



**Hochschule für Technik
und Wirtschaft Berlin**

University of Applied Sciences

Untersuchung von Image Colorization Methoden anhand
Convolutional Neuronal Networks

Abschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft Berlin

Fachbereich IV: Informatik, Kommunikation und Wirtschaft

Studiengang Angewandte Informatik

1. Prüfer: Prof. Dr. Christin Schmidt

2. Prüfer: M.Sc. Patrick Baumann

Eingereicht von: Adrian Saiz Ferri

Immatrikulationsnummer: s0554249

Eingereicht am: XX.XX.2020

Abstract

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Zielsetzung	1
1.3	Vorgehensweise und Aufbau der Arbeit	1
2	Grundlagen	2
2.1	Neuronale Netze	2
2.1.1	Feedforward Neural Network	3
2.1.2	Fully-connected Neural Network	3
2.1.3	Aktivierungsfunktionen	5
2.1.4	Convolutional Neural Networks	7
2.1.5	Loss Functions	10
2.1.6	Optimierungsalgorithmen	11
2.1.7	Backpropagation	13
2.1.8	Transposed Convolution	14
2.1.9	Autoencoder	14
2.2	<i>Lab</i> -Farbraum	15
2.3	Verwandte Arbeiten	16
3	Konzeption	17
3.1	Netzwerkarchitektur	17
3.2	Datensätze	17
3.3	Framework	17
4	Implementierung	18
5	Evaluation	19
5.1	Vergleich der Modelle	19

6 Fazit	20
6.1 Zusammenfassung	20
6.2 Kritischer Rückblick	20
6.3 Ausblick	20
Abbildungsverzeichnis	I
Tabellenverzeichnis	II
Source Code Content	III
Glossar	IV
Literaturverzeichnis	V
Onlinereferenzen	VI
Bildreferenzen	VII
Anhang A	VIII
Eigenständigkeitserklärung	IX

Kapitel 1

Einleitung

1.1 Motivation

TODO

1.2 Zielsetzung

TODO

1.3 Vorgehensweise und Aufbau der Arbeit

TODO

Kapitel 2

Grundlagen

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

2.1 Neuronale Netze

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

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

2.1.1 Feedforward Neural Network

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

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

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

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

2.1.2 Fully-connected Neural Network

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

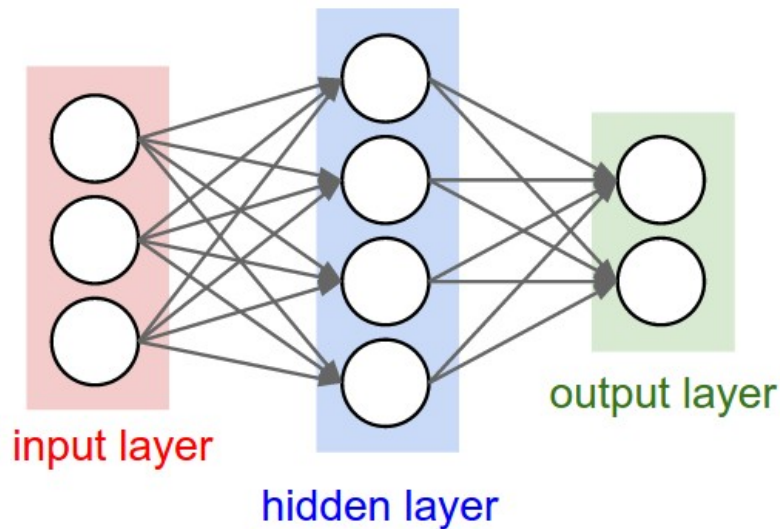


Abbildung 2.1: Fully-connected Neural Network mit 2 Layers (ein Hidden Layer mit 4 Neuronen) und ein Output Layer mit 2 Neuronen [Fei20a]

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

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

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

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

Forward Pass

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

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

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

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

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

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

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

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

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

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 verschiedene Nachteile, es neigt dazu den Gradienten zu verschwinden und die Outputs sind nicht Null zentriert. Sigmoid ist definiert als:

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

wobei x einem Input ist.

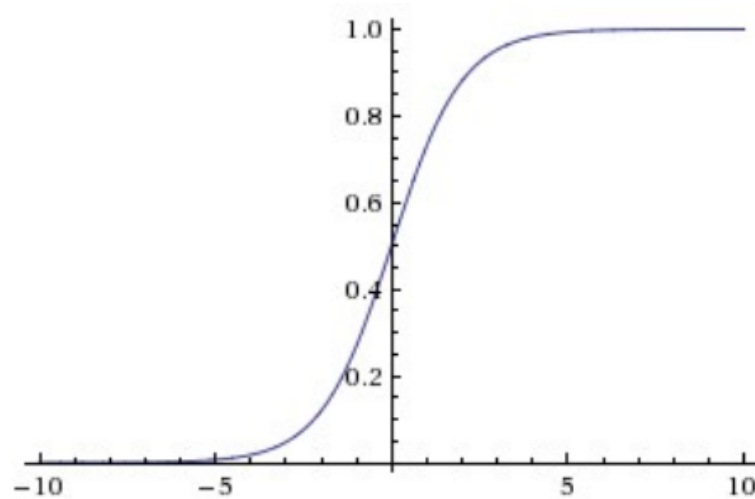


Abbildung 2.2: Sigmoid Aktivierungsfunktion [Fei20b]

ReLU

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

$$f(x) = \max(0, x) \quad (2.8)$$

wobei x einem Input ist.

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

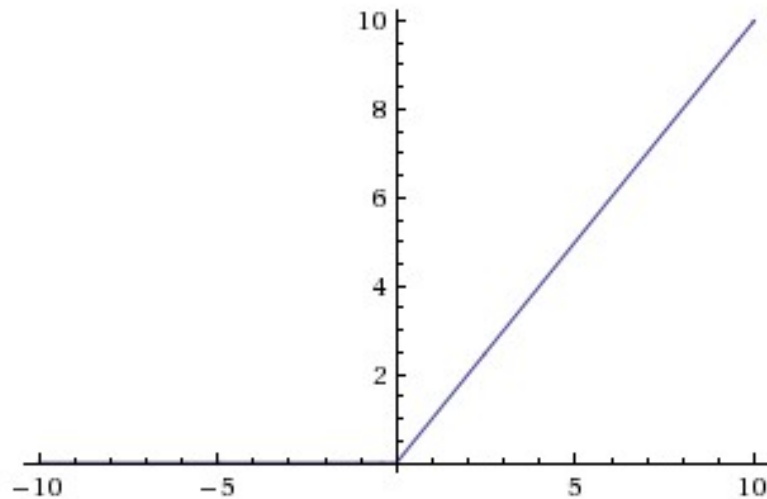


Abbildung 2.3: Rectified Linear Unit (ReLU) [Fei20b]

2.1.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN) sind eine besondere Form von künstliche neuronale Netzwerke, das speziell für die Verarbeitung von Bilddaten, unter anderen, vorgesehen ist [Dip19].

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

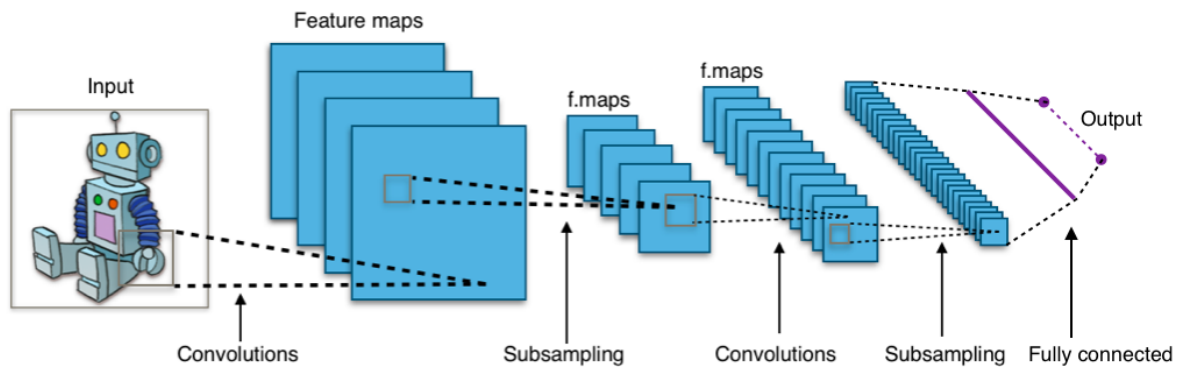


Abbildung 2.4: Typische Struktur von einem Convolutional Neural Network [Com15]

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

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

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

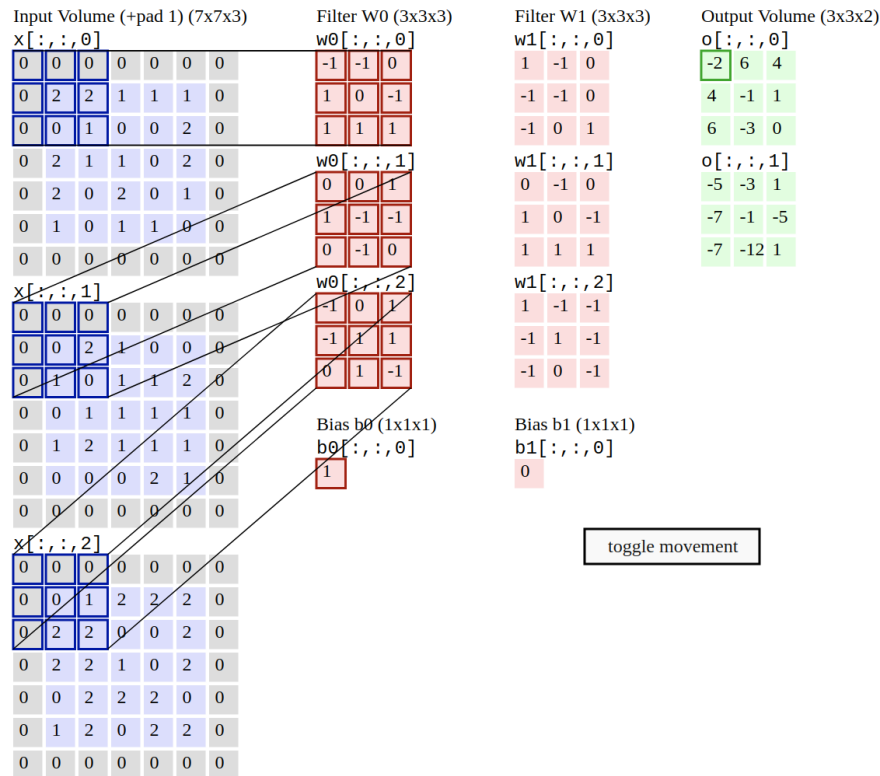
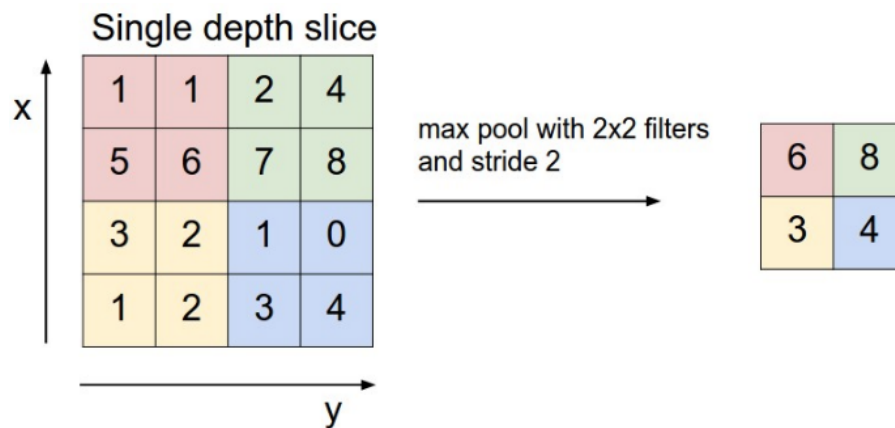


Abbildung 2.5: Beispiel Forward pass von einem Convolutional Layer mit einem $7 \times 7 \times 3$ Input Volumen, zwei $3 \times 3 \times 3$ Filter, Padding 1 und Stride 2. [Fei20c]

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

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

Abbildung 2.6: Max pooling Operation mit 2×2 Filtern und Stride 2 [Fei20c]

2.1.5 Loss Functions

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

Mean Square Error Loss

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

$$J = \frac{1}{N} \sum (y - \hat{y})^2 \quad (2.9)$$

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

Cross Entropy Loss

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

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

$$J = -\frac{1}{N} \left(\sum_{i=0}^N y_i * \log(\hat{y}_i) \right) \quad (2.10)$$

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

Weighted Cross Entropy Loss

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

2.1.6 Optimierungsalgorithmen

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

Gradient Descent

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

$$w_{x+1} = w_x - \alpha * \nabla J(w_x) \quad (2.11)$$

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

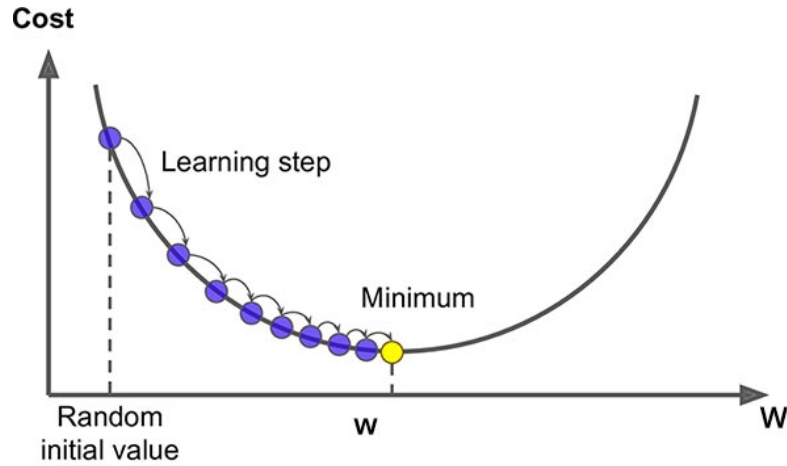


Abbildung 2.7: Gradient descent visualisiert [Bha18]

Es gibt verschiedene Arten von Gradient Descent, Mini-Batch Gradient Descent und Stochastisches Gradient Descent. Bei den normalen Gradient Descent werden die Gradienten im Bezug zu den gesamten Datensatz berechnet und damit das Update durchgeführt. Mini-Batch Gradient Descent berechnet die Gradienten im Bezug zu einen kleinen Teil von dem Datensatz und führt einen Update für alle Parameter durch. Letztendlich, Stochastisches Gradient Descent berechnet den Gradient bezogen auf einem einzigen Element von dem Datensatz und führt einen Update für alle Parameter durch.

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

$$\begin{aligned} v_{t+1} &= \rho v_t - \alpha * \nabla J(w_x) \\ w_{x+1} &= w_x + v_{t+1} \end{aligned} \tag{2.12}$$

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

Adam

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

2.1.7 Backpropagation

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

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

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} \quad (2.13)$$

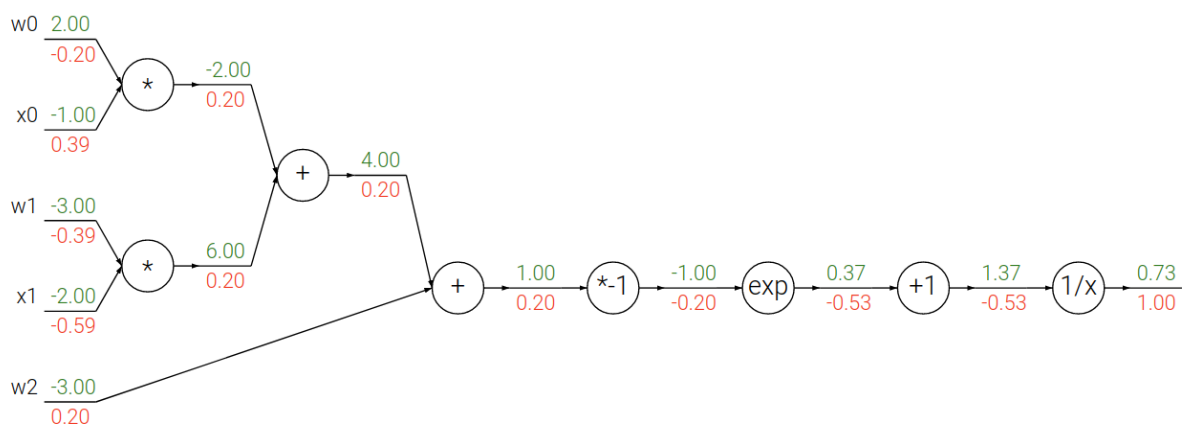


Abbildung 2.8: Backpropagation Beispiel anhand einer 2D Neuron mit der Aktivierungsfunktion Sigmoid [Fei20d]

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

2.1.8 Transposed Convolution

Im gegensatz zu einem Pooling Layer, ermöglicht einer Transposed Convolutional Layer die Dimensionen von einem Volumen zu vergrößern. Die funktionsweise einer Transposed Convolution wird anhand von einem Beispiel erklärt.

Ausgehend von einer 2×2 Input Matrix die auf 3×3 vergrößert werden soll, ein 2×2 Filter, Null Padding und Stride 1, ergibt sich den Folgenden Output.

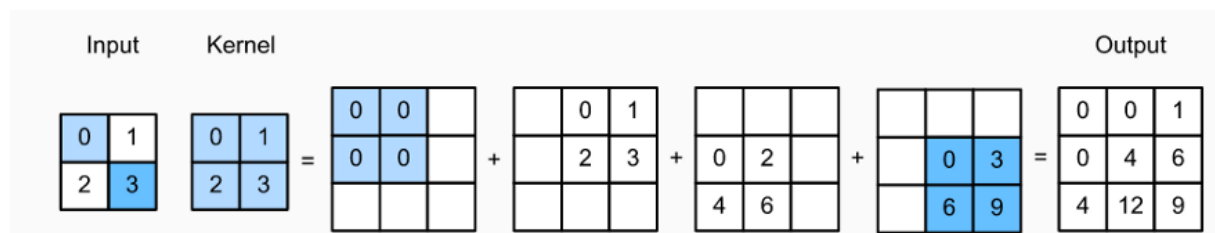


Abbildung 2.9: Die Komplette Transposed Convolution Operation [Zha20]

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

2.1.9 Autoencoder

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

Encoder

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

$$h = f(x) \quad (2.14)$$

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

Decoder

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

$$\hat{x} = g(h) \quad (2.15)$$

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

In der vorliegende Arbeit werden Autoencoders basierend auf Convolutional Neural Networks eingesetzt.

2.2 *Lab*-Farbraum

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

In der vorliegende Arbeit wird den *Lab*-Farbraum verwendet, da es unkompliziert ist, den “ L ” Kanal von beide Farbkanäle “ a ” und “ b ” zu trennen. Außerdem bildet den *Lab*-Farbraum das Menschliche Sehvermögen besser als den RGB-Farbraum ab.

2.3 Verwandte Arbeiten

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

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

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

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

Kapitel 3

Konzeption

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

3.1 Netzwerkarchitektur

TODO

3.2 Datensätze

TODO

3.3 Framework

Kapitel 4

Implementierung

TODO

Kapitel 5

Evaluation

5.1 Vergleich der Modelle

TODO

Kapitel 6

Fazit

TODO

6.1 Zusammenfassung

TODO

6.2 Kritischer Rückblick

TODO (Reflexion und Bewertung der Zielsetzung gegenüber erreichtem Ergebnis)

6.3 Ausblick

TODO

Abbildungsverzeichnis

2.1	Fully-connected Neural Network mit 2 Layers (ein Hidden Layer mit 4 Neuronen) und ein Output Layer mit 2 Neuronen [Fei20a]	4
2.2	Sigmoid Aktivierungsfunktion [Fei20b]	6
2.3	Rectified Linear Unit (ReLU) [Fei20b]	7
2.4	Typische Struktur von einem Convolutional Neural Network [Com15]	8
2.5	Beispiel Forward pass von einem Convolutional Layer mit einem $7 \times 7 \times 3$ Input Volumen, zwei $3 \times 3 \times 3$ Filter, Padding 1 und Stride 2. [Fei20c]	9
2.6	Max pooling Operation mit 2×2 Filtern und Stride 2 [Fei20c]	10
2.7	Gradient descent visualisiert [Bha18]	12
2.8	Backpropagation Beispiel anhand einer 2D Neuron mit der Aktivierungsfunktion Sigmoid [Fei20d]	13
2.9	Die Komplette Transposed Convolution Operation [Zha20]	14

Tabellenverzeichnis

Source Code Content

Glossar

Bin Behälter. 15

CIE Internationale Beleuchtungskommission. 15

CNN Convolutional Neural Network (Gefaltetes Neuronales Netzwerk). 7, 8, IV

Layer Schicht. 2, 8, 9, I

Loss Function Kostenfunktion. 10, 11, 13, 15

Stride Schrittweite einer Faltung bei einem CNN. 8, 9, I

Literaturverzeichnis

- [DHS11] John Duchi, Elad Hazan und Yoram Singer. „Adaptive subgradient methods for online learning and stochastic optimization“. In: *Journal of Machine Learning Research* 12.Jul (2011), S. 2121–2159.
- [GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [LLW04] Anat Levin, Dani Lischinski und Yair Weiss. „Colorization Using Optimization“. In: *ACM Trans. Graph.* 23.3 (Aug. 2004), S. 689–694. ISSN: 0730-0301. DOI: 10.1145/1015706.1015780. URL: <https://doi.org/10.1145/1015706.1015780>.
- [NH10] Vinod Nair und Geoffrey E. Hinton. „Rectified Linear Units Improve Restricted Boltzmann Machines“. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, 2010, S. 807–814. ISBN: 9781605589077. (Besucht am 27.07.2020).
- [Zha20] Aston Zhang u. a. *Dive into Deep Learning*. <https://d2l.ai>. 2020.

Onlinereferenzen

- [15] *Backpropagation*. 2015. URL: <http://www.inztitut.de/blog/glossar/backpropagation/> (besucht am 01.08.2020).
- [Bec19] Roland Becker. *Convolutional Neural Networks – Aufbau, Funktion und Anwendungsgebiete*. 2019. URL: <https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsgebiete-1691/> (besucht am 01.08.2020).
- [Dip19] Nico Litzel Dipl.-Ing. (FH) Stefan Luber. *Was ist ein Convolutional Neural Network?* 2019. URL: <https://www.bigdata-insider.de/was-ist-ein-convolutional-neural-network-a-801246/> (besucht am 01.08.2020).
- [Fei17a] Serena Yeung Fei-Fei Li Justin Johnson. *Neural Networks 1*. 2017. URL: <https://cs231n.github.io/neural-networks-1/> (besucht am 12.07.2020).
- [Fei17b] Serena Yeung Fei-Fei Li Justin Johnson. *Optimization 2*. 2017. URL: <https://cs231n.github.io/optimization-2/> (besucht am 01.08.2020).
- [Moe18] Julian Moeser. *Funktionsweise und Aufbau künstlicher neuronaler Netze*. 2018. URL: <https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/> (besucht am 10.07.2020).
- [Ngu20] Hoang Tu Nguyen. *Einführung in die Welt der Autoencoder*. 2020. URL: <https://data-science-blog.com/blog/2020/04/01/einfuehrung-in-die-welt-der-autoencoder/%7D> (besucht am 01.08.2020).

Bildreferenzen

- [Bha18] Saugat Bhattarai. *What is gradient descent in machine learning?* 2018. URL: <https://saugatbhattarai.com/np/what-is-gradient-descent-in-machine-learning/> (besucht am 01.08.2020).
- [Com15] Wikimedia Commons. *Typical CNN architecture*. 2015. URL: https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png (besucht am 03.04.2018).
- [Fei20a] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: <https://cs231n.github.io/neural-networks-1/> (besucht am 10.07.2020).
- [Fei20b] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: <https://cs231n.github.io/neural-networks-1/> (besucht am 12.07.2020).
- [Fei20c] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: <https://cs231n.github.io/convolutional-networks/> (besucht am 01.08.2020).
- [Fei20d] Serena Yeung Fei-Fei Li Justin Johnson. 2020. URL: <https://cs231n.github.io/optimization-2/> (besucht am 01.08.2020).

Anhang A

TODO

Eigenständigkeitserklärung

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

Berlin, den XX.XX.2018

Adrian Saiz Ferri