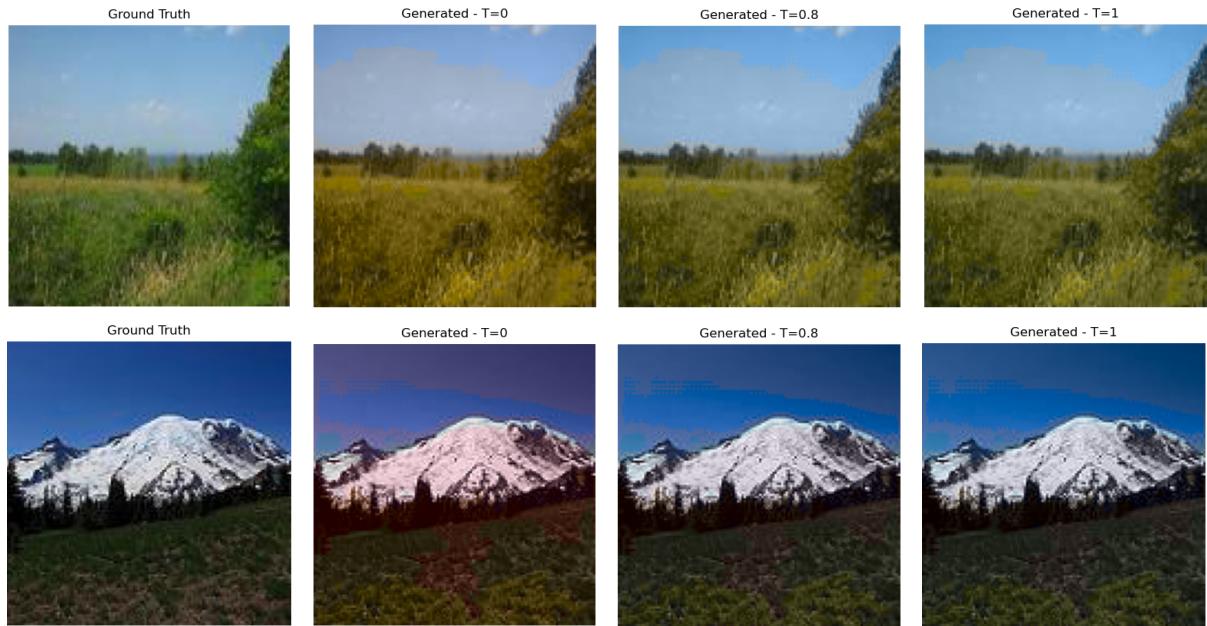


Abbildung 5.6: Ergebnisse mit 324 Bins. Die erste Spalte zeigt das Originale Bild, die zweite zeigt das generierte Bild mit einer Temperatur von 0, die dritte Spalte zeigt das generierte Bild mit einer Temperatur von 0.8 und die vierte Spalte zeigt das generierte Bild mit einer Temperatur von 1

Nach der Auswertung der Experimente wurden 324 Bins und ein Temperaturwert von 0.8 bis 1 bevorzugt, um die bestmöglichen Ergebnisse zu bekommen.

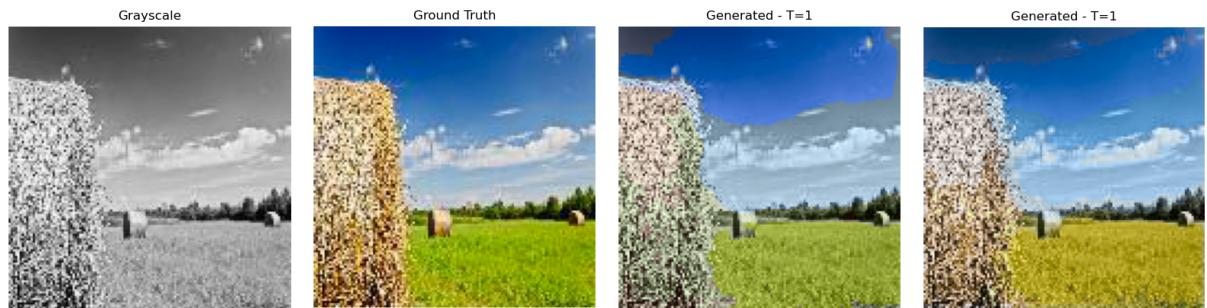


Abbildung 5.7: Ergebnisse mit 36 und 324 Bins. Die erste Spalte zeigt das Graustufenbild, die zweite zeigt das Originale Bild, die dritte Spalte zeigt das mit 36 Bins generierte Bild mit einer Temperatur von 1 und die vierte Spalte zeigt das mit 324 Bins generierte Bild mit einer Temperatur von 1

Kapitel 6

Evaluation

Im Kapitel Evaluation werden die im Kapitel 5 vorgestellten Tests evaluiert. Außerdem wird die gewählte Methode ausgewertet und mit anderen Methoden verglichen.

6.1 Evaluationsmetrik

Für das Problem von Image Colorization existiert keine relevante Evaluationsmetrik, die die Farben von den Objekten auf einem Bild auswerten kann. Das während des Trainings angewendete Cross Entropy Loss ist nicht relevant für die Auswertung der Ergebnisse aus dem Test Datensatz. Aus diesem Grund wurde die Evaluation der Ergebnisse durch eine Menschliche Auswertung wie bei Zhang et al. und Billaut et al. durchgeführt.

Eine andere Evaluationsmetrik wäre ein vor-trainiertes Image Segmentation Modell. Wenn das Modell die Merkmale auf den generierten Bildern trotzdem richtig klassifiziert, wäre davon auszugehen dass die Bilder richtig eingefärbt sind. Diese Methode wurde nicht angewendet.

6.2 Evaluation des Spiel-Datensatzes

Mit dem Spiel-Datensatz wurde die Funktionsweise der Methode bestätigt. Die von Billaut et al. vorgeschlagene Netzwerkarchitektur für Image colorization hat beeindruckende Ergebnisse nach wenigen Epochen erreicht.

Die gewählte Methode für das Binning funktionierte und hat ermöglicht, die originalen

Farben wiederherzustellen. Die Anzahl an Bins war für diesen Datensatz nicht relevant da es nur 9 mögliche Farben gab.

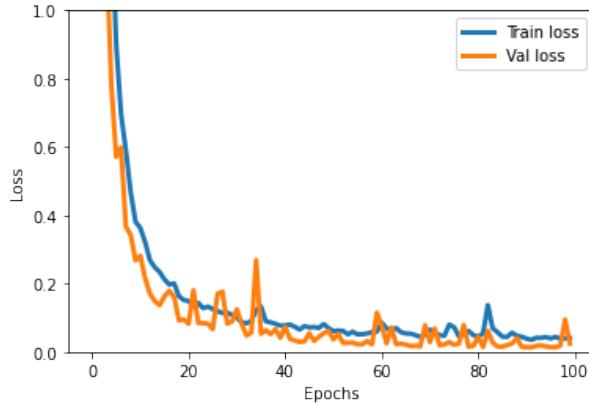


Abbildung 6.1: Training und Validation Loss Verlauf des Spiel-Datensatzes

6.3 Evaluation des CIFAR-100 Subsets

Die Ergebnissen aus dem CIFAR-100 Subset zeigten, dass das Modell nach wenigen Epochen viele Merkmale lernen konnte. Ein wichtiger Faktor, der erwähnt werden muss, ist dass die Gewichte des Netzwerks zufällig initialisiert wurden und nicht vortrainiert waren.

In diesem Datensatz wurde die Auswirkung der Bin Anzahl gemessen. Die Nutzung von 36 Bins zeigte im Vergleich zu 324 eine Verschlechterung der Farben in der Vorhersage. Dies ist darauf zurückzuführen, dass das Modell nur 36 mögliche Farben zu Verfügung hat. Eine Erhöhung der Bins auf 324, ermöglichte es dem Modell eine Auswahl an mehreren Farben zu treffen. Der Ansatz von Billaut et al. verwendet nur 32 Bins und erzielt ähnliche Ergebnisse wie die Methode dieser Arbeit mit 324 Bins. Dies wurde erreicht in dem die Pixel von jedem Trainingsbild vor dem Training in Bins klassifiziert wurden und daraus nur die am meisten vorkommenden 32 Bins ausgewählt wurden. Pixel die nicht in den gewählten 32 Bins klassifiziert werden konnten, wurden in das nächstliegende Bin zugeordnet [BRT18].

Die Methode mit einem MSE Loss liefert eine um ein vielfaches größere Auswahl an Farben

für die Vorhersage. Bei dieser Methode treten die im Kapitel 2.7 erwähnten Schwierigkeiten auf, wobei im Falle dieses Datensatzes, die Schwierigkeiten nur sehr latent ausgeprägt waren. Da die Klassifikationsmethode bessere Ergebnissen liefert hat und der Fokus der Arbeit auf Klassifikationsmethoden gesetzt war, wurden alle Experimente des Landscape Datensatzes mit der Klassifikationsmethode durchgeführt.

Der Verlauf von dem Training und Validation Loss deutete bei diesem Datensatz auf Overfitting, was bei der Größe des Datensatzes nicht auszuschließen war. Wie bei 5.2 beschrieben wurden das U-net und die Hyperparameter angepasst, um dieses zu verhindern.

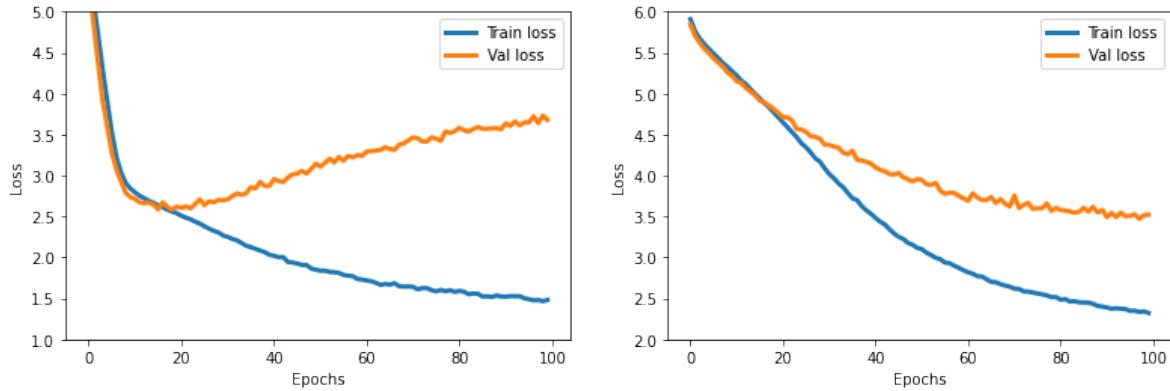


Abbildung 6.2: Overfitting auf dem CIFAR-100 Subset. **Links:** 100 Epochen mit Adam, ReLU und einer Lernrate von 0.001. **Rechts:** 100 Epochen mit Adam, ReLU und einer Lernrate von 0.0001.

Eine Anpassung der Lernrate führte nur zu einem langsameren Training. Eine Reduktion der lernbaren Parameter von 135684 auf 35748 zeigte eine deutliche Verbesserung der Performance des Modells und reduzierte das Overfitting.

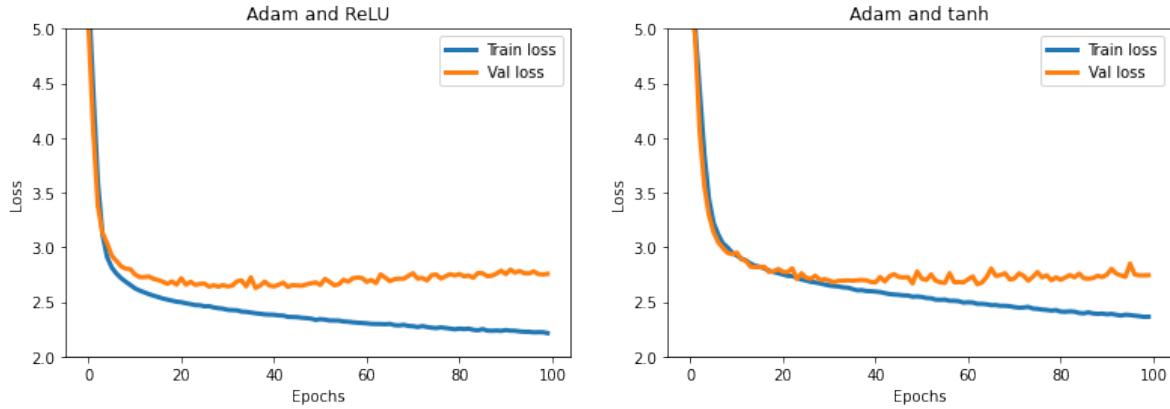


Abbildung 6.3: Loss Verlauf mit einer reduzierten Anzahl an Parametern und verschiedenen Aktivierungsfunktionen. **Links:** 100 Epochen mit Adam, ReLU und einer Lernrate von 0.001. **Rechts:** 100 Epochen mit Adam, Tanh und einer Lernrate von 0.001.

Der Grund für das Overfitting ist in diesem Fall auf die Größe des Subsets zurückzuführen. Um das zu prüfen wurde das Modell auf dem kompletten CIFAR-100 Datensatz für 150 Epochen trainiert, was einen guten Loss Verlauf zeigte. Andererseits, hat das Modell nichts relevantes gelernt und konnte kein Bild richtig einfärben, was bei der Anzahl der Klassen und der Anzahl der Epochen nichts Ungewöhnliches ist.

6.4 Evaluation des Landscape Datensatzes

Die Ergebnisse dieses Modells zeigen, dass die ausgewählte Methode mit wenigen Bildexemplaren und Epochen, sehr gute Ergebnisse erreichen kann. Die Anzahl der Bins für diesen Datensatz wurde durch die Ergebnisse der vorgeführten Experimenten auf 324 gesetzt, da diese Anzahl die beste Kombination aus Trainingszeit und Performance gezeigt hat.

Der Loss Verlauf der Experimente deutete bei diesem Datensatz ebenfalls nach wenige Epochen auf Overfitting hin. Nach zahlreichen Experimenten konnte das Overfitting durch Anpassung der Hyperparameter nur minimiert werden. Für die Auswertung der Ergebnisse wurde das Modell mit dem besten Validation Loss verwendet.

Bei der Evaluation der Ergebnisse ist zu erkennen, dass schon nach den wenigen Epochen das Modell in der Lage ist die wichtigsten Entitäten wie den Himmel, Wolken, Wasser, Grass und Erde richtig einzufärben. Es fällt auf, dass bei einigen Bildern die Vorhersage des Modells realer aussieht, als das Original Bild, wie im zweiten Beispiel von 5.6 zu sehen ist. Im Vergleich zum Modell von Billaut et al. sehen die vorhergesagten Farben von diesem Modell sehr ähnlich aus. Das zeigt, dass die Performance der Methode ohne die Optimierungstechniken vergleichbar ist. Die Auswirkung des Temperaturwerts ist bei beiden Modellen ähnlich. Der Modus der Verteilung zeigt einen rötlichen Ton bei den Ergebnissen und der Durchschnitt zeigt bei einigen Fällen gesättigte Bilder.

Da die Datensatz Größe von Billaut et al. fast identisch zu der Datensatz Größe dieser Arbeit ist, könnte das Problem mit dem Overfitting auf die Qualität des Datensatzes zurückzuführen sein. Der Loss Verlauf zeigt kein Overfitting und das verwendete U-Net ist ähnlich zum U-Net der vorliegenden Arbeit, was ein Overfitting wegen der angewendeten Netzwerkarchitektur ausschließt.

Ein Vergleich mit den Ergebnissen von Zhang et al. wäre nur angemessen mit den Ergebnissen aus deren Klassifikationsmodell ohne Rebalancing. In diesem Fall sind die Resultate dieses Modells nicht weit entfernt von deren Ergebnissen. Es ist zu erkennen, dass deren Ergebnisse ähnliche Merkmale mit den Ergebnissen dieser Arbeit vorweisen, wie z.B. die Ähnlichkeit der Vorhersagen zwischen Klassifikationsmethode und Regressionsmethode. Die Ergebnisse mit Class Rebalancing unterscheiden sich deutlicher von den Ergebnissen der Regressionsmethode. Da deren Datensatz über 1.5 Millionen Bilder beinhaltet, kann das Modell mehr Objekte auf den Bildern einfärben, was sich bei einigen Bildern z.B. mit Menschen, bemerkbar macht.

Im allgemeinen sind die Ergebnisse dieses Modells mit anderen Ergebnissen von Klassifikationsmethoden ohne Optimierungstechniken wie Class Rebalancing vergleichbar.

Kapitel 7

Fazit

7.1 Zusammenfassung

Das Ziel dieser Bachelorarbeit war es, die Methoden von Image Colorization anhand Convolutional Neural Networks zu untersuchen, insbesondere Klassifikationsmethoden. Durch die ausführliche Auseinandersetzung mit Klassifikationsmethoden und die Implementierung der verwendeten Techniken wie Binning, konnte eine Methode, basierend auf dem letzten Stand der Technik, implementiert werden. Diese Methode wurde angewandt und mit verschiedenen Datensätzen getestet, um die Ergebnisse zu untersuchen.

Die Ergebnisse dieser Arbeit haben gezeigt, dass die Methoden funktionieren und sogar mit weniger Bildern und Epochen gute Ergebnisse produzieren. Zahlreiche Experimente bestätigten, dass Klassifikationsmethoden bessere Ergebnisse als Regressionsmethoden liefern. Diese Differenz ist deutlicher bei Methoden die Optimierungstechniken, wie Class Rebalancing, anwenden.

7.2 Kritischer Rückblick

Der Fokus dieser Arbeit wurde auf die Untersuchung der Methoden gelegt. Bei der Erstellung, Implementierung und Durchführung der Methode wurde festgestellt, dass das Problem von Image Colorization sehr komplex ist. Obwohl die erreichten Ergebnisse ausreichend für den Vergleich mit anderen Methoden sind, wurde festgestellt, dass ohne Optimierungstechniken eine Verbesserung der Qualität der Bilder nur schwer zu erreichen ist.

Die Auswahl eines balancierten Datensatzes ist ein wichtiger Faktor für die Qualität der Ergebnisse und kann Overfitting verhindern. Die Größe des Datensatzes ist auch entscheidend bei dem Training eines nicht vor-trainierten Modells.

Die Zielsetzung wurde erreicht in dem die Klassifikationsmethode bestätigt wurde, obwohl die Qualität der Ergebnisse durch weiteres Training und Optimierung verbessert werden kann.

7.3 Ausblick

Die implementierte Methode dieser Arbeit hat viel Potential und kann durch Anpassung der Techniken und Anwendung von Optimierungstechniken verbessert werden. Eine Anwendung auf andere und größere Datensätze wird mit Sicherheit bessere Ergebnisse produzieren. Die Methode auf Videos anzuwenden würde ebenfalls interessante Ergebnisse liefern.

Methoden aus dem Bereich des unüberwachten Lernens werden heutzutage angewendet um das Image Colorization Problem zu lösen. Generative adversarial Networks (GANs) werden in der Praxis öfter angewendet, als normale CNNs da diese viel bessere Ergebnisse erzeugen.

Abbildungsverzeichnis

2.1	Fully-connected Neural Network mit 2 Layers (eine Hidden Layer mit 4 Neuronen und eine Output Layer mit 2 Neuronen) [Fei20a]	5
2.2	Sigmoid Aktivierungsfunktion [Fei20b]	7
2.3	Tanh Aktivierungsfunktion [Fei20b]	8
2.4	Rectified Linear Unit (ReLU) [Fei20b]	9
2.5	Leaky ReLU [ccs20]	9
2.6	Gradient descent visualisiert [Bha18]	12
2.7	Backpropagation Beispiel anhand eines 2D Neurons mit der Aktivierungsfunktion Sigmoid [Fei20c]	13
2.8	Typische Struktur von einem Convolutional Neural Network [Com15]	14
2.9	Beispiel eines Forward pass von einer Convolutional Layer mit einem $7 \times 7 \times 3$ Input Volumen, zwei $3 \times 3 \times 3$ Filtern, Padding 1 und Stride 2. [Fei20d]	15
2.10	Max pooling Operation mit 2×2 Filtern und Stride 2 [Fei20d]	16
2.11	Die komplette Transposed Convolution Operation [Zha20]	16
3.1	Original Bild in RGB oben links, Belichtungskanal "L" oben rechts, Farbkanal "a" unten links und Farbkanal "b" unten rechts.	21
3.2	Grid mit 36 bins. Die x-Achse bildet die Werte des Farbkanals "a" und die y-Achse die Werte des Farbkanals "b" ab. (Eigene Darstellung)	22
3.3	U-net Architektur (Beispiel für 32×32 Pixel in der niedrigsten Auflösung). Jede blaue Box entspricht einer multi-Kanal Feature Map. Die Tiefe der Feature Maps ist gekennzeichnet durch die Zahl über der Box. Die Breite und Höhe ist durch die Zahl unten links erkennbar. Die weißen Boxen repräsentieren die kopierten Feature Maps. Die Pfeile bestimmen die verschiedenen Operationen. [RFB15]	26
3.4	Beispiel von Trainingsbildern mit einer der möglichen Farben pro Klasse	27
4.1	ConvBlock (Eigene Darstellung)	33
4.2	U-Net Architektur für 128×128 Input Bilder (Eigene Darstellung)	34

5.1	Beispiele von sehr guten Ergebnissen aus dem Spiel-Datensatz	35
5.2	Beispiele von generalisierten Ergebnissen	36
5.3	Beispiele von guten Ergebnissen aus dem Subset von CIFAR-100 mit 324 Bins. Die erste Spalte beinhaltet das Graustufenbild, die zweite Spalte beinhaltet das Original Bild und die letzte Spalte stellt das generierte Bild dar. Das generierte Bild wurde mit einer Temperatur von 1 erzeugt, was bedeutet, dass die rekonstruierten Farben den Durchschnitt aus jedem Bin repräsentieren.	37
5.4	Einfluss der Anzahl der Bins auf die Ergebnisse. Das zweite Bild ist das original, das dritte Bild wurde mit 36 Bins und einem Temperaturwert von 1 generiert und das vierte mit 324 Bins und ebenfalls mit einem Tempe- raturwert von 1.	38
5.5	Vergleich von Klassifikation mit Binning gegenüber Regression. Das zweite Bild wurde mit 324 Bins und dem Cross Entropy Loss generiert. Das dritte Bild wurde ohne Binning und mit einem MSE Loss generiert.	39
5.6	Ergebnisse mit 324 Bins. Die erste Spalte zeigt das Originale Bild, die zweite zeigt das generierte Bild mit einer Temperatur von 0, die dritte Spalte zeigt das generierte Bild mit einer Temperatur von 0.8 und die vierte Spalte zeigt das generierte Bild mit einer Temperatur von 1	40
5.7	Ergebnisse mit 36 und 324 Bins. Die erste Spalte zeigt das Graustufenbild, die zweite zeigt das Originale Bild, die dritte Spalte zeigt das mit 36 Bins generierte Bild mit einer Temperatur von 1 und die vierte Spalte zeigt das mit 324 Bins generierte Bild mit einer Temperatur von 1	40
6.1	Training und Validation Loss Verlauf des Spiel-Datensatzes	42
6.2	Overfitting auf dem CIFAR-100 Subset. Links: 100 Epochen mit Adam, ReLU und einer Lernrate von 0.001. Rechts: 100 Epochen mit Adam, ReLU und einer Lernrate von 0.0001.	43
6.3	Loss Verlauf mit einer reduzierten Anzahl an Parametern und verschiedenen Aktivierungsfunktionen. Links: 100 Epochen mit Adam, ReLU und einer Lernrate von 0.001. Rechts: 100 Epochen mit Adam, Tanh und einer Lernrate von 0.001.	44
A.1	Ergebnisse mit 324 Bins. Die erste Spalte zeigt das Originale Bild, die zweite zeigt das generierte Bild mit einer Temperatur von 0, die dritte Spalte zeigt das generierte Bild mit einer Temperatur von 0.8 und die vierte Spalte zeigt das generierte Bild mit einer Temperatur von 1	XI

A.2 Ergebnisse mit 324 Bins. Die erste Spalte zeigt das Originale Bild, die zweite zeigt das generierte Bild mit einer Temperatur von 0, die dritte Spalte zeigt das generierte Bild mit einer Temperatur von 0.8 und die vierte Spalte zeigt das generierte Bild mit einer Temperatur von 1	XII
A.3 Ergebnisse mit 324 Bins wo das Original Bild ein Graustufenbild ist.	XIII

Source Code Content

4.1	Binning eines normalisierten Lab Bildes	31
4.2	Leere Dictionary Erzeugung für n_bins	31
4.3	“Bin-zu-Farbe” Umwandlung	31
4.4	“Bin-zu-Farbe” Berechnung mit einem Temperaturwert	32

Glossar

Bin Behälter. 18, 21, 22

Grid Raster. 22, 30, I

Ground Truth Zielwerte. 10

Layer Schicht. 3, 14, 15, I

Loss Function Kostenfunktion. 10, 11, 13, 18

Overfitting Überanpassung eines Modells auf die Trainings Datenpunkte. 35, 36, 38, 39, 42, 43, 44, 46, II

Stride Schrittweite einer Faltung bei einem CNN. 14, 15, 16, I

Abkürzungsverzeichnis

CIE Internationale Beleuchtungskommission. 17

CNN Convolutional Neural Network (Gefaltetes Neuronales Netzwerk). 14, 23, 47, V

Literaturverzeichnis

- [BRT18] Vincent Billaut, Matthieu de Rochemonteix und Marc Thibault. *ColorUNet: A convolutional classification approach to colorization*. 2018. arXiv: 1811.03120 [cs.CV]. (Besucht am 01.08.2020).
- [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.
- [KB14] Diederik P. Kingma und Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].
- [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).
- [Özb19] Gökhan Özbulak. *Image Colorization By Capsule Networks*. 2019. arXiv: 1908.08307 [eess.IV].
- [RFB15] Olaf Ronneberger, Philipp Fischer und Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].
- [Zha20] Aston Zhang u. a. *Dive into Deep Learning*. <https://d2l.ai>. 2020.
- [ZIE16] Richard Zhang, Phillip Isola und Alexei A. Efros. *Colorful Image Colorization*. 2016. arXiv: 1603.08511 [cs.CV]. (Besucht am 01.08.2020).

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 Anwendungsbiete*. 2019. URL: <https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsbiete-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/einfuehrung-in-die-welt-der-autoencoder/%7D](https://data-science-blog.com/blog/2020/04/01/einfuehrung-in-die-welt-der-autoencoder/) (besucht am 01.08.2020).

Bildreferenzen

- [Bha18] Saugat Bhattacharai. *What is gradient descent in machine learning?* 2018. URL: <https://saugatbhattacharai.com.np/what-is-gradient-descent-in-machine-learning/> (besucht am 01.08.2020).
- [ccs20] ccs96307. 2020. URL: <https://clay-atlas.com/us/blog/2020/02/03/machine-learning-english-note-relu-function/> (besucht am 31.07.2020).
- [Com15] Wikimedia Commons. *Typical CNN architecture.* 2015. URL: https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png (besucht am 20.12.2020).
- [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/optimization-2/> (besucht am 01.08.2020).
- [Fei20d] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: <https://cs231n.github.io/convolutional-networks/> (besucht am 01.08.2020).

Anhang A

A.1 Ergebnisse aus dem Landscape Datensatz

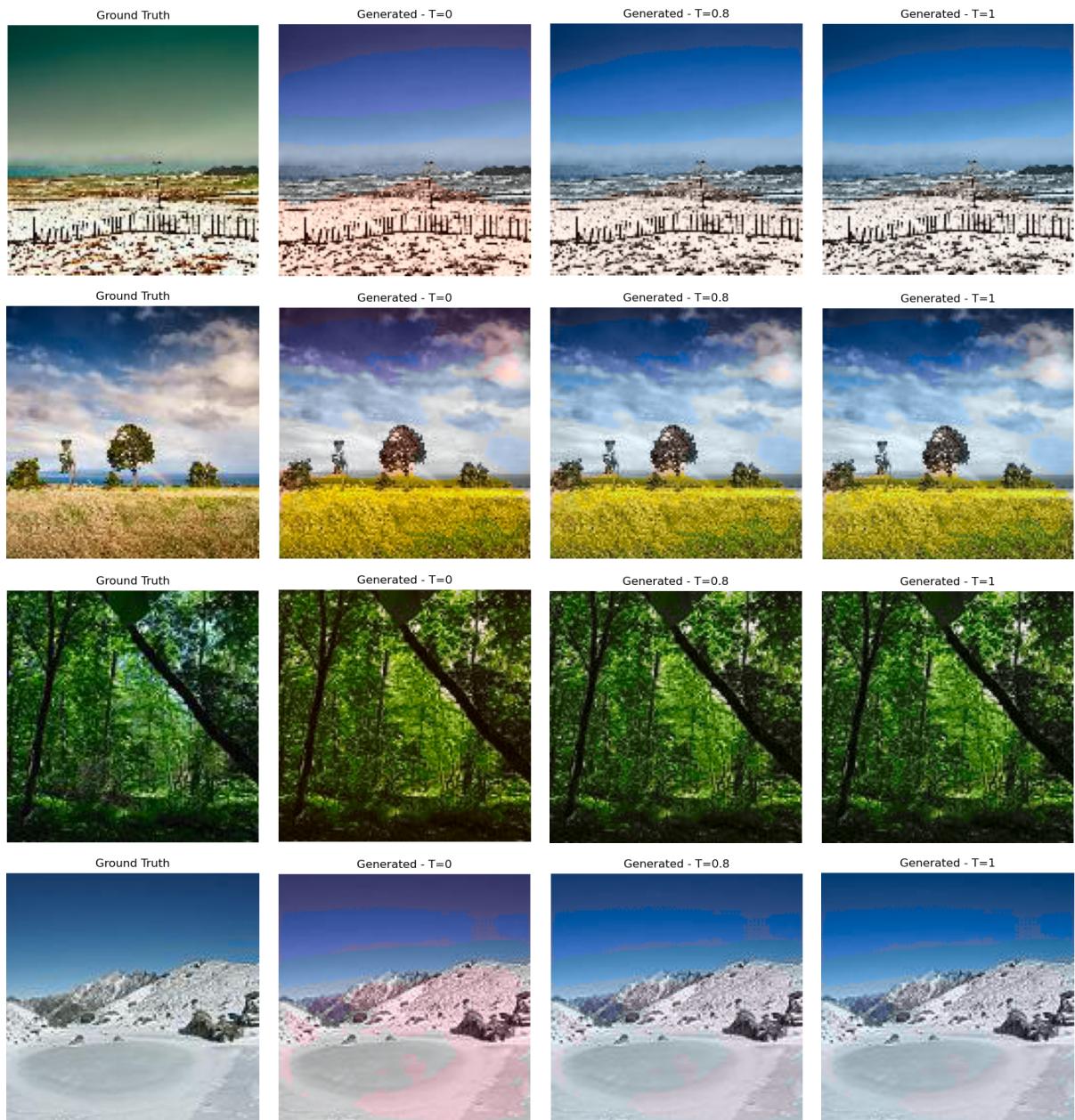


Abbildung A.1: Ergebnisse mit 324 Bins. Die erste Spalte zeigt das Originale Bild, die zweite zeigt das generierte Bild mit einer Temperatur von 0, die dritte Spalte zeigt das generierte Bild mit einer Temperatur von 0.8 und die vierte Spalte zeigt das generierte Bild mit einer Temperatur von 1

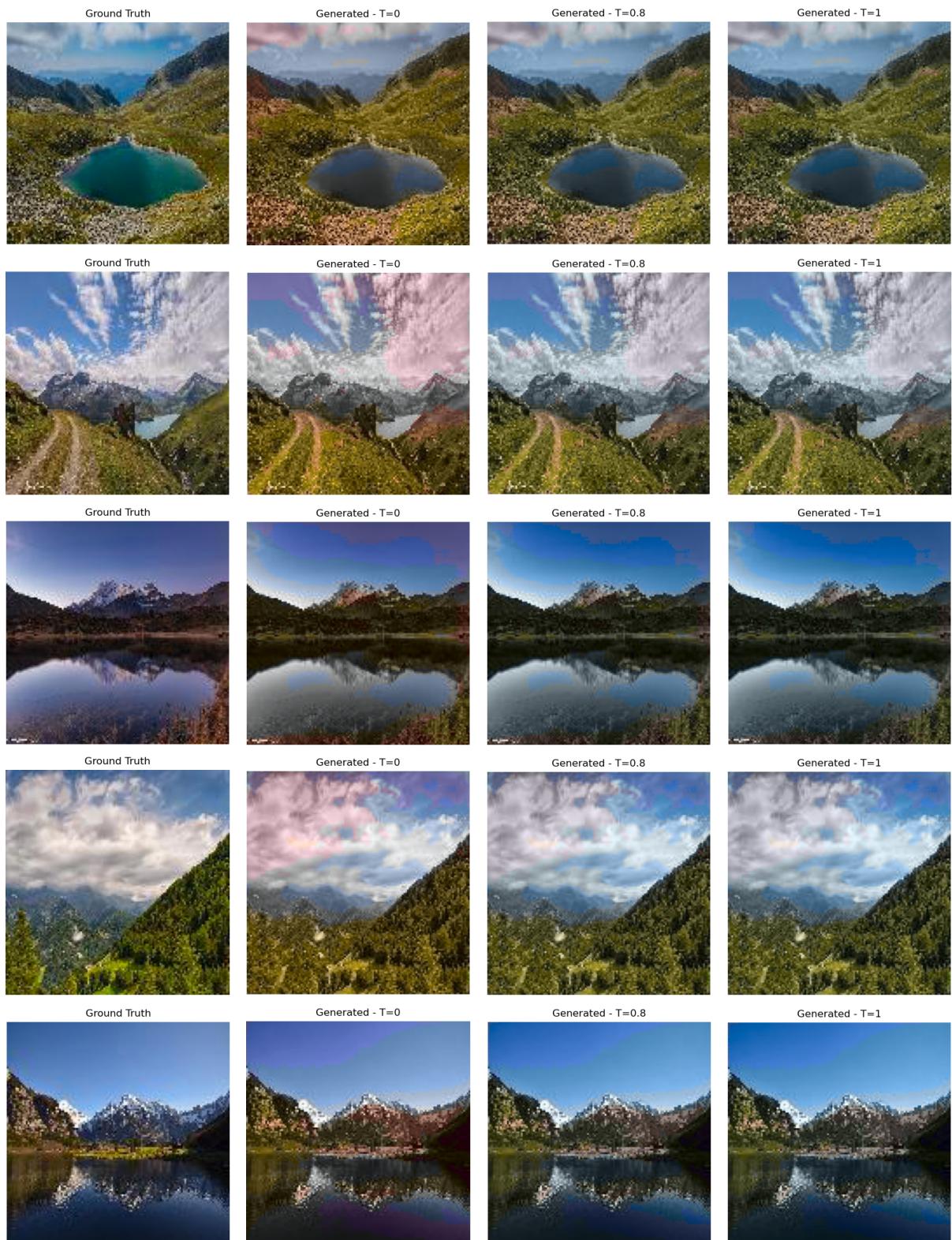


Abbildung A.2: Ergebnisse mit 324 Bins. Die erste Spalte zeigt das Originale Bild, die zweite zeigt das generierte Bild mit einer Temperatur von 0, die dritte Spalte zeigt das generierte Bild mit einer Temperatur von 0.8 und die vierte Spalte zeigt das generierte Bild mit einer Temperatur von 1



Abbildung A.3: Ergebnisse mit 324 Bins wo das Original Bild ein Graustufenbild ist.

A.2 Notebook CIFAR-100 Subset Colorization

```

1   from __future__ import print_function, division
2   import time
3   import datetime
4   import os
5   import copy
6   import csv
7
8   import matplotlib.pyplot as plt
9
10  # For conversion
11  from skimage.color import lab2rgb, rgb2lab, rgb2gray
12  from skimage import io
13
14  import numpy as np
15  import torch
16  import torch.utils.data
17  import torch.nn as nn
18  import torch.nn.functional as F
19  import torch.optim as optim
20  from torch.optim import lr_scheduler
21  from torch.optim.lr_scheduler import ReduceLROnPlateau

```

```
22     import torchvision
23     from torchvision import datasets, models, transforms
24
25     use_gpu = torch.cuda.is_available()
26     N_BINS = 324
27     W_BIN = np.sqrt(N_BINS).astype(int)
28
29     cifar_data_train = datasets.CIFAR100(".", train=True,
30                                         transform=transforms.Compose([
31                                             transforms.RandomResizedCrop(32),
32                                             transforms.RandomHorizontalFlip()
33                                         ]), target_transform=None, download=True)
34
35     cifar_data_val = datasets.CIFAR100(".", train=False,
36                                         transform=transforms.Compose([
37                                             transforms.Resize(32),
38                                             transforms.CenterCrop(32)
39                                         ]), target_transform=None, download=True)
40
41     meta = np.load('cifar-100-python/meta', allow_pickle=True)
42
43     def calculate_bin(a, b, width):
44         return (width * b) + a
45
46         """
47         Encode each pixel from the image into a bin
48
49         Output: (W, H) where each value is a bin
50         """
51
52     def encode_bins(ab_image, n_bins):
53         x = np.linspace(0, 1, W_BIN+1)
54         indices = np.digitize(ab_image, x) - 1
55
56         bins = np.vectorize(calculate_bin)(indices[:, :, 0], indices[:, :, 1], W_BIN)
57
58         return bins
59
60     def to_rgb(grayscale_input, ab_input):
61         """
62             Convert to rgb
63             """
64
65         color_image = torch.cat((grayscale_input, ab_input), 0).numpy() #combine
66                                         → channels
67         color_image = color_image.transpose((1, 2, 0)) # rescale for matplotlib
```

```
63     color_image[:, :, 0:1] = color_image[:, :, 0:1] * 100
64     color_image[:, :, 1:3] = color_image[:, :, 1:3] * 255 - 128
65     color_image = lab2rgb(color_image)
66
67     return (color_image * 255).astype(int)
68
69 class GrayscaleDataLoader(torch.utils.data.Dataset):
70     def __init__(self, dataset):
71         self.images = dataset
72
73     def __getitem__(self, index):
74         image, target = self.images[index]
75
76         img_original = np.asarray(image)
77
78         # rgb to lab
79         img_lab = rgb2lab(img_original)
80         img_lab = (img_lab + 128) / 255
81         img_ab = img_lab[:, :, 1:3]
82
83         # form bins
84         bins = torch.from_numpy(encode_bins(img_ab, N_BINS))
85
86         # ab channels
87         img_ab = torch.from_numpy(img_ab.transpose((2, 0, 1))).float()
88
89         # greyscale image
90         img_original = rgb2gray(img_original)
91         img_original = torch.from_numpy(img_original).unsqueeze(0).float()
92
93         return img_original, img_ab, bins
94
95     def __len__(self):
96         return len(self.images)
97
98 classes = [0, 23, 33, 47, 49, 52, 56, 59, 70, 71, 82, 96]
99
100 def filter_image(image):
101     if image[1] in classes:
102         return image
103
104 filtered_data_train = list(filter(None, map(filter_image, cifar_data_train)))
105 train_images = GrayscaleDataLoader(cifar_data_train)
106
```

```
107     filtered_data_val = list(filter(None, map(filter_image, cifar_data_val)))
108     val_images = GrayscaleDataLoader(cifar_data_val)
109
110     print(len(train_images))
111     print(len(val_images))
112
113     train_loader = torch.utils.data.DataLoader(train_images, batch_size=64,
114         ↵ shuffle=True)
114     val_loader = torch.utils.data.DataLoader(val_images, batch_size=64,
115         ↵ shuffle=True)
116
116     class Conv2dBlock(nn.Module):
117         def __init__(self, D_in, n_filters, kernel_size=3):
118             super(Conv2dBlock, self).__init__()
119
120             # first layer
121             self.conv1      = nn.Conv2d(D_in, n_filters, kernel_size, stride=1,
122             ↵ padding=1)
122             self.batch_norm1 = nn.BatchNorm2d(n_filters)
123
124             # second layer
125             self.conv2      = nn.Conv2d(n_filters, n_filters, kernel_size, stride=1,
126             ↵ padding=1)
126             self.batch_norm2 = nn.BatchNorm2d(n_filters)
127
128         def forward(self, x):
129             x = self.conv1(x)
130             x = self.batch_norm1(x)
131             x = F.relu(x)
132             x = self.conv2(x)
133             x = self.batch_norm2(x)
134             out = F.relu(x)
135
136             return out
137
138     class Model(nn.Module):
139         def __init__(self, n_out):
140             super(Model, self).__init__()
141
142             # Encoder
143             self.conv1 = Conv2dBlock(1, 8)
144             self.pool1 = nn.MaxPool2d(2, 2)
145
146             self.conv2 = Conv2dBlock(8, 16)
```

```
147         self.pool2 = nn.MaxPool2d(2, 2)
148
149         self.conv3 = Conv2dBlock(16, 32)
150
151     # Decoder
152     self.upconv1 = nn.ConvTranspose2d(32, 16, kernel_size=3, stride=2,
153                                     padding=1, output_padding=1)
154
155     self.conv4    = Conv2dBlock(32, 16)
156     self.upconv2 = nn.ConvTranspose2d(16, 8, kernel_size=3, stride=2,
157                                     padding=1, output_padding=1)
158
159     self.conv5    = Conv2dBlock(16, 8)
160     self.conv6    = nn.Conv2d(8, n_out, kernel_size=1, stride=1, padding=0)
161
162     def forward(self, x):
163         c1 = self.conv1(x)
164         x  = self.pool1(c1)
165
166         c2 = self.conv2(x)
167         x  = self.pool2(c2)
168
169         c3 = self.conv3(x)
170
171         u1 = self.upconv1(c3)
172         x  = torch.cat([u1, c2], dim=1)
173
174         x = self.conv4(x)
175         u2 = self.upconv2(x)
176         x  = torch.cat([u2, c1], dim=1)
177
178         x = self.conv5(x)
179         out = self.conv6(x)
180
181     # Classify mode colors to bins
182     dataset_bin_colors = {i: [[], []] for i in range(N_BINS)}
183
184     def get_dataset_bin(colors_dict, mode='mode'):
185         _dict = copy.deepcopy(colors_dict)
186
187         for bin in _dict:
188             for channel, _ in enumerate(_dict[bin]):
```

```
189         if (len(_dict[bin][channel]) > 0):
190             if mode == 'mode':
191                 _dict[bin][channel] = np.max(np.array(_dict[bin][channel]))
192             elif mode == 'mean':
193                 _dict[bin][channel] = np.mean(np.array(_dict[bin][channel]))
194             else:
195                 _dict[bin][channel] = 0
196
197             np.save(mode + '_color_bins_'+str(N_BINS)+'.npy', _dict)
198             del _dict
199
200     def add_to_dict(bin, a, b):
201         dataset_bin_colors[bin][0].append(a)
202         dataset_bin_colors[bin][1].append(b)
203
204     def _encode_bins(ab_image):
205         x = np.linspace(0,1,W_BIN+1)
206         indices = np.digitize(ab_image, x) - 1
207         indices = indices.transpose(1, 2, 0)
208
209         bins = np.vectorize(calculate_bin)(indices[:, :, 0], indices[:, :, 1], W_BIN)
210         np.vectorize(add_to_dict)(bins, ab_image[0, :, :], ab_image[1, :, :])
211
212     return bins
213
214 counter = 0
215
216 for index, y in enumerate(train_loader):
217     gray_images, ab_images, bins = y
218
219     for i, ab in enumerate(ab_images):
220         _encode_bins(ab)
221
222     get_dataset_bin(dataset_bin_colors, 'mode')
223     get_dataset_bin(dataset_bin_colors, 'mean')
224
225     """
226         Return the mode of the predicted bin for each color channel
227     """
228     def decode_pixel(bin, T, dataset_bin_colors_mode, dataset_bin_colors_mean):
229         a_mode = dataset_bin_colors_mode[bin][0]
230         b_mode = dataset_bin_colors_mode[bin][1]
231
232         a_mean = dataset_bin_colors_mean[bin][0]
```

```
233     b_mean = dataset_bin_colors_mean[bin][1]
234
235     if a_mode == 0:
236         a_mode = assign_next_bin(bin, a_mode, 0, dataset_bin_colors_mode)
237
238     if b_mode == 0:
239         b_mode = assign_next_bin(bin, b_mode, 1, dataset_bin_colors_mode)
240
241     if a_mean == 0:
242         a_mean = assign_next_bin(bin, a_mean, 0, dataset_bin_colors_mean)
243
244     if b_mean == 0:
245         b_mean = assign_next_bin(bin, b_mean, 1, dataset_bin_colors_mean)
246
247     a_distance = a_mode - a_mean
248     b_distance = b_mode - b_mean
249
250     a = a_mode - (a_distance * T)
251     b = b_mode - (b_distance * T)
252
253     return a, b
254
255     """
256     Assign predicted color from next bin
257     """
258     def assign_next_bin(bin, channel, index, colormap):
259         counter = [1, -1]
260         length = len(colormap) - 1
261
262         while channel == 0:
263             plus_index = bin + counter[0] if bin + counter[0] <= length else length
264             channel = colormap[plus_index][index]
265
266             if channel == 0:
267                 counter[0] += 1
268                 minus_index = bin + counter[1] if bin + counter[1] >= 0 else 0
269                 channel = colormap[minus_index][index]
270                 counter[1] -= 1
271
272         return channel
273
274     def deserialize_bins(bins, T, mode_path, mean_path):
275         bins = bins.numpy()
276         dataset_bin_colors_mode = np.load(mode_path, allow_pickle=True)
```

```
277     dataset_bin_colors_mean = np.load(mean_path, allow_pickle=True)
278
279     return np.array(np.vectorize(decode_pixel)(bins, T, dataset_bin_colors_mode,
280                                → dataset_bin_colors_mean))
281
282     """
283     A class from the PyTorch ImageNet tutorial
284     """
285
286     class AverageMeter(object):
287         def __init__(self):
288             self.reset()
289         def reset(self):
290             self.val, self.avg, self.sum, self.count = 0, 0, 0, 0
291         def update(self, val, n=1):
292             self.val = val
293             self.sum += val * n
294             self.count += n
295             self.avg = self.sum / self.count
296
297         def train(train_loader, model, criterion, optimizer, epoch, use_gpu):
298             print('Starting training epoch {}'.format(epoch))
299             model.train()
300
301             # Prepare value counters and timers
302             batch_time, data_time, losses = AverageMeter(), AverageMeter(),
303             → AverageMeter()
304
305             end = time.time()
306             for i, (input_gray, input_ab, bins) in enumerate(train_loader):
307                 bins = bins.squeeze(0)
308                 # Use GPU if available
309                 if use_gpu: input_gray, input_ab, bins = input_gray.cuda(),
310                                → input_ab.cuda(), bins.cuda()
311
312                 # Record time to load data (above)
313                 data_time.update(time.time() - end)
314
315                 # Run forward pass
316                 output_bins = model(input_gray)
317                 loss = criterion(output_bins, bins)
318                 losses.update(loss.item(), input_gray.size(0))
319
320                 # Compute gradient and optimize
321                 optimizer.zero_grad()
```

```
318         loss.backward()
319         optimizer.step()
320
321         # Record time to do forward and backward passes
322         batch_time.update(time.time() - end)
323         end = time.time()
324
325         # Print model accuracy -- in the code below, val refers to value, not
326         # validation
327         if i % 25 == 0:
328             print('Epoch: [{0}][{1}/{2}]\n'
329                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\n'
330                  'Data {data_time.val:.3f} ({data_time.avg:.3f})\n'
331                  'Loss {loss.val:.4f} ({loss.avg:.4f})'.format(
332                     epoch, i, len(train_loader), batch_time=batch_time,
333                     data_time=data_time, loss=losses))
334
335             print('Finished training epoch {}'.format(epoch))
336
337             return losses.avg
338
339     def validate(val_loader, model, criterion, epoch, use_gpu):
340         model.eval()
341
342         # Prepare value counters and timers
343         batch_time, data_time, losses = AverageMeter(), AverageMeter(),
344         # AverageMeter()
345
346         end = time.time()
347
348         for i, (input_gray, input_ab, bins) in enumerate(val_loader):
349             data_time.update(time.time() - end)
350             bins = bins.squeeze(0)
351
352             # Use GPU
353             if use_gpu: input_gray, input_ab, bins = input_gray.cuda(),
354             # input_ab.cuda(), bins.cuda()
355
356             # Run model and record loss
357             output_bins = model(input_gray)
358             loss = criterion(output_bins, bins)
359             losses.update(loss.item(), input_gray.size(0))
360
361             # Record time to do forward passes and save images
```

```
359         batch_time.update(time.time() - end)
360         end = time.time()
361
362         # Print model accuracy -- in the code below, val refers to both value and
363         # validation
364         if i % 25 == 0:
365             print('Validate: [{0}/{1}]\t'
366                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
367                  'Loss {loss.val:.4f} ({loss.avg:.4f})'.format(i, len(val_loader),
368                        batch_time=batch_time, loss=losses))
368
369         print('Finished validation.')
370
371     model = Model(324)
372     criterion = nn.CrossEntropyLoss()
373     optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
374
375     if use_gpu:
376         criterion = criterion.cuda()
377         model = model.cuda()
378
379     epochs = 150
380     best_losses = 2.5
381
382     output = csv.writer(open('train_log.csv', 'w'))
383     output.writerow(['epoch', 'train_loss', 'val_loss'])
384
385     with open('train_log.csv', 'w') as output:
386         writer = csv.writer(output)
387         writer.writerow(['epoch', 'train_loss', 'val_loss'])
388
389     for epoch in range(60, epochs):
390         if use_gpu and epoch > 0:
391             model.cuda()
392             # Train for one epoch, then validate
393             train_loss = train(train_loader, model, criterion, optimizer, epoch,
394                               use_gpu)
394             with torch.no_grad():
395                 val_loss = validate(val_loader, model, criterion, epoch, use_gpu)
396                 # scheduler.step(val_loss)
397                 writer.writerow([epoch, train_loss, val_loss])
398                 # Save checkpoint and replace old best model if current model is better
399                 print('Val Loss = {}'.format(val_loss))
```

```

400     print('Best Loss = {}'.format(best_losses))
401     if val_loss < best_losses:
402         best_losses = val_loss
403         torch.save(model.to('cpu').state_dict(),
404                     'model-cifar-{}-{}-{:3f}.pth'.format(N_BINS, epoch+1, val_loss))
405
406     def evaluate(gray, ab_input, bins, model, temperature):
407         model.eval()
408
409         with torch.no_grad():
410             bins = bins.squeeze(0)
411
412             # Run model and record loss
413             # add batch size 1 for single image
414             output_bins = model(gray.unsqueeze(1))
415
416             # remove batch size and get the max index of the predicted bin for each
417             # pixel
418             color_image = to_rgb(
419                 gray,
420                 torch.from_numpy(
421                     deserialize_bins(
422                         output_bins.squeeze(0).argmax(0),
423                         temperature,
424                         'mode_color_bins_'+str(N_BINS)+'.npy',
425                         'mean_color_bins_'+str(N_BINS)+'.npy',
426                         ),
427                         ).float()
428             )
429
430             return color_image

```

A.3 Skripte Lanscape Datensatz Colorization

A.3.1 Image augmentation Skript

```

1   import os
2   from glob import glob
3
4   import imageio
5   import imgaug as ia

```

```
6      import imgaug.augmenters as iaa
7      import numpy as np
8      import pandas as pd
9      import matplotlib.pyplot as plt
10     import matplotlib.patches as patches
11     import matplotlib
12
13     DATASET_PATH = "dataset_clean/train/"
14     DATASET_VAL_PATH = "dataset_clean/val/"
15     AUGUMENTED_PATH = "dataset_resized/train/"
16     AUGUMENTED_VAL_PATH = "dataset_resized/val/"
17     CLASSES = ["field", "forest", "glacier", "lake", "mountain", "road", "sea",
18                 ← "uncategorized"]
18     SIZE = 128
19
20     for _class in CLASSES:
21         train_files = glob("./{}{}/{}".format(AUGUMENTED_PATH, _class))
22         val_files = glob("./{}{}/{}".format(AUGUMENTED_VAL_PATH, _class))
23
24         for f in train_files:
25             os.remove(f)
26
27         for f in val_files:
28             os.remove(f)
29
30     aug = iaa.Resize({"height": SIZE, "width": SIZE}, "cubic")
31     rotate=iaa.Affine(rotate=(-30, 30))
32     crop = iaa.Crop(percent=(0, 0.3))
33     flip_hr=iaa.Fliplr(p=1.0)
34     flip_vr=iaa.Flipud(p=1.0)
35
36     # Resize training images to the given size
37     for i, _class in enumerate(CLASSES):
38         for filename in os.listdir(DATASET_PATH + "/" + CLASSES[i]):
39             image = imageio.imread(DATASET_PATH + "/" + CLASSES[i] + "/" + filename)
40             augmented_img = aug.augment_image(image)
41             imageio.imwrite(AUGUMENTED_PATH + "/" + CLASSES[i] + "/" + filename,
42                           ← augmented_img)
42
43     # Resize validation images to the given size
44     for i, _class in enumerate(CLASSES):
45         for filename in os.listdir(DATASET_VAL_PATH + "/" + CLASSES[i]):
46             image = imageio.imread(DATASET_VAL_PATH + "/" + CLASSES[i] + "/" +
47                                   ← filename)
```

```

47     augmented_img = aug.augment_image(image)
48     imageio.imwrite(AUGUMENTED_VAL_PATH + "/" + CLASSES[i] + "/" + filename,
49                      ↵     augmented_img)
50
51     # Apply argumentation to the images
52     for i, _class in enumerate(CLASSES):
53         for filename in os.listdir(AUGUMENTED_PATH + "/" + CLASSES[i]):
54             image = imageio.imread(AUGUMENTED_PATH + "/" + CLASSES[i] + "/" +
55                                   ↵     filename)
56
57             # Rotate image between -30 and 30 degrees
58             rotated_image=rotate.augment_image(image)
59             imageio.imwrite(AUGUMENTED_PATH + "/" + CLASSES[i] + "/rotated_" +
60                             ↵     filename, rotated_image)
61
62             # Crop image by a factor of 30%
63             cropped_image=crop.augment_image(image)
64             imageio.imwrite(AUGUMENTED_PATH + "/" + CLASSES[i] + "/croped_" + filename,
65                             ↵     cropped_image)
66
67             # Flip image horizontally
68             flip_hr_image= flip_hr.augment_image(image)
69             imageio.imwrite(AUGUMENTED_PATH + "/" + CLASSES[i] + "/hr_flipped_" +
70                             ↵     filename, flip_hr_image)
71
72             # Flip image vertically
73             flip_vr_image= flip_vr.augment_image(image)
74             imageio.imwrite(AUGUMENTED_PATH + "/" + CLASSES[i] + "/vr_flipped_" +
75                             ↵     filename, flip_vr_image)

```

A.3.2 Modell Skript

```

1   from __future__ import print_function, division
2
3   import numpy as np
4   import torch
5   import torch.nn as nn
6   import torch.nn.functional as F
7
8   class Conv2dBlock(nn.Module):
9       def __init__(self, D_in, n_filters, kernel_size=3):
10          super(Conv2dBlock, self).__init__()
11

```

```
12     # first layer
13     self.conv1      = nn.Conv2d(D_in, n_filters, kernel_size, stride=1,
14                               ← padding=1)
14     self.batch_norm1 = nn.BatchNorm2d(n_filters)
15
15
16     # second layer
17     self.conv2      = nn.Conv2d(n_filters, n_filters, kernel_size, stride=1,
18                               ← padding=1)
18     self.batch_norm2 = nn.BatchNorm2d(n_filters)
19
20
20     def forward(self, x):
21         x = self.conv1(x)
22         x = self.batch_norm1(x)
23         x = F.relu(x)
24         x = self.conv2(x)
25         x = self.batch_norm2(x)
26         out = F.relu(x)
27
27
28     return out
29
30
30     class Model(nn.Module):
31         def __init__(self, n_out, divider=1):
32             super(Model, self).__init__()
33
34             # Encoder
35             self.conv1 = Conv2dBlock(1, int(16 / divider))
36             self.pool1 = nn.MaxPool2d(2, 2)
37
38             self.conv2 = Conv2dBlock(int(16 / divider), int(32 / divider))
39             self.pool2 = nn.MaxPool2d(2, 2)
40
41             self.conv3 = Conv2dBlock(int(32 / divider), int(64 / divider))
42             self.pool3 = nn.MaxPool2d(2, 2)
43
44             self.conv4 = Conv2dBlock(int(64 / divider), int(128 / divider))
45             self.pool4 = nn.MaxPool2d(2, 2)
46
46
47             self.conv5 = Conv2dBlock(int(128 / divider), int(256 / divider))
48
48             # Decoder
49             self.upconv6 = nn.ConvTranspose2d(int(256 / divider), int(128 / divider),
50                                           ← kernel_size=3, stride=2, padding=1, output_padding=1)
51
52             self.conv6   = Conv2dBlock(int(256 / divider), int(128 / divider))
```

```
53     self.upconv7 = nn.ConvTranspose2d(int(128 / divider), int(64 / divider),
54         kernel_size=3, stride=2, padding=1, output_padding=1)
55
56     self.conv7    = Conv2dBlock(int(128 / divider), int(64 / divider))
57     self.upconv8 = nn.ConvTranspose2d(int(64 / divider), int(32 / divider),
58         kernel_size=3, stride=2, padding=1, output_padding=1)
59
60     self.conv8    = Conv2dBlock(int(64 / divider), int(32 / divider))
61     self.upconv9 = nn.ConvTranspose2d(int(32 / divider), int(16 / divider),
62         kernel_size=3, stride=2, padding=1, output_padding=1)
63
64     def forward(self, x):
65         c1 = self.conv1(x)
66         x  = self.pool1(c1)
67
68         c2 = self.conv2(x)
69         x  = self.pool2(c2)
70
71         c3 = self.conv3(x)
72         x  = self.pool3(c3)
73
74         c4 = self.conv4(x)
75         x  = self.pool4(c4)
76
77         c5 = self.conv5(x)
78
79         u6 = self.upconv6(c5)
80         x  = torch.cat([u6, c4], dim=1)
81
82         x = self.conv6(x)
83
84         u7 = self.upconv7(x)
85         x  = torch.cat([u7, c3], dim=1)
86
87         x = self.conv7(x)
88
89         u8 = self.upconv8(x)
90         x  = torch.cat([u8, c2], dim=1)
91
92         x  = self.conv8(x)
```

```
93
94     u9 = self.upconv9(x)
95     x = torch.cat([u9, c1], dim=1)
96
97     x = self.conv9(x)
98
99     out = self.conv10(x)
100
101    return out
```

A.3.3 Utils Skript

```
1   from __future__ import print_function, division
2   import time
3   import os
4   import copy
5
6   from skimage.color import lab2rgb, rgb2lab, rgb2gray
7   from skimage import io
8
9   import numpy as np
10  import torch
11  from torchvision import datasets, models, transforms
12
13  class AverageMeter(object):
14      '''A handy class from the PyTorch ImageNet tutorial'''
15      def __init__(self):
16          self.reset()
17      def reset(self):
18          self.val, self.avg, self.sum, self.count = 0, 0, 0, 0
19      def update(self, val, n=1):
20          self.val = val
21          self.sum += val * n
22          self.count += n
23          self.avg = self.sum / self.count
24
25      def load_img(img, N_BINS):
26          img_original = np.asarray(img)
27
28          # rgb to lab
29          img_lab = rgb2lab(img_original)
30          img_lab = (img_lab + 128) / 255
```

```
31     img_ab = img_lab[:, :, 1:3]
32
33
34     # form bins
35     bins = torch.from_numpy(encode_bins(img_ab, N_BINS))
36
37     #ab channels
38     img_ab = torch.from_numpy(img_ab.transpose((2, 0, 1))).float()
39
40     # greyscale image
41     img_original = rgb2gray(img_original)
42     img_original = torch.from_numpy(img_original).unsqueeze(0).float()
43
44     return img_original, img_ab, bins
45
46     """
47     Generate a dictionary with the mode of every bin
48
49     Output: BIN_NUMBER: [A_COLOR_MODE, B_COLOR_MODE]
50     """
51
52     def get_dataset_bin_mode(colors_dict):
53         mode_dict = copy.deepcopy(colors_dict)
54
55         for bin in mode_dict:
56             for channel, _ in enumerate(mode_dict[bin]):
57                 if (len(mode_dict[bin][channel]) > 0):
58                     mode_dict[bin][channel] = np.max(np.array(mode_dict[bin][channel]))
59                 else:
60                     mode_dict[bin][channel] = 0
61
62         return mode_dict
63
64     """
65     Calculate the bin from the indices
66
67     Output: N
68     """
69     def calculate_bin(a, b, width):
70         return (width * b) + a
71
72     """
73     Add color to each bin dictionary value
74     """
75     def add_to_dict(bin, a, b):
```

```
75     dataset_bin_colors[bin][0].append(a)
76     dataset_bin_colors[bin][1].append(b)
77
78
79     '''
80     Encode each pixel from the image into a bin
81
82     Output: (W, H) where each value is a bin
83     '''
84
85     def encode_bins(ab_image, n_bins):
86         w_bin = np.sqrt(n_bins).astype(int)
87
88         x = np.linspace(0,1,w_bin+1)
89         indices = np.digitize(ab_image, x) - 1
90
91         bins = np.vectorize(calculate_bin)(indices[:, :, 0], indices[:, :, 1], w_bin)
92
93         return bins
94
95     '''
96     Return the mode of the predicted bin for each color channel
97     '''
98
99     def decode_pixel(bin, T, dataset_bin_colors_mode, dataset_bin_colors_mean):
100        a_mode = dataset_bin_colors_mode[bin][0]
101        b_mode = dataset_bin_colors_mode[bin][1]
102
103        a_mean = dataset_bin_colors_mean[bin][0]
104        b_mean = dataset_bin_colors_mean[bin][1]
105
106        if a_mode == 0:
107            a_mode = assign_next_bin(bin, a_mode, 0, dataset_bin_colors_mode)
108
109        if b_mode == 0:
110            b_mode = assign_next_bin(bin, b_mode, 1, dataset_bin_colors_mode)
111
112        if a_mean == 0:
113            a_mean = assign_next_bin(bin, a_mean, 0, dataset_bin_colors_mean)
114
115        if b_mean == 0:
116            b_mean = assign_next_bin(bin, b_mean, 1, dataset_bin_colors_mean)
117
118        a_distance = a_mode - a_mean
119        b_distance = b_mode - b_mean
```

```
119         a = a_mode - (a_distance * T)
120         b = b_mode - (b_distance * T)
121
122     return a, b
123
124     """
125     Assign predicted color from next bin
126     """
127     def assign_next_bin(bin, channel, index, colormap):
128         counter = [1, -1]
129         length = len(colormap) - 1
130
131         while channel == 0:
132             plus_index = bin + counter[0] if bin + counter[0] <= length else length
133             channel = colormap[plus_index][index]
134
135         if channel == 0:
136             counter[0] += 1
137             minus_index = bin + counter[1] if bin + counter[1] >= 0 else 0
138             channel = colormap[minus_index][index]
139             counter[1] -= 1
140
141     return channel
142
143     def serialize_bins(ab_image):
144         return encode_bins(ab_image)
145
146     def deserialize_bins(bins, T, mode_path, mean_path):
147         bins = bins.numpy()
148         dataset_bin_colors_mode = np.load(mode_path, allow_pickle=True)
149         dataset_bin_colors_mean = np.load(mean_path, allow_pickle=True)
150
151         return np.array(np.vectorize(decode_pixel)(bins, T, dataset_bin_colors_mode,
152             → dataset_bin_colors_mean))
152
153     def to_rgb(grayscale_input, ab_input):
154         """
155         Convert to rgb
156         """
157         color_image = torch.cat((grayscale_input, ab_input), 0).numpy() #combine
158             → channels
159         color_image = color_image.transpose((1, 2, 0)) # rescale for matplotlib
160         color_image[:, :, 0:1] = color_image[:, :, 0:1] * 100
160         color_image[:, :, 1:3] = color_image[:, :, 1:3] * 255 - 128
```

```
161     color_image = lab2rgb(color_image)
162
163     return (color_image * 255).astype(int)
```

A.3.4 Train Skript

```
1      import time
2      from utils import AverageMeter
3
4      def train(train_loader, model, criterion, optimizer, epoch, use_gpu=False):
5          print('Starting training epoch {}'.format(epoch))
6          model.train()
7
8          # Prepare value counters and timers
9          batch_time, data_time, losses = AverageMeter(), AverageMeter(),
10             AverageMeter()
11
12         end = time.time()
13         for i, (input_gray, input_ab, bins) in enumerate(train_loader):
14             bins = bins.squeeze(0)
15             # Use GPU if available
16             if use_gpu: input_gray, input_ab, bins = input_gray.cuda(),
17                           input_ab.cuda(), bins.cuda()
18
19             # Record time to load data (above)
20             data_time.update(time.time() - end)
21
22             # Run forward pass
23             output_bins = model(input_gray)
24
25             if len(output_bins) != len(bins):
26                 bins = bins.unsqueeze(0)
27
28             loss = criterion(output_bins, bins)
29             losses.update(loss.item(), input_gray.size(0))
30
31             # Compute gradient and optimize
32             optimizer.zero_grad()
33             loss.backward()
34             optimizer.step()
35
36             # Record time to do forward and backward passes
```

```

35         batch_time.update(time.time() - end)
36     end = time.time()
37
38     # Print model accuracy -- in the code below, val refers to value, not
39     # → validation
40     if i % 25 == 0:
41         print('Epoch: [{0}] [{1}/{2}]\t'
42               'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
43               'Data {data_time.val:.3f} ({data_time.avg:.3f})\t'
44               'Loss {loss.val:.4f} ({loss.avg:.4f})\t'.format(
45                   epoch, i, len(train_loader), batch_time=batch_time,
46                   data_time=data_time, loss=losses))
47
48     print('Finished training epoch {}'.format(epoch))
49     return losses.avg

```

A.3.5 Validation Skript

```

1      import time
2      from utils import AverageMeter
3
4      def validate(val_loader, model, criterion, epoch, use_gpu=False):
5          model.eval()
6
7          # Prepare value counters and timers
8          batch_time, data_time, losses = AverageMeter(), AverageMeter(),
9          # → AverageMeter()
10
11         end = time.time()
12
13         for i, (input_gray, input_ab, bins) in enumerate(val_loader):
14             data_time.update(time.time() - end)
15             bins = bins.squeeze(0)
16
17             # Use GPU
18             if use_gpu: input_gray, input_ab, bins = input_gray.cuda(),
19             # → input_ab.cuda(), bins.cuda()
20
21             # Run model and record loss
22             output_bins = model(input_gray)
23
24             if len(output_bins) != len(bins):

```

```

23         bins = bins.unsqueeze(0)
24
25         loss = criterion(output_bins, bins)
26         losses.update(loss.item(), input_gray.size(0))
27
28         # Record time to do forward passes and save images
29         batch_time.update(time.time() - end)
30         end = time.time()
31
32         # Print model accuracy -- in the code below, val refers to both value and
33         # validation
34         if i % 25 == 0:
35             print('Validate: [{0}/{1}]\t'
36                   'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
37                   'Loss {loss.val:.4f} ({loss.avg:.4f})\t'.format(i, len(val_loader),
38                                         batch_time=batch_time, loss=losses))
39
40         print('Finished validation.')
41
42         return losses.avg

```

A.3.6 Main Skript

```

1      import os
2      import sys
3      import argparse
4      import time
5      import copy
6      import shutil
7      import random
8      import csv
9
10     from PIL import Image
11
12     # For conversion
13     from skimage.color import lab2rgb, rgb2lab, rgb2gray
14     from skimage import io
15
16     import matplotlib.pyplot as plt
17
18     import numpy as np
19     import torch
20     import torch.nn as nn

```

```
21     import torch.utils.data
22     import torchvision
23     from torch.optim.lr_scheduler import ReduceLROnPlateau
24     from torchvision import datasets, models, transforms
25     import torchvision.transforms.functional as TF
26
27     from model import Model
28     from utils import to_rgb, encode_bins, deserialize_bins, load_img
29     from train import train
30     from validate import validate
31     from evaluate import evaluate
32
33     # Arguments
34     parser = argparse.ArgumentParser()
35
36     parser.add_argument(
37         dest='data_dir', type=str,
38         help='Data: Path to read-only directory containing image *.jpeg files.'
39     )
34
36     parser.add_argument(
37         '--saved-model-dir', type=str, default=None,
38         help='Data: Path of dir of last saved model'
39     )
34
36     parser.add_argument(
37         '--saved-model-file', type=str, default=None,
38         help='Data: File of last saved model'
39     )
34
36     parser.add_argument(
37         '--checkpoints-dir', type=str, default=None,
38         help='Data: Path to writable directory for the checkpoint files'
39     )
34
36     parser.add_argument(
37         '--learning-rate', type=float, default=0.001,
38         help='Training: Learning rate. Default: 0.001'
39     )
34
36     parser.add_argument(
37         '--model-divider', type=int, default=1,
38         help='Model divider to divide number of the filters in the conv layers.
39             → Default 1.'
```

```
64      )
65
66      parser.add_argument(
67          '--batch-size', type=int, default=64,
68          help='Training: Batch size. Default: 64'
69      )
70
71      parser.add_argument(
72          '--num-bins', type=int, default=36,
73          help='Training: Number of bins. Default: 36'
74      )
75
76      parser.add_argument(
77          '--from-epoch', type=int, default=0,
78          help='Training: From epoch. Default: 0'
79      )
80
81      parser.add_argument(
82          '--num-epochs', type=int, default=100,
83          help='Training: Number of epochs. Default: 100'
84      )
85
86      parser.add_argument(
87          '--log-dir', type=str, default=None,
88          help='Debug: Path to writable directory for a log file to be created.
89          ↳ Default: log to stdout / stderr'
90      )
91
92      parser.add_argument(
93          '--log-file-name', type=str, default='training.csv',
94          help='Debug: Name of the log file, generated when --log-dir is set. Default:
95          ↳ training.csv'
96      )
97
98      parser.add_argument(
99          '--seed', type=int, default=None,
100         help='Parameter: Seed for the visualization'
101     )
102
103     parser.add_argument(
104         '--temperature', type=float, default=1,
105         help='Parameter: Temperature parameter to tune the predictions.'
```

```
106
107     args = parser.parse_args()
108     temperature = args.temperature
109     seed = args.seed
110
111     if seed is not None:
112         random.seed(seed)
113         torch.manual_seed(seed)
114         torch.cuda.manual_seed(seed)
115         np.random.seed(seed)
116
117     # Redirect output streams for logging
118     if args.log_dir:
119         log_file = open(os.path.join(os.path.expanduser(args.log_dir),
120                                     args.log_file_name), 'w')
121
122     data_dir = os.path.expanduser(args.data_dir)
123
124     TRAIN_PATH = os.path.join(data_dir, 'train')
125     VAL_PATH = os.path.join(data_dir, 'val')
126
127     if args.checkpoints_dir is not None:
128         CHECKPOINTS_PATH = os.path.expanduser(args.checkpoints_dir)
129
130     SAVED_MODEL_PATH = None
131
132     if args.saved_model_dir is not None and args.saved_model_file is not None:
133         SAVED_MODEL_PATH = os.path.join(os.path.expanduser(args.saved_model_dir),
134                                         args.saved_model_file)
135
136     N_BINS = args.num_bins
137     W_BIN = np.sqrt(N_BINS).astype(int)
138
139     use_gpu = torch.cuda.is_available()
140
141     class GrayscaleImageFolder(datasets.ImageFolder):
142         def __getitem__(self, index):
143             path, target = self.imgs[index]
144             img = self.loader(path)
145
146             img_original = np.asarray(img)
147
148             # rgb to lab
149             img_lab = rgb2lab(img_original)
```

```
148     img_lab = (img_lab + 128) / 255
149     img_ab = img_lab[:, :, 1:3]
150
151     # form bins
152     bins = torch.from_numpy(encode_bins(img_ab, N_BINS))
153
154     # ab channels
155     img_ab = torch.from_numpy(img_ab.transpose((2, 0, 1))).float()
156
157     # grayscale image
158     img_original = rgb2gray(img_original)
159     img_original = torch.from_numpy(img_original).unsqueeze(0).float()
160
161     return img_original, img_ab, bins
162
163     # Training
164     train_transforms = transforms.Compose([
165         transforms.Resize((128, 128)),
166     ])
167     train_imagefolder = GrayscaleImageFolder(TRAIN_PATH, train_transforms)
168     train_loader = torch.utils.data.DataLoader(train_imagefolder,
169         batch_size=args.batch_size, shuffle=False)
170
171     # Validation
172     val_transforms = transforms.Compose([
173         transforms.Resize((128, 128))
174     ])
175     val_imagefolder = GrayscaleImageFolder(VAL_PATH, val_transforms)
176     val_loader = torch.utils.data.DataLoader(val_imagefolder,
177         batch_size=args.batch_size, shuffle=False)
178
179     """
180     """
181     model = Model(N_BINS, args.model_divider)
182
183     if SAVED_MODEL_PATH is not None:
184         model.load_state_dict(torch.load(SAVED_MODEL_PATH))
185         print(SAVED_MODEL_PATH)
186         print('Model loaded')
187
188     criterion = nn.CrossEntropyLoss()
189     optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)
```

```
190     if use_gpu:
191         criterion = criterion.cuda()
192         model = model.cuda()
193
194     epochs = args.num_epochs
195     best_losses = 3
196
197     with log_file as output:
198         writer = csv.writer(output)
199         writer.writerow(['epoch', 'train_loss', 'val_loss'])
200
201     for epoch in range(args.from_epoch, epochs):
202         if use_gpu and epoch > args.from_epoch:
203             model.cuda()
204             # Train for one epoch, then validate
205             train_loss = train(train_loader, model, criterion, optimizer, epoch,
206             ↪ use_gpu)
207             with torch.no_grad():
208                 val_loss = validate(val_loader, model, criterion, epoch, use_gpu)
209
210             writer.writerow([epoch, train_loss, val_loss])
211             # Save checkpoint and replace old best model if current model is better
212             if val_loss < best_losses:
213                 best_losses = val_loss
214                 torch.save(model.to('cpu').state_dict(),
215                 ↪ '{}/model-{}-{}-{:3f}.pth'.format(CHECKPOINTS_PATH, N_BINS,
216                 ↪ epoch+1, val_loss))
```

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 03.01.2021

Adrian Saiz Ferri