



**HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG**  
UNIVERSITY OF APPLIED SCIENCES

# **Methoden des Deep Learning im Bereich Convolutional Neural Networks**

**Matthias Hermann**

Konstanz, 30.09.2015

**MASTERARBEIT**

# **MASTERARBEIT**

**zur Erlangung des akademischen Grades**

**Master of Science (M. Sc.)**

**an der**

**Hochschule Konstanz**

Technik, Wirtschaft und Gestaltung

**Fakultät Informatik**

Studiengang Master of Science, Informatik

**Thema:** **Methoden des Deep Learning im Bereich Convolutional Neural Networks**

**Masterkandidat:** Matthias Hermann, Zimmererweg 3, 78467 Konstanz

1. Prüfer: Prof. Dr. Matthias O. Franz  
2. Prüfer: Martin Schall, M. Sc.

Ausgabedatum: 01.04.2015  
Abgabedatum: 30.09.2015

## **Abstract**

Thema: Methoden des Deep Learning im Bereich Convolutional Neural Networks

Masterkandidat: Matthias Hermann

Firma: HTWG Konstanz

Betreuer: Prof. Dr. Matthias O. Franz  
Martin Schall, M. Sc.

Abgabedatum: 30.09.2015

Schlagworte: Machine Learning, Deep Learning, Convolutional Neural Network, Backpropagation, Gradient Descent, Regularization, MNIST, CIFAR10

## Ehrenwörtliche Erklärung

Hiermit erkläre ich *Matthias Hermann, geboren am 04.12.1988 in Wangen im Allgäu*, dass ich

- (1) meine Masterarbeit mit dem Titel

### **Methoden des Deep Learning im Bereich Convolutional Neural Networks**

bei der HTWG Konstanz unter Anleitung von Prof. Dr. Matthias O. Franz selbstständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

- (2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 30.09.2015

---

(Unterschrift)

## Danksagung

# Inhaltsverzeichnis

<b>Danksagung</b>	<b>IV</b>
<b>Inhaltsverzeichnis</b>	<b>VI</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Deep Learning . . . . .	2
1.2 Ziel der Arbeit . . . . .	4
1.3 Kapitelübersicht . . . . .	5
<b>2 Grundlagen Neuronaler Feedforward Netze</b>	<b>6</b>
2.1 Perzeptron und Multilayer Perceptron . . . . .	6
2.2 Aktivierungsfunktionen . . . . .	8
2.3 Fehlermaße . . . . .	11
2.3.1 Mittlerer Quadratischer Fehler . . . . .	11
2.3.2 Cross-Entropy . . . . .	12
2.4 Backpropagation-Algorithmus . . . . .	12
2.5 Basisarchitekturen . . . . .	14
2.5.1 Softmax-Regression (Klassifikation) . . . . .	14
2.5.2 Funktionsapproximation (Regression) . . . . .	16
2.6 Convolutional Neural Networks . . . . .	17
2.6.1 Modellbeschreibung . . . . .	17
2.6.2 Training mit Backpropagation . . . . .	21
<b>3 Spezielle Methoden im Deep Learning</b>	<b>24</b>
3.1 Vorverarbeitung . . . . .	26
3.1.1 Kontrastnormalisierung . . . . .	27
3.1.2 ZCA-Whitening . . . . .	27
3.2 Initialisierung . . . . .	28
3.2.1 Normalisierte Fehlerpropagierung . . . . .	28
3.2.2 Unüberwachtes Vortraining . . . . .	30
3.3 Gradientenabstieg . . . . .	33
3.3.1 Momentum . . . . .	35
3.3.2 Adaptive Lernrate . . . . .	36
3.3.3 Hessian free optimazation (HF) . . . . .	40

3.4	Regularisierung und Generalisierung . . . . .	40
3.4.1	A-priori Annahmen . . . . .	41
3.4.2	Modellkapazität . . . . .	43
3.4.3	Erweitern der Trainingsdaten . . . . .	46
3.5	Visualisierungsmethoden . . . . .	48
3.5.1	Primitiv . . . . .	48
3.5.2	Gradientenbasiert . . . . .	50
3.5.3	Nachverarbeitung . . . . .	52
	<b>Literaturverzeichnis</b>	<b>55</b>

# Kapitel 1

## Einleitung

Maschinelles Lernen stellt den Oberbegriff für Computerprogramme dar, welche hinsichtlich eines Performanzmaßes  $P$  (*Performance measure*) eine Aufgabe A mit zunehmender Erfahrung E besser lösen können. Erfüllt ein Programm diese Anforderung wird es als lernend bezeichnet (vgl. Mitchell, 1997, S. 2). Die verschiedenen Teilgebiete dieser Disziplin unterscheiden sich in der Struktur der Aufgabe A sowie des auf die Erfahrung beziehungsweise Trainingsdaten angewandten Lernverfahrens. Seit der Erfindung des Computers wurden so eine Vielzahl verschiedener Modelle entwickelt (vgl. z.B. Hastie et al., 2009). Ganz allgemein können diese Modelle in vier Gruppen aufgeteilt werden. Diese entsprechen den verschiedenen Anwendungsgebiete des maschinellen Lernens und sind in Tabelle 1.1 aufgeführt (siehe Franz, 2014):

Klassifikation	Regression
Dichteschätzung	Rangfolgebestimmung

Tabelle 1.1: Die verschiedenen Methoden im maschinellen Lernen

Ein weiteres integrales Merkmal eines Models im maschinellen Lernen ist das angewandte Lernverfahren, auch Lernparadigma genannt (vgl. im Folgenden Haykin (1998), S. 63 ff. und Becker (1991)).

- Überwachtes Lernen (*Supervised learning*): Die Trainingsdaten  $D_1 = \{(x_1, y_1), \dots, (x_n, y_n)\}$  liegen in Eingabe-Ausgabe-Beispielen vor, mit dem Ziel die Abbildung von Ein- nach Ausgabe beziehungsweise Trainingsbeispiel  $x_i$  und Label  $y_i$  zu lernen.
- Unüberwachtes Lernen (*Unsupervised learning*): Die Trainingsdaten  $D_2 = \{(x_1), \dots, (x_n)\}$  liegen ohne Labels vor. Das Modell lernt statistische Merkmale der Daten und somit eine interne Repräsentation.

- Bekräftigungs Lernen (*Reinforcement learning*): Hierbei wird ebenfalls eine Abbildung von Ein- nach Ausgabe gelernt. Allerdings erhält das System lediglich zeitlich verzögert Rückmeldung über die Performance der Abfolge von Ausgaben. Ziel ist es, hinsichtlich eines skalaren Performanzmaßes optimale Ausgaben zu erzeugen.

## 1.1 Deep Learning

Moderne Lernalgorithmen wie Support Vector Machines (SVMs) bieten viele interessante Eigenschaften, wie Schlußvariablen, maximale Trennbreite und eine konvexe Kostenfunktion. Analysen der letzten Jahre zeigen jedoch, dass solche modernen parameterlosen Lernalgorithmen fundamentalen Restriktionen unterliegen (vgl. im Folgenden Bengio and LeCun, 2007).

Zum einen ruhen diese auf einem dem *Fluch der Dimensionalität* sehr ähnlichen Problem (siehe Duda and Hart, 1973). Außerdem auf der Berechnung der lokalen Ähnlichkeit (Euklidischen Distanz) einer neuen Eingabe zu bereits bekannten Beispielen. Kern-basierte Verfahren mit lokalem Kern, wie dem Gauß-Kern, arbeiten unter der Annahme, dass die Zielfunktion glatt ist und verwandte Daten im Eingaberaum einer gewissen Ähnlichkeit unterliegen. Dies ist beispielsweise nicht mehr gegeben, wenn verwandte Daten in veränderter Form, wie affin transformiert, in unterschiedlichen Ausprägungen oder verstärkt, vorliegen. Nach Bengio and LeCun (2007) erfordert das Lernen von Funktionen mit großen Variationen im Eingaberaum ebenso viele Trainingsdaten, was in Verbindung zum *Fluch der Dimensionalität* steht. Des weiteren können Verfahren, welche auf lokaler Generalisierung beruhen, nicht zwischen relevanten und irrelevanten Merkmalen (z.B. Vorder- und Hintergrund) unterscheiden, was gerade in der Bilderkennung zu erheblichen Beschränkungen führt. Wünschenswert ist somit ein Verfahren, das nicht lokal generalisiert und mit steigender Anzahl Trainingsdaten (mehrere 10.000) gut skaliert.

Ein Lösungsansatz, um sehr komplexe Funktionen kompakt mit wenigen Parametern darzustellen, ist die Kombination vieler nicht-linearer Funktionen in Form einer Gleichung 1.1.

$$f(x) = f''(f'(x)) \quad (1.1)$$

Hierdurch kann  $f(x)$  stark von Trainingsbeispielen, welche weit entfernt von der Eingabe  $x$  selbst sind, abhängen. Dies ermöglicht nicht-lokale Merkmale und erlaubt es große Variationen im Eingaberaum abzubilden, was gerade im Bereich Bildverarbeitung, natürliche Sprache und Robotik von großer Relevanz ist. Durch die Wahl eines parametrisierten Modells kann das Modell darüber hinaus mit steigender Anzahl Trainingsdaten skalieren —Derartige Modelle heißen Künstliche Neuronale Netze (KNNs) (vgl. McCulloch and Pitts, 1943).

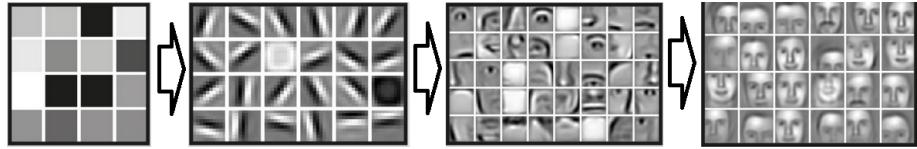


Abbildung 1.1: Merkmale unterschiedlichen Abstraktionsgrads von links (Pixel) bis rechts (Gesichter) (Einzelbilder: Andrew Ng, Stanford Artificial Intelligence Lab)

Angetrieben von der architektonischen Tiefe biologischer Gehirne, obgleich kein Beleg für eine derartige Funktionsweise dieser existiert (vgl. Becker, 1991), wurde so in der Vergangenheit oftmals versucht tiefe Netze (DNNs) überwacht zu trainieren (*Deep Learning*). Allerdings mit bedingtem Erfolg (vgl. Bengio and LeCun, 2007). Denn trotz der Ausdrucksstärke tiefer Modelle, stellt das Training solcher eine enorme Schwierigkeit dar (vgl. Becker, 1991). Oft genannte Gründe stehen in Zusammenhang mit zufälliger Initialisierung der Parameter und der Optimierung nicht-konvexer Zielfunktionen, welche durch viele Plateaus und lokale Minima charakterisiert sind (vgl. Bengio, 2009). Bis auf bemerkenswerte Erfolge sogenannter *Convolutional Neural Networks* (vgl. LeCun et al., 1989) sind somit tiefe Architekturen in der Vergangenheit weitestgehend unbeachtet geblieben.

### Renaissance durch Vortraining

Den Beginn des heutigen *Deep Learning* stellt die Arbeit von Hinton et al. (2006) dar: Ein unüberwachtes Lernverfahren, welches tiefe Architekturen schichtweise mittels eines *Greedy*-Algorithmus trainiert. Dadurch können, wie in Abbildung 1.1 dargestellt, unterschiedliche Schichten Merkmale mit unterschiedlichem Abstraktionsgrad extrahieren. Die Gesamtstrategie sieht dabei vor, zuerst Merkmale unüberwacht zu extrahieren und diese im Anschluss für ein etwaiges überwachtes Lernverfahren, wie beispielsweise Kernbasierte Verfahren (vgl. Hinton and Salakhutdinov, 2008), zu verwenden. Dies ähnelt stark den Ideen von Becker (1991), die Principle Component Analysis (PCA) zur Vorverarbeitung der Trainingsdaten zu verwenden. Durch die tiefe, schichtweise Extraktion der Merkmale sind diese jedoch nicht exklusiv, sondern verteilt, was auch als *distributed representation* bezeichnet wird (vgl. Hinton, 1986).

In den vergangenen Jahren konnten so bahnbrechende Fortschritte im Bereich der Spracherkennung (vgl Sainatha et al., 2015), der Verarbeitung natürlicher Sprache (NLP) (vgl. Socher et al., 2011) sowie der Bilderkennung erzielt werden. Den aktueller Benchmark in Letzterem stellt das *GoogLeNet* von Szegedy et al. (2014) mit 26% Top-5 Fehlern auf dem bekannten *Image-*

*Net*<sup>1</sup> Problem (vgl. Russakovsky et al., 2015). Von den neuen Möglichkeiten im *Deep Learning* profitieren auch die im letzten Kapitel erwähnten *Convolutional Neural Networks* insofern, dass bestehende Ideen weiterentwickelt oder fehlende gelabelte Trainingsdaten kompensiert werden (vgl. Le et al., 2012).

Heute werden unter *Deep Learning* meist folgende fünf Modelle unterschieden:

- *Recurrent Neural Network* (RNN) mit Long Short Term Memory (LSTM) (vgl. Hochreiter and Schmidhuber (1997))
- *Convolutional Neural Network* (CNN) (vgl. LeCun et al. (1998a), Krizhevsky et al. (2012) und Simonyan and Zisserman (2014))
- *Deep Belief Network* (DBNs) mit beschränkten Boltzmann-Maschinen (RBMs) (vgl. Bengio et al. (2007) und Ranzato et al. (2007a))
- *Stacked Denoising Autoencode* (SdA) (vgl. Vincent et al. (2008) und Vincent et al. (2010))
- *Deep Reinforcement Learning* (DRL) (vgl. Mnih et al. (2013))

## 1.2 Ziel der Arbeit

*Convolutional Neural Networks* (CNNs) sind eine besondere Klasse von Neuronalen Netzen. Diese von der Neurobiologie inspirierten Modelle in Verbindung mit neueren Methoden aus dem Bereich *Deep Learning*, stellen derzeit die besten Systeme für viele Probleme aus Bilderkennung, Spracherkennung und Verarbeitung natürlicher Sprache. Grundsätzlich werden drei Typen klassischer neuronaler Netze unterschieden: *Single-Layer Feedforward Networks*, *Multilayer Feedforward Networks* und *Recurrent Networks* (vgl. Haykin, 1998, S. 22f), wobei rekurrente Varianten in dieser Arbeit nicht weiter betrachtet werden.

Diese Arbeit beschränkt sich auf *Feedforward Networks* in den Bereichen Klassifikation und Regression und beschreibt somit das klassische, überwachte, neuronale Lernmodell. Für den Entwurf dieser neuronalen Netze existieren bereits eine Vielzahl an Softwaretools. Als Beispiel werden hier folgende bekannte Softwareprodukte angeführt:

- Theano (Python) (vgl. Bergstra et al., 2010)
- Torch (Lua) (vgl. Collobert et al., 2011)

---

<sup>1</sup>Der ImageNet-Datensatz besteht aus 200 Klassen, rund 450.000 Trainingsbeispielen, rund 20000 Validierungsgbeispielen und rund 40.000 Testbeispielen. Jedes Bild hat eine Größe von  $482 \times 415$  (<http://www.image-net.org/> (26.08.2015)).

- Cuda-Convnet (Cuda) (vgl. Nouri, 2013)
- PyLean2 (Python) (vgl. Goodfellow et al., 2013a)
- Caffe (C/C++) (vgl. Jia et al., 2014)

Ziel dieser Arbeit ist es zum einen die Methoden im Bereich *Deep Learning* aufzubereiten und zu analysieren. Darauf aufbauend wird ein eigener funktionierender Prototyp eines *Convolutional Neural Networks* entwickelt. Dieser soll möglichst effizient sein, wenige Abhängigkeiten haben sowie eine Python-Schnittstelle bereitstellen. Im Zentrum der Untersuchungen stehen dabei überwachte Lernverfahren mit dem Ziel, Bilddaten mit *State of the Art*-Performance zu klassifizieren, was allgemein als Bilderkennung bezeichnet wird. Darüber hinaus werden Methoden zum unüberwachten Vortraining und zur Visualisierung neuronaler Netze untersucht und vorgestellt.

Diese Arbeit entsteht am Institut für Optische Systeme (IOS) der HT-WG Konstanz und dient zusammen mit zwei weiteren Arbeiten im Bereich *Stacked Denoising Autoencoder* und *LSTM-Recurrent Neural Network* als Grundlage für weitere Aktivitäten im Bereich *Deep Learning* im IOS.

### 1.3 Kapitelübersicht

Diese Thesis ist unterteilt in sechs Kapitel. Kapitel 1 beginnt mit der thematischen Einordnung dieser Arbeit sowie der Motivation. Der Hauptteil ist in zwei Blöcke unterteilt. In Kapitel 2 und 3 werden notwendige Grundlagen für die Implementierung des Prototyps vorgestellt. So führt Kapitel 2 die Methode Neuronale Netze ausgehend vom Perzeptron ein und beschreibt im Speziellen die Besonderheiten der *Convolutional Neural Networks* (CNNs). In Kapitel 3 werden spezielle Methoden des *Deep Learning* vorgestellt. Der zweite Block besteht aus Kapitel 4 und 5. Kapitel 4 beschreibt die Entwicklung des Prototyps und Kapitel 5 die verschiedenen durchgeföhrten Experimente mit dem Ziel den Prototyp auf Richtigkeit zu überprüfen. Im abschließenden 6. Kapitel werden die relevanten Ergebnisse der Arbeit nochmals zusammengefasst und ein Ausblick auf über diese Arbeit hinausgehende Aspekte gegeben.

## Kapitel 2

# Grundlagen Neuronaler Feedforward Netze

Neuronale Netze sind äußerst vielseitig. Inspiriert von Gehirnen von Lebewesen, die einem komplexen, nichtlinearen, parallelen Computer ähneln, finden sie heute Anwendung in der Modellierung, Zeitreihenanalyse, Mustererkennung und Signalverarbeitung. Eine sehr wichtige Eigenschaft ist hierbei die Möglichkeit, von Eingabedaten beziehungsweise Trainingsdaten überwacht, unüberwacht und bekräftigend zu lernen (vgl. Haykin, 1998, S. 1). Dies erlaubt einen universellen Einsatz in allen Teilgebieten des maschinellen Lernens (vgl. Kapitel 1). Grundsätzlich werden drei Typen neuronaler Netze unterschieden: *Single-Layer Feedforward Networks*, *Multilayer Feedforward Networks* und *Recurrent Networks* (vgl. Haykin, 1998, S. 22f). Diese Arbeit beschränkt sich jedoch auf nicht rekurrente *Multilayer Feedforward Networks* in den Bereichen Klassifikation und Regression und beschreibt somit das klassische überwachte, neuronale Lernmodell (siehe Kapitel 1).

### 2.1 Perzeptron und Multilayer Perceptron

Heutige Neuronale Netze basieren auf dem in den späten 1950er von Rosenblatt (1962) entwickelten Perzeptron. Wie Abbildung 2.1 zeigt, ist die Architektur des Perzeptron an biologische Neuronen angelehnt. Die Ausgabe wird berechnet indem die Summe über die Produkte zwischen der Eingabe  $x_i$  und den Gewichten  $w_i$  gebildet wird. Erreicht die Summe einen definierten Schwellwert (*bias*)  $-b$ , wird das Neuron aktiviert und gibt die binäre Eins aus, ansonsten die binäre Null. Formal berechnet das Perzeptron somit die in Gleichung 2.1 aufgeführte Funktion.

$$f(x) = \text{sign}\left(\sum_i^n w_i x_i + b\right) = \text{sign}(\langle w, x \rangle + b) \quad (2.1)$$

Obwohl das Perzeptron sich in dieser Form bereits als linearer Klassifikator

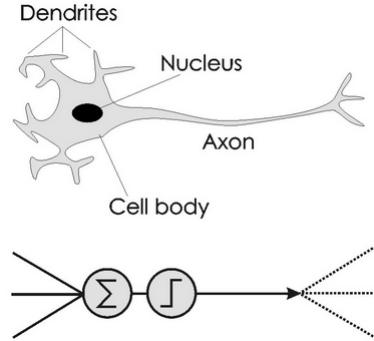


Abbildung 2.1: Schematischer Vergleich von biologischen und künstlichen Neuronen von McCulloch and Pitts (1943) (nach Winston (1992), S. 444ff.))

eignet, ist es in seiner Ausdrucksstärke sehr eingeschränkt. So ist es beispielsweise nicht möglich Entscheidungshierarchien, wie in Netzwerken von McCulloch and Pitts (1943), zu bilden und somit das bekannte XOR-Problem zu lösen.

Die Erweiterung zum Perzeptron stellen die in den 1980er Jahren von Rumelhart et al. (1986a) entwickelten *Multilayer Perceptrons* (MLP) dar. Die Architektur eines MLP wird bestimmt durch die Anzahl an Schichten (*Layers*) sowie der pro Schicht enthaltenen Neuronen. Abbildung 2.2 zeigt die generische Organisation mehrerer Neuronen in der Eingabeschicht (Input-Layer), der Verdeckten Schicht (Hidden-Layer) und der Ausgabeschicht (Output-Layer). Neuronen verallgemeinern hierbei das Perzeptron insofern, dass diese die Schwellwert-Aktivierungsfunktion durch eine beliebige Funktion  $\phi(\cdot)$  ersetzen können. Somit ermöglichen diese auf die reellen Zahlen  $\mathbb{R}$  abzubilden. Darüber hinaus kann durch die Darstellung von Entscheidungshierarchien auch das XOR-Problem gelöst werden (vgl. Rojas, 1996, S. 125).

Zusammenfassend lässt sich sagen, dass durch die richtige Wahl der Architektur und Aktivierungsfunktion mit MLPs eine beliebige Funktion  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  beliebig genau approximiert werden kann, was durch das *Universal Approximation Theorem* beschrieben ist. Die Schwierigkeit besteht jedoch darin, dass die zu approximierende Funktion lediglich durch Trainingsbeispiele gegeben ist und die Gewichte und Schwellwerte durch Training so angepasst werden müssen, dass neue Werte möglichst gut extrapoliert werden (Generalisierung) (vgl. Rojas, 1996, S. 24ff.).

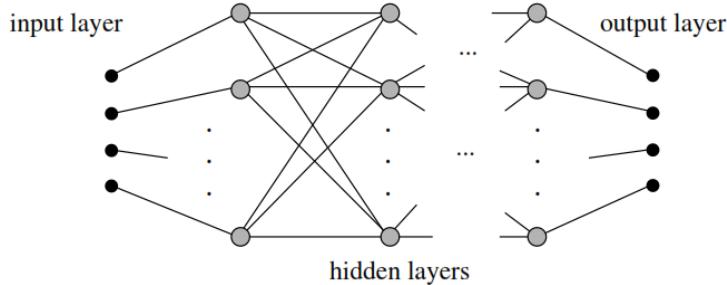


Abbildung 2.2: Generische mehrschichtige Architektur eines MLP (siehe Rojas (1996), S. 126)

Ein einfaches Beispiel soll an dieser Stelle die Verbindung zwischen MLP und *Deep Learning* herstellen. Die einfachste Architektur eines MLP umfasst die Eingabe  $x$  (Eingabeneuronen), ein *Hidden*-Neuron  $w_2$  sowie ein Ausgabeneuron  $w_1$ . Formal wird diese Architektur durch die Funktion in Gleichung 2.2 beschrieben, welche in unmittelbarer Verbindung zu den in Kapitel 1.1 beschriebenen tiefen Architekturen steht.

$$f(x) = \phi(\langle w_1, \phi(\langle w_2, x \rangle + b_2) \rangle + b_1) \quad (2.2)$$

Analog dazu fusionieren die Gewichtsvektoren  $w_i$  bei mehrere Neuronen in einer Schicht zu einer Matrix  $W$  in der die einzelnen  $w_i$  der Neuronen zeilenweise organisiert sind. Das innere Produkt in der Gleichung 2.2 wird so durch eine Matrix-Vektor-Multiplikation ersetzt und der skalare Schwellwert  $b_i$  mit einem Vektor  $b$ . Eine Schicht eines MLP kann folglich mit  $\phi(Wx + b)$  formal dargestellt werden.

## 2.2 Aktivierungsfunktionen

Im letzten Kapitel wurde vorgestellt wie sich das Perzeptron zum Neuron und insgesamt zu einem *Multilayer perceptron* (MLP) verallgemeinern lässt. In diesem Kapitel werden nun die wichtigsten Aktivierungsfunktionen für neuronale Netze vorgestellt (vgl. Hagan et al., 2014, S. 2-17).

### Linear

Die einfachste Funktion ist die Identitätsfunktion. Diese ist in Abbildung 2.3 dargestellt.

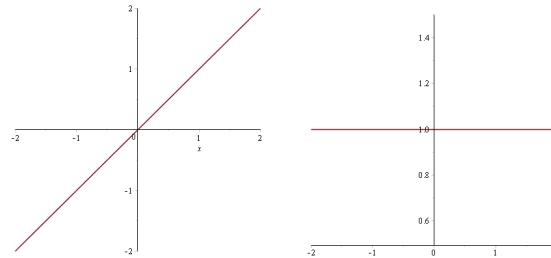


Abbildung 2.3: Lineare Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$f(x) = x \quad (2.3)$$

- Ableitung:

$$f'(x) = 1 \quad (2.4)$$

### Sigmoid

Die Sigmoid-Funktion ist eine nichtlineare Funktion, welche  $\mathbb{R} \rightarrow (0, 1)$  abbildet (siehe Abbildung 2.4). Besonderheit der Sigmoidfunktion ist, dass die Ableitung an einer bestimmten Stelle durch die Funktion selbst berechnet werden kann.

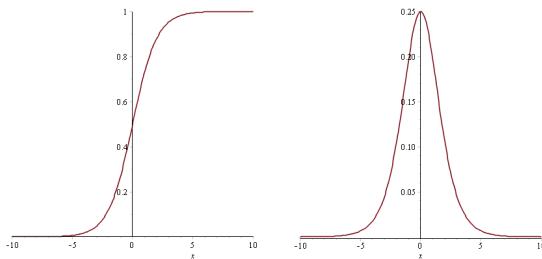


Abbildung 2.4: Sigmoid Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$\text{sig}(x) = \frac{1}{1 + \exp^{-t}} \quad (2.5)$$

- Ableitung:

$$\text{sig}'(x) = \text{sig}(x)(1 - \text{sig}(x)) \quad (2.6)$$

### Tangens Hyperbolicus

Die Tangens Hyperbolicus-Funktion ist eine nichtlineare Funktion, welche  $\mathbb{R} \rightarrow (-1, 1)$  abbildet (siehe Abbildung 2.5). Wie bei der Sigmoidfunktion, kann die Ableitung an einer bestimmten Stelle ebenfalls durch die Funktion selbst berechnet werden.

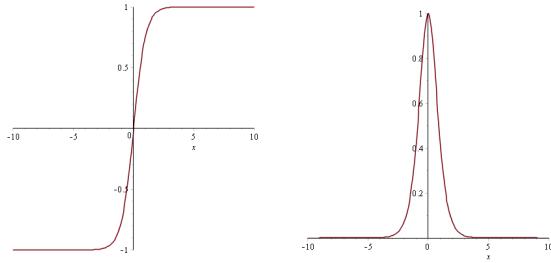


Abbildung 2.5: Tangens Hyperbolicus (tanh) Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$\tanh(x) = \frac{\exp^x - \exp^{-x}}{\exp^x + \exp^{-x}} \quad (2.7)$$

- Ableitung:

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

### Rectified Linear

Die *Rectified Linear*-Funktion (ReLU) ist eine von Glorot et al. (2011) eingeführte Aktivierungsfunktion mit Eigenschaften, die im Bereich der Neuronalen Netze von Interesse sind. So ist bei dieser Aktivierungsfunktion für alle Werte, welche eine von 0 verschiedene Ausgabe erzeugen auch die Ableitung von 0 verschieden. Außerdem können keine negativen Werte auftreten. Die Funktion bildet  $\mathbb{R} \rightarrow [0, \infty)$  ab und ist in Abbildung 2.6 dargestellt.

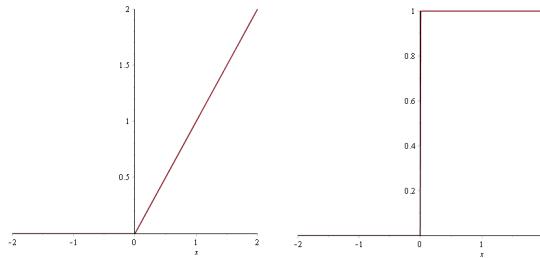


Abbildung 2.6: Rectified Linear (ReLU) Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

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

- Ableitung:

$$f'(x) = \begin{cases} 0 & \text{für } 0 \geq x \\ 1 & \text{für } 0 < x \end{cases} \quad (2.10)$$

## Softplus

Die *Softplus*-Funktion ist eine glatte Variante der ReLu-Funktion, welche  $\mathbb{R} \rightarrow (0, \infty)$  abbildet. Die *Softplus*-Funktion ist in Abbildung 2.7 dargestellt.

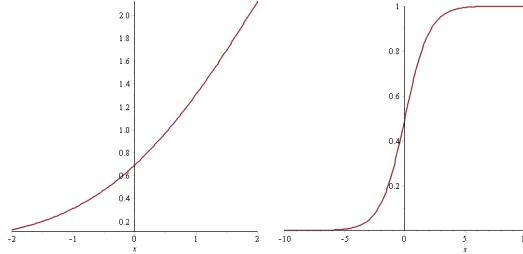


Abbildung 2.7: Softplus Aktivierungsfunktion (links) und Ableitung (rechts)

- Funktion:

$$f(x) = \ln(1 + e^x) \quad (2.11)$$

- Ableitung:

$$f'(x) = \frac{1}{1 + e^{-x}} \quad (2.12)$$

## 2.3 Fehlermaß

Feedforward-Netze basieren auf dem sogenannten *error-correction learning*. Das bedeutet, dass das Ausgabesignal  $y_k$  eines Neuron  $k$  (*Output*) des Netzwerks zu einem Eingangssignal  $x$  mit einem Zielsignal  $d_k$  (*Target*) verglichen und so das Fehlersignal  $e_k = d_k - y_k$  erzeugt wird (vgl. Haykin, 1998, S. 51f). Um das Fehlersignal in MLPs zu messen, wird ein entsprechendes Fehlermaß benötigt. Dabei werden meist der Mittlere Quadratische Fehler (*Mean Squared Error*) für Regression- und die negative Log-Likelihood (*Cross-Entropy*) für Klassifikationsaufgaben verwendet (vgl. Golik et al., 2013).

### 2.3.1 Mittlerer Quadratischer Fehler

Der *Mean Squared Error* (MSE) ist für ein gegebenes MLP  $f(x)$  und gegebene Trainingsdaten  $D = \{(x_1, y_1), \dots (x_{n,n})\}$  wie in Gleichung 2.13 definiert. Eine Besonderheit stellt der zusätzliche Faktor  $\frac{1}{2}$  dar, welcher letztlich aber nur die Ableitung vereinfacht.

$$MSE = \frac{1}{2N} \sum_i^N (f(x_i) - y_i)^2 \quad (2.13)$$

### 2.3.2 Cross-Entropy

Das *Cross-Entropy*-Fehlermaß (CE) ist in Gleichung 2.14 dargestellt. Es beschreibt die negative maximale Log-Likelihood über den Trainingsdaten  $D = \{(x_1, y_1), \dots (x_{n,n})\}$  (vgl. Mitchell, 1997, S. 118). Als Vorgriff auf Kapitel 3.3 wird hier bereits CE über die Trainingsdaten gemittelt angegeben.

$$CE = -\frac{1}{N} \sum_i^N (y_i \ln(f(x_i)) + (1 - y_i) \ln(1 - f(x_i))) \quad (2.14)$$

Durch die  $\ln(\cdot)$ -Funktion beachtet CE im Gegensatz zu MSE wie nah das Ausgabesignal am Zielsignal ist, vernachlässigt allerdings im Gegenzug fehlerbehaftete Ausgaben. Dieser Zusammenhang trägt dazu bei, dass CE für Klassifikationsaufgaben besser geeignet erscheint.

## 2.4 Backpropagation-Algorithmus

In den vergangenen Kapiteln wurden, ausgehend vom Perzeptron, die nicht-linearen MLPs eingeführt und verschiedene Aktivierungsfunktionen vorgestellt. Außerdem wurde gezeigt, dass der Fehler eines Neuronalen Netzes mittels zwei verschiedenen Fehlermaßen angegeben werden kann. Wie im vergangenen Kapitel festgestellt wurde, wird bei Neuronalen Netzen ein sogenanntes *error-correction learning* angewandt. Dazu ist es nötig formal eine Regel zu definieren, mit welcher die Gewichte des Netzwerks verändert werden, um die Performance hinsichtlich des gewählten Fehlermaßes zu optimieren (Gradientenverfahren): Die sogenannte Delta-Regel (vgl. Widrow and Hoff (1960) und Widrow and Hoff (1988)).

Für das Anpassen der Gewichte und Schwellwerte in einem MLP mit nicht-linearen Aktivierungsfunktionen wird eine Verallgemeinerung, der von Rumelhart et al. (1986b) entwickelte Backpropagation-Algorithmus, verwendet. Dieser besteht abstrakt gesehen aus zwei Phasen. In der ersten Phase wird die Ausgabe des Netzes sowie der entsprechende Fehler berechnet (*Forward Pass*), in der zweiten der Fehler mittels Delta-Regel zurück propagiert und so der Gradient schichtweise durch Anwendung der Kettenregel berechnet (*Backward Pass*).

Im Folgenden wird der Backpropagation-Algorithmus vorgestellt (vgl. z.B. Rojas, 1996, S. 151ff.). Beispielhaft dient hier der MSE als Fehlermaß.

- $W^l$ : Gewichtsmatrix
- $b^l$ : Vektor mit Schwellwerten
- $J(\cdot)$ : Kostenfunktion
- $x$ : Eingabevektor
- $x^l$ : Eingabevektor Layer
- $f(\cdot)$ : Ausgabevektor
- $\phi(\cdot)$ : Aktivierungsfunktion
- $z^l$ : Linearer Ausgabevektor

- $a^l$ : Aktivierter Ausgabevektor
- $N$ : Anzahl Trainingsbeispiele
- $L$ : Anzahl Layer
- $\circ$ : Hadamard-Produkt
- $\eta$ : Lernrate

Gleichung 2.15 definiert die Kostenfunktion:

$$J(W, b) = \frac{1}{2N} \sum_{i=1}^N (f(x_i) - y_i)^2 \quad (2.15)$$

Die Gleichungen 2.16 und 2.17 beschreiben den allgemeinen Gradientenabstieg:

$$W_{t+1}^l = W_t - \eta \nabla W_t \quad (2.16)$$

$$b_{t+1}^l = b_t - \eta \nabla b_t \quad (2.17)$$

Der Backpropagation-Algorithmus arbeitet sowohl im *Online*- wie auch im *Batch*-Modus. Das bedeutet um den Gradienten der Kostenfunktion  $\nabla J(W, b)$  zu berechnen, wird zuerst für jedes Trainingsbeispiel  $\nabla J(W, b, x_i, y_i)$  einzeln berechnet und aufsummiert. Folgende Formeln 2.18 und 2.19 beschreiben die Delta-Regel des Backpropagation-Algorithmus:

Delta Output-Layer:

$$\delta^L = (a_i^L - y_i) \circ \phi'(z^L) \quad (2.18)$$

Delta für Hidden-Layer  $l$ :

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \circ \phi'(z^l) \quad (2.19)$$

Die partiellen Ableitungen pro Schicht (Gleichung 2.20 und 2.21) werden über alle Trainingsbeispiele  $1..N$  aufsummiert und am Ende eines Durchlaufs mit  $\frac{1}{N}$  gemittelt.

Partielle Ableitung für Layer  $l$ :

$$\frac{\partial J(W, b)}{\partial W^l} = \frac{1}{N} \sum_{i=1}^N \delta^l (x_i^l)^T \quad (2.20)$$

$$\frac{\partial J(W, b)}{\partial b^l} = \frac{1}{N} \sum_{i=1}^N \delta^l \quad (2.21)$$

Betrachtet man die Delta-Formeln 2.18 und 2.19, wird ersichtlich, dass sich die Verwendung einer Aktivierungsfunktion, deren Ableitung sich durch den Funktionswert selbst berechnen lässt, besonders eignet. Im Falle der Sigmoid-Funktion vereinfacht sich  $\phi'(z^l)$  so zu  $\text{sig}'(z^l) = \text{sig}(a^l)(1 - \text{sig}(a^l))$ , was das Speichern von  $z^l$  erübrigt.

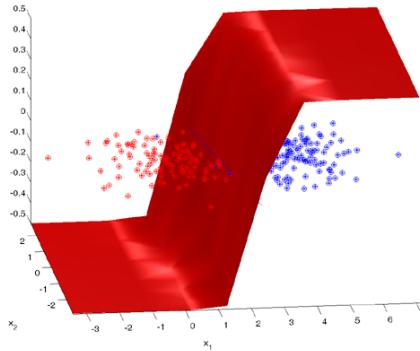


Abbildung 2.8: Logistische Regression mit zwei Klassen (Bild: Vadim Strijov, Computing Centre of the Russian Academy of Sciences)

## 2.5 Basisarchitekturen

Dieses Kapitel beschreibt die zwei Grundarchitekturen Neuronaler Netze (MLPs). Diese umfassen einerseits die Softmax-Regression zur Klassifikation und andererseits die Nichtlineare-Regression.

### 2.5.1 Softmax-Regression (Klassifikation)

Die Logistische Regression stellt ein bekanntes lineares statistisches Regressionsverfahren aus den 1950er dar, welches ursprünglich nichts mit dem Neuronalen Lernmodell zu tun hat. Praktischerweise entspricht dieses jedoch der gleichen Architektur, was die Verwendung als Ausgabeschicht für die Klassifikation mit MLPs nahelegt. Abbildung 2.8 zeigt das Verfahren für zwei Klassen. Die Logistische Regression berechnet nach der Formel 2.22 die Wahrscheinlichkeit einer binären Ausgabe  $y_i \in \{0, 1\}$  für eine gegebene Eingabe  $x_i$  (vgl. Hastie et al., 2009, S. 119ff).

$$p(y|X, W) = \prod_{i=1}^n \left[ \frac{1}{1 + \exp^{-Wx_i}} \right]^{y_i} \left[ 1 - \frac{1}{1 + \exp^{-Wx_i}} \right]^{1-y_i} \quad (2.22)$$

Maximierte man nun die Likelihood beziehungsweise minimierte die negative Log-Likelihood erhält man die Formel 2.23 mit  $f(x_i) = \frac{1}{1+\exp^{-Wx_i}}$ . Dies entspricht proportional einem einschichtigen MLP mit Sigmoid-Aktivierungsfunktion sowie *Cross-Entropy*-Fehlermaß und lässt sich somit mittels des Backpropagation-Algorithmus optimieren.

$$-\ln(p(y|X, W)) = - \sum_{i=1}^n y_i \ln(f(x_i)) + (1 - y_i) \ln(1 - f(x_i)) \quad (2.23)$$

## Multinomial Regression

Mehrdimensionale Ausgaben können in neuronalen Netzen durch einfaches Hinzufügen von Neuronen zum Output-Layer erzeugt werden. Ein gern verwendeter Code, um mehrere Klassen darzustellen, ist der sogenannte *Grandmother-cell*-Code (vgl. LeCun et al., 1998a). Hier wird für  $d$ -Klassen ein  $d$ -dimensionaler Zielvektor oder erweiterter Labelvektor definiert mit  $d_j = 1$  für die Klasse  $j$ . Das Modell der Logistischen Regression lässt sich ebenso leicht auf mehrere Klassen erweitern. Dazu muss die Ausgabe, um ein valides probabilistisches Modell zu erhalten, normalisiert werden. Gleichung 2.24 beschreibt die Kostenfunktion der multinomialen Regression. Diese ist im Bereich Neuronale Netze eher bekannt als Softmax-Regression (vgl. z.B. Krizhevsky et al. (2012) und Bengio et al. (2015)).

$$-\ln(p(y|X, W)) = -\sum_{i=1}^n \sum_{j=1}^d 1\{y^i = j\} \ln\left[\frac{\exp^{w_j^T x^i}}{\sum_{l=1}^d \exp^{w_l^T x^i}}\right] \quad (2.24)$$

Neben der Softmax-Regression wird teilweise auch ein Output-Layer mit radialen Basisfunktionen (RBFs) verwendet (vgl. LeCun et al., 1998a). Dies hat den Vorteil, dass für jede Klasse ein eigener spezifischer Labelvektor definiert werden kann. Ein Nachteil ist jedoch, dass diese Variante die Ausgabe nicht normalisiert und somit nicht probabilistisch interpretiert werden kann. Als Basisfunktionen im Hidden-Layer eignen sich RBFs nur sehr bedingt, da diese lokal im Eingaberaum sind und daher sehr viele Einheiten für hochdimensionale Räume benötigt werden (vgl. LeCun et al., 1998b).

## Besonderheit in Verbindung mit Backpropagation

Wird Backpropagation angewandt, muss im Output-Layer die Ableitung der Aktivierungsfunktion berechnet werden. Gleichung 2.25 zeigt diese für die Softmax-Funktion.  $\delta$  entspricht hierbei dem Kronecker-Delta (vgl. im Foldenden Bengio et al., 2015, Kap. 6.3.2).

$$\text{softmax}'(x)_{ij} = \text{softmax}(x)_i (\delta_{ij} - \text{softmax}(x)_j) \quad (2.25)$$

Dieser Ausdruck ist sehr unhandlich. Es lässt sich allerdings zeigen, dass die Softmax-Regression in Verbindung mit einem *Cross-Entropy*-Fehlermaß die Berechnung von  $\delta^L$  nicht erschwert, sondern zu Gleichung 2.26 vereinfacht.

Delta Output-Layer:

$$\delta^L = a_i^L - y_i \quad (2.26)$$

Dies bietet damit den großen Vorteil, dass  $\delta^L$  nur noch vom Fehler abhängt und nicht mehr von der Ableitung der Aktivierungsfunktion und damit die Nichtlinearität am Ausgang nicht gesättigt wird.

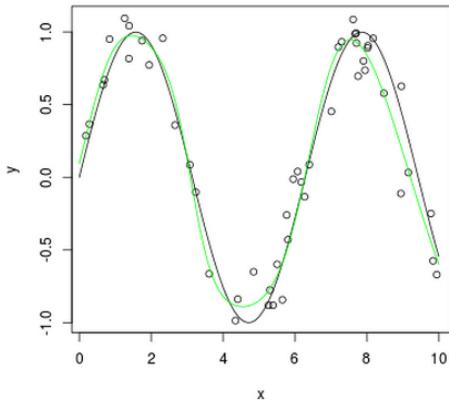


Abbildung 2.9: Approximierte Sinuskurve mit einem Hidden-Layer mit 6 Neuronen (grün)

### 2.5.2 Funktionsapproximation (Regression)

Neuronale Netze können beliebige Funktionen  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  approximieren und so mittels Backpropagation auf beliebige Daten trainiert werden (vgl. Rojas, 1996, S. 269ff). Es ist lediglich darauf zu achten, dass der Wertebereich des Output-Layers zu dem der Labels passt. So werden im Output-Layer meist lineare Aktivierungsfunktionen verwendet. Alternativ können auch die Labels auf den Wertebereich der verwendeten Aktivierungsfunktion skaliert werden.

Abbildung 2.9 zeigt eine Sinuskurve, welche von einem MLP mit 6 Neuronen im Hidden-Layer approximiert wird. Die zugrundeliegende Trainingsdaten sind aus einer Sinuskurve mit additivem normalverteilten Rauschen generiert.

## 2.6 Convolutional Neural Networks

Bildklassifikation, Objektlokalisierung und Objekterkennung rücken derzeit immer mehr in den Fokus. Dies ist beispielsweise an einer immer größeren Verbreitung von Drohnen und autonomen Fahrzeugen zu erkennen ist. Diese Entwicklung geht einher mit den Fortschritten im Bereich Neuronale Netze und den Möglichkeiten des *Deep Learning*. Neuere Arbeiten aus den Neurowissenschaften stützen darüber hinaus getroffene Annahmen mit realen Beobachtungen an Lebewesen, was die Entwicklung weiter unterstützt. In diesem Zusammenhang wird gerne das berühmte *Halle Berry*-Neuron im menschlichen Temporallappen genannt, welches Hally Berry erkennt. Dies deutet darauf hin, dass das biologischen Modell des Sehens auf einem invarianten, dünnbesetzten (*spare*), expliziten Code basiert (vgl. Quiroga et al., 2005).

Dieses Kapitel behandelt eine spezielle Klasse Neuronaler Netze, die das biologische Modell des Sehens und Hörens versuchen zu imitieren. Die sogenannten *Convolutional Neural Networks*.

### 2.6.1 Modellbeschreibung

*Convolutional Neural Networks* (CNNs) sind eine Erweiterung der *Multi Layer Perceptrons* (MLPs), welche den biologischen visuellen Cortex zu imitieren versuchen. Die Idee selbst beruht auf den Arbeiten von Hubel and Wiesel (1962) bezüglich des Sehempfindens von Katzen, die zeigen, dass der visuelle Cortex aus komplex angeordneten Zellen besteht, wobei jede einzelne Zelle einen kleinen Bereich des Sichtfeldes abdeckt. Diese Architektur erlaubt es, räumlich voneinander getrennte Muster zu erfassen.

Im künstlichen Modell setzt sich ein CNN aus mehreren Convolution- und Pooling-Layern sowie einem klassischen MLP zusammen. Dieses spezielle Design erlaubt es, eine 2D-Struktur im Input-Layer zu erfassen. Dies wird durch sogenannte lokale Verbindungen (*Local Connection*) erzielt.<sup>1</sup>. Das erste künstliche Modell dieser Art ist das *NeoCognitron* von Fukushima (1980), welches zwei verschiedene Arten lokaler rezeptiver Felder (siehe Abbildung 2.10) definiert: Eines zur Detektion von Kanten und eines mit lokaler Invarianz hinsichtlich Translation.

---

<sup>1</sup>Auch wenn die Ursprünge des Modells im Bereich der visuellen Wahrnehmung liegen, wird es heute beispielsweise auch für Spracherkennung und NLP verwendet (vgl. dazu Socher et al. (2011) und Sainatha et al. (2015))

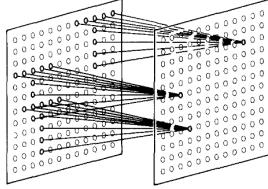


Abbildung 2.10: Lokales rezeptives Feld (siehe Fukushima (1980))

Das *LeNet* von LeCun et al. (1989) ist die Weiterentwicklung dieses Models. Hier teilen sich mehrere rezeptive Felder dieselben Gewichte beziehungsweise Parameter (*Parameter Sharing*), was die Verbindungen und dadurch auch die Anzahl der Gewichte reduziert. Die Anzahl ist damit von der Dimensionalität der Eingabe entkoppelt. Algebraisch entspricht dies einer Faltung (*Convolution*), wovon das Model seinen Namen ableitet. Diese Architektur ist die Grundlage heutiger CNNs und definiert die damit verbundenen Eigenschaften (vgl. LeCun et al. (1998a), Bengio and LeCun (2007) und Zeiler and Fergus (2014)):

- Lokale Extraktion von Merkmalen (*Local Feature Extraction*)
- Translationsinvarianz hinsichtlich Eingabedaten (geringe Skalierungs- und Rotationsinvarianz)
- Einfacheres Training durch weniger Parameter und Verbindungen als MLP mit gleich vielen Neuronen
- Nicht-lokale Generalisierung durch Verschachtelung nichtlinearer Funktionen

Abbildung 2.11 stellt ein CNN mit 8 Schichten zur Klassifikation von  $32 \times 32$  Pixel großen Eingabebildern dar. Die ersten 6 Schichten zählen zum CNN, während die letzten beiden Schichten ein klassisches MLP repräsentieren. Der zugrundeliegende Ansatz ist hierbei, mittels CNN Merkmale (*Features*) derart zu extrahieren, sodass die Klassen am Ende möglichst gut trennbar sind.

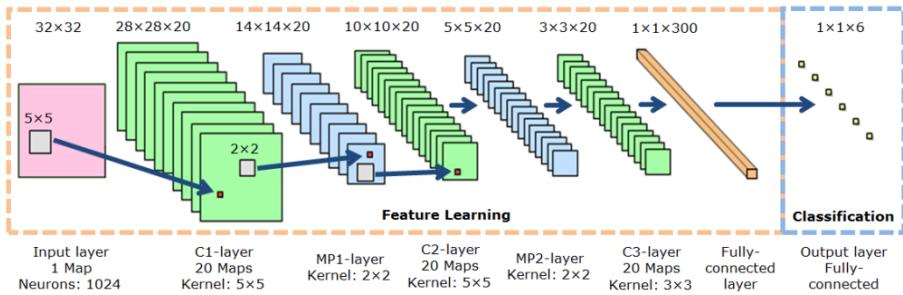


Abbildung 2.11: Convolutional Neural Network mit 8 Schichten (siehe Nagi et al., 2011)

Aus Gründen der Vereinfachung werden im weiteren Verlauf des Kapitels lediglich quadratische Eingaben sowie quadratische Masken betrachtet.

### Convolution-Layer

Der Convolution-Layer ist einer der beiden zentralen Bestandteile eines CNN (vgl. im Folgenden LeCun et al., 1998a). Dieser transformiert eine 3D-Eingabe  $x$  mit Tiefe  $j$ , bezeichnet als *m-Input-Maps*, zu einer 3D-Ausgabe  $y$ . Die Tiefe  $n$  der Ausgabe wird bestimmt durch die Anzahl an Faltungskernen und selbst meist als *n-Feature-Maps* bezeichnet. Ein Faltungskern  $i$  ist in Abbildung 2.12 dargestellt. Er entspricht ebenfalls einer 3D-Struktur, welche durch Höhe (*height*)  $k_w$  und Breite (*width*)  $k_w$  der Faltungsmaske sowie der Tiefe *depth*  $k_d$  bestimmt ist. Die Tiefe  $k_d$  muss der Tiefe der 3D-Eingabe  $j$  entsprechen. Es gibt somit gleich viele Faltungsmasken pro *Feature-Map* wie es *Input-Maps* gibt: Es gilt  $k_d = j$ . Alle Faltungskerne zusammen werden kompakt mit  $W$  bezeichnet und einzelne Filtermasken mit  $W_{ij}$  eindeutig adressiert. Bei der Berechnung der Netz-Ausgabe (*Forward Pass*) wird für die *Input-Map*  $x_j$  die Filtermaske  $W_{ij}$  aus dem aktuellen Faltungskern  $i$  gezogen und damit die Ausgabe  $a_i$  berechnet. Dies geschieht mittels diskreter Faltung. Die 3D-Ausgabe  $y$  kann ebenfalls als Bild interpretiert werden, indem jedes Pixel der Ausgabe eines Neurons entspricht. Folglich wird pixelweise sowohl ein *bias*  $b_i$  addiert als auch die Aktivierungsfunktion auf jede der erzeugten *n-Feature-Maps* angewandt.

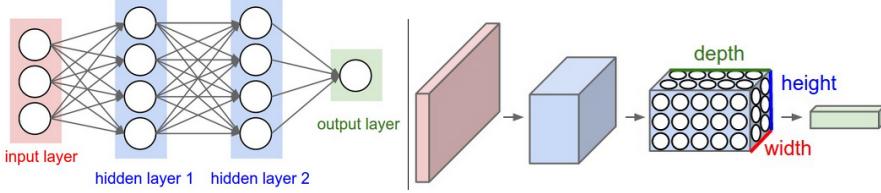


Abbildung 2.12: Jeder Convolution-Layer transformiert eine 3D-Eingabe (z.B. RGB-Bild) in eine 3D-Ausgabe (siehe Fei-Fei and Karpathy (2014))

Folgende Hyperparameter definieren einen Convolution-Layer:

- *Feature-Maps*:  $n$
- *Input-Maps*:  $m = k_d$
- Größe Faltungskern:  $k_w$

Formal berechnet der Convolution-Layer für jede der  $n$  *Feature-Maps* an jeder gültigen Stelle  $iijj$  der  $N \times N$ -großen  $m$  *Input-Maps* zuerst die 2D-Faltung 2.27 (Operator:  $*$ ) und im Anschluss die Aktivierungsfunktion 2.28:

$$z_i = \sum_{j=0}^m x_j * W_{ij} \quad (2.27)$$

$$a_i = \phi(z_i + b_i) \quad (2.28)$$

Durch die Randbehandlung *valid* reduziert sich die Größe der *Feature-Maps* auf  $(N - k_w + 1) \times (N - k_w + 1)$ .

Die Anzahl der Gewichte berechnet sich aus  $n \cdot k_d \cdot k_w \cdot k_w$ . Die Anzahl Pixel in der 3D-Ausgabe entspricht der Anzahl an Neuronen. Im Rahmen des Trainings gilt es, diese Gewichte zu trainieren. Die exakte mathematische Berechnung von Delta  $\delta$  und Gradienten  $\frac{\partial J(W,b;x,y)}{\partial W_{ij}^l}$  und  $\frac{\partial J(W,b;x,y)}{\partial b_i^l}$  für das Training mit Backpropagation folgt in Kapitel 2.6.2.

### Pooling-Layer

Der zweite wichtige Bestandteil eines CNN ist das sogenannte *Pooling* (vgl. im Folgenden LeCun et al., 1998a). LeCun et al. (1998a) bezeichnen diesen Vorgang als *Subsampling*, wobei heute der Begriff *Pooling* eher geläufig ist (vgl. Zeiler and Fergus (2013) oder Glorot et al. (2011)). Allgemein berechnet der Pooling-Layer ein DownSampling, wobei dessen Art nicht definiert ist. Abbildung 2.13 stellt das sogenannte Max-Pooling dar (vgl. Glorot et al. (2011) und Zeiler et al. (2011)). Jede *Input-Map*  $x_j$  der 3D-Eingabe wird vom Pooling-Layer einzeln bearbeitet und ausgegeben. Die Anzahl  $m$  ändert sich nicht. Für jedes  $x_j$  wird nun ein Bereich der Größe  $k_h \times k_w$ , wobei  $k_h$  der

Filterhöhe und  $k_w$  der Filterbreite entspricht, ausgeschnitten und verarbeitet. Im Falle des Max-Pooling wird das Maximum berechnet und in der Ausgabe gespeichert.

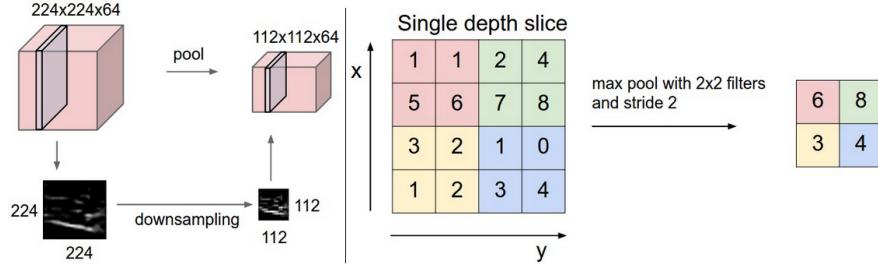


Abbildung 2.13: Ein Pooling-Layer transformiert eine 3D-Eingabe (z.B. RGB-Bild) in eine kleinere 3D-Ausgabe (Downsampling) (siehe Fei-Fei and Karpathy (2014))

Der einzige notwendige Hyperparameter für einen Pooling-Layer ist somit die Filtergröße  $k_w$

Formal teilt der Max-Pooling-Layer jede *Input-Map*  $x_j$  in  $k_w \times k_w$  disjunkte Bereiche und wählt in jedem Bereich das Maximum. Dadurch reduziert sich die Ausgabe auf  $\frac{N}{k_w} \times \frac{N}{k_w}$ .

Alternative, ebenfalls verbreitete Formen des Poolings stellen das Average-Pooling (vgl. LeCun et al., 1998a) oder das  $L_p$ -Pooling (vgl. Pierre Sermanet and LeCun, 2012) dar. Die Besonderheit beim Pooling-Layer sind die fehlende Aktivierungsfunktion sowie Gewichte und Schwellwerte, weshalb Convolution- und Pooling-Layer auch zusammen in einer Schicht berechnet werden können (vgl. Simard et al., 2003). Die exakte mathematische Beschreibung von Delta  $\delta$  für das Training mit Backpropagation folgt in Kapitel 2.6.2.

## 2.6.2 Training mit Backpropagation

Im letzten Teil des Kapitels wurde bereits das Modell eines CNN und dessen elementaren Bestandteile beschrieben. Außerdem wurden die enthaltenen Parameter vorgestellt. Das Ziel dieses Abschnitts ist es zu beschreiben, wie die beiden neuen Schichten mittels Backpropagation trainiert werden können (vgl. z.B. Bouvrie, 2006). Wie in Kapitel 2.4 beschrieben, ist für Backpropagation (Backward Pass) pro Trainingsbeispiel die Berechnung dreier Größen notwendig:  $\delta^l$  sowie  $\frac{\partial J(W, b; x_i, y_i)}{\partial W_{ij}^l}$  und  $\frac{\partial J(W, b; x_i, y_i)}{\partial b_i^l}$ .

Um die Formeln zu vereinfachen, werden an dieser Stelle sogenannte *Messages* eingeführt. Dies erleichtert die Notation insofern, dass die Berechnung von  $\delta^l$  nicht mehr von den Gewichten der vorherigen Schicht  $W^{l+1}$  abhängt.

Delta  $\delta_{message}^l$  bezeichnet so den über die Schicht hinaus zurückpropagierten Fehler. Im Falle von gewöhnlichen MLPs gilt  $\delta_{message}^{l+1} = (W^{l+1})^T \delta^{l+1}$ . Für Convolutional-Layer wird das Delta  $\delta_i^l$  korrespondierend zu jeder einzelnen der  $n$ -Feature-Maps aus dem eingehenden  $\delta_{message_j}^{l+1}$  berechnet. An dieser Stelle sei nochmals betont, dass die Anzahl Feature-Maps  $n$  der Anzahl Input-Maps  $m$  der nächsten Schicht entspricht. Es gilt folglich  $n^l = m^{l+1}$ . Im Folgenden werden zuerst die notwendigen Formeln für Pooling-Layer und im Anschluss die der Convolution-Layer vorgestellt.

Formel 2.29 berechnet das Delta  $\delta_{message_j}^l$  im Pooling-Layer mittels Kronecker-Produkt  $kron(\cdot)$ . Da der Pooling-Layer keine Parameter besitzt ist die Berechnung von  $\delta_i^l$  irrelevant.

Delta Average-Pooling-Layer:

$$\delta_{message_j}^l = kron(delta_{message_j}^{l+1}, ones(k_w, k_w)) \circ \frac{1}{k_w^2} \quad (2.29)$$

Je nach Art des Poolings muss hier unterschiedlich vorgegangen werden. Für einen Max-Pooling-Layer müssen im *Forward pass* die Positionen der Maxima gespeichert und bei der Rückpropagierung (Backward pass) der Fehler an die gespeicherten Positionen geschrieben werden. Im Lp-Pooling muss der Fehler entsprechend Norm  $p$  und Filtermaske aufgeteilt werden.

Delta Convolution-Layer:

Für jede Feature-Map  $i$  im Convolution-Layer wird das zugehörige Delta  $\delta_i^l$  mit Formel 2.30 berechnet.

$$\delta_i^l = \delta_{message_j}^{l+1} \circ \phi'(z_i) \quad (2.30)$$

Jedes  $\delta_{message_j}^l$  für die Rückpropagierung des Fehlers berechnet sich mit Formel 2.31. Die Funktion  $rot180(\cdot)$  bezeichnet das Drehen einer Matrix um  $180^\circ$  beziehungsweise das Vertauschen beider Axen (*Flipping*).

$$\delta_{message_j}^l = \sum_{i=0}^n \delta_i^l * rot180(W_{ij}) \quad (2.31)$$

Durch die Randbehandlung *full* erhöht sich die Größe des Delta  $\delta_{message_j}^l$  von  $(N - k_w + 1) \times (N - k_w + 1)$  wieder auf  $N \times N$ .

Gradient Convolution-Layer:

Der Gradient für den Convolution-Layer kann mittels Formel 2.32 und 2.33 aus der Eingabe und Delta  $\delta^l$  berechnet werden. Die Randbehandlung *valid* ist anzuwenden.

$$\frac{\partial J(W, b)}{\partial W_{ij}^l} = \frac{1}{M} \sum_{m=1}^M rot180(x_{mj}^l * rot180(\delta_{mi}^l)) \quad (2.32)$$

$$\frac{\partial J(W, b)}{\partial b_i^l} = \frac{1}{M} \sum_{m=1}^M \sum_{u=0}^{k_w} \sum_{v=0}^{k_w} \delta_{miuv}^l \quad (2.33)$$

Analog zum Vorgehen bei MLPs wird der Gradient (Gleichung 2.32 und 2.33) ebenfalls über alle Trainingsbeispiele  $1..M$  aufsummiert und am Ende eines Durchlaufs mit  $\frac{1}{M}$  gemittelt. Außerdem kann bei Aktivierungsfunktionen, deren Ableitung sich durch den Funktionswert selbst darstellen lässt, auf die Speicherung von  $z_i$  verzichtet werden.

## Kapitel 3

# Spezielle Methoden im Deep Learning

Bereits zu Beginn (siehe Kapitel 1.1), wie auch im vergangenen Kapitel (siehe 2.6.1) wurde darauf hingewiesen, dass Convolutional Neural Networks (CNNs) allgemeinen Multilayer Perceptrons (MLPs) in zwei zentralen Punkten überlegen sind. So reduzieren sie das Problem des *Overfittings*, wobei es sich um die Überanpassung des Modells an die Trainingsdaten mit dem Resultat schlechter Generalisierung auf neue unbekannte Daten handelt. Dies geschieht durch Reduktion der zu lernenden Gewichte und Verbindungen. Des Weiteren erlauben CNNs die lokale Extraktion von Merkmalen durch Einbezug räumlicher Korrelationen im Eingaberaum (vgl. LeCun et al. (1998a)).

Auch durch die Verwendung von CNNs bleiben dennoch zentrale Schwierigkeiten im Deep Learning bestehen:

- Verschwinden des Gradienten (*Vanishing Gradient*) in tiefen Netzen (vgl. Hochreiter (1991))
- Gradientenabstieg bei nicht-konvexer Zielfunktion (vgl. Martens (2010) und Dauphin et al. (2014))
- *Overfitting* bei großen Netzen, insbesondere in nachgeschalteten MLPs (vgl. Hinton et al. (2012))
- Schlechte Konditionierung der Fehlerlandschaft aufgrund von *Parameter Sharing* (vgl. LeCun et al. (1998a))

Die ersten beiden Problemen lassen sich unter der Problematik der sogenannten *Pathological Curvature* zusammenfassen (vgl. Martens (2010)). Historisch betrachtet liefert die Arbeit von Hochreiter (1991) die Grundlage zu dem Problem, das heute als *Vanishing Gradient*-Effekt bekannt ist. Dieser beschreibt den Zusammenhang zwischen der abnehmenden Größe des Fehlersignals und der Tiefe des Netzwerks. Das heißt, je mehr Hidden-Layer ein

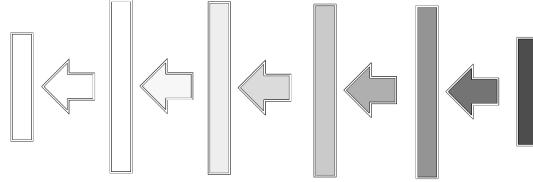


Abbildung 3.1: Der Vanishing Gradient-Effekt beschreibt bei der Rückpropagierung das abklingende Fehlersignal von Output- hin zu Input-Layer

Netz besitzt, desto kleiner wird der Gradient in den vorderen Schichten und desto langsamer lernen diese Schichten in der Folge (siehe Abbildung 3.1). Deutlich wird der Effekt bei der Betrachtung der Formel 3.1 für das zurückpropagierte Fehlersignal  $\delta_{message}^l$  im Layer  $l$ , in der aus Kapitel 2.6.2 bekannten  $\delta_{message}$ -Notation.

$$\delta_{message}^l = (W^l)^T (\delta_{message}^{l+1} \circ \phi'(z^l)) \quad (3.1)$$

Setzt man klassische Aktivierungsfunktionen wie  $sigmoid(\cdot)$  oder  $tanh(\cdot)$  ein (siehe Kapitel 2.2), gilt  $\phi'(z^l) \leq 1$ . Dies führt zwangsläufig zu einem abklingenden Fehlersignal (*Vanishing Gradient*). Man könnte argumentieren, dass dies durch ein großes  $W^l$ , sodass  $(W^l)^T (\delta_{message}^{l+1} \circ \phi'(z^l)) \geq 1$ , verhindert werden könnte. Dem ist nicht der Fall, da gleichzeitig  $z^l = W^l x^l + b^l$  gilt und damit ein großes  $W^l$  unweigerlich zu einem großen  $z^l$  und damit zu einer kleineren Ableitung der Aktivierungsfunktion, welche ihr Maximum bei 0 besitzt, führt. Ist zusätzlich zu einem großen  $W^l$  der Schwellwert  $b^l$  negativ, sodass  $z^l \approx 0$ , führt dies ebenso nicht zum Verschwinden des Effekts sondern zu einem, nicht weniger ungünstigen, *Exploding Gradient*-Effekt. Selbst im linearen Fall ohne Aktivierungsfunktion tritt dieser Effekt auf, wenn die Gewichte jedes Neurons nicht die Norm  $\|w\| \approx 1$  besitzen.

Mit normalisierter Initialisierung der Gewichte lässt sich der Effekt kaschieren (vgl. Glorot and Bengio (2010)). Allerdings scheint erst die Verwendung der neuartigen ReLu-Aktivierungsfunktion (vgl. Kapitel 2.2) den Effekt gänzlich zu unterdrücken und führt darüber hinaus zu oftmals erwünschten, dünnbesetzten (*sparse*) Aktivierungen (vgl. Glorot et al. (2011)). Der Nachteil von ReLu-Neuronen ist allerdings, dass diese bei zu großen Lernraten und damit zu großen Änderungen der Gewichte unwiderruflich sterben (*Dying ReLu*) und somit Teile des Netzes auslöschen können (vgl. Maas et al. (2013)).

$$f(x) = \max(0, x) \quad (3.2)$$

Viele der in diesem Kapitel vorgestellten Methoden zielen auf den Vanishing-Gradient-Effekt ab, selbst wenn heute diskutiert wird, dass der Effekt nicht

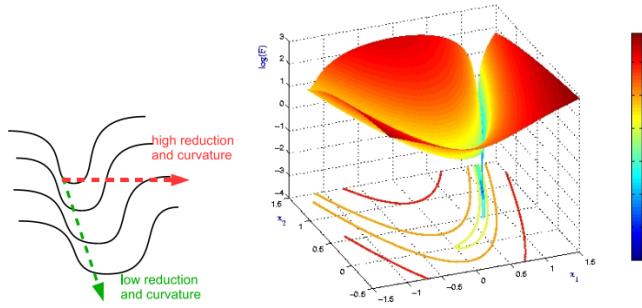


Abbildung 3.2: Pathological Curvature in der Rosenbrock-Funktion:  $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$  (vgl. Martens (2010))

ausschließlich für die Schwierigkeiten im Deep-Learning verantwortlich ist. So zeigt Martens (2010) empirisch, dass der Effekt ebenso auf eine ungünstige Fehlerlandschaft (*Pathological Curvature*) zurückgeführt werden kann. Eine solche ist in Form der Rosenbrock-Funktion in Abbildung 3.2 dargestellt. Neben den benannten Problemen hinsichtlich der Fehlerfunktion, führt *Overfitting* häufig ebenso zu schlechten Ergebnissen und entsprechende Regularisierungsmethoden werden benötigt.

### 3.1 Vorverarbeitung

Der Bereich Vorverarbeitung steht nicht unmittelbar in Zusammenhang mit Deep Learning, wird jedoch zwecks der Vollständigkeit, besonders in Verbindung zu CNNs, vorgestellt. Becker (1991) stellte bereits Anfang der 90er Jahre fest, dass durch Dekorrelation der Trainingsdaten MLPs effizienter trainiert werden können. LeCun et al. (1998b) fassen die klassische Vorverarbeitung in linearen MLPs wie folgt zusammen:

- Mittelwertfreie Trainingsdaten verhindern eine steile Fehlerlandschaft.
- Die Normalisierung der Varianz unterschiedlicher Merkmale verhindert deren unterschiedliche Gewichtung.
- Die Dekorrelation der Trainingsdaten führt zwar zu einer diagonalen Hesse-Matrix, allerdings zeigt der Gradient nicht in Richtung Minimum. Dies muss durch dedizierte Lernraten, entsprechend den Kehrwerten der Eigenwerte pro Gewicht, korrigiert werden.
- Das Whitening der Daten führt zu einer kreisförmigen Fehlerlandschaft mit korrektem Gradienten.

Diese Hinweise gelten nur für lineare MLPs und somit quadratische Fehlerlandschaften. Die Fehlerlandschaft der hier verwendeten nicht-linearen ML-

$P_s$  kann jedoch lokal quadratisch approximiert werden (vgl. Hinton et al. (2015)).

Die genannten Techniken können allerdings nicht unmittelbar auf CNNs übertragen werden, da diese versuchen lokale Korrelationen an verschiedenen Orten im Eingaberaum zu extrahieren. So zeigt sich bei der Verwendung von CNNs, dass die erfolgreichsten Architekturen lediglich den Mittelwert über die gesamten Trainingsdaten berechnen und diesen von jedem Pixel subtrahieren (vgl. Krizhevsky et al. (2012) und Simonyan and Zisserman (2014)). Fei-Fei and Karpathy (2014) kommen zu einem ähnlichen Schluss und beschreiben für CNNs lediglich die Zentrierung der Daten mittels globalen Mittelwert, alternativ pro Farbkanal oder Pixel, als nötige Vorverarbeitung.

Teilweise werden für CNNs dennoch zwei weitere Techniken als Praxis beschrieben (vgl. Andrade (2014) und Goodfellow et al. (2013b)) und deshalb im Folgenden kurz eingeführt.

### 3.1.1 Kontrastnormalisierung

Das Ziel der Kontrastnormalisierung ist es, unterschiedliche Trainingsbeispiele desselben Objekts in einen ähnlichen Kontrastbereich zu transformieren und so Helligkeitsunterschiede zu kompensieren (vgl. Zeiler and Fergus (2013)).

Die globale Kontrastnormalisierung GCN bearbeitet jedes Trainingsbeispiel einzeln und berechnet den Mittelwert und die Varianz über alle Merkmale beziehungsweise Pixel. Im Anschluss wird der Mittelwert subtrahiert und durch die Varianz geteilt (vgl. Pierre Sermanet and LeCun, 2012). Neben der GCN wird häufig auch die lokale Kontrastnormalisierung (LCN) angewandt. Diese nimmt pro Trainingsbeispiel als Eingabe jeweils nur kleine Umgebungen  $k_h \times k_w$  und berechnet die Kontrastnormalisierung für diese Bereiche einzeln (vgl. Jarrett et al., 2009).

Zeiler and Fergus (2013) verwenden für Farbbilder den RGB-Raum und normalisieren jeden Kanal einzeln. Dies kann jedoch zu einer Verschiebung des Farbtöns führen. Soll dies vermieden werden, wird für Farbbilder entweder der HSV-Raum vorgeschlagen, in welchen nur die Intensität  $V$  normalisiert wird (vgl. Pink, 2011, S. 56). Oder es wird der YUV-Farbraum verwendet und die Normalisierung lediglich auf den Y-Kanal angewandt (vgl. Pierre Sermanet and LeCun, 2012).

### 3.1.2 ZCA-Whitening

Der Begriff *Zero Component Analysis* (ZCA) beschreibt ein der *Principle Component Analysis* (PCA) sehr ähnliches Verfahren (vgl. im Folgenden Krizhevsky and Hinton (2009)). Es zielt darauf ab, die Eingangsdaten so zu

dekorrelieren, sodass die Transformation so nahe wie möglich am Original ist.

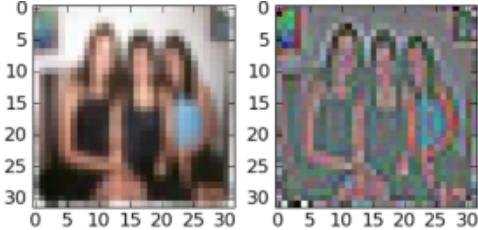


Abbildung 3.3: Originales Bild (links) und ZCA-transformiertes Bild (rechts) (siehe. Krizhevsky and Hinton (2009))

Die Transformationsmatrix  $W_{ZCA}$  ist in Gleichung 3.3 angegeben, wobei die Kovarianzmatrix  $C = X^T X$  ist. Der einzige Unterschied zum PCA-Whitening liegt darin, dass eine weitere Rotation zurück in den Bildraum durchgeführt wird. Dies ist möglich, da *whitened-data* auch nach einer Rotation mit einer orthogonalen Matrix *whitened* bleibt.

$$W_{ZCA} = C^{-\frac{1}{2}} = P(D + \epsilon)^{-\frac{1}{2}} P^T = PW_{PCA} \quad (3.3)$$

Abbildung 3.3 zeigt die Transformation für ein Beispiel.

## 3.2 Initialisierung

Die Initialisierung eines neuronalen Netzes ist äußerst wichtig, da die Gewichte ungleich Null sein müssen. Ansonsten berechnen alle Neuronen die gleiche Ausgabe und somit die gleichen Gradienten. (*Breaking the Symmetry*) (vgl. Rojas, 1996, S. 201). LeCun et al. (1989) initialisieren die Gewichte beispielsweise zufällig zwischen  $-\frac{2.4}{fan_{in}}$  und  $\frac{2.4}{fan_{in}}$ , wobei  $fan_{in}$  der Anzahl Eingangsneuronen entspricht. Das hat zum Ziel die Nichtlinearität mit Werten um 0 zu versorgen und damit nicht zu sättigen (*Vanishing Gradient*). Das erfolgreiche Netz von Krizhevsky et al. (2012) verwendet beispielsweise die Standard-Initialisierung, eine einfache Normalverteilung mit  $W \sim \mathcal{N}(0, 0.01)$ . Dies ist möglich, da als Aktivierungsfunktionen ReLus verwendet werden, welche nicht sättigen können.

Daneben existieren im Deep Learning heute zwei Arten zur Initialisierung, welche im Folgenden aufgeführt sind.

### 3.2.1 Normalisierte Fehlerpropagierung

Stellvertretend für eine derartige Initialisierung der Gewichte, sodass die Varianz des Signals sowohl bei der Berechnung der Ausgabe (*Forward Pass*), als auch bei der Rückpropagierung des Fehlers (*Backward Pass*) gleich groß

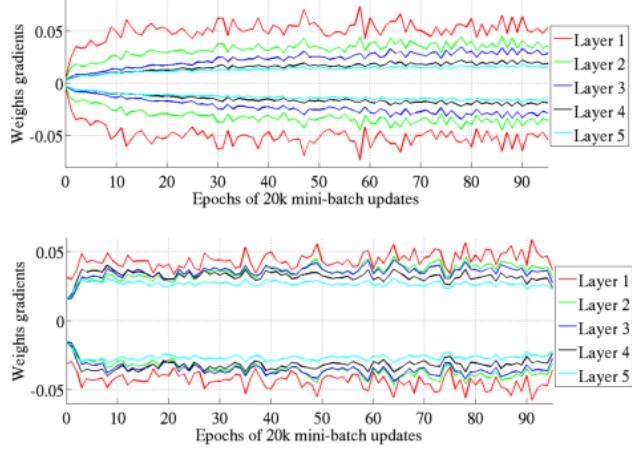


Abbildung 3.4: Vergleich der Varianz der Gradienten im Laufe des Trainings (Standard-Initialisierung (oben) und Xavier-Initialisierung (unten)): *Layer 1* entspricht dem Output-Layer mit der größten Varianz im Gradient (siehe Glorot and Bengio (2010))

bleibt (vgl. z.B. Rojas, 1996, S. 199 ff.), steht heute die sogenannte *Xavier-Initialization* (vgl. im Folgenden Glorot and Bengio (2010)). Dies führt dazu, dass die Varianz im Gradienten in jeder Schicht in etwa gleich groß ist, was aus Abbildung 3.4 zu entnehmen ist. Die *Xavier-Initialization* kombiniert beide Aspekte und initialisiert die Gewichte in der Art, dass die Varianz des Signals von Schicht zu Schicht sowohl im *Forward Pass* als auch im *Backward Pass* erhalten bleibt. Dies wird durch Formel 3.4 erreicht, was letztlich der Anwendung der Varianz-Formel für Gleichverteilung mit  $Var(W) = \frac{2}{fan_{in} + fan_{out}}$  entspricht.

$$W \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}\right] \quad (3.4)$$

Für den Fall, dass ReLu-Funktionen eingesetzt werden, verallgemeinern He et al. (2015) diese Formel insofern, dass eine lineare Approximation um 0 für diese und davon abgeleitete Aktivierungsfunktionen nicht mehr gültig ist. Formel 3.5 entspricht der Verallgemeinerung für Funktionen dieser Art.

$$W \sim \mathcal{N}(0, \sqrt{\frac{2}{fan_{out}}}) \quad (3.5)$$

Mit der Xavier-Initialisierung werden in Folge häufig bessere Ergebnisse erreicht, als mit der Standard-Initialisierung (*Random Initialization*) in Verbindung mit Informationen über die Krümmung (Approximation der Hesse-Matrix - siehe Kapitel 3.3.2) (vgl. Chapelle and Erhan (2011) und Glorot and Bengio (2010)). Für Convolutional Neural Networks berechnen sich

$fan_{in}$  und  $fan_{out}$  durch Multiplikation der Filtergröße mit der Anzahl *Input-Maps* respektive *Feature-Maps*  $c$ :  $k_w \cdot k_h \cdot c$  (vgl. He et al., 2015).

### 3.2.2 Unüberwachtes Vortraining

Dieses Kapitel weicht die in Kapitel 1 gemachte Einschränkung auf das klassische überwachte, neuronale Lernmodell insofern auf, als dass ein unüberwachtes Lernverfahren vorgestellt wird. Dieses Vorgehen dient jedoch zum einen dem höheren Ziel, das überwachte Verfahren durch eine bessere Initialisierung der Gewichte besser zu konditionieren und in der Performance zu verbessern. Zum anderen wird sich zeigen, dass es sich bei genauerer Betrachtung nicht um ein klassisches unüberwachtes Lernverfahren handelt: Es wird lediglich ein unüberwachtes Problem als überwachtes Problem angesehen. Dies lässt sich am einfachen Beispiel eines nichtlinearen Autoencoders verdeutlichen (vgl. im Folgenden Masci et al., 2011). Der Encoder in Formel 3.6 nimmt als Eingabe einen Vektor  $x \in \mathcal{R}^d$  und berechnet einen Code  $h \in \mathcal{R}^{d'}$ .

$$h = \phi(Wx + b) \quad (3.6)$$

Dieser Code wird im zweiten Schritt durch den Decoder in Formel 3.7 dekodiert und so die Ausgabe berechnet. Das Ziel des Autoencoders ist es den Fehler zwischen Eingabe und Rekonstruktion, wie z.B.  $MSE = \mathbb{E}[(x - y)^2]$ , durch Anpassen der Gewichte  $\theta = W, b$ , wobei  $W' = W^T$  (*Tied Weights*), zu minimieren.

$$y = \phi(W'h + b') \quad (3.7)$$

Der Autoencoder prädiziert folglich die Eingabe selbst und lernt die sogenannte Identität (*Identity Mapping*). Die Ideen von Hinton et al. (2006), die gleichzeitig auch den Beginn der Renaissance von Deep Learning darstellen, beschreiben ein Verfahren große Netzwerke mit vielen Schichten schichtweise *greedy* mit beschränkten Boltzmann Maschinen (RBMs) zu trainieren (vgl. Bengio et al. (2007)). Das bedeutet, dass jede Schicht als Eingabe die Ausgabe der vorherigen Schicht erhält und versucht, diese wiederum zu prädizieren (vgl. Ranzato et al. (2006)). Dies wirkt ähnlich wie ein Regularisierer und initialisiert die Gewichte hinsichtlich der Optimierung und Generalisierung in einer besseren Ausgangssituation (vgl. Erhan et al., 2010). Hiervon profitieren besonders große MLPs, welche bis dato aufgrund des *Vanishing Gradient*-Effekts als schwer zu trainieren galten.

Eine wichtige Weiterentwicklung des Konzepts sind sogenannte *Denoising Autoencoder* (DAs) beziehungweise *Stacked Denoising Autoencoder* (SDAs), welche versuchen die Eingabe  $x$  ausgehend von einer korrumptierten Version  $\hat{x}$  zu prädizieren. Dies erlaubt das Erlernen robuster Merkmale und kann



Abbildung 3.5: Filter der ersten Schicht eines Convolutional Autoencoders  
(a) Kein Max-Pooling, 0 % Rauschen, (b) Kein Max-Pooling, 50 % Rauschen,  
(c) Max-Pooling 2x2, (d) Max-Pooling 2x2, 30 % Rauschen (siehe  
Masci et al. (2011))

als diskriminative Alternative zum generativen stochastischen RBM-Modell gesehen werden (vgl. Vincent et al. (2008)).

Convolutional Neural Networks wie das *LeNet 5* von LeCun et al. (1998a) können aufgrund ihrer besonderen Struktur dennoch trainiert werden. Trotzdem ist es auch bei CNNs schwierig die ersten Schichten zu trainieren, da die hinteren Schichten generell schneller unscheinbares *Overfitting* mit entsprechend kleinem Gradienten aufweisen (vgl. Erhan et al. (2010)). Der erste Versuch unüberwachtes Vortraining auf CNNs anzuwenden stammt von Ranzato et al. (2006). Hier wird die erste Schicht eines veränderten *LeNet 5* mittels Vortraining initialisiert, was den Fehler auf dem MNIST-Datensatz<sup>1</sup> von 0.7% auf 0.6% verringert. Das Vortraining der  $5 \times 5$ -Filter geschieht mit zufällig gewählten  $5 \times 5$ -Bildausschnitten auf der Basis von RBMs.

### Convolutional Autoencoder

Es existieren verschiedenste Architekturvorschläge für Autoencoder und RBMs zum Vortraining von CNNs. Die Grundlage hierfür ist meist die Extraktion den Filter entsprechend großer Bildausschnitte und das Trainieren gewohnter Modelle (vgl. Desjardins and Bengio (2008), Ranzato et al. (2007b) oder Lee et al. (2009)). Im Folgenden wird der Convolutional Autoencoder von Masci et al. (2011) vorgestellt, da dieser auf Faltungen basiert und, wie Abbildung 3.5 zeigt, die gewünschten translationsinvarianten Filter erzeugt. Die vorgestellte Variante verbessert den Fehler auf dem MNIST-Datensatz von 0.79% auf 0.71%.

Die Architektur des vorgestellten Convolutional Autoencoders besteht aus einem Convolution-Layer sowie einem nachgeschalteten Pooling-Layer. Das bedeutet, dass sich der gesamte Autoencoder letztendlich aus zwei hintereinander ausgeführte Layern dieser Art zusammensetzt.

---

<sup>1</sup>Der MNIST-Datensatz handgeschriebener Ziffern besteht aus 60,000 Trainingsbeispielen und 10,000 Testbeispielen. Es stellt eine Untermenge des größeren NIST-Datensatzes dar. Die Bilder sind  $28 \times 28$  und die Ziffern zentriert (<http://yann.lecun.com/exdb/mnist/> (26.08.2015)).

Gleichung 3.8 zeigt die Berechnung des Codes  $h_i$  der  $i$ -ten *Feature-Map*. Dieser wird wie in gewöhnlichen CNNs durch Faltung mit Randbehandlung *valid* berechnet.

$$h_i = \phi\left(\sum_{j=0}^m x_j * W_{ij} + b_i\right) \quad (3.8)$$

Der Decoder berechnet die Funktion 3.9, wobei ähnlich dem *Backward Pass* im Backpropagation die Filtermaske über beide Seiten geflippt beziehungsweise um  $180^\circ$  gedreht und die Randbehandlung *full* verwendet wird. Die Dekodierung ist beeinflusst von gewöhnlichen Autoencodern mit *Tied Weights*, bei denen als Dekodierungsfunktion ebenfalls die transponierte Gewichtsmatrix des *Forward Pass* verwendet wird.

$$y_j = \phi\left(\sum_{i=0}^n h_i * \text{rot180}(W_{ij}) + c_j\right) \quad (3.9)$$

Der Gradient für den Autoencoder berechnet sich mit Formel 3.10. Diese setzt sich, wie der Autoencoder, aus zwei Teilen zusammen. Der erste Teil wird für den Encoder, der zweite für den Decoder benötigt. Im Unterschied zur Berechnung des Gradienten beim CNN wird anstatt  $\delta y$  der Code  $h$  der Flip-Operation unterzogen. Dies tritt auf, da der *Backward Pass* dem eigentlichen *Forward Pass* entspricht.<sup>2</sup> Eine zusätzliche Besonderheit besteht in Zusammenhang mit der Randbehandlung *valid*. Um den Code  $\text{rot180}(h)$  mit dem räumlich größeren  $\delta y$  der Ausgabe zu falten, muss dieser doppelt mit Nullen erweitert werden (*Padding*). Zum einen um die gleiche Größe wie die Ausgabe zu erhalten und zum anderen um die Faltung zu ermöglichen.

$$\frac{\partial J(W, b, c)}{\partial W_{ij}^l} = \frac{1}{M} \sum_{m=1}^M \text{rot180}(x_{mj}^l * \text{rot180}(\delta h_{mi}^l) + \text{rot180}(h_{mi}^l) * \delta y_{mj}^l) \quad (3.10)$$

Analog zum bekannten Convolution-Layer berechnen sich die beiden Schwellwerte  $b$  und  $c$  aus der Summe der zugehörigen Deltas  $\delta y$  und  $\delta h$ , wie in Gleichung 3.11 und 3.12 dargestellt.

$$\frac{\partial J(W, b, c)}{\partial b_i^l} = \frac{1}{M} \sum_{m=1}^M \sum_{u=0}^{k_w} \sum_{v=0}^{k_w} \delta h_{miuv}^l \quad (3.11)$$

$$\frac{\partial J(W, b, c)}{\partial c_j^l} = \frac{1}{M} \sum_{m=1}^M \sum_{u=0}^{k_w} \sum_{v=0}^{k_w} \delta y_{miuv}^l \quad (3.12)$$

---

<sup>2</sup>Im Unterschied zur von Masci et al. (2011) beschriebenen Berechnung wird in Konsequenz auch auf die Flip-Operation von  $\delta h$  nicht verzichtet und der Gradient am Ende gesamtheitlich geflippt. Der Gradient für den Encoder wird damit äquivalent zu dem in einem CNN berechnet. In Kapitel ?? wird die Richtigkeit dieser Änderung überprüft.

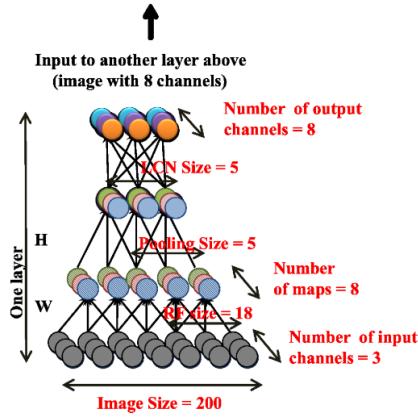


Abbildung 3.6: Autoencoder mit lokalen rezeptiven Feldern aber ohne *Parameter Sharing* (siehe Le et al. (2012))

Weiterentwicklungen wie die *Tiled CNNs* versuchen weitere Invarianzen, wie beispielsweise die Rotationsinvarianz, zu lernen (vgl. Le et al. (2010)). Einer der größten Autoencoder, der *Google Autoencoder* bedient sich ebenfalls der lokalen rezeptiven Felder, verzichtet allerdings auf *Parameter Sharing*, um verschiedene Invarianzen zu lernen (vgl. Le et al. (2012)). Eine Schicht dieser Architektur ist in Abbildung 3.6 abgebildet.

Auch wenn das Verwenden von unüberwachtem Vortraining für die ersten Schichten eines CNNs passend erscheinen mag, führt es nicht zu einer solchen Verbesserung der Ergebnisse wie bei tiefen MLPs (DNNs) (vgl. Abdel-Hamid et al. (2013)). Durch die Entwicklung neuer Aktivierungsfunktionen wie ReLu und raffinierter Initialisierung rückt das Thema somit in solche Bereiche, in denen wenig gelabelte Trainingsdaten zur Verfügung stehen, um das CNN vollständig überwacht zu trainieren (vgl. Masci et al. (2011) und Le et al. (2012)). Darüber hinaus können auch verbesserte Optimierungsverfahren das unüberwachte Vortraining hinsichtlich einer besseren Initialisierung ersetzen (vgl. Martens (2010) und Sutskever et al. (2013)).

Eine neue Herangehensweise ist es, bereits trainierte CNNs für andere Probleme zu übernehmen, was als sogenanntes Transferlernen bezeichnet wird (vgl. Wagner et al. (2013)). So liefert zum Beispiel die 7. Schicht des *AlexNet* von Krizhevsky et al. (2012) bereits sehr gute Bildbeschreibungen (vgl. Bell and Bala (2015)).

### 3.3 Gradientenabstieg

Dieses Kapitel behandelt verschiedene Methoden zur Verbesserung des Gradientenabstiegs. Diese können das Training erheblich beschleunigen, da es sich im Allgemeinen im Deep Learning um nicht-konvexe Fehlerlandschaften

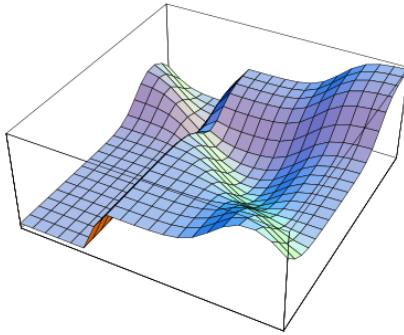


Abbildung 3.7: Fehlerfunktion beziehungsweise Zielfunktion im Gewichtsraum mit lokalem Minimum (siehe Rojas, 1996, S. 155)

wie in Abbildung 3.7 handelt.

Die Grundlage für das effiziente Lernen mit vielen Trainingsdaten ist der sogenannte stochastische Gradientenabstieg (SGD) (vgl. im Folgenden Bottou (1998)). Hierbei geht es um die Frage, was die erwartete durchschnittliche Richtung des Gradienten ist. Durch die Einführung des MSE-Fehlermaßes ist der Ansatz des SGD bereits gegeben. So entspricht die MSE-Fehlerfunktion dem Erwartungswert der einzelnen Fehlerquadrate. Wie in Gleichung 3.13 zu sehen ist, entspricht analog dazu der Erwartungswert der einzelnen Gradienten dem Gradient der Fehlerfunktion.

$$\nabla J(W, b) = \mathbb{E}[\nabla(f(x_i) - y_i)^2] = \frac{1}{N} \sum_i^N \nabla(f(x_i) - y_i)^2 \quad (3.13)$$

Für eine allgemeine Fehlerfunktion  $e(x)$  gilt: Nimmt man an Stelle der gesamten Trainingsmenge  $N$  (Batch) einzelne Trainingsbeispiele  $i$  (Online) oder eine Teilmenge  $M$  (Averaged SGD), so ergibt sich Formel 3.14 für den Gradientenabstieg mit Lernrate beziehungsweise Schrittweite  $\eta$ . Der Gradient zeigt somit in Richtung des Durchschnitts und damit in Richtung des Erwartungswerts.

Neben einer effizienteren Berechnung des Gradientenabstiegs, liefert SGD eine Zufallskomponente (*Randomization*). Diese kann zu beschleunigter Konvergenz führen, da durch variierende (*noisy*) Gewichtsupdates ein größeres Gebiet untersucht werden kann. Außerdem erlaubt SGD Training ohne unmittelbarem Zugriff auf die gesamte Trainingsmenge. Üblicherweise werden die für die Berechnung entnommenen Teilmengen als *Mini-Batch* bezeichnet (Bengio, 2012, vgl.).

$$W_{t+1} = W_t - \eta \frac{1}{M} \sum_{i=1}^M \nabla e(x_i) \quad (3.14)$$

SGD bildet die Grundlage für viele Erweiterungen und stellt so die erste Optimierung des Gradientenabstiegs dar. Um die Formeln zu vereinfachen, wird im Folgenden auf die sonst übliche Mittelung des Gradienten verzichtet. An den entsprechenden Stellen wird darauf hingewiesen, ob es sich um die Erweiterung für den Online-/Mini-Batch-Modus oder um den klassischen Batch-Modus, welcher für Deep Learning in Verbindung mit vielen Trainingsdaten unpraktikabel ist, handelt.

### Lernrate und Mini-Batch-Größe

Wird der klassische Gradientenabstieg ohne Optimierungen verwendet, ist es besonders wichtig  $\eta$  richtig zu wählen. Aufgrund des *Vanishing Gradients* sollte die Lernrate in den letzten Schichten kleiner gewählt werden. Außerdem ist darauf zu achten, dass die Lernrate durch *Parameter Sharing* proportional zur Wurzel der Verbindungen ist, die sich ein Gewicht teilen. Letzteres ist gerade bei CNNs zu beachten, um eine unnötig schlechte Konditionierung der Fehlerlandschaft zu vermeiden (vgl. LeCun et al. (1998b)). Die Bestimmung der Mini-Batch-Größe  $M$  erlaubt es zwischen Online- und Batch-Training zu interpolieren. Dies ist wichtig, da die richtige Wahl der Größe von den Trainingsdaten abhängt und nicht allgemein vorhergesagt werden kann. So kann beispielsweise Online-Training in vielen Situationen schneller sein als Batch-Training. Als Stellgröße kann der Grad an Redundanz in den Trainingsdaten dienen. Mit steigender Redundanz sollten die Batches entsprechend kleiner gewählt werden, um nicht unnötig Rechenzeit zu verbrauchen (vgl. Wilson and Martinez, 2003). Bengio (2012) nennt  $M = 32$  einen guten Standardwert.

#### 3.3.1 Momentum

Eine der ersten erfolgreichen Erweiterungen für iterative Verfahren ist die sogenannte Momentum Methode von Polyak (1964). Die Momentum-Methode ist durch die Physik motiviert, dadurch dass dem Gradienten eine potentielle Energie zugewiesen wird. Dies erlaubt es in gleichbleibender Richtung des Gradienten Geschwindigkeit aufzunehmen. Dies führt letztendlich dazu, dass in flachen Regionen der Fehlerlandschaft größere und in unwegsamen Bereichen kleinere Schrittweiten effektiv ausgeführt werden (vgl. LeCun et al. (1998b)). Dies wird durch das Hinzufügen eines weiteren Terms in das Gewichtsupdate erzielt. Wie Formeln 3.15 und 3.16 zeigen, bildet die Momentum-Methode einen gewichteten Durchschnitt der vergangenen Gradienten der Gewichte.

$$\Delta W_t = \mu \Delta W_{t-1} - \eta \nabla J(W_t, b_t) \quad (3.15)$$

$$W_{t+1} = W_t + \Delta W_t \quad (3.16)$$

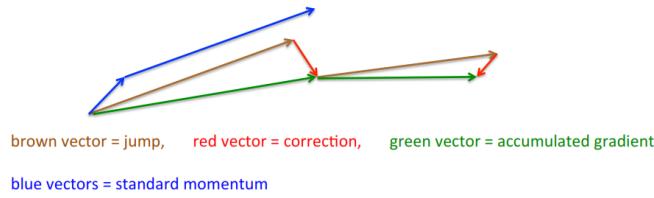


Abbildung 3.8: Das effektivere Nesterov-Momentum korrigiert Fehler im Nachhinein (siehe Hinton et al., 2015)

Typische Werte für die Momentum-Rate  $\mu$  sind Werte größer 0.9. Diese sind allerdings abhängig vom Lernproblem und können nicht allgemein formuliert werden (vgl. Fei-Fei and Karpathy (2014)). Die Momentum-Methode ist eine sehr einfache Optimierung für SGD und findet in erfolgreichen Netzen häufig Anwendung (vgl. Krizhevsky et al. (2012)).

### Nesterov-Methode

Die Nesterov-Methode (NAG) ist eine verbesserte Momentum-Methode, die von Nesterov (1983) beeinflusst ist (vgl. im Folgenden Sutskever, 2013). Diese Methode führt zunächst den über die vergangenen Perioden gewichteten Momentum-Term aus und berechnet dann den Gradient an dieser Stelle. Die Methode fungiert somit als eine Art Ausblick auf die zukünftige Fehlerlandschaft. Der eigentliche Gradient ergibt sich schließlich aus Ausblick und Korrektur, wie Abbildung 3.8 zeigt.

Da diese Ausführung die Reihenfolge des üblichen Trainings beeinflussen würde, wird die Methode derart umgeschrieben, dass die aktuellen Gewichte immer dem Ausblick entsprechen. Daraus ergeben sich die Regeln 3.17 und 3.18 für die Aktualisierung der Gewichte (vgl. Fei-Fei and Karpathy (2014)).

$$\Delta W_t = \mu \Delta W_{t-1} - \eta \nabla J(W_t, b_t) \quad (3.17)$$

$$W_{t+1} = W_t + (-\mu \Delta W_{t-1}) + (1 + \mu) \Delta W_t \quad (3.18)$$

Die beschriebene Methode ist robuster hinsichtlich der Momentum-Rate und lässt höhere Raten zwischen 0.95 und 0.99 zu. Der NAG wird großes Potenzial zugesprochen, das klassische Momentum möglicherweise dauerhaft zu ersetzen (vgl. Sutskever et al. (2013)).

### 3.3.2 Adaptive Lernrate

Ein allgemeines Problem in Verbindung von Gradientenabstieg und dem Training im Deep Learning ist der *Vanishing Gradient*-Effekt Hochreiter (1991). Ein Algorithmus der direkt auf diesen Effekt abzielt, ist der sogenannte Resilient Propagation, auch Rprop genannt (vgl. Riedmiller and

Braun (1992) und Igel and Hüskens (2000)). Rprop weist jedem Gewicht eine eigene Lernrate zu und verzichtet gänzlich auf die Werte des Gradienten. Die grundlegende Arbeitsweise sieht vor, bei gleichbleibendem Vorzeichen der partiellen Ableitung eines Gewichts die zugehörige Lernrate zu erhöhen. Tritt ein Vorzeichenwechsel auf, so liegt es nahe, dass in der Richtung dieser Ableitung eine Senke übersprungen wurde und die Lernrate wird reduziert. Rprop ist äquivalent zum Gradientenabstieg im Batch-Modus, wenn bei letzterem durch die Länge des Gradienten geteilt wird. Hier liegt ein fundamentales Problem hinsichtlich Deep Learning: Um die Länge des Gradients korrekt zu schätzen, bedarf es der gesamten Trainingsmenge, was die Verwendung von SGD ausschließt. Somit ist Rprop nur für Batch-Training geeignet (vgl. Hinton et al. (2015)).

Der bekannte Newton-Algorithmus für Optimierung in Gleichung 3.19 multipliziert nicht mit einer globalen Lernrate, sondern mit der Inversen der Hesse-Matrix. Unter bestimmten Annahmen, wie etwa einer quadratischen Zielfunktion, erreicht dieser Algorithmus innerhalb eines Schrittes das Minimum, was als Newton-Schritt bezeichnet wird (vgl. Bottou (1998)).

$$W_{t+1} = W_t - H_t^{-1} \nabla J(W_t, b_t) \quad (3.19)$$

Die Hesse-Matrix ist im Allgemeinen sehr teuer zu berechnen, da jede partielle Ableitung der  $N$  Gewichte nochmals abgeleitet werden müsste. Dies führt auf eine  $N \times N$ -Matrix, welche in neuronalen Netzen nicht effizient zu berechnen ist und deshalb approximiert werden muss (vgl. LeCun et al., 1998b).

Algorithmen mit sogenannter adaptiver Lernrate gehen so vor, dass sie lediglich die Diagonale der Hesse-Matrix  $\text{diag}(H)$  approximieren. Dies führt zu eigenen Lernraten pro Gewicht, welche idealerweise den echten Eigenwerten der Hesse-Matrix entsprechen (vgl. LeCun et al., 1998b). Adaptive Lernraten erlauben es, Schrittweiten in Gegenden schwacher Krümmung zu erhöhen und entsprechend in anderen zu verkürzen. Dieser Mechanismus wirkt damit ebenso unmittelbar dem *Vanishing Gradient*-Effekt entgegen, da kleiner werdende Gradienten zu den ersten Schichten hin skaliert werden (vgl. Martens, 2010). Problematisch bei der Verwendung der Hesse-Matrix für nicht-konvexe Probleme sind negative Eigenwerte an lokalen Maxima oder gemischt positive und negative Eigenwerte an Sattelpunkten, die zu einer falschen Richtung des Gradienten führen (vgl. Dauphin et al., 2014). Aus diesem Grund ist es notwendig, eine Methode mit entsprechender Vorkonditionierung zu verwenden (vgl. Dauphin et al., 2015).

### Stochastic Levenberg-Marquardt

Der bekannte stochastische Levenberg-Marquardt-Algorithmus (LMA) von LeCun et al. (1998b) ist ein Algorithmus im Bereich Neuronale Netze und SGD, der versucht die  $\text{diag}(H)$  zu approximieren. Hierzu sind allerdings

zwei weitere Approximationen notwendig. So wird die Gauß-Newton Approximation zur Vermeidung negativer Eigenwerte, wie im klassischen LMA-Algorithmus, angewandt und die  $\text{diag}(H)$  nur auf einer kleinen Teilmenge der Trainingsdaten (Mini-Batch) berechnet. Darüber hinaus wird die Berechnung nur einmal pro Epoche durchgeführt. Die Kosten entsprechen denen eines normalen *Forward Pass* sowie einem angepassten *Backward Pass*, der zur Rückpropagierung der zweiten Ableitungen dient, und sind somit zu vernachlässigen. Neben der Kompensation des *Vanishing-Gradients*, kompensiert der stochastische LMA den Effekt der schlechten Konditionierung der Fehlerlandschaft durch *Parameter Sharing* (vgl. LeCun et al., 1998b). Die neue Lernrate pro Gewicht  $i$  ergibt sich mit Formel 3.20.

$$\eta_i = \frac{\eta}{\mu + H_{ii}} \quad (3.20)$$

Wie im klassischen LMA wird ein Dämpfungsfaktor  $\mu$  verwendet, um zu verhindern, dass die Schrittweite zu groß wird. Dieser Faktor kann auch als Vorannahme gesehen werden, die angibt, dass sich das Gewicht nicht ändern soll, wenn die Krümmung entsprechend klein wird (vgl. Martens (2010)). Im *LeNet 5* wird  $\mu = 0.02$  verwendet.

### Equilibrium SGD

Eine Verbesserung zum stochastischen LMA ist die von Dauphin et al. (2015) beschriebene Equilibrium-Methode, welche die Matrix  $D^{Eq} = \sqrt{\text{diag}(H^2)}$  berechnet. Diese Vorkonditionierung bietet einige Vorteile hinsichtlich der Behandlung der genannten indefiniten Hesse-Matrizen, die durch Sattelpunkten innerhalb der Fehlerlandschaft von MLPs auftreten. Darüber hinaus kann diese Methode effizient durch den Zusammenhang in Formel 3.21 mit  $v \sim \mathcal{N}(0, 1)$ , beispielsweise mit dem  $R\{\cdot\}$ -Operator (vgl. Pearlmutter (1994)), berechnet werden.

$$D^{Eq} = \text{diag}(H^2) = E[(Hv)^2] \quad (3.21)$$

Im Unterschied zum stochastischen LMA bildet die Equilibrium-Methode, wie Formel 3.22 zeigt, einen *Root Mean Square*-Durchschnitt (RMS) über die vergangenen Perioden. Dauphin et al. (2015) schlagen eine Neuberechnung von  $D^{eq}$  nach jeweils 20 Iterationen und  $\mu \in [10^{-4}, 10^{-6}]$  vor.

$$D_{t+1}^{eq} = \rho D_t^{eq} + (1 - \rho)(Hv)^2 \quad (3.22)$$

Damit ergibt sich die in Formel 3.23 gezeigte effektive Lernrate, welche ebenfalls einen Dämpfungsfaktor  $\mu$  zur Vermeidung großer Schrittweiten beinhaltet.

$$\eta_i = \frac{\eta}{\mu + \sqrt{D_{ii}^{eq}}} \quad (3.23)$$

## RMSprop

Der RMSprop-Algorithmus stammt von Hinton et al. (2015) und kombiniert zwei Ideen. Einerseits die Idee des Rprop nur auf das Vorzeichen des Gradienten zu achten, indem er den *Root Mean Square*-Durchschnitt (RMS) der partiellen Ableitungen über mehrere Zeitschritte berechnet. Andererseits wird das Hauptproblem von AdaGrad der stetig kleiner werdenden effektiven Lernraten dadurch vermieden (vgl. Bengio et al., 2015, Kap. 8.4.1, S. 257). Die Formel 3.24 beschreibt die Berechnung des RMS. Die effektive Lernrate berechnet sich mit der Formel 3.25, wobei ebenfalls ein Dämpfungsfaktor  $\mu$  verwendet wird.

$$\hat{H}_{t+1} = \rho \hat{H}_t + (1 - \rho) \nabla J(W_t, b_t)^2 \quad (3.24)$$

$$\eta_i = \frac{\eta}{\mu + \sqrt{\hat{H}_{ii}}} \quad (3.25)$$

Eine sehr interessante Eigenschaft von RMSprop ist die Fähigkeit zur Approximation der Equilibrium-Matrix  $D^{Eq} = \sqrt{diag(H^2)}$  (vgl. Dauphin et al. (2015)). RMSprop stellt eine sehr effektive und praktische Optimierung dar, welche dazu einfach zu implementieren ist. Dies macht diese derzeit sehr beliebt (vgl. Bengio et al. (2015)).

## AdaDelta

AdaDelta ist ein von Zeiler (2012) entwickeltes Verfahren, welches ebenfalls das Problem der immer kleiner werdenden Lernrate von AdaGrad adressiert (vgl. Bengio et al. (2015)). Darüber hinaus versucht AdaDelta aus der ersten Ableitung die Krümmung der Funktion (*Second Order Information*) zu berechnen. Der Divisor in Formel 3.26 entspricht, bis auf dem unter der Wurzel stehenden  $\mu$ , dem des RMSprop aus Formel 3.24. Eine Neuheit stellt der RMS über den Gewichtsupdates  $RMS[\Delta x]$ , der einen Zeitschritt versetzt berechnet wird und als eine Art Momentum wirkt. Insgesamt schätzt die AdaDelta-Methode die absolute Hesse-Matrix  $|diag(H)|$  aus Gradient und Gewichtsupdate (vgl. Zeiler, 2012).

$$\Delta x_t = \frac{\sqrt{\mu + RMS[\Delta x]_{t-1}^2}}{\sqrt{\mu + \hat{H}_t}} \quad (3.26)$$

Die effektive Lernrate ergibt sich aus Formel 3.27.

$$\eta_i = \Delta x_i \quad (3.27)$$

Eine Besonderheit von AdaDelta ist, dass gänzlich auf eine globale Lernrate verzichtet und lediglich  $\mu$  als Hyperparameter benötigt wird. Da sich  $\mu$  unter

der Wurzel befindet ist dieses entsprechend kleiner als bei den vorherigen Methoden. Zeiler (2012) schlägt hierbei Werte bis  $\mu = 10^{-8}$  vor.

### 3.3.3 Hessian free optimazation (HF)

Algorithmen mit adaptiver Lernrate vernachlässigen, im Sinne der Approximation, alle nicht-diagonalen Werte der Hesse-Matrix. Damit wird ein gewisser Fehler akzeptiert, da diese Werte gerade die Interaktion der Gewichte hinsichtlich der Zielfunktion beschreiben (vgl. Martens (2010)).

Die *Hessian free optimazation* (HF) von Martens (2010) versucht, im Gegensatz zu den vorherigen Methoden, nicht den *Vanishing Gradient*-Effekt zu eliminieren oder Plateaus zu überwinden, sondern zielt direkt auf die Optimierung von ungünstigen Fehlerlandschaften ab *Pathological Curvature*. Für eine gegebene Stelle  $w_t$  im Gewichtsraum wird ein lokales quadratisches Modell angenommen und dieses mittels Conjugate Gradient (CG) optimiert. Deshalb kann HF nur im *Batch*-Modus betrieben oder, analog zum stochastischen LMA, mit einer hinreichend großen Teilmenge der Trainingsdaten. Eine Besonderheit dieser Methode ist, dass der gefundene Vektor von CG im letzten Zeitschritt als Initialisierung für den aktuellen Zeitschritt gewählt werden und dadurch Informationen über mehrere Iterationen weitergeben kann.

Unterzieht man diese Methode einer kritischen Betrachtung, kann festgestellt werden, dass sie, aufgrund der Verwendung des linearen CG, keine Methode 2. Ordnung und daher den anderen Methoden nicht allgemein überlegen ist. So zeigt Sutskever et al. (2013), dass die vorgestellte HF-Methode sehr viele Ähnlichkeiten zum Nesterov-Momentum aufweist und zu diesem unter der Annahme, dass nur ein Schritt im CG-Algorithmus ausgeführt wird, sogar äquivalent ist. Die interessante Eigenschaft der beiden Methoden HF und NAG, ist die Möglichkeit die Schrittweiten in Bereichen mit geringer Krümmung zu erhöhen und in Bereichen großer Krümmung entsprechend zu verringern. Hauptsächlich aus diesem Grund wird der NAG-Methode gerade im Bereich tiefer Netze hohes Potenzial zugesprochen (vgl. Sutskever et al. (2013)).

## 3.4 Regularisierung und Generalisierung

Lernprobleme selbst lassen sich beispielsweise mit bayesianischer Inferenz anschaulich beschreiben (vgl. im Folgenden Mitchell, 1997, S. 159f).

$$P(h|\mathcal{D}) = \frac{P(\mathcal{D}|h)P(h)}{P(\mathcal{D})} \quad (3.28)$$

So beschreibt die Formel 3.28 den Zusammenhang zwischen gegebenen Trainingsdaten  $\mathcal{D}$  und der Hypothese  $h$ , welche den zu erlernenden Parametern

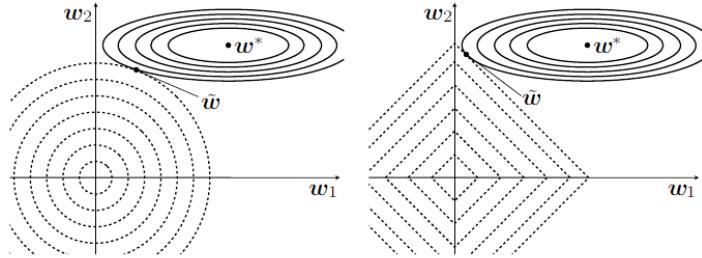


Abbildung 3.9: Effekt der  $L^2$ - (links) und  $L^1$ -Norm (rechts) auf den Wert der optimalen Gewichte  $W$  mit empirischem Optimum  $w^*$  (vgl. Bengio et al., 2015, Kap. 7.2, S. 199)

$\Theta = \{W, b\}$  entspricht. Sie soll die gegebenen Daten  $\mathcal{D}$  möglichst gut abbilden können sowie eine gute Generalisierung für unbekannte Daten leisten. In der bayesianischen Interpretation korrespondieren Regularisierer mit der A-priori Verteilung der Hypothese  $h$  (vgl. im Folgenden Bengio et al., 2015, Kap. 7, S. 196).

Im Deep Learning existieren verschiedenste Methoden der Regularisierung, welche auf eine bessere Generalisierung abzielen. Manche betreffen die Parameter selbst, andere codieren Expertenwissen oder regulieren die Kapazität von Modellen mit sehr vielen freien Parametern. Im Umfeld neuronaler Netze betrifft die Regularisierung von Parametern meist lediglich die Gewichte  $W$ , der Schwellwert  $b$  bleibt unbeachtet.

### 3.4.1 A-priori Annahmen

Die A-priori Verteilung der Parameter  $\Theta$  gibt an wie wahrscheinlich verschiedene Werte für die einzelnen Parameter sind. Die klassischen Methoden des maschinellen Lernens sowie der Statistik bestrafen die Norm der Parameter (*Parameter Norm Penalty*) und vermeiden so große Werte dieser. Die Norm wird als  $\Omega(\Theta)$  definiert und zur Zielfunktion, wie in Gleichung 3.29, gewichtet mit dem Hyperparameter  $\alpha$  addiert (vgl. Bengio et al., 2015, Kap. 7.2, S. 200).

$$\hat{J}(W, b) = J(W, b) + \alpha\Omega(\Theta) \quad (3.29)$$

Typische Normen sind die  $L^1$ - und  $L^2$ -Norm. Abbildung 3.9 stellt die beiden Varianten gegenüber.

#### $L^1$ -Norm

Bei der Verwendung der  $L^1$ -Norm zur Berechnung von  $\Omega\Theta$  bleibt nach dem Ableiten lediglich das Vorzeichen  $sign(w_i)$  der einzelnen Gewichte übrig.

Damit ergibt sich die in Gleichung 3.30 dargestellte Regel für die Berechnung des neuen Gradienten  $\nabla_w \hat{J}(W, b)$  pro Gewicht  $w$ .

$$\nabla_w \hat{J}(W, b) = \nabla_w J(W, b) + \alpha \text{sign}(w) \quad (3.30)$$

Die  $L^1$ -Regularisierung bestraft die Summe aller Absolutwerte der Gewichte und verkleinert diese somit während des Gradientenabstiegs stetig mit gleicher Rate. Die rechte Grafik in Abbildung 3.9 zeigt, dass  $L^1$ -Regularisierung einerseits zu größeren Werten führt, im Gegenzug jedoch auch sehr kleine Werte bevorzugt. Diese Eigenschaft führt zu *Sparsity*, da kleine Gewichte  $w$  zwangsläufig im Laufe des Trainings den Wert Null annehmen (vgl. Bengio et al., 2015, Kap. 7.2, S. 203). Die bayesianische Interpretation der Methode ist die A-priori Annahme einer isotropischen Laplace-Verteilung der Gewichte (vgl. Bengio et al., 2015, Kap. 7.2, S. 206).

### **$L^2$ -Norm**

Trotz der interessanten *Sparsity* Eigenschaften der  $L^1$ -Regularisierung wird im Deep Learning meist die  $L^2$ -Norm zur Regularisierung verwendet und die *Sparsity* mit ReLu-Funktionen erzwungen (vgl. z.B. Krizhevsky et al., 2012). Diese aus der *Ridge Regression* bekannte Form der Regularisierung, führt zu Parameterwerten näher dem Ursprung. In der bayesianische Interpretation entspricht diese somit einer Gauss-Verteilung mit Mittelwert Null (vgl. Bengio et al., 2015, Kap. 7.2, S. 200). Der Gradient der  $L^2$  regularisierten Fehlerfunktion ist in Gleichung 3.31 aufgeführt.

$$\nabla_w \hat{J}(W, b) = \nabla_w J(W, b) + \alpha w \quad (3.31)$$

Diese Form der Regularisierung führt dazu, dass das Modell eine größere Varianz in den Trainingsdaten  $X$  annimmt. Somit werden Gewichte zu Merkmalen mit geringer Kovarianz zur Ausgabe verkleinert (vgl. Bengio et al., 2015, Kap. 7.2, S. 200 f.). Die *Ridge Regression* in Gleichung 3.32 verdeutlicht diesen Aspekt, indem ein Vielfaches der Identitätsmatrix  $I$  auf die Kovarianzmatrix  $X^T X$  addiert wird.

$$w = (X^T X + \alpha I)^{-1} X^T y \quad (3.32)$$

In Abbildung 3.9 erkennt man diesen Effekt daran, dass das Gewicht  $w_1$ , in dessen Richtung die Zielfunktion im Gewichtsraum eher flach ist, geschrumpft wird.

### **Max- $L^2$ -Norm**

Max- $L^2$ -Norm Regularisierung beschreibt eine Methode, welche die  $L^2$ -Norm der Gewichte eines Neurons  $p$  beschränkt sodass  $\|w_p\|_2 \leq c$  gilt. Werte für den Hyperparameter  $c$  liegen meist zwischen 3 und 4 (vgl. Srivastava

et al., 2014). Diese Art der Regularisierung projiziert folglich die Gewichte eines Neurons  $w_p$  auf eine Kugel mit Radius  $c$ , sobald die Norm diese verlässt. Nach Srivastava et al. (2014) verhindert dies den *Exploding Gradient*-Effekt, da die Gewichte den Fehler beim Rückpropagieren nicht mehr überproportional verstärken können. Dies ermöglicht größere Lernraten im Vergleich zum Training ohne Max-Norm-Regularisierung, was gerade in Verbindung mit Dropout-Learning (siehe Kapitel 3.4.2) von Vorteil ist, da so größere Gebiete im Gewichtsraum untersucht werden können.

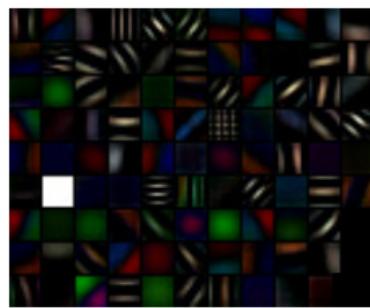


Abbildung 3.10: Ein dominierender Filter ohne Max-Norm Regularisierung (siehe Zeiler and Fergus, 2014)

Abbildung 3.10 zeigt einen weiteren Grund für die Notwendigkeit dieser Form der Regularisierung. Die Dominanz eines Filters über alle anderen kann effektiv verhindert werden (vgl. Zeiler and Fergus, 2014).

### 3.4.2 Modellkapazität

Beim Training großer Modelle mit hoher Kapazität ist *Overfitting* von zentraler Bedeutung. *Overfitting* beschreibt den Effekt eines kleiner werdenden Fehlers auf den Trainingsdaten bei gleichzeitiger Verschlechterung der Performance auf den Testdaten. Abbildung 3.11 stellt diesen Effekt grafisch dar.

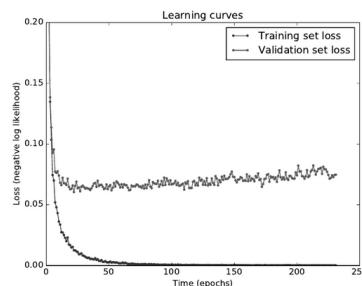


Abbildung 3.11: *Overfitting* am Beispiel des MNIST-Datensatzes (siehe Ben-gio et al., 2015, Kap. 7.3, S. 216)

Im Folgenden werden Methoden zusammengefasst, welche die Kapazität des Modells regulieren und damit aktiv versuchen *Overfitting* zu verhindern.

### **Early Stopping**

Die einfachste, sehr häufig angewandte Methode *Overfitting* aktiv zu verhindern, ist das sogenannte *Early Stopping*. Diese Methode teilt die Trainingsmenge zuerst in zwei Teilmengen. Sie setzt somit einen Satz Validierungsdaten voraus (vgl. im Folgenden Bengio et al., 2015, Kap. 7.3, S. 216 ff.).

Beim *Early Stopping* wird während des Trainings in regelmäßigen Abständen der Fehler auf den Validierungsdaten beobachtet. Verbessert sich dieser, werden die aktuellen Gewichte und Schwellwerte des Modells gesondert gespeichert. Am Ende des Trainings werden folglich nicht die aktuellsten Werte zurückgegeben sondern diejenigen, welche den kleinsten Validierungsfehler erzeugten. Anstelle eines lokalen Minimums über den Trainingsdaten wird somit ein lokales Minimum des Validierungsfehlers gesucht. Das Training stoppt, sobald sich der Validierungsfehler über mehrere Epochen nicht verbessert hat. Die Kapazität des Modells wird dahingehend beschränkt, dass die Trainingszeit limitiert ist. Bei der  $L^2$ -Regularisierung wurde beobachtet, dass Gewichte in Richtungen hoher Krümmung hinsichtlich der Zielfunktion weniger stark reguliert werden als andere. Diese Beobachtung lässt eine Ähnlichkeit zum *Early Stopping* erkennen, da die Gewichte in Verbindung mit hoher Krümmung relativ zu den anderen Gewichten früher im Trainingsprozess gelernt werden.

### **Parameteranzahl limitieren**

Die Kapazität des Modells lässt sich durch Vereinfachung beziehungsweise Verkleinerung und somit einer Verringerung der zu lernenden Gewichte regulieren. Im Umfeld des Deep Learning werden grundsätzlich zwei verschiedene Methoden beschrieben: Teilen der Gewichte (*Parameter Sharing*) und Reduktion der Verbindungen zwischen Merkmalen.

#### *Parameter Sharing*

Diese Methode ist am meisten verbreitet in CNNs und die Grundlage für die großen Erfolge im maschinellen Lernen und Deep Learning. CNNs berechnen an verschiedenen Stellen des Eingabebeispiels mit den selben Gewichten eine gewichtete Summe, was der algebraischen Faltung entspricht (vgl. Le-Cun et al., 1998a). Durch *Parameter Sharing* ist es möglich große neuronale Netze zu trainieren, ohne die Trainingsmenge entsprechend zu vergrößern. Damit sind CNNs ein herausragendes Beispiel für die Integration von Expertenwissen in einem neuronalen Netz (vgl. Bengio et al., 2015, Kap 7.8, S. 224).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X		X	X	X		X	X	X	X	X	X	X	X	X	
1	X	X		X	X	X		X	X	X	X	X	X	X	X	
2	X	X	X		X	X	X		X		X	X	X	X	X	
3	X	X	X		X	X	X	X		X		X	X	X	X	
4		X	X	X		X	X	X	X		X	X	X	X	X	
5		X	X	X		X	X	X	X		X	X	X	X	X	

Abbildung 3.12: Verbindungsma trix der Merkmale beziehungsweise *Feature-Maps* (Zeilen) mit den dazugehörigen Gewichten beziehungsweise Faltungsmasken (Spalten) zweier aufeinanderfolgenden Schichten (siehe LeCun et al., 1998a)

#### *Limitierung von Verbindungen zwischen Merkmalen*

Bei Nutzung dieser Methode wird ebenfalls das Ziel verfolgt Parameter zu reduzieren, wobei allerdings ein anderer Ansatz als bereits vorgestellt gewählt wird. Anstatt die Eingabe nur in überlappende, rezeptive Felder zu unterteilen und mit den selben Gewichten zu gewichten, werden zusätzlich die unterschiedlichen Dimensionen der Eingabe (z. B. RGB-Kanäle) überlappend aufgeteilt. Die bekannte Verbindungsma trix von *LeNet 5* ist zur Veranschaulichung in Abbildung 3.12 dargestellt.

Neben der Reduktion von zu trainierenden Parametern erlaubt dies auch eine bessere Extraktion unabhängiger Merkmale (vgl. LeCun et al., 1998a). Darüber hinaus bedient sich die Architektur von Krizhevsky et al. (2012) zwecks Parallelisierung des Netzes auf mehrere Rechenkerne (GPUs) derselben Methode, wie Abbildung 3.13 zeigt.

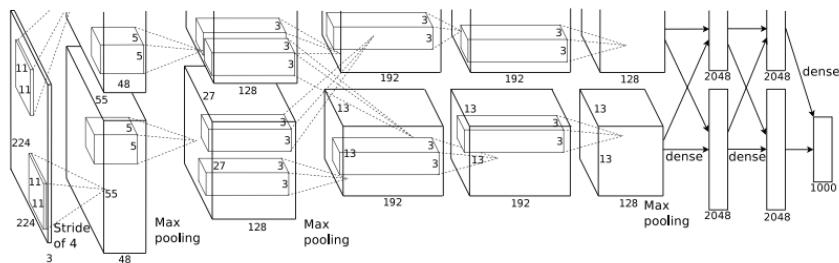


Abbildung 3.13: Architektur für *ImageNet 2012* mit limitierten Verbindungen zwischen *Feature-Maps* zwecks Parallelisierung (siehe Krizhevsky et al., 2012)

#### **Dropout**

Die Dropout-Methode wurde von Hinton et al. (2012) eingeführt und bezeichnet im eine Methode, mehrere kleinere Modelle zu lernen und die verschiedenen Ausgaben zu kombinieren (*Model Averaging*). Hierbei ist hervor-

zuheben, dass dies gleichzeitig und in einem neuronalen Netz geschieht. (vgl. Srivastava et al., 2014).

Die Dropout-Methode wurde mit dem Ziel entwickelt MLPs wenigen Trainingsdaten zu trainieren, was typischerweise in *Overfitting* resultiert. Hierbei soll die Architektur des Netzes im Training zufällig für jedes Trainingsbeispiel verändert werden, um Abhängigkeiten (*Co-Adaptions*) extrahierter Merkmale zu vermeiden (vgl. Hinton et al., 2012). Dazu werden zufällig einzelne Neuronen deaktiviert. Dieses Grundprinzip ist in Abbildung 3.14 dargestellt. Um den Gradienten richtig zu berechnen, ist es wichtig, dass die deaktivierte Neuronen auch im Backpropagation beim *Backward Pass* deaktiviert bleiben.

Wird das zufällige ausschalten von Neuronen deaktiviert, existiert ein trainiertes Netz, welches implizit mehrere Modelle kombiniert und somit eine deutliche Verbesserung gegenüber bestehenden Regularisierungs-Methoden hinsichtlich *Overfitting* darstellt (vgl. Srivastava et al., 2014). Damit die Gewichte durch die Kombination der Modelle im Testlauf nicht zu groß sind, müssen diese mit  $1 - p$  skaliert werden. Als Hyperparameter muss für Dropout lediglich die Wahrscheinlichkeit  $p$  ein Neuron zu deaktivieren angegeben werden. Typische Werte reichen von 0.1 bis 0.5 (vgl. Krizhevsky et al. (2012), Srivastava et al. (2014) oder Simonyan and Zisserman (2014)).

Wird die Standardinitialisierung in Verbindung mit ReLu-Funktionen verwendet, muss darauf geachtet werden, dass die Neuronen positive Ausgaben erzeugen, um ebenso positive Gradienten für das Lernen zu erhalten. Dies kann mittels des Schwellwertes  $b = 1$  oder einer hinreichend großen Varianz erreicht werden (vgl. Hinton et al., 2012).

Die beschriebene Methode kann auch bestehende, gut funktionierende Netze verbessern. Da die effektive Kapazität durch Dropout verringert wird, ist es sinnvoll die Anzahl existierender Neuronen  $n$  auf  $\frac{n}{1-p}$  zu erhöhen (vgl. Srivastava et al., 2014). Aufgrund der Tatsache, dass CNNs die Modellkapazität durch *Parameter Sharing* bereits stark regulieren, ist Dropout in Convolution-Layer nicht gleichermaßen effektiv wie in Hidden-Layer (vgl. Hinton et al., 2012).

### 3.4.3 Erweitern der Trainingsdaten

Die Kapazität eines Modells ist im Allgemeinen eine relative Größe basierend auf der zur Verfügung stehenden Trainingsmenge. Folglich lässt sich das Problem des *Overfittings* auch umgekehrt formulieren: Anstatt die Kapazität zu regulieren, kann die Menge der Trainingsdaten erhöht werden. Diese Methode wird häufig als *Data Augmentation* bezeichnet und umfasst drei Bereiche: Affine Transformationen, Elastische Transformationen und Additives Rauschen. Diese Formen der Regularisierung haben oftmals großen Einfluss auf die Performanz des Systems hinsichtlich Invarianzen und müssen deshalb beim Vergleich unterschiedlicher Algorithmen und Lernmaschinen

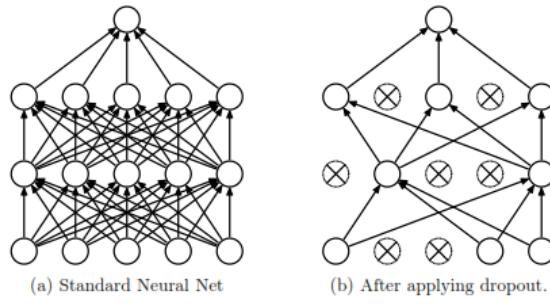


Abbildung 3.14: Schematische Darstellung eines gewöhnlichen MLP (links) und eines Dropout-MLP (rechts) (siehe Srivastava et al., 2014)

berücksichtigt werden. (vgl. Bengio et al., 2015, Kap. 7.5, S. 210 f.)

### Affine Transformationen

Affine Transformationen können die Generalisierung des Netzes immens verbessern. Eingaben sollen hierbei so verändert werden, dass gewünschte Invarianzen besser trainiert werden. Dies ist auch dann von Vorteil, wenn das Modell selbst bereits gewisse Invarianzen, wie beispielsweise die Translationsinvarianz von CNNs, enthält. So finden affine Transformationen Anwendung im Training von *LeNet 5* von LeCun et al. (1998a) und im *AlexNet* von Krizhevsky et al. (2012). Eine analytische Methode *Data Augmentation* in den Trainingsprozess einzubauen, beschreibt Simard et al. (1992) mittels des Tangent-Prop-Algorithmus.

### Elastische Transformationen

Elastische Transformationen erweitern das Set an verfügbaren affinen Operationen und verbessern die Generalisierung dadurch weiter. Zunächst wird ein Vektorfeld pro Dimension generiert, welches zufällige Verschiebungen der einzelnen Eingabemerkmale (z. B. Pixel) beschreibt. Die generierten Felder werden anschließend mit einem Gauss-Kern gefaltet, beziehungsweise weichgezeichnet, und auf die originale Eingabe angewandt (vgl. Simard et al., 2003). Das Ergebnis elastischer Transformationen ist in Abbildung 3.15 dargestellt.

### Additives Rauschen

Eine weitere Möglichkeit das System robuster zu machen, ist das Training mit additivem Rauschen. So korrumptiert beispielsweise Vincent et al. (2008) die Trainingsdaten zu einem gewissen Grad mit additiven Rauschen, Maskierung oder *Salt-and-pepper* und konstruiert damit einen *Denoising Autoencoder*.

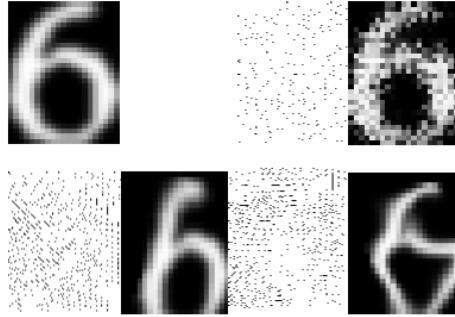


Abbildung 3.15: Originale Eingabe (oben links) und drei Beispiele elastisch veränderter Eingaben mit zugehörigem Vektorfeld (siehe Simard et al., 2003)

der (DAE). Mehrere DAEs bilden zusammen *Stacked Denoising Autoencoder* und schließen damit die Lücke zu den beschränkten Boltzmann-Maschinen und (DBNs) im Bereich des unüberwachten Vortrainings (vgl. Vincent et al., 2010).

Gerade die Maskierung einzelner Merkmale im Eingaberaum liefert die Grundlage für das spätere Dropout, welches sowohl für die Hidden-Layer als auch im Input-Layer geeignet ist und die Generalisierung deutlich verbessern kann (vgl. Srivastava et al., 2014).

## 3.5 Visualisierungsmethoden

Gerade die Verbindung von Bilderkennung und CNNs macht die Anwendung von Visualisierungsmethoden besonders attraktiv, da sich gewisse Verhaltensweisen des neuronalen Netzes so besser verstehen lassen. Die meisten der vorgestellten Methoden haben das Ziel der Kritik, *Features* aus neuronalen Netzen ließen sich nicht interpretieren, entgegenzuwirken (vgl. Zeiler and Fergus, 2014).

### 3.5.1 Primitiv

Die einfachste Möglichkeit des Sichtbarmachens des Neuronalen Netzes, ist die Visualisierung der Ausgabe beziehungsweise der Gewichte und wird deshalb als primitiv bezeichnet. Dennoch können diese Techniken wichtige Einblicke gewähren.

#### Neuronen-Aktivierung

Die Visualisierung der Ausgabe eines *AlexNet* von Krizhevsky et al. (2012) ist in Abbildung 3.16 (links) dargestellt. Die Eingabe ist ein Bild einer Katze. Es fällt sofort auf, dass die Aktivierungen sehr dünnbesetzt (*sparse*) sind,

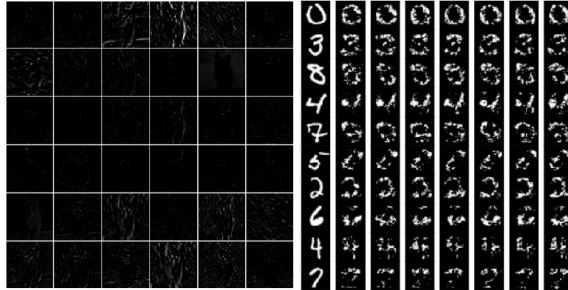


Abbildung 3.16: Aktivierungen des *AlexNet* nach erstem Layer (links) (siehe Fei-Fei and Karpathy, 2014) und Rekonstruktionen eines Autoencoders (rechts) (siehe Vincent et al., 2010)

was letztendlich auf die ReLU-Funktionen zurückzuführen ist (vgl. Glorot et al., 2011).

Diese Visualisierungstechnik kann in mehrerer Hinsicht unterstützen. Einerseits können so tote Neuronen identifiziert werden (vgl. *Dying ReLU*-Effekt in Maas et al. (2013)). Andererseits kann überprüft werden, ob die Aktivierungen die gewünschte *Sparsity* und Lokalität aufweisen. Wird ein Autoencoder verwendet, kann darüber hinaus interessant sein, die Aktivierung beziehungsweise Ausgabe des Netzes zu visualisieren, da diese der Rekonstruktion der Eingabe entsprechen. Abbildung 3.16 (rechts) zeigt derartige Rekonstruktionen auf dem MNIST-Datensatz.

### Faltungskerne

Eine weitere einfache Möglichkeit, Informationen über das CNN zu erhalten ist die Visualisierung der gelernten Filtermasken. Abbildung 3.17 zeigt die Faltungskerne des ersten Layers eines trainierten *AlexNet*.

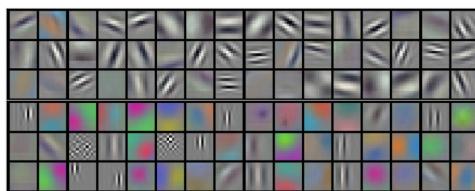


Abbildung 3.17: Die Faltungskerne des ersten Layers eines trainierten *AlexNet* (siehe Krizhevsky et al., 2012)

Trainierte Netze haben typischerweise glatte, Gabor-Filtern ähnliche Filtermasken. Verrauschte Filtermasken können hingegen entweder ein Indikator für eine fehlende Regularisierung und damit einhergehendes *Overfitting* sein

oder darauf hinweisen, dass das Netz nicht ausreichend konvergiert ist (vgl. Fei-Fei and Karpathy, 2014).

### 3.5.2 Gradientenbasiert

Eine weitere Klasse von Visualisierungsmethoden arbeiten auf Basis des Backpropagation-Algorithmus. Dies bedeutet, dass sie sich der Fehlerrückführung (*Backward Pass*) bedienen und diese zur Visualisierung verwenden.

#### Neuronen-Visualisierung

Die erste Methode ist die sogenannte Deconvnet-Visualisierung von Zeiler and Fergus (2014). Diese kann als Neuronen-Visualisierung bezeichnet werden, da letztlich Aktivierungen einzelner Neuronen visualisiert werden. Die Grundidee des Verfahrens ist es, einzelne Aktivierungen im Netz im Eingaberaum sichtbar zu machen. Dazu wird ein bestimmtes Neuron beziehungsweise eine bestimmte *Feature-Map* ausgewählt und die Aktivierungen werden über die gesamte Testmenge gemessen. Die Beispiele mit den höchsten Aktivierungen werden gespeichert. Im Anschluss nutzt die Visualisierung diese Beispiele und rekonstruiert die erzeugten Aktivierungen, indem alle anderen *Feature-Maps* dieses Layers Null gesetzt werden. Für die Rekonstruktion  $R$  der Aktivierung wird im Grunde die Fehlerrückführung des Backpropagation-Algorithmus verwendet. Damit gilt in einem Convolution-Layer die Gleichung 3.33 zur Rekonstruktion einer Feature-Map.

$$R_j^l = \sum_{i=0}^n \hat{R}_i^{l+1} * \text{rot180}(W_{ij}) \quad (3.33)$$

Im Unterschied zur Berechnung des Gradienten mit Backpropagation, wird bei diesem Verfahren die Aktivierungsfunktion nicht abgeleitet. Stattdessen wird die Aktivierungsfunktion  $\phi$  selbst mit der Rekonstruktion  $R^{l+1}$  im Sinne des *Forward Pass* berechnet. Formel 3.34 verdeutlicht diesen Zusammenhang für den Layer  $l$ .

$$\hat{R}_i^{l+1} = \phi_l(R_i^{l+1}) \quad (3.34)$$

Eine weitere Besonderheit stellen die Pooling-Layer dar, da diese im Prinzip nicht umkehrbar sind. Die von Zeiler and Fergus (2014) vorgeschlagene Methode sieht deshalb lediglich die Variante Max-Pooling vor und verbietet alle anderen Pooling-Verfahren. Max-Pooling bietet hierbei den Vorteil, dass im *Forward Pass* die Stellen der Maxima vermerkt werden.

Acht ausgewählte Neuronen im dritten Layer mit den jeweiligen zur höchsten Aktivierung korrespondierenden Eingabebildern, sind in Abbildung 3.18 dargestellt.

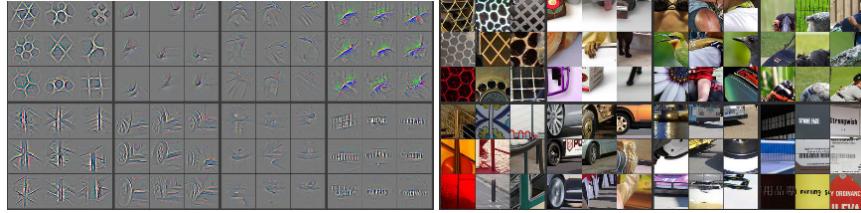


Abbildung 3.18: Mit Deconvnet rekonstruierte Aktivierungen im dritten Layer (links) und dazugehörige originale Eingabebilder (rechts) (siehe Zeiler and Fergus, 2014)

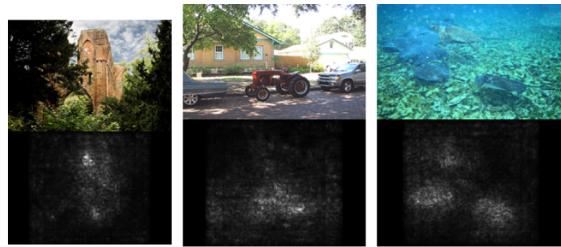


Abbildung 3.19: Originale Eingabebilder (oben) und dazu gehörige *Siliency-Maps* (unten) (siehe Simonyan et al., 2013)

### Saliency-Visualisierung

Die sogenannte Saliency-Visualisierung von Simonyan et al. (2013) ist dem Deconvnet-Verfahren im Grunde sehr ähnlich. Es unterscheidet sich allerdings in einem zentralen Punkt. Während das Verfahren von Zeiler and Fergus (2014) Veränderungen im Vergleich zum Backpropagation-Algorithmus vornimmt, hält sich dieses Verfahren strikt an die Fehlerrückführung und berechnet damit die Ableitungen der Aktivierungsfunktionen, wie Formel 3.35 zeigt.

$$\hat{R}_i^{l+1} = R_i^{l+1} \circ \phi'(z_i) \quad (3.35)$$

Dies bietet den Vorteil, dass sich das Verfahren somit sowohl auf die Convolution-Layer als auch die Hidden-Layer anwenden lässt. Dies lässt sich damit begründen, dass das Verfahren im ursprünglichen Sinne einzelne Klassen im Eingaberaum visualisiert (vgl. Simonyan et al., 2013).

Abbildung 3.19 zeigt die beschriebene Technik, angewandt auf Beispiele aus dem *ImageNet*-Datensatz. Die *Saliency-Map* zeigt die Rekonstruktion der entsprechenden Aktivierung im Output-Layer.



Abbildung 3.20: t-SNE Einbettung einer Auswahl an *ImageNet*-Bildern auf Basis der Aktivierung des letzten Layers im *AlexNet* (Bild: Laurens van der Maaten, Facebook AI Research)

### 3.5.3 Nachverarbeitung

Methoden im Bereich Nachverarbeitung sind dadurch charakterisiert, dass ein trainiertes MLP, beziehungsweise CNN, lediglich zur Erzeugung von Merkmalsvektoren oder zur Berechnung von Ausgaben herangezogen wird. Diese Art der Visualisierung beschränkt sich demnach auf den Kontext und betrachtet das Netz als *Blackbox*, beziehungsweise als eine Funktion  $h(x)$ .

#### t-SNE

Die Methode t-Distributed Neighbor Embedding (t-SNE) beschreibt ein von Maaten et al. (2008) entwickeltes Optimierungsverfahren zur Dimensionsreduktion. Es verbessert das klassische SNE-Verfahren insofern, dass die Gradienten für die Optimierung einfacher zu berechnen sind. Darüber hinaus wird anstelle einer Gauß- eine Student-t-Verteilung als Ähnlichkeitsmaß im niedrigdimensionalen 2D oder 3D Zielraum verwendet (vgl. Maaten et al., 2008). Der Grundmechanismus von t-SNE ist die Minimierung des Unterschieds zweier Wahrscheinlichkeitsverteilungen (Kullback-Leibler-Divergenz) — Der Wahrscheinlichkeitsverteilung von Paaren im hochdimensionalen  $P$  Raum sowie der im niedrigdimensionalen Raum  $Q$ . Damit ergibt sich die zu minimierende Zielfunktion  $C$  in Formel 3.36.

$$C = KL(P||Q)) = \sum_i \sum_j p_{ij} \log\left(\frac{p_{ij}}{q_{ij}}\right) \quad (3.36)$$

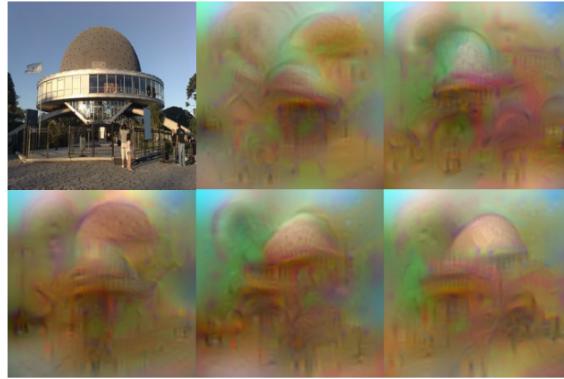


Abbildung 3.21: Mögliche Rekonstruktionen des originalen Eingabebilds (oben links) (siehe Simard et al., 2003)

Grundsätzlich können die von neuronalen Netzen erzeugten Merkmalsvektoren (*Features*) auch mittels der Hauptkomponentenanalyse (PCA) visualisiert werden. Die t-SNE unterscheidet sich von PCA allerdings in einigen zentralen Eigenschaften (vgl. im Folgenden Maaten et al., 2008):

- Wenn ein Großteil der Varianz nicht in den ersten zwei, beziehungsweise drei, Hauptkomponenten beschrieben ist, kann t-SNE bessere Ergebnisse liefern, da es versucht die gesamte Information abzubilden.
- t-SNE ist im Vergleich zur PCA keine orthogonale, lineare Transformation sondern eine nichtlineare Reduktion mit nicht zwingend orthogonalen Komponenten.
- t-SNE findet, aufgrund der nicht-konvexen Zielfunktion, nicht immer zum globalen Minimum.
- t-SNE ist darauf spezialisiert hochdimensionale Daten auf maximal zwei beziehungsweise drei Dimensionen zu reduzieren.

Abbildung 3.20 zeigt die t-SNE 2D-Transformation des Merkmalsvektors vor dem Output-Layer eines *AlexNet*. Als Eingabedaten dient eine Teilmenge des *ImageNet*-Datensatzes. Diese Art der Visualisierung zeigt, dass das CNN Merkmale extrahiert und darauf aufbauend relevante von irrelevanten Merkmalen trennen kann. Dies lässt sich sehr deutlich daran erkennen, dass inhaltlich ähnliche Bilder näher zusammen sind als andere und somit Kategorien oder Gruppen und nicht Hintergrund oder Färbung die Daten charakterisieren (vgl. Bell and Bala, 2015).

## Inceptionism

Ein weiteres Verfahren, welches als *Inceptionism*<sup>3</sup> bezeichnet wird, wählt eine andere Herangehensweise. Durch Nutzung dieses Verfahrens wird versucht, ausgehend von einer Aktivierung, beziehungsweise einem Merkmalsvektor eines Layers, eine dazu passende Eingabe zu rekonstruieren. Das Grundverfahren, um Merkmalsvektoren zu invertieren, stammt von Mahendran and Vedaldi (2014). Dieses Verfahren nimmt ein beliebiges Eingabebild  $x_0$  und berechnet mittels eines trainierten CNN die Ausgabe  $h_0 = h_l(x_0)$  in einem beliebigen Layer  $l$ . Das Ziel des Verfahrens ist es, die Zielfunktion  $C$  in Gleichung 3.37 mit Regularisierer  $R(x)$  zu minimieren und ausgehend vom Rauschen ein optimales  $x$  zu finden.

$$C = \|h(x) - h_0\|^2 + \lambda R(x) \quad (3.37)$$

Der Regularisierer sorgt dafür, dass das optimale  $x$  innerhalb eines für Bilder üblichen Intervalls  $B = [-128, 128]$  liegt (Formel 3.38) und das resultierende Bild stückweise konstant ist (Formel 3.39).

$$R_1 = \|x\|_6^6 \quad (3.38)$$

$$R_2 = \sum_{i,j} [(x_{i,j+1} - x_{i,j})^2 + (x_{i+1,j} - x_{i,j})^2]^{\frac{1}{2}} \quad (3.39)$$

Abbildung 3.21 zeigt fünf Optimierungen auf Basis des Eingabebilds oben links.

---

<sup>3</sup>Das Kunswort *Inceptionism* geht auf einen Blog-Eintrag von *Research at Google* zurück (<http://googleresearch.blogspot.de/2015/06/inceptionism-going-deeper-into-neural.html> (26.08.2015)).

# Literaturverzeichnis

- Abdel-Hamid, O., Deng, L., and Yu, D. (2013). Exploring convolutional neural network structures and optimization techniques for speech recognition. In *Proceedings of the International Conference of the International Speech Communication Association, Interspeech*.
- Abuzaid, F., Hadjis, S., Zhang, C., and Ré, C. (2015). Caffe con troll: Shallow ideas to speed up deep learning. *CoRR*, abs/1504.04343.
- Andrade, A. d. (2014). Best practices for convolutional neural networks applied to object recognition in images. Thesis <http://www.cs.toronto.edu/~adeandrade/assets/bpfcnnatorii.pdf> (15.08.2015).
- Becker, S. (1991). Unsupervised learning procedures for neural networks. *International Journal of Neural Systems*, 2:17–33.
- Bell, S. and Bala, K. (2015). Learning visual similarity for product design with convolutional neural networks. *ACM Trans. Graph.*, 34(4):98:1–98:10.
- Bengio, Y. (2009). Learning deep architectures for ai. *Foundation and Trends in Machine Learning*, 2(1):1–127.
- Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533.
- Bengio, Y., Goodfellow, I. J., and Courville, A. (2015). Deep learning. Book in preparation for MIT Press <http://www.iro.umontreal.ca/~bengioy/dlbook/> (15.08.2015).
- Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In Schölkopf, B., Platt, J., and Hoffman, T., editors, *Proceedings of the 20th International Conference on Advances in Neural Information Processing Systems, NIPS*, pages 153–160. MIT Press. ISBN: 9781622760381.
- Bengio, Y. and LeCun, Y. (2007). Scaling learning algorithms towards AI. In Bottou, L., Chapelle, O., DeCoste, D., and Weston, J., editors, *Large*

*Scale Kernel Machines.* MIT Press, Cambridge, Massachusetts. ISBN: 9780262026253.

- Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a cpu and gpu math expression compiler. In Varoquaux, G., Walf, S. v. d., and Millman, K. J., editors, *Proceedings of the 8th Python for Scientific Computing Conference, SciPy*, volume 4, page 3. Caltech. ISBN: 978-0-557-23212-3.
- Bottou, L. (1998). Stochastic gradient tricks. In Montavon, G., Orr, G. B., and Müller, K.-R., editors, *Neural networks: Tricks of the trade*, pages 430–445. Springer, Heidelberg, Dodrecht, London, NewYork. ISBN: 978-3-642-35289-8.
- Bouvie, J. (2006). Notes on convolutional neural networks. Technical Report [http://cogprints.org/5869/1/cnn\\_tutorial.pdf](http://cogprints.org/5869/1/cnn_tutorial.pdf) (15.08.2015).
- Chapelle, O. and Erhan, D. (2011). Improved preconditioner for hessian free optimization. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.
- Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759.
- Ciresan, D. C., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745.
- Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *Proceedings of the NIPS Workshop, BigLearn*.
- Dauphin, Y., Pascanu, R., Gülcühre, Ç., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572.
- Dauphin, Y. N., de Vries, H., Chung, J., and Bengio, Y. (2015). Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *CoRR*, abs/1502.04390.
- Desjardins, G. and Bengio, Y. (2008). Empirical evaluation of convolutional rbms for vision. Technical Report.
- Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. A Wiley Interscience Publication. John Wiley & Sons. ISBN: 978-0471223610.

- Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660.
- Fei-Fei, L. and Karpathy, A. (2014). CS231n Convolutional Neural Networks for Visual Recognition, University of Stanford. Course Material <http://cs231n.github.io/> (14.08.2015).
- Franz, M. O. (2014). Maschinelles Lernen, University of Applied Sciences Konstanz. Course Material.
- Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202.
- Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. and Titterington, M., editors, *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, AISTATS*, pages 249–256. JMLR W&CP.
- Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Teh, Y. and Titterington, M., editors, *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics, AISTATS*, pages 315–323. JMLR W&CP.
- Golik, P., Doetsch, P., and Ney, H. (2013). Cross-entropy vs. squared error training: a theoretical and experimental comparison. In Bimbot, F., editor, *Proceedings of the International Conference of the International Speech Communication Association, Interspeech*, pages 1756 –1760. ISCA. ISBN: 9781629934433.
- Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., and Bengio, Y. (2013a). Pylearn2: a machine learning research library. *arXiv preprint*, 1308.421.
- Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013b). Maxout networks. In Lawrence, N. and Reid, M., editors, *Proceedings of the 30th International Conference on Machine Learning, ICML*.
- Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. Linear Algebra Library.
- Hagan, M. T., Demuth, H. B., Beale, M., and Jesús, O. D. (2014). *Neural Network Design*. Martin Hagan, 2 edition. ISBN: 978-0971732117.
- Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Minint, Inference and Prediction*. Springer, 2 edition. ISBN: 978-0387848570.

- Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation*. Pearson, Hamilton, Ontario, 2 edition. ISBN: 978-0132733502.
- He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852.
- Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the 8th Annual Conference of the Cognitive Science Society, CogSci*, pages 1–12.
- Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.
- Hinton, G. E. and Salakhutdinov, R. R. (2008). Using deep belief nets to learn covariance kernels for gaussian processes. In Platt, J., Koller, D., Singer, Y., and Roweis, S., editors, *Proceedings of the 20th International Conference on Advances in Neural Information Processing Systems, NIPS*, pages 1249–1256. MIT Press.
- Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.
- Hinton, G. E., Srivastava, N., and Swersky, K. (2015). CSC321 Neural Networks for Machine Learning, University of Toronto. Course Material [http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\\_slides\\_lec6.pdf](http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) (14.08.2015).
- Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen netzen. Thesis <http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf> (15.08.2015).
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computing*, 9(8):1735–1780.
- Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction, and functional architecture in the cat’s visual cortex. *J. Physiol*, 160:106–154.
- Igel, C. and Hüskens, M. (2000). Improving the rprop learning algorithm. In *Proceedings of the 2nd international symposium on neural computation, ICSC*, volume 2000, pages 115–121. Springer.
- Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *Proceedings of the 12th IEEE International Conference on Computer Vision, ICCV*, pages 2146–2153. IEEE.

- Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint*, 1408.5093.
- Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. Python Library.
- Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997.
- Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Thesis <http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf> (15.08.2015).
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Proceedings of the 25th International Conference on Advances in neural information processing systems, NIPS*, pages 1097–1105. MIT Press. ISBN: 9781627480031.
- Le, Q., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G., Dean, J., and Ng, A. (2012). Building high-level features using large scale unsupervised learning. In Langford, J. and Pineau, J., editors, *Proceedings of the 29th International Conference on Machine Learning, ICML*. ACM. ISBN: 978-1-4503-1285-1.
- Le, Q. V., Ngiam, J., Chen, Z., Chia, D., Koh, P. W., and Ng, A. Y. (2010). Tiled convolutional neural networks. In *Proceedings of the 20th International Conference on Advances in Neural Information Processing Systems, NIPS*.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (1998b). Efficient backprop. In Montavon, G., Orr, G. B., and Müller, K.-R., editors, *Neural networks: Tricks of the trade*, pages 9–48. Springer, Heidelberg, Dodrecht, London, NewYork. ISBN: 978-3-642-35289-8.
- Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z., and Tu, Z. (2014). Deeply-supervised nets. *CoRR*, abs/1409.5185v2.

- Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th International Conference on Machine Learning, ICML*, ICML '09, pages 609–616, New York, NY, USA. ISBN: 978-1-60558-516-1.
- Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning, ICML*.
- Maaten, L., Hinton, G. E., and Bengio, Y. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605.
- Mahendran, A. and Vedaldi, A. (2014). Understanding deep image representations by inverting them. *CoRR*, abs/1412.0035.
- Martens, J. (2010). Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning, ICML*.
- Masci, J., Meier, U., Cireşan, D., and Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. In *Proceedings of the International Conference on Artificial Neural Networks and Machine Learning, ICANN*, pages 52–59. Springer. ISBN: 978-3-642-21735-7.
- McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill. ISBN: 978-0071154673.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.
- Nagi, J., Ducatelle, F., Caro, G. A. D., Cires, D., Meier, U., Giusti, A., Nagi, F., Schmidhuber, J., and Gambardella, L. M. (2011). Max-pooling convolutional neural networks for vision-based hand gesture recognition. In *Proceedings of the IEEE International Conference on Signal and Image Processing Applications, ICSIPA*.
- Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate  $O(1/k^2)$ . *Soviet Mathematics Doklady*, 27(2):372–376.
- Nouri, D. (2013). cuda-convnet. Cuda Framework.
- OpenMP Architecture Review Board (2008). OpenMP application program interface version 3.0. Compiler Extension.

- Pearlmutter, B. A. (1994). Fast exact multiplication by the hessian. *Neural Computation*, 6:147–160.
- Pierre Sermanet, S. C. and LeCun, Y. (2012). Convolutional neural networks applied to house numbers digit classification. *CoRR*, abs/1204.3968.
- Pink, O. (2011). *Bildbasierte Selbstlokalisierung von Straßenfahrzeugen*. KIT Scientific Publishing. ISBN: 978-3866447080.
- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics*, 4(5):1–17.
- Quiroga, R. Q., Reddy, L., Kreiman, G., Koch, C., and Fried, I. (2005). Invariant visual representation by single neurons in the human brain. *Nature*, 435:1102–1107.
- Ranzato, M., Poultney, C. S., Chopra, S., and Lecun, Y. (2006). Efficient Learning of Sparse Representations with an Energy-Based Model. In Bernhard Schölkopf, J. P. and Hofmann, T., editors, *Proceedings of the 19th Conference on Neural Information Processing Systems, NIPS*, pages 1137–1144. MIT Press.
- Ranzato, M., Poultney, C. S., Chopra, S., and Lecun, Y. (2007a). Efficient learning of sparse representations with an energy-based model. In Schölkopf, B., Platt, J., and Hoffman, T., editors, *Proceedings of the 19th International Conference on Advances in Neural Information Processing Systems, NIPS*, pages 1137–1144. MIT Press.
- Ranzato, M. A., Huang, F. J., Boureau, Y.-L., and LeCun, Y. (2007b). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 1–8. IEEE. ISBN: 1-4244-1179-3.
- Riedmiller, M. and Braun, H. (1992). Rprop-a fast adaptive learning algorithm. In *Proceedings of the 7th International Conference on Computer and Information Sciences, ISCIS*.
- Rojas, R. (1996). *Neural Networks: A Systematic Introduction*. Springer, Berlin Heidelberg New York. ISBN: 978-3540605058.
- Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books. ISBN: 9783642709135.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Learning internal representations by error propagation. In Rumelhart, D. E. and McClelland, J., editors, *Parallel Distributed Processing: Implications for Psychology and Neurobiology*, volume 1, pages 318–362. MIT Press, Cambridge. ISBN: 0-262-01097-6.

- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning representations by back-propagating errors. *Nature*, 323:533–536.
- Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 1:1–42.
- Sainatha, T. N., Kingsburya, B., Saona, G., Soltau, H., rahman Mohamedb, A., Dahlb, G., and Ramabhadran, B. (2015). Deep convolutional neural networks for large-scale speech tasks. *Neural Networks: Deep Learning of Representations*, 64:32–48.
- Schling, B. (2011). *The Boost C++ Libraries*. XML Press. ISBN: 9780982219195.
- Simard, P., Steinkraus, D., and Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the 7th International Conference on Document Analysis and Recognition, ICDAr*, pages 958–963. IEEE. ISBN: 0-7695-1960-1.
- Simard, P., Victorri, B., LeCun, Y., and Denker, J. (1992). Tangent prop: a formalism for specifying selected invariances in adaptive networks. In Moody, J. M., Hanson, S. J., and Lippman, R. P., editors, *Proceedings of the 4th International Conference on Advances in Neural Information Processing Systems, NIPS*, volume 4.
- Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034.
- Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.
- Socher, R., Lin, C. C.-Y., Ng, A., and Manning, C. D. (2011). Parsing natural sciences and natural language with recursive neural networks. In Getoor, L. and Scheffer, T., editors, *Proceedings of the 28th International Conference on Machine Learning, ICML*. ACM. ISBN: 978-1-4503-0619-5.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.
- Sutskever, I. (2013). Training recurrent neural networks. Thesis [http://www.cs.utoronto.ca/~ilya/pubs/ilya\\_sutskever\\_phd\\_thesis.pdf](http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf) (20.08.2015).

- Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning, ICML*.
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *CoRR*, abs/1409.4842.
- Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning, ICML*, pages 1096–1103.
- Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11:3371–3408.
- Wagner, R., Thom, M., Schweiger, R., Palm, G., and Rothermel, A. (2013). Learning convolutional neural networks from few samples. In *Proceedings of the International Joint Conference on Neural Networks, IJCNN*, pages 1–7.
- Wan, L., Zeiler, M. D., Zhang, S., LeCun, Y. A., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning, ICML*.
- Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record, Part 4*, pages 96–104.
- Widrow, B. and Hoff, M. E. (1988). Adaptive switching circuits. In Anderson, J. A. and Rosenfeld, E., editors, *Neurocomputing: Foundations of Research*, pages 123–134. MIT Press, Cambridge, MA, USA. ISBN: 0-262-01097-6.,
- Wilson, D. R. and Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural Netw.*, 16(10):1429–1451.
- Winston, P. H. (1992). *Artificial Intelligence*. Pearson, 3 edition. ISBN: 978-0201533774.
- Zander, A. (2001). *Neuronale Netze zur Analyse von nichtlinearen Strukturmodellen mit latenten Variablen*. Deutscher Universitäts-Verlag. ISBN: 978-3824472598.
- Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.

- Zeiler, M. D. and Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. In *Proceedings of the International Conference on Learning Representations, ICLR*.
- Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Proceedings of the 13th European Conference on Computer Vision, ECCV*, pages 818–833. Springer. ISBN: 978-3-319-10590-1.
- Zeiler, M. D., Taylor, G. W., and Fergus, R. (2011). Adaptive deconvolutional networks for mid and high level feature learning. In *Proceedings of the 14th IEEE International Conference on Computer Vision, ICCV*.