

# **Klassifizierung mit Convolutional Neural Networks**

- Bildverarbeitung mit Matlab und Python -

## **Praxisphasenbericht**

im Studiengang  
Bachelor Elektrotechnik

Matthias Korf  
Matr.-Nr.: 951922  
an der Hochschule Niederrhein

Praxisphasenbetreuer: Prof. Dr. Hirsch  
Krefeld, 1. Okt. - 18. Dez. 2016

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis .....</b>	<b>2</b>
<b>1     Einleitung .....</b>	<b>3</b>
<b>2   Convolutional Neural Networks .....</b>	<b>4</b>
2.1   Generelle Netzstruktur .....	4
2.2   Convolutional Layer .....	4
2.3   Activation Layer .....	7
2.4   Pooling Layer .....	8
2.5   Fully Connected Layer .....	10
2.6   Klassifizierung .....	11
<b>3     Backpropagation.....</b>	<b>13</b>
3.1   Berechnung der Deltas .....	13
3.2   Gradienten Update .....	14
3.3   Optimierer.....	15
<b>4     Optimierung von CNNs .....</b>	<b>18</b>
4.1   Initialisierung der Filtergewichte .....	18
4.2   Batch Normalisierung.....	18
4.3   L1/L2 Regularisierung.....	19
4.4   Dropout.....	19
<b>5     Zusammenfassung .....</b>	<b>20</b>
<b>Anhang.....</b>	<b>21</b>
<b>Literaturverzeichnis .....</b>	<b>24</b>

# 1 Einleitung

Neuronale Netze (NN) [1] haben seit dem Durchbruch in 2012 [2] im Bereich der Mustererkennung, Maschinellen Lernens, Künstlichen Intelligenz einen starken Forschungsboom erfahren. In der Computer Vision (CV) hat das Convolutional Neural Network (CNN) [3] in vielen Bereichen wie z.B. der Klassifizierung, Lokalisierung, Objekt Detektion, Segmentierung, OCR Erkennung und Image Captioning andere Modelle verdrängt und ist damit aktuell State of the Art [4] [5] [6] [7] [8]. In der Klassifizierung werden mittlerweile beeindruckende Erkennungsraten von über 97% [4] erreicht, was laut [9] Human bis Superhuman Level entspricht.

Auf Grund der hohen Komplexität einer hohen Anzahl von Netzwerkschichten werden Neuronale Netze häufig auch unter dem Begriff des Deep Learnings (DL) zusammengefasst. Abgrenzend dazu werden alle anderen Ansätze wie z.B. Optischer Fluss, Sift Features, Deformable Part Models etc. als sogenannte hand-crafted Feature Modelle bezeichnet [10].

Neben der Computer Vision finden CNNs erfolgreiche Verwendung in der Spracherkennung (ASR) [11] und beim Deep Reinforcement Learning [12]. In einer Kombination aus CNN und Q-Learning konnte ein künstlicher Agent in 2016 zum ersten Mal das Brettspiel Go gegen den weltbesten Go-Spieler gewinnen [13].

Im folgenden Bericht werden nun die Basiselemente und Grundstruktur eines Convolutional Neural Networks erklärt. Dazu gibt es praktische Beispiele zur Vertiefung des Verständnisses von CNNs.

Im Anhang befindet sich ein Code Auszug für ein einfaches 3 Layer CNN. Das vollständige Code-Beispiel kann in den Sprachen Matlab und Python von folgender Github Repository [14] heruntergeladen werden:

<https://github.com/roboball/DeepLearningToymodels>

## 2 Convolutional Neural Networks

### 2.1 Generelle Netzstruktur

Die allgemeine Netzstruktur besteht bei einem CNN aus einem Input Layer, einer beliebigen Anzahl von Hidden Layern und einem Output Layer an den sich ein Klassifizierer anschliesst [17] (siehe Abb. 1).

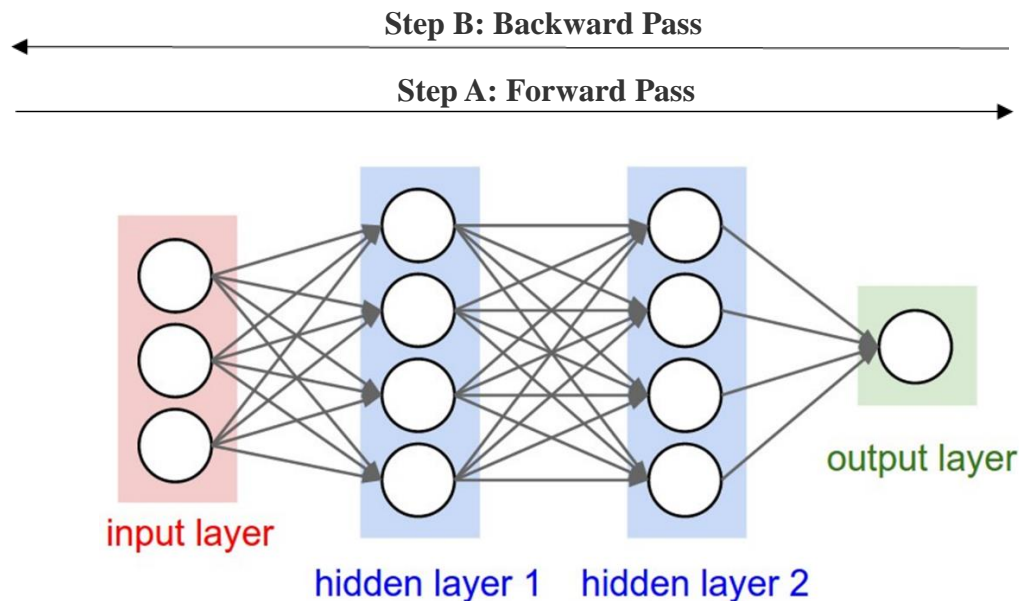


Abbildung 1: Generelle Netzstruktur

Im Fall der Bildverarbeitung stellen Bilder den Input dar. Als Bilder wählt man entweder eindimensionale Grauwert- bzw. Binärbilder oder dreidimensionale RGB, HSV etc. Farbraum Bilder. Auf Grund der Mehrdimensionalität bezeichnet man die Matrizen Arrays des Bildes auch als Inputvolumen bzw. nach einer Faltung mit Filtern als Outputvolumen. Im Standardfall ist das Inputvolumen 4 dimensionaler Tensor:

4 Dim Input Array (Höhe x Breite x Farbkanäle x Batchgröße)

Höhe und Breite bezieht sich auf die Bildgröße, ein Beispiel für Farbkanäle ist RGB und die Batchgröße bezieht sich auf die Anzahl von Bildern, die parallel vom Netz verarbeitet werden. Als Batchgröße wählt man in der Regel Minibatches der Größe 20 bis 100 Bilder.

### 2.2 Convolutional Layer

Convolution bedeutet zu deutsch Faltung. Die Faltung ist in der Praxis ein Filter z. B. Hochpass (Kantendetektion) oder Tiefpass (Blureffekt). Bei einem CNN Layer handelt es sich also um Filterbänke, die das Inputvolumen mit Filtergewichten zu einem Outputvolumen transformieren. In den vorderen Convolutional Layern erkennen die

Filter nach dem Training des CNNs meist Kanten, Ecken und Farben. In den nächsten Layern primitive Strukturen wie Kreise, Rechtecke kombiniert mit Farbtexturen. Schliesslich ergeben sich in den hinteren Layern aus den einfachen Strukturen komplexere Teilmuster wie z. B. Gesichter, Autoräder bis hin zu höheren komplexen Objekten wie z. B. Menschen oder Autos (siehe Abb. 2).

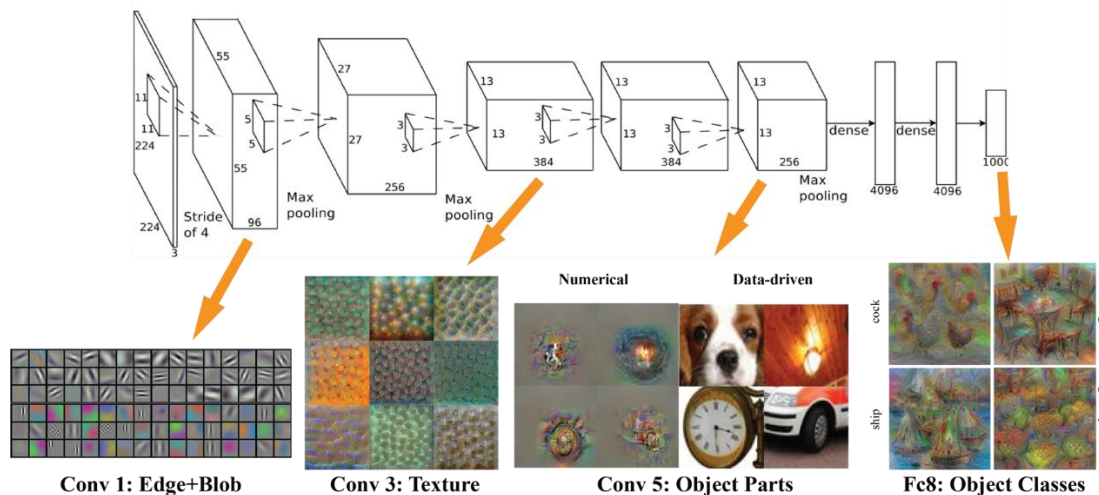


Abbildung 2: Filterbänke

In der Praxis gilt es zu beachten, dass die Deep Learning Community den Faltungsbegriff der klassischen Signalverarbeitung um verschiedene Hyperparameter erweitert hat. Diese Hyperparameter umfassen z.B. den horizontalen und vertikalen Stride, den Betrag an Zero Padding (bei Matlab die Option: valid, same, full) oder der Dilationfaktor für Dilated Convolutions [15]. Aus Grund der höheren Komplexität der einzelnen Funktion werden spezielle Deep Learning Libraries (z. B. Tensorflow [16] für Python, Matconvnet [15] für Matlab) zur praktischen Implementation eines CNNs empfohlen.

Weiterhin gilt es zu beachten, dass die meisten Deep Learning Libraries (z.B. Tensorflow) an Stelle der Faltung eine Kreuzkorrelation im Forward Pass und die Faltung im Backward Pass implementieren. Die Kreuzkorrelation ist auch bekannt als Template Matching Algorithmus. Das Template wird mit dem Inputbild auf Ähnlichkeit überprüft. Man kann den Forward Pass also vielleicht auch als eine Art stufenweises Multi Template Matching kombiniert mit einer Downsampling Pyramide interpretieren.

Im folgenden werden nun die Formeln und ein Beispiel für ein 2D Bild z. B. Grauwertbild beschrieben. Die Formeln für die Kreuzkorrelation (hier im Forward Pass) und Faltung (im Backward Pass) lauten wie folgt:

$$\text{Forward: } Y = \sum_{-M}^M \sum_{-N}^N \text{Input}(i, j) * W(x - i, y - j) \quad // \text{Kreuzkorrelation}$$

$$\text{Backward: } Y = \sum_{-M}^M \sum_{-N}^N \text{Input}(i, j) * W(x + i, y + j) \quad // \text{Faltung}$$

Man beachte, dass sich die beiden Formeln nur durch ein um 180 Grad in horizontaler und vertikaler Richtung gedrehtes Filter  $W$  unterscheiden.

Beispiel für ein 2D Bild:

Kreuzkorrelation Forward Pass:

1	2	3	4	3
2	3	1	4	2
3	5	1	2	0
4	1	2	3	2
1	3	2	5	0

 $\ast$ 

3	0	2
1	4	-1
1	-2	2

 $=$ 

21	23	20
31	20	13
7	41	14

Faltung Backward Pass:

In der Backproagation wird nun das Delta (oder auch Gradient, hier eine 3x3 Matrix) mit Nullen gepaddet, um wieder zu der ursprünglichen Größe (5x5) zu kommen. Zusätzlich wird das Filter W in horizontaler und vertikaler Richtung geflipped. Als Ergebnis erhält man die Faltung.

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	1	3	2	0	0
0	0	1	2	1	0	0
0	0	0	1	2	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

 $\ast$ 

2	-2	1
-1	4	1
2	0	3

 $=$ 

2	4	-1	-1	2
1	3	10	11	3
1	10	17	12	9
2	3	7	15	5
0	2	4	3	6

Zur Überprüfung können folgende Befehle in Matlab benutzt werden:

```
Y1 = conv2(I,rot90(W,2),'valid')+ b ; %Kreuzkorrelation (Forward Pass)
```

```
Y2 = conv2(Y1,W),'full'); %Faltung (Backward Pass)
```

Alternativ kann die Größe für Input und Output mit entsprechend angepasstem Zeropadding (Option: 'same') auch konstant bleiben:

```
Y1 = conv2(I,rot90(W,2),'same')+ b; %Kreuzkorrelation (Forward Pass)
```

```
Y2 = conv2(Y1,W),'same'); %Faltung (Backward Pass)
```

Das  $b$  im Forward Pass steht für das Bias. Der Bias ist eine Konstante und wird einfach als Offset zur Kreuzkorrelation hinzuaddiert.

## 2.3 Activation Layer

Aktivierungsfunktionen haben die Aufgabe das Outputvolumen des Convolutional Layers zu entlinearisieren. Das CNN Modell bekommt dadurch einen nicht linearen Charakter und ist im Vergleich zu linearen Modellen in der Lage komplexere Klassifizierungsfunktionen zu erlernen.

### 2.3.1 Sigmoid

Die sigmoide Funktion ist eine nichtlineare Aktivierungsfunktion mit Wertebereich zwischen 0 und 1. Die Formeln für den Forward und Backward Pass lauten:

$$\text{Forward: } \sigma = \frac{1}{1 + e^{-x}}$$

$$\text{Backward: } d\sigma = \sigma (1 - \sigma)$$

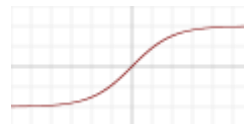


### 2.3.2 Tanh

Die tanh Funktion ist eine nichtlineare Aktivierungsfunktion mit Wertebereich zwischen -1 und 1. Die Formeln für den Forward und Backward Pass lauten:

$$\text{Forward: } \tanh(x)$$

$$\text{Backward: } d \tanh(x) = 1 - (\tanh(x))^2$$



### 2.3.3 Relu

ReLu steht für Rectifier Linear Unit. Die Formeln für den Forward und Backward Pass berechnet sich wie folgt:

$$\text{Forward: } \text{relu} = \max(x, 0) \text{ ; } x \text{ für } x > 0, \text{ sonst } 0$$

$$\text{Backward: } d \text{ relu} = 1 \text{ für } x(\text{input}) > 0, \text{ sonst } 0$$




Man beachte, dass für den Backward Pass der Input des Relu Funktion aus dem Forward Pass benötigt wird.

So auch im folgenden Beispiel:

Relu Forward Pass:


2	3	2
5	0	0
-7	4	-1



2	3	2
5	0	0
0	4	0

Relu Backward Pass:

2	0	5
-5	1	3
2	-3	-1



2	0	5
-5	0	0
2	-3	-1

## 2.4 Pooling Layer

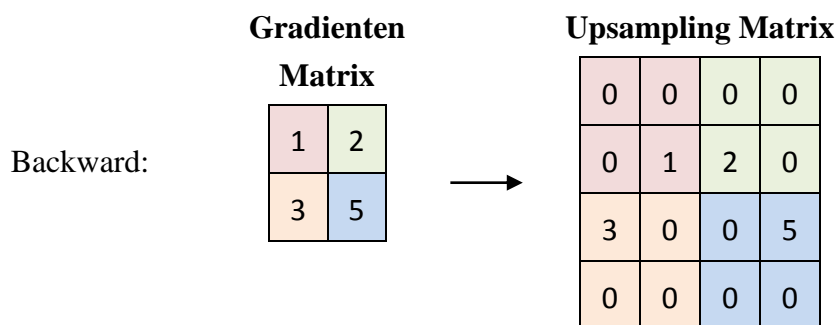
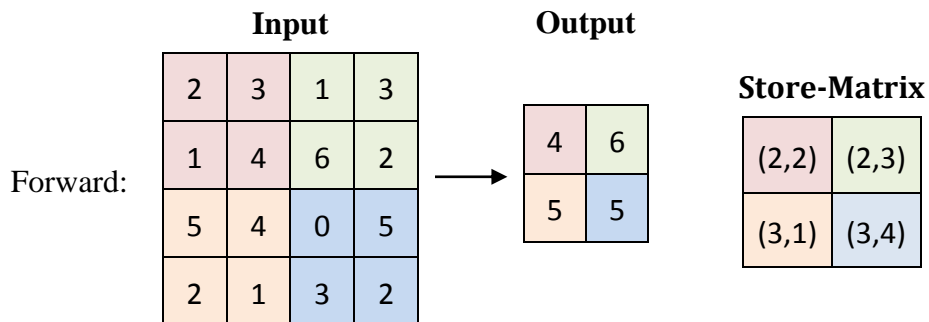
Pooling steht für Downsampling. Die Höhe und Breite des Outputvolumens aus dem Convolutional Layer wird verkleinert. Es kommt zu einer Informationsverdichtung. Möglich wird dies durch ein gewisses Maß an Skaleninvarianz des Netzwerks. Das Pooling hat den Vorteil den Rechenaufwand auf dem Computer zu verringern.

### 2.4.1 Max Pooling

Max Pooling extrahiert die Maxima in einem Sliding Window. Die restlichen Werte fallen weg. Das Sliding Window wird durch die zwei Hyperparameter horizontaler Stride (HS) und vertikaler Stride (VS) festgelegt. Der Stride bestimmt die Größe des Sliding Windows. Für einen 4x4 Input mit Stride (HS=2, VS=2) ergibt sich folglich ein 2x2 Output. Zu beachten gilt es zusätzlich, dass die Position des Maximums in einer weiteren Store-Matrix gespeichert werden muss, damit die Werte in die Upsampling Matrix im Backward Pass an die richtige Stelle zurückgeschrieben werden können.

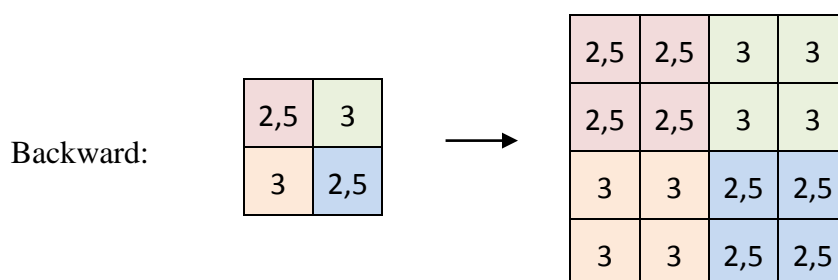
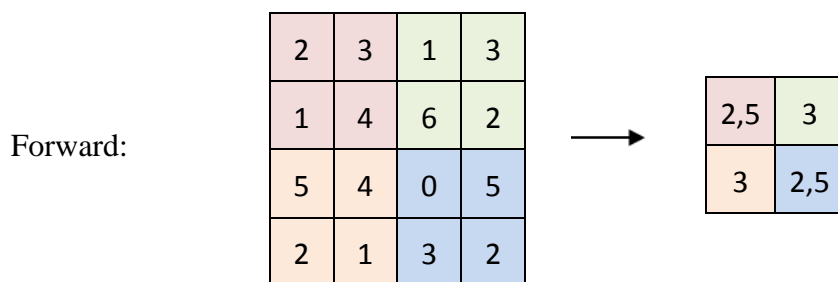


Beispiel: Max\_Pooling(Input 4x4, HS=2,VS=2)



### 2.4.2 Average Pooling

Beim Average Pooling nimmt man jeweils den Durchschnitt des Sliding Windows. In diesem Fall ist keine Store-Matrix notwendig.



## 2.5 Fully Connected Layer

Der Fully Connected (FC) Layer hat die Aufgabe das Outputvolumen der vorherigen Schicht in einen eindimensionalen Vektor zu transformieren und die Länge des Vektors auf die Anzahl der Klassen zu reduzieren. Der FC Layer entspricht der Netzwerkschicht eines klassischen NNs.

Forward:

Im ersten Schritt wird die Output Matrix aus dem letzten Pooling Layer in einen Vektor transformiert:

Output Matrix

2	3
1	4



Inputvektor

2	3	1	4
---	---	---	---

Danach erfolgt eine Vektor Matrix Multiplikation wie bei einem klassischen NN. Hier ein Beispiel für 3 Outputklassen:

Weights		Inputvektor		Output																			
<table> <tr><td>0,1</td><td>0,3</td><td>0,1</td><td>0,2</td></tr> <tr><td>0,1</td><td>0</td><td>0,2</td><td>0,1</td></tr> <tr><td>0,2</td><td>0,1</td><td>0,1</td><td>0,4</td></tr> </table>	0,1	0,3	0,1	0,2	0,1	0	0,2	0,1	0,2	0,1	0,1	0,4	*	<table> <tr><td>2</td></tr> <tr><td>3</td></tr> <tr><td>1</td></tr> <tr><td>4</td></tr> </table>	2	3	1	4	=	<table> <tr><td>2</td></tr> <tr><td>0,8</td></tr> <tr><td>2,4</td></tr> </table>	2	0,8	2,4
0,1	0,3	0,1	0,2																				
0,1	0	0,2	0,1																				
0,2	0,1	0,1	0,4																				
2																							
3																							
1																							
4																							
2																							
0,8																							
2,4																							

Backward:

Die Berechnung der Deltas erfolgt nach der Formel wie beim klassischen NN für ein Delta im Hidden Layer.

Output Delta		Weights		Hidden Delta																			
<table> <tr><td>1</td><td>2</td><td>3</td></tr> </table>	1	2	3	*	<table> <tr><td>0,1</td><td>0,3</td><td>0,1</td><td>0,2</td></tr> <tr><td>0,1</td><td>0</td><td>0,2</td><td>0,1</td></tr> <tr><td>0,2</td><td>0,1</td><td>0,1</td><td>0,4</td></tr> </table>	0,1	0,3	0,1	0,2	0,1	0	0,2	0,1	0,2	0,1	0,1	0,4	=	<table> <tr><td>0,9</td><td>0,6</td><td>0,8</td><td>1,6</td></tr> </table>	0,9	0,6	0,8	1,6
1	2	3																					
0,1	0,3	0,1	0,2																				
0,1	0	0,2	0,1																				
0,2	0,1	0,1	0,4																				
0,9	0,6	0,8	1,6																				

## 2.6 Klassifizierung

Die Klassifizierung hat die Aufgabe die einzelnen Elemente aus dem Outputvektor des FC Layers in diskrete Klassen einzuteilen [17]. Im kontinuierlichen Fall entspricht die Klassifizierung einer Regression (siehe 2.6.2). Zu jedem Klassifizierer muss man zusätzlich eine Verlustfunktion (Errorfunktion, Kostenfunktion) definieren, die den Trainingszustand des CNNs misst.

### 2.6.1 Crossentropy Loss (Softmax)

Softmax bezeichnet die Multinomiale Logistische Klassifikation. Zu beachten gilt es, dass man bei nur 2 Klassen den Spezialfall der Binären Logistischen Klassifikation erhält. Zweck der Softmax Klassifikation ist es die Werte Outputschicht in Wahrscheinlichkeiten zwischen 0 und 1 zu transformieren. Die Transformation hat den Vorteil, dass Wahrscheinlichkeiten relativ einfach interpretierbar sind und man den Zielvektor als One Hot Vektor darstellen kann. Das Ziel bekommt in diesem Fall eine 1 für 100 Prozent und alle anderen Werte bekommen eine 0 zugewiesen.

$$\text{Softmax Klassifizierer: } y(x_i) = \frac{e^{x_i}}{\sum_{n=1}^N e^{x_n}}$$

Die dazugehörige Errorfunktion wird Crossentropy Loss genannt. Die Funktion benötigt einen Targetvektor (t) enkodiert als One Hot Presentation und den Output aus dem Softmax Klassifizierer (y). Der Targetvektor (t) definiert sich aus den Labels der Trainingsdaten.

$$\text{Multiclass Crossentropy Loss: } L(y, t) = -\sum_{i=1}^N t_i \log(y_i)$$

$$\text{Delta Output: } \frac{d L(y, t)}{d y_i} = y_i - t_i$$

Beispiel für 3 Outputklassen (siehe: 2.5)

Input		Exponent		Output y		Targetvektor t		Delta Output
2		7,39		0,36		0		0,36
0,8	→	2,23	→	0,11	-	1	=	-0,89
2,4		11,02		0,53		0		0,53

Spezialfall bei nur 2 Klassen:

$$\text{Binäres Cross Entropy Loss: } L(y, t) = -\sum_{i=1}^N t_i \log(y_i) + (1 - t_i) \log(1 - y_i)$$

$$\text{Delta Output: } \frac{d L(y, t)}{d y_i} = y_i - t_i$$

### 2.6.2 Quadratisches Loss (Lineare Regression)

Die Quadratische Verlustfunktion (L2 Loss, Mean Squared Error) wird im Regelfall benutzt, falls das Model einen linearen kontinuierlichen Charakter aufweist. Die lineare Regression stellt zum Beispiel diesen Fall dar.

Forward:

$$\text{Quadratisches Loss: } L(y, t) = 0.5 * \sum_{i=1}^N (y_i - t_i)^2$$

Backward:

$$\text{Delta Output: } \frac{d L(y, t)}{d y_i} = y_i - t_i$$

### 2.6.3 Hinge Loss (Support Vektor Maschine)

Die Support Vektor Maschine ist ein weiterer Klassifizierer. Als Verlustfunktion wählt man das Hinge Loss. Im Unterschied zum Softmax Klassifizierer kann der Output nicht als Wahrscheinlichkeit interpretiert werden. Das Multiclass Hinge Loss für einen einzelnen Datenpunkt  $x_i$  mit Gewichten  $w$  und Hyperparameter  $\Delta$  (frei wählbar positive Konstante) ergibt sich wie folgt [9]:

Forward:

$$\text{Hinge Loss: } L_i = \sum_{j \neq y_i} [\max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)]$$

Backward:

$$\text{Delta Output: } \frac{d L_i}{d w_{y_i}} = -(\sum_{j \neq y_i} 1 (w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)) x_i$$

Für die anderen Zeilen  $j \neq y_i$  ergibt sich das Delta als:

$$\text{Delta Output: } \frac{d L_i}{d w_j} = 1 (w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

## 3 Backpropagation

Im vorherigen Abschnitt sind bereits die Formeln für den Backward Pass der Deltas vorgestellt worden. Diese Formeln beruhen auf dem Backpropagation Algorithmus [18] [17]. Der Algorithmus besteht aus zwei Schritten. Der erste Schritt hat zum Ziel den Error der Verlustfunktion rückwärts auf die einzelnen Netzknoten zu verteilen. Dies geschieht mit Hilfe der Deltas (siehe 3.1). Im zweiten Schritt erfolgt die Berechnung des Gradienten Updates für die Filtergewichte (siehe 3.2). Das eigentliche Update der Filtergewichte geschieht durch verschiedene Optimierungsformeln (siehe 3.3).

Vorgehensweise:

- (1) Berechne das Delta für jeden Netzknoten
- (2) Berechne das Gradienten Update für jeden Filter
- (3) Benutze einen Optimierer für das Filter Update.

### 3.1 Berechnung der Deltas

Die Berechnung der Deltas hat zum Ziel den Error auf die Output und Hidden Nodes zu verteilen. Deltas sind aus mathematischer Sicht Gradienten (wie auch in 3.2). Die Berechnung erfolgt mit Hilfe der Kettenregel. Als einfache Regel kann man sich merken: Zu jedem Netzknoten im Forward Pass gibt es ein Delta im Backward Pass, wobei die Deltas pro Schicht zu einer n-dimensionalen Operation (z.B. Faltung) zusammengefasst werden.

Man unterscheidet wie beim klassischen Neuralen Netz in Berechnung der Output Deltas und der Hidden Deltas. Die Output Deltas werden analog zum klassischen NN durch die transponierte Matrix Vektor Operation berechnet. Die Hidden Deltas berechnen sich falls im Forward Pass die Kreuzkorrelation verwendet wurde als Faltungen (ansonsten vice versa). Dies ist der wesentliche Unterschied zwischen CNN und klassischen NN.

Siehe als Beispiel Abbildung 1. Es gibt einen Outputlayer und zwei Hiddenlayer. Zuerst wird der Error vom Klassifizierer zum Outputlayer propagiert. Danach erfolgt eine transponierte Matrix-Vektor Multiplikation vom Outputlayer zum Hiddenlayer2 (siehe klassisches NN).

Für die Hiddenlayer gilt dann folgende Reihenfolge:

- (1) Umkehr der Pooling Operation
- (2) Umkehr der Aktivierungsoperation
- (3) Berechnung einer n-dimensionalen Faltung

Hier noch einmal die Formeln für den 2D Fall in Matlab Code aus dem vorherigen Abschnitt (siehe Abschnitt 2) zusammengefasst:

Berechnung Output Delta:

```
hidden_delta(n) = output_delta * weight(FC_Layer)';
```

```
hidden_delta_reshape(n) = reshape(hidden_delta(n) , x_Dim, y_Dim)';
```

Das Hidden Delta des FC Layers ergibt sich aus dem Output Delta und dem Gewicht des FC Layers transponiert. Danach erfolgt ein Reshape von Vektor in Matrix Form.

Berechnung Hidden Delta:

```
hidden_delta(l) = maxpool_reverse(hidden_delta_reshape(n), storepool , stride);
```

```
hidden_delta(k) = relu_reverse (hidden_delta(l), relu_forward_pass );
```

```
hidden_delta(j) = conv2(hidden_delta(k), weight , 'full');
```

Im Hidden Layer wird die Reshaped-Matrix zuerst durch die umgekehrte Pooling Operation upsampled. Danach erfolgt die Umkehrung der Aktivierungsfunktion (hier Relu). Im letzten Schritt erfolgt die Faltung des Hidden Deltas mit dem Gewicht des Layers.

## 3.2 Gradienten Update

Das Gradienten Update erfolgt nachdem der Error durch die Deltas auf die Netzwerkknoten backpropagiert wurde. Die Berechnung erfolgt für jeden Filter (Weight und Bias) nach folgenden Formeln im einfachen 2D Fall in Matlab:

Gradienten Update für den FC Layer:

```
gradient_weight(n) = output_layer_reshaped' * output_delta;
```

```
gradient_bias(n) = output_delta;
```

Für das Gradienten Update des Gewichts  $w$  wird das Output Delta mit dem transponierten Vektor aus dem Forward Pass des FC Layers multipliziert. Das Gradienten Update des Bias ist einfach das Output Delta.

Gradienten Update für einen Hidden Layer:

```
gradient_weight(j) = conv2(layer_pool, rot90(hidden_delta(k), 2), 'valid');
```

```
gradient_bias(j) = sum(sum(hidden_delta(k)));
```

Für das Gradienten Update des Gewichts ergibt sich die Kreuzkorrelation aus dem Outputvolumen der Pooling Operation aus dem Forward Pass und dem Hidden Delta aus der vorhergehenden Relu Reverse Operation. Im Fall Forward Pass 'same' muss auch im Gradienten Update die Option 'same' gesetzt werden. Für den Biasterm ergibt sich die Aufsummierung aller Gradienten im Array zu einem einzelnen Skalarwert.

### 3.3 Optimierer

Die Optimierer kommen zum Einsatz, um die Reduzierung des Errors zu beschleunigen [19] [17]. Der Gradient befindet sich in einem komplexen Errorgebirge und versucht den Weg in das Tal (lokales Minimum) zu finden. Wendet man das Gradienten Update ohne Optimierung an, kann es teilweise sehr lange dauern bis der Error auf einen minimalen Wert konvergiert. Andere Effekte sind, dass der Error sehr heftig oszilliert und teilweise gar nicht konvergiert, sondern gegen unendlich strebt (siehe Abb. 5).

Generell lässt sich sagen, dass kein Verfahren Konvergenz des Errors garantiert. Optimierungsstrategien bieten jedoch durch verschiedene Strategien wie z. B. mit Moving Average den Vorteil den Verlauf zu glätten und auf diese Weise zu einem besseren Konvergenzverhalten zu führen.

#### 3.3.1 Stochastic Gradient Descent und Momentum

Stochastic Gradient Descent (SGD) führt nach jeder Trainingsiteration ein Update der Gewichte  $w$  aus. Dazu wird das alte Gewicht  $w_t$  mit dem Produkt aus Lernrate  $\eta$  und Gradienten Update  $\nabla_w L(w_t, x_i, t_i)$  subtrahiert.  $x_i$  steht dabei für ein Training Sample,  $t_i$  für das Target und  $L$  für die Verlustfunktion.

Stochastic Gradient Descent ohne Momentum:

$$w_{t+1} = w_t - \eta \nabla_w L(w_t, x_i, t_i)$$

Bei SGD mit Momentum fließt zusätzlich das Momentum  $v_t$  mit dem Discountfaktor Gamma  $\gamma$  als Gewichtungsterm mit ein. Der zusätzliche Term hat eine Beschleunigung von SGD zur Folge. Der SGD Signalverlauf bekommt eine niedrigere Frequenz und die Oszillationen werden leicht gedämpft (siehe Abb.3).

Stochastic Gradient Descent mit Momentum:

$$v_t = \gamma v_{t-1} + \eta \nabla_w L(w_t, x_i, t_i)$$

$$w_{t+1} = w_t - v_t$$

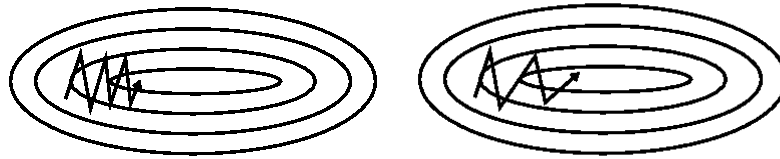


Abbildung 3: links SGD ohne Momentum vs. rechts SGD mit Momentum

### 3.3.2 Minibatch Gradient Descent

Minibatch Gradient Descent ist eine Variante von SGD. Im Unterschied zu SGD wird das Parameterupdate nur nach jedem Minibatch mit Größe  $n$  durchgeführt. Das Gradienten Update  $\nabla_w L(w_t, x_{i:i+n}, t_{i:i+n})$  ergibt sich aus dem Mittelwert des Gradienten für ein einzelnes Training Sample  $x_i$  (siehe Code Beispiel im Anhang). Vorteil ist meist eine Beschleunigung in der Rechenzeit, da sich Minibatch-Updates in der Regel sehr gut parallelisieren lassen (benötigt lediglich einen hohen RAM Speicher). Wie bei SGD kann optional auch ein Momentum Term eingefügt werden.

$$w_{t+1} = w_t - \eta \nabla_w L(w_t, x_{i:i+n}, t_{i:i+n})$$

### 3.3.3 RMSprop

RMSprop gehört zu den adaptiven Lernratenverfahren. Die Methode ist eng verwandt mit zwei weiteren adaptiven Lernratenverfahren: Adagrad und Adadelta.  $E|g_t^2|$  steht für die rekursive Aufsummierung von quadrierten Gradienten. Es ergibt sich ein Moving Average Filter bestehend aus dem quadrierten Gradient  $g_{t-1}^2$  zum Zeitpunkt t-1 und dem quadrierten Gradient  $g_t^2$  zum Zeitpunkt t. Dieser Ausdruck fließt dann in den Nennerterm für das Gradienten Update der Gewichte  $w_{t+1}$  mit ein.

$$E|g_t^2| = 0,9 E|g_{t-1}^2| + 0,1 g_t^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E|g_t^2| + \varepsilon}} g_t$$



### 3.3.4 Adam

Adaptive Moment Estimation (ADAM) ist eine weitere fortgeschrittene Optimierungsmethode, die für jeden einzelnen Parameter adaptive Lernraten enthält. Für das eigentliche Gewichtsupdate werden die zwei Momente  $m_t$  und  $v_t$  benötigt, die dem Momentum Term (siehe Abschnitt 3.3.1) sehr ähnlich sind.  $m_t$  steht für das erste Moment (arithmetischer Mittelwert) und  $v_t$  für das zweite Moment (unzentrierte Varianz) der Gradienten  $g_t$ .  $\beta_1$  und  $\beta_2$  sind zwei Parameter für die ein Wert von  $\beta_1 = 0,9$  und  $\beta_2 = 0,999$  empfohlen wird.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

In der Praxis hat sich herausgestellt, dass beide Momente zu stark gegen Null streben. Deswegen führt man einen Korrekturterm  $\hat{m}_t$  und  $\hat{v}_t$  ein, der einen verbesserten Schätzer der Momente liefert.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Beide Schätzer werden schließlich in die Update Regel für Adam eingesetzt. Für Epsilon wird ein Wert von  $\varepsilon = 10^{-8}$  empfohlen.

Adam Update-Regel:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t$$

## 4 Optimierung von CNNs

Die Abschnitte 1 bis 3 beschreiben bereits die Netzstruktur und Netzelemente eines vollständigen CNNs. In diesem Abschnitt sollen nun noch einige Tipps und Tricks zur weiteren Optimierung von CNNs beschrieben werden [20] [17].

### 4.1 Initialisierung der Filtergewichte

Die Initialisierung der Filtergewichte erfolgt in einem naiven Versuch durch gauss oder uniform verteilte Zufallszahlen. In Matlab geschieht dies z. B. mit dem Befehl `rand(n)` oder `randn(n)` (siehe dazu [14]). In der Praxis hat sich mittlerweile eine fortgeschrittenere Methode nach Xavier durchgesetzt, bei dem die Varianz der Verteilung vom Input und Output abhängen.  $n_{in}$  und  $n_{out}$  stehen für die Anzahl der Inputs und Outputs pro Layer. Auf diese Weise wird sichergestellt, dass die Varianz der Gewichte weder zu groß noch zu klein ist.

$$Var(W) = \frac{1}{n_{in}}$$

In der Praxis kann es vorkommen, dass die Formel oben zu restriktiv ist. Deswegen verwendet man meist folgende Formel zur Initialisierung der Gewichte:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

### 4.2 Batch Normalisierung

Batch Normalisierung kann potenziell helfen das Lernen und die Test Accuracy zu verbessern. Die Methode beruht auf der Annahme, dass unnormalisierte Daten im Netzwerk zu kleine bzw. zu große Werte annehmen können. Aus diesem Grund wird vor jeder Operation im Netzwerk das Minibatch normalisiert.

Es findet quasi in jedem Netzwerk Layer ein Pre-processing des Inputs statt. Der Algorithmus besteht aus einer Berechnung des Mittelwerts und Varianz für jedes Minibatch, mit deren Hilfe der Input normalisiert wird.

Siehe dazu folgende Abbildung:

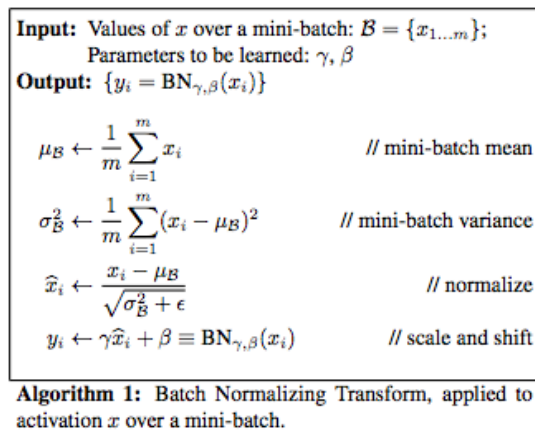


Abbildung 4: Batch Normalisierung

### 4.3 L1/L2 Regularisierung

L1/L2 Regularisierung ist eine sehr verbreitete Methode, um Overfitting zu verhindern. Mit Overfitting ist gemeint, dass das Netz an die Training Daten überangepasst ist und die Erkennungsraten im Trainings- bzw. Testdurchlauf wieder sinken (siehe dazu im Code Beispiel den Verlauf der Training Accuracy).

L1 und L2 bezeichnen zwei mathematische Normen. Die Normen werden als zusätzlicher Strafterm in die Zielfunktion eingefügt. Für L2 benutzt man den Term  $0,5 \lambda w^2$  und für L1 den Term  $\lambda |w|$ . In den Ausdrücken steht  $\lambda$  für die Regularisierungstärke und  $w$  für das Gewicht. Eine Kombination beider Ausdrücke ist auch möglich und wird dann Elastic Net Regularisierung genannt.

### 4.4 Dropout

Dropout ist eine Methode wie L1/L2 Regularisierung eine alternative Methode um Overfitting zu verhindern. Die Idee dahinter ist, dass man einer gewissen Wahrscheinlichkeit  $p$  verschiedene Neuronen im Netzwerk an und ausschaltet. Die Methode hat sich als sehr simple und effektiv herausgestellt.

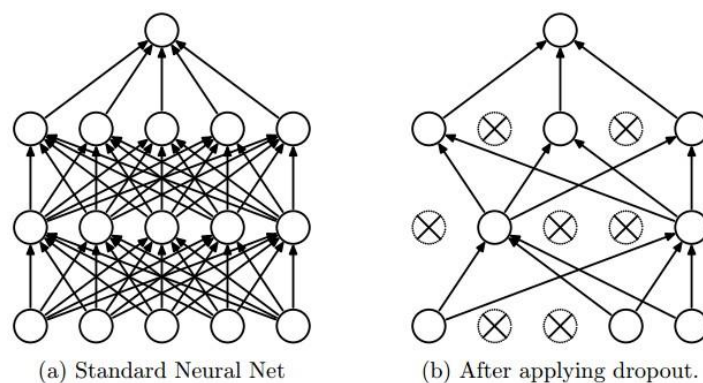


Abbildung 5: Dropout

## 5 Zusammenfassung

Der vorliegende Bericht befasst sich mit dem Thema Convolutional Neural Network. In den einzelnen Abschnitten werden die Elemente und Netzwerkstrukturen eines CNNs erläutert. Zusätzlich gibt es praktische Übungsbeispiele.

Im Abschnitt 1 erfolgt eine kurze Einführung zum aktuellen Stand des CNNs in der Computer Vision und benachbarten Disziplinen.

Im Abschnitt 2 werden die Basiselemente für einen Layer: Kreuzkorrelation, Aktivierungsfunktion und Pooling erläutert. Danach erfolgt eine Erklärung der meist verwendeten Klassifizierungsfunktionen. Im Standardfall nimmt man die Kreuzkorrelation, Relu und MaxPooling gefolgt von einem Softmax Klassifizierer.

Im Abschnitt 3 werden verschiedene Optimierer erklärt. Der Standardfall ist Minibatch Gradient Descent meist kombiniert mit Adam.

Im Abschnitt 4 werden zusätzliche Optimierungsmöglichkeiten dargestellt. In der Praxis haben die meisten Techniken bereits breite Verwendung gefunden. Im Code Beispiel [14] wurde aus Gründen der Vereinfachung bewusst auf diese Techniken verzichtet.

Im Anhang befindet sich zusätzlich ein Code Auszug. Der Auszug enthält die Standardfunktionen für den Forward und Backward Pass in einer Matlab Implementierung (siehe oben). Das Code Beispiel ist bewusst so konzipiert, dass man schnell auf einen Blick alle Funktionen und Variablen im Matlab Workspace ansehen kann. Alternativ kann auch auf eine Python-Tensorflow Implementation zurückgegriffen werden. Für den professionellen Einsatz wird die Kombination Python-Tensorflow empfohlen.

Den gesamten Code und weitere Infos befinden sich auf der Github Repo [14].

## Anhang

Code Auszug: Forward und Backward Pass

Netzwerk: 3 Layer CNN trainiert mit Minibatch SGD in Matlab.

Links: Volle Matlab und Python Implementation zu finden unter [14].

```
%*****

% FORWARD PASS

%*****

%input from minibatch:
x1 = double(reshape(minibatch(:,sample),[28,28]));
target = minibatchlabel(:,sample);

%1.Layer

l1_conv = conv2(x1,rot90(w1,2),'valid') + b1;
l1_relu = relu(l1_conv);
[ l1_pool , l1_storepool ] = maxpool(l1_relu, 2);

%2.Layer

l2_conv = conv2(l1_pool,rot90(w2,2),'valid') + b2;
l2_relu = relu(l2_conv);
[ l2_pool , l2_storepool ] = maxpool(l2_relu, 2);

%3.Layer: FC1

l3_reshape = reshape(l2_pool',1,4*4);
l3_fclayer = w3' * l3_reshape' + b3';

% Output Layer: softmax probs, cross-entropy loss and output deltas
[ softmaxprobs, ce_loss, deltaout ] = softmax(l3_fclayer, target);
```

```

%*****

% BACKWARD PASS

%*****

%*****

%hidden deltas:

%*****

%3.Layer:

l3_backfclayer = deltaout * w3';
l3_backreshape = reshape(l3_backfclayer ,4,4)';

%2.Layer:

l2_backpool = maxpoolup( l3_backreshape, l2_storepool , 2 );
l2_backrelu = reluup( l2_backpool, l2_relu );
l2_backconv = conv2(l2_backrelu,w2,'full');

%1.Layer:

l1_backpool = maxpoolup( l2_backconv,l1_storepool, 2 );
l1_backrelu = reluup( l1_backpool, l1_relu );
l1_backconv = conv2(l1_backrelu,w1,'full');

%*****

%gradients for update:

%*****

%weight gradients:

grad_weight3 = l3_reshape' * deltaout;
grad_weight2 = conv2(l1_pool,rot90(l2_backrelu,2),'valid');
grad_weight1 = conv2(x1,rot90(l1_backrelu,2),'valid');

%bias gradients:

grad_bias3 = deltaout;
grad_bias2 = sum(sum(1 * l2_backrelu));
grad_bias1 = sum(sum(1 * l1_backrelu));

```

```

%*****

%store: gradients and loss

%*****

%weight gradients:

gradw1(:,:,,sample) = grad_weight1;
gradw2(:,:,,sample) = grad_weight2;
gradw3(:,:,,sample) = grad_weight3;

%bias gradients:

gradb1(:,:,,sample) = grad_bias1;
gradb2(:,:,,sample) = grad_bias2;
gradb3(:,:,,sample) = grad_bias3;

%loss

batchloss(sample) = ce_loss;

end

%*****

%gradient update: Minibatch Gradient Descent:

%*****

%average the gradients:

avg_gradw1 = sum(gradw1,4)./batsize_train;
avg_gradw2 = sum(gradw2,4)./batsize_train;
avg_gradw3 = sum(gradw3,4)./batsize_train;
avg_gradb1 = sum(gradb1,4)./batsize_train;
avg_gradb2 = sum(gradb2,4)./batsize_train;
avg_gradb3 = sum(gradb3,4)./batsize_train;

%weight updates:

w1 = w1 - eta * avg_gradw1;
w2 = w2 - eta * avg_gradw2;
w3 = w3 - eta * avg_gradw3;
b1 = b1 - eta * avg_gradb1;
b2 = b2 - eta * avg_gradb2;
b3 = b3 - eta * avg_gradb3';

```

## Literaturverzeichnis

- [1] Ivakhnenko, Alekseï Grigor'evich, and Valentin Grigorévich Lapa. Cybernetic predicting devices. CCM Information Corporation, 1965.
- [2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.
- [3] LeCun, Yann, et al. "Backpropagation applied to handwritten zip code recognition." Neural computation 1.4 (1989): 541-551.
- [4] Russakovsky, Olga, et al. "Imagenet large scale visual recognition challenge." International Journal of Computer Vision 115.3 (2015): 211-252.
- [5] Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2014) 580–587.
- [6] Ren, S., He, K., Girshick, R., Sun, J.: Faster r-cnn: Towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems. (2015) 91–99.
- [7] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman. Deep structured output learning for unconstrained text recognition. In ICLR, 2015.
- [8] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In ICML, 2015.
- [9] Schmidhuber, Jürgen. "Deep learning in neural networks: An overview." Neural Networks 61 (2015): 85-117.
- [10] Lee, Chen-Yu, and Simon Osindero. "Recursive Recurrent Nets with Attention Modeling for OCR in the Wild." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.
- [11] Xiong, Wayne, et al. "Achieving human parity in conversational speech recognition." arXiv preprint arXiv:1610.05256 (2016).
- [12] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.
- [13] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." Nature 529.7587 (2016): 484-489.



- [14] <https://github.com/roboball/DeepLearningToymodels>
- [15] [http://www.vlfeat.org/matconvnet/mfiles/vl\\_nnconv](http://www.vlfeat.org/matconvnet/mfiles/vl_nnconv)
- [16] <https://www.tensorflow.org>
- [17] <http://www.deeplearningbook.org>
- [18] Kelley, Henry J. "Gradient theory of optimal flight paths." *Ars Journal* 30.10 (1960): 947- 954.
- [19] <http://sebastianruder.com/optimizing-gradient-descent/index.html>
- [20] <http://cs231n.github.io/neural-networks-2/#datapre>

### Abbildungsnachweis:

Abbildung 1: [cs231n.github.io/neural-networks-1/](http://cs231n.github.io/neural-networks-1/)

Abbildung 2: [http://vision03.csail.mit.edu/cnn\\_art/index.html](http://vision03.csail.mit.edu/cnn_art/index.html)

Abbildung 3: <http://sebastianruder.com/optimizing-gradient-descent/index.html>

Abbildung 4: <https://gab41.lab41.org/batch-normalization-what-the-hey-d480039a9e3b#.u2poiaukl>

Abbildung 5: <http://cs231n.github.io/neural-networks-2/#datapre>