



Ian
Goodfellow
Yoshua
Bengio
Aaron
Courville

Deep Learning

Das umfassende Handbuch

Grundlagen, aktuelle Verfahren und
Algorithmen, neue Forschungsansätze



Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)

Der Verlag räumt Ihnen mit dem Kauf des ebooks das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und Einspeicherung und Verarbeitung in elektronischen Systemen.

Der Verlag schützt seine ebooks vor Missbrauch des Urheberrechts durch ein digitales Rechtemanagement. Bei Kauf im Webshop des Verlages werden die ebooks mit einem nicht sichtbaren digitalen Wasserzeichen individuell pro Nutzer signiert.

Bei Kauf in anderen ebook-Webshops erfolgt die Signatur durch die Shopbetreiber. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Ian Goodfellow
Yoshua Bengio
Aaron Courville

Deep Learning

Das umfassende Handbuch

Grundlagen, aktuelle Verfahren und
Algorithmen, neue Forschungsansätze

Übersetzung aus dem Amerikanischen
von Guido Lenz



Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie. Detaillierte bibliografische Daten sind im Internet über <<http://dnb.d-nb.de>> abrufbar.

ISBN 978-3-95845-701-0

1. Auflage 2018

www.mitp.de
E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 7189 - 079
Telefax: +49 7953 7189 - 082

© 2018 mitp Verlags GmbH & Co. KG, Frechen

Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Authorized German translation from the English language edition, entitled Deep Learning by Ian Goodfellow, Yoshua Bengio, and Aaron Courville
ISBN 9780262035613

Original English language edition published by The MIT Press, a department of Massachusetts Institute of Technology

Copyright © 2016 Massachusetts Institute of Technology

German-language edition copyright © 2018 by mitp-Verlag. All rights reserved.

Lektorat und fachliche Prüfung: Sabine Schulz, Janina Bahlmann, Lisa Kresse
Sprachkorrektorat: Petra Heubach-Erdmann
Fachkorrektur Mathematik: Stefan Niedergriese
Fachkorrektur Deep Learning: Alexander Bresk
Coverbild: madgooch/ stock.adobe.com
Satz: Dr. Joachim Schlosser, www.schlosser.info

Inhaltsverzeichnis

Website zum Buch	xi
Danksagung	xiii
Über die Fachkorrektoren zur deutschen Ausgabe	xvii
Notation	xix
1 Einleitung	1
1.1 Für wen ist dieses Buch gedacht?	10
1.2 Historische Entwicklungen im Deep Learning	12
I Angewandte Mathematik und Grundlagen für das Machine Learning	31
2 Lineare Algebra	33
2.1 Skalare, Vektoren, Matrizen und Tensoren	34
2.2 Multiplizieren von Matrizen und Vektoren	36
2.3 Einheits- und Umkehrmatrizen	38
2.4 Lineare Abhängigkeit und lineare Hülle	40
2.5 Normen	42
2.6 Spezielle Matrizen und Vektoren	43
2.7 Eigenwertzerlegung	45
2.8 Singulärwertzerlegung	48
2.9 Die Moore-Penrose-Pseudoinverse	49
2.10 Der Spuroperator	50
2.11 Die Determinante	51
2.12 Beispiel: Hauptkomponentenanalyse	51
3 Wahrscheinlichkeits- und Informationstheorie	57
3.1 Warum Wahrscheinlichkeit?	58

3.2	Zufallsvariablen	61
3.3	Wahrscheinlichkeitsverteilungen	61
3.4	Randwahrscheinlichkeit	64
3.5	Bedingte Wahrscheinlichkeit	64
3.6	Die Produktregel der bedingten Wahrscheinlichkeiten	65
3.7	Unabhängigkeit und bedingte Unabhängigkeit	65
3.8	Erwartungswert, Varianz und Kovarianz	66
3.9	Häufig genutzte Wahrscheinlichkeitsverteilungen	67
3.10	Nützliche Eigenschaften häufig verwendeter Funktionen	74
3.11	Satz von Bayes	76
3.12	Technische Einzelheiten stetiger Variablen	77
3.13	Informationstheorie	79
3.14	Strukturierte probabilistische Modelle	83
4	Numerische Berechnung	87
4.1	Überlauf und Unterlauf	87
4.2	Schlechte Konditionierung	89
4.3	Optimierung auf Gradientenbasis	90
4.4	Optimierung unter Nebenbedingungen	101
4.5	Beispiel: Lineare kleinste Quadrate	104
5	Grundlagen für das Machine Learning	107
5.1	Lernalgorithmen	108
5.2	Kapazität, Überanpassung und Unteranpassung	121
5.3	Hyperparameter und Validierungsdaten	133
5.4	Schätzer, Verzerrung und Varianz	135
5.5	Maximum-Likelihood-Schätzung	145
5.6	Bayessche Statistik	149
5.7	Algorithmen für überwachtes Lernen	154
5.8	Algorithmen für unüberwachtes Lernen	161
5.9	Stochastisches Gradientenabstiegsverfahren	167
5.10	Entwickeln eines Machine-Learning-Algorithmus	170
5.11	Probleme, an denen Deep Learning wächst	171
II	Tiefe Netze: Zeitgemäße Verfahren	183
6	Tiefe Feedforward-Netze	185
6.1	Beispiel: Erlernen von XOR	189
6.2	Lernen auf Gradientenbasis	195
6.3	Verdeckte Einheiten	211

6.4	Architekturdesign	218
6.5	Backpropagation und andere Algorithmen zur Differentiation	225
6.6	Historische Anmerkungen	248
7	Regularisierung	253
7.1	Parameter-Norm-Strafterme	255
7.2	Norm-Strafterme als Optimierung unter Nebenbedingungen	263
7.3	Regularisierung und unterbestimmte Probleme	265
7.4	Erweitern des Datensatzes	266
7.5	Robustheit gegen Rauschen	268
7.6	Halb-überwachtes Lernen	270
7.7	Multitask Learning	271
7.8	Früher Abbruch	273
7.9	Parameter Tying und Parameter Sharing	281
7.10	Dünnsbesetzte Repräsentationen	283
7.11	Bagging und andere Ensemblemethoden	285
7.12	Dropout	287
7.13	Adversarial Training	299
7.14	Tangentendistanz, Tangenten-Propagation und Mannigfaltigkeit-Tangenterklassifikator	301
8	Optimierung beim Trainieren von tiefen Modellen	305
8.1	Unterschied zwischen Lernen und reiner Optimierung	306
8.2	Herausforderungen bei der Optimierung neuronaler Netze	315
8.3	Grundlegende Algorithmen	327
8.4	Verfahren zur Parameterinitialisierung	335
8.5	Algorithmen mit adaptiven Lernraten	342
8.6	Approximative Verfahren zweiter Ordnung	347
8.7	Optimierungsverfahren und Meta-Algorithmen	354
9	CNNs	369
9.1	Die Faltungsoperation	370
9.2	Motivation	374
9.3	Pooling	379
9.4	Faltung und Pooling als unendlich starke A-priori- Wahrscheinlichkeit	385
9.5	Varianten der grundlegenden Faltungsfunktion	386
9.6	Strukturierte Ausgaben	398
9.7	Datentypen	400
9.8	Effiziente Faltungsalgorithmen	402

9.9	Zufällige oder unüberwachte Merkmale	403
9.10	Die neurowissenschaftliche Basis für CNNs	405
9.11	CNNs und die Geschichte des Deep Learnings	413
10	Sequenzmodellierung: RNNs und rekursive Netze	415
10.1	Auffalten von Berechnungsgraphen	417
10.2	RNNs	420
10.3	Bidirektionale RNNs	436
10.4	Sequenz-zu-Sequenz-Architekturen	439
10.5	Tiefe RNNs	441
10.6	Rekursive neuronale Netze	443
10.7	Die Herausforderung langfristiger Abhängigkeiten	445
10.8	Echo-State-Netze	448
10.9	Leaky-Einheiten und andere Verfahren für mehrere Zeitskalen	451
10.10	Das Long Short-Term Memory und andere Gated RNNs .	453
10.11	Optimierung für langfristige Abhängigkeiten	458
10.12	Explizites Gedächtnis	462
11	Praxisorientierte Methodologie	467
11.1	Performance-Kriterien	468
11.2	Default-Baseline-Modell	471
11.3	Prüfen, ob mehr Daten gesammelt werden sollen	473
11.4	Auswählen von Hyperparametern	475
11.5	Debugging-Verfahren	485
11.6	Beispiel: Erkennen mehrstelliger Zahlen	490
12	Anwendungen	493
12.1	Deep Learning im großen Maßstab	493
12.2	Computer Vision	504
12.3	Spracherkennung	511
12.4	Verarbeitung natürlicher Sprache	514
12.5	Weitere Anwendungen	533
III	Deep-Learning-Forschung	543
13	Lineare Faktorenmodelle	547
13.1	Probabilistische PCA und Faktorenanalyse	548
13.2	Unabhängigkeitsanalyse	549
13.3	Slow Feature Analysis	553

13.4 Sparse Coding	556
13.5 Interpretation der Mannigfaltigkeit der PCA	560
14 Autoencoder	563
14.1 Untervollständige Autoencoder	564
14.2 Regularisierte Autoencoder	565
14.3 Repräsentationsleistung, Schichtgröße und Tiefe	570
14.4 Stochastische Encoder und Decoder	571
14.5 Denoising Autoencoder	573
14.6 Erlernen von Mannigfaltigkeiten mit Autoencodern	578
14.7 Contractive Autoencoder	584
14.8 Prädiktive dünnbesetzte Zerlegung	587
14.9 Anwendungen für Autoencoder	588
15 Representation Learning	591
15.1 Schichtweises unüberwachtes Pretraining mit Greedy-Algorithmen	593
15.2 Transfer Learning und Domänenadaption	602
15.3 Halb-überwachtes Separieren kausaler Faktoren	607
15.4 Verteilte Repräsentation	614
15.5 Exponentielle Verbesserungen durch Tiefe	621
15.6 Hinweise zum Aufdecken der zugrunde liegenden Ursachen	623
16 Strukturierte probabilistische Modelle für Deep Learning	627
16.1 Die Herausforderung der unstrukturierten Modellierung .	628
16.2 Verwenden von Graphen zum Beschreiben der Modellstruktur	633
16.3 Stichprobenentnahme aus graphischen Modellen	651
16.4 Vorteile der strukturierten Modellierung	653
16.5 Lernen anhand von Abhängigkeiten	654
16.6 Inferenz und approximative Inferenz	656
16.7 Der Deep-Learning-Ansatz für strukturierte probabilistische Modelle	657
17 Monte-Carlo-Verfahren	663
17.1 Stichprobenentnahme und Monte-Carlo-Verfahren	663
17.2 Importance Sampling	666
17.3 Markow-Ketten-Monte-Carlo-Verfahren	668
17.4 Gibbs-Sampling	673
17.5 Die Herausforderung, zwischen getrennten Modi zu mischen	674

18 Die Partitionsfunktion	681
18.1 Der Log-Likelihood-Gradient	682
18.2 Stochastische Maximum Likelihood und kontrastive Divergenz	684
18.3 Pseudo-Likelihood	692
18.4 Score Matching und Ratio Matching	695
18.5 Denoising Score Matching	697
18.6 Noise-Contrastive Estimation	698
18.7 Schätzen der Partitionsfunktion	701
19 Approximative Inferenz	711
19.1 Inferenz als Optimierung	713
19.2 Erwartungsmaximierung	714
19.3 MAP-Inferenz und Sparse Coding	716
19.4 Variational Inference und Variational Learning	718
19.5 Erlernte approximative Inferenz	733
20 Tiefe generative Modelle	737
20.1 Boltzmann-Maschinen	737
20.2 Restricted Boltzmann Machines	740
20.3 Deep-Belief-Netze	743
20.4 Deep Boltzmann Machines	747
20.5 Boltzmann-Maschinen für reellwertige Daten	762
20.6 Gefaltete Boltzmann-Maschinen	769
20.7 Boltzmann-Maschinen für strukturierte und sequenzielle Ausgaben	772
20.8 Weitere Boltzmann-Maschinen	773
20.9 Backpropagation durch Zufallsoperationen	775
20.10 Gerichtete generative Netze (Directed Generative Nets) .	780
20.11 Ziehen von Stichproben aus Autoencodern	802
20.12 Generative stochastische Netze	806
20.13 Andere Generierungskonzepte	807
20.14 Bewerten von generativen Modellen	809
20.15 Schlussbemerkungen	812
Literaturverzeichnis	813
Abkürzungsverzeichnis	871
Index	875

Website zum Buch

Auf der Website des Verlags finden Sie Hinweise zur Übersetzung sowie Errata, sofern Fehler bekannt sind unter

www.mitp.de/700

Darüber hinaus gibt es zu diesem Buch eine englischsprachige Website. Sie enthält ergänzende Materialien auf Englisch, darunter Übungen, Vortragsfolien, Korrekturen und andere Ressourcen, die für Leser und Lehrkräfte hilfreich sind. Diese finden Sie unter

www.deeplearningbook.org

Danksagung

Dieses Buch wäre ohne die Hilfe vieler Menschen nicht entstanden.

Wir danken all denen, die unser Buchkonzept unterstützt und uns dabei geholfen haben, seinen Inhalt und Aufbau zu planen: Guillaume Alain, Kyunghyun Cho, Çağlar Gülçehre, David Krueger, Hugo Larochelle, Razvan Pascanu und Thomas Rohée.

Wir danken auch den Menschen, die uns Feedback zum Text selbst gegeben haben. Einige davon haben zu vielen Kapiteln Rückmeldung gegeben: Martín Abadi, Guillaume Alain, Ion Androutsopoulos, Fred Bertsch, Olexa Bilaniuk, Ufuk Can Biçici, Matko Bošnjak, John Boersma, Greg Brockman, Alexandre de Brébisson, Pierre Luc Carrier, Sarath Chandar, Paweł Chilinski, Mark Daoust, Oleg Dashevskii, Laurent Dinh, Stephan Dreseitl, Jim Fan, Miao Fan, Meire Fortunato, Frédéric Francis, Nando de Freitas, Çağlar Gülçehre, Jürgen Van Gael, Javier Alonso García, Jonathan Hunt, Gopi Jeyaram, Chingiz Kabytayev, Lukasz Kaiser, Varun Kanade, Asifullah Khan, Akiel Khan, John King, Diederik P. Kingma, Yann LeCun, Rudolf Mathey, Matías Mattamala, Abhinav Maurya, Kevin Murphy, Oleg Mürk, Roman Novak, Augustus Q. Odena, Simon Pavlik, Karl Pichotta, Eddie Pierce, Kari Pulli, Roussel Rahman, Tapani Raiko, Anurag Ranjan, Johannes Roith, Mihaela Rosca, Halis Sak, César Salgado, Grigory Sapunov, Yoshinori Sasaki, Mike Schuster, Julian Serban, Nir Shabat, Ken Shirriff, Andre Simpelo, David Slate, Scott Stanley, David Sussillo, Ilya Sutskever, Carles Gelada Sáez, Graham Taylor, Valentin Tolmer, Massimiliano Tomassoli, An Tran, Shubhendu Trivedi, Alexey Umnov, Vincent Vanhoucke, Marco Visentini-Scarzanella, Martin Vita, David Warde-Farley, Dustin Webb, Kelvin Xu, Wei Xue, Ke Yang, Li Yao, Zygmunt Zająć und Ozan Çağlayan.

Unser Dank geht auch an jene, deren Feedback einzelnen Kapiteln gewidmet war:

- Notation: Zhang Yuanhang
- Kapitel 1, Einleitung: Yusuf Akgul, Sebastien Bratieres, Samira Ebrahimi, Charlie Gorichanaz, Brendan Loudermilk, Eric Morris, Cosmin Pârvulescu und Alfredo Solano
- Kapitel 2, Lineare Algebra: Amjad Almahairi, Nikola Banić, Kevin Bennett, Philippe Castonguay, Oscar Chang, Eric Fosler-Lussier, Andrej Khalyavin, Sergey Oreshkov, István Petrás, Dennis Prangle, Thomas Rohée, Gitanjali Gulve Sehgal, Colby Toland, Alessandro Vitale und Bob Welland
- Kapitel 3, Wahrscheinlichkeits- und Informationstheorie: John Philip Anderson, Kai Arulkumaran, Vincent Dumoulin, Rui Fa, Stephan Gouws, Artem Oboturov, Antti Rasmus, Alexey Surkov und Volker Tresp
- Kapitel 4, Numerische Berechnung: Tran Lam An Ian Fischer und Hu Yuhuang
- Kapitel 5, Grundlagen für das Machine Learning: Dzmitry Bahdanau, Justin Domingue, Nikhil Garg, Makoto Otsuka, Bob Pepin, Philip Popien, Bharat Prabhakar, Emmanuel Rayner, Peter Shepard, Kee-Bong Song, Zheng Sun und Andy Wu
- Kapitel 6, Tiefe Feedforward-Netze: Uriel Berdugo, Fabrizio Bottarel, Elizabeth Burl, Ishan Durugkar, Jeff Hlywa, Jong Wook Kim, David Krueger, Aditya Kumar Praharaj und Sten Sootla
- Kapitel 7, Regularisierung: Morten Kolbaek, Kshitij Lauria, Inkyu Lee, Sunil Mohan, Hai Phong Phan und Joshua Salisbury
- Kapitel 8, Optimierung beim Trainieren von tiefen Modellen: Marcel Ackermann, Peter Armitage, Rowel Atienza, Andrew Brock, Tegan Maharaj, James Martens, Mostafa Nategh, Kashif Rasul, Klaus Strobl und Nicholas Turner
- Kapitel 9, CNNs: Martín Arjovsky, Eugene Brevdo, Konstantin Divilov, Eric Jensen, Mehdi Mirza, Alex Paino, Marjorie Sayer, Ryan Stout und Wentao Wu

- Kapitel 10, Sequenzmodellierung: RNNs und rekursive Netze: Gökçen Eraslan, Steven Hickson, Razvan Pascanu, Lorenzo von Ritter, Rui Rodrigues, Dmitriy Serdyuk, Dongyu Shi und Kaiyu Yang
- Kapitel 11, Praxisorientierte Methodologie: Daniel Beckstein
- Kapitel 12, Anwendungen: George Dahl, Vladimir Nekrasov und Riba Roscher
- Kapitel 13, Lineare Faktorenmodelle: Jayanth Koushik
- Kapitel 15, Representation Learning: Kunal Ghosh
- Kapitel 16, Strukturierte probabilistische Modelle für Deep Learning: Minh Lê und Anton Varfolom
- Kapitel 18, Die Partitionsfunktion: Sam Bowman
- Kapitel 19, Approximative Inferenz: Yujia Bao
- Kapitel 20, Tiefe generative Modelle: Nicolas Chapados, Daniel Galvez, Wenming Ma, Fady Medhat, Shakir Mohamed und Grégoire Montavon
- Bibliographie: Lukas Michelbacher und Leslie N. Smith

Wir danken auch allen, die uns erlaubt haben, Abbildungen, Grafiken oder Daten aus ihren Werken zu verwenden. Wir haben diese Wiedergabe in den Bildunterschriften angemerkt.

Wir danken Lu Wang dafür, dass er pdf2htmlEX programmiert hat, mit dem wir die Webversion des Buchs erstellt haben, sowie für sein Angebot, uns bei der Verbesserung der Qualität des HTML-Codes zu unterstützen.

Wir danken Ians Ehefrau Daniela Flori Goodfellow für ihre Geduld, die sie Ian während der Abfassung des Buchs entgegengebracht hat. Und natürlich für ihre Hilfe beim Korrektorat.

Wir möchten dem Team hinter Google Brain dafür danken, dass es eine geeignete Umgebung geschaffen hat, in der wir so viel Zeit für die Arbeit an diesem Buch verbringen und gleichzeitig Feedback und Hilfe von den Kollegen annehmen konnten. Ganz besonders möchten wir Ians ehemaligem Vorgesetzten Greg Corrado und seinem derzeitigen Vorgesetzten Samy Bengio für ihre Unterstützung bei diesem Projekt danken. Und schließlich gebührt unser Dank Geoffrey Hinton, der uns bei Schwierigkeiten stets ermuntert hat, weiterzumachen.

Über die Fachkorrektoren zur deutschen Ausgabe

Fachkorrektor für den Bereich Deep Learning

Alexander Bresk ist Unternehmer, Data Engineer, Machine Teacher, Autor und Basketball-Nerd. Er arbeitet als Senior Data Engineer bei der LOVOO GmbH. Dort beschäftigt er sich mit den Themen Recommendation und Online Segmentation. Seinen Master of Science erwarb Alexander an der HTW Dresden, wo er Angewandte Informationstechnologien studierte. Seit seinem Master im Bereich des Question Answering forscht er aktiv an Verfahren zur Verarbeitung natürlicher Sprache (NLP), wobei er unter anderem Deep Learning Frameworks einsetzt. Außerdem organisiert er zusammen mit Kollegen und Freunden Meetups, Konferenzen und andere Veranstaltungen in den Bereichen Tech, Machine Learning und Startups. Mit seiner Firma Machine Rockstars berät er Unternehmen beim Einstieg in das Thema Digitalisierung sowie bei der Einführung von Machine Learning.

Alexander Bresks Website: <http://alexander.bre.sk>

Fachkorrektor für den Bereich Mathematik

Stefan Niedergriese hat Mathematik studiert und arbeitet bei einem Versicherungsunternehmen in der mathematischen Abteilung.

Notation

Dieser Abschnitt dient als Referenz. Er listet die im gesamten Buch verwendete Notation auf. Falls Ihnen einzelne der mathematischen Konzepte unbekannt sind, sollten Sie die Kapitel 2 bis 4 lesen – dort gehen wir auf die meisten davon näher ein.

Zahlen und Vektoren

- a Ein Skalar (ganzzahlig oder reell)
- \mathbf{a} Ein Vektor
- \mathbf{A} Eine Matrix
- \mathbf{A} Ein Tensor
- \mathbf{I}_n Einheitsmatrix mit n Zeilen und n Spalten
- \mathbf{I} Einheitsmatrix mit aus dem Kontext ersichtlicher Dimensionalität
- $\mathbf{e}^{(i)}$ Standardbasisvektor $[0, \dots, 0, 1, 0, \dots, 0]$ mit einer 1 an der Stelle i
- $\text{diag}(\mathbf{a})$ Eine quadratische Diagonalmatrix mit Diagonaleinträgen aus \mathbf{a}
 - a Eine skalare Zufallsvariable
 - \mathbf{a} Eine vektorwertige Zufallsvariable
 - \mathbf{A} Eine matrixwertige Zufallsvariable

Mengen und Graphen

- \mathbb{A} Eine Menge
- \mathbb{R} Die Menge der reellen Zahlen

$\{0, 1\}$	Die Menge, die 0 und 1 enthält
$\{0, 1, \dots, n\}$	Die Menge aller ganzen Zahlen zwischen 0 und n
$[a, b]$	Das reellwertige Intervall der Mengen a und b
$(a, b]$	Das reellwertige Intervall ohne a , jedoch mit b
$\mathbb{A} \setminus \mathbb{B}$	Mengendifferenz (Komplement), also die Menge der Elemente aus \mathbb{A} , die nicht in \mathbb{B} enthalten sind
\mathcal{G}	Ein Graph
$Pa_{\mathcal{G}}(x_i)$	Die Eltern von x_i in \mathcal{G}

Indizes

a_i	Element i des Vektors \mathbf{a} , die Indexmenge beginnt mit 1
a_{-i}	Alle Elemente des Vektors \mathbf{a} bis auf das Element i
$A_{i,j}$	Element i, j der Matrix \mathbf{A}
$A_{i,:}$	Zeile i der Matrix \mathbf{A}
$A_{:,i}$	Spalte i der Matrix \mathbf{A}
$A_{i,j,k}$	Element (i, j, k) eines 3-D-Tensors \mathbf{A}
$\mathbf{A}_{::,i}$	2-D-Schnitt eines 3-D-Tensors
a_i	Element i des Zufallsvektors \mathbf{a}

Lineare Algebra

\mathbf{A}^{\top}	Transponierte der Matrix \mathbf{A}
\mathbf{A}^{+}	Moore-Penrose-Pseudoinverse von \mathbf{A}
$\mathbf{A} \odot \mathbf{B}$	Elementweises Produkt (Hadamard-Produkt) aus \mathbf{A} und \mathbf{B}
$\det(\mathbf{A})$	Determinante von \mathbf{A}

Analysis

$\frac{dy}{dx}$	Ableitung von y bezüglich x
$\frac{\partial y}{\partial x}$	Partielle Ableitung von y bezüglich x

$\nabla_{\mathbf{x}}y$	Gradient von y bezüglich \mathbf{x}
$\nabla_{\mathbf{X}}y$	Matrixableitungen von y bezüglich \mathbf{X}
$\nabla_{\mathbf{X}}y$	Tensor mit Ableitungen von y bezüglich \mathbf{X}
$\frac{\partial f}{\partial \mathbf{x}}$	Jacobi-Matrix $\mathbf{J} \in \mathbb{R}^{m \times n}$ von $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$
$\nabla_{\mathbf{x}}^2 f(\mathbf{x})$ oder $\mathbf{H}(f)(\mathbf{x})$	Hesse-Matrix von f im Eingabepunkt \mathbf{x}
$\int f(\mathbf{x})d\mathbf{x}$	Bestimmtes Integral über gesamten Definitionsbereich \mathbf{x}
$\int_{\mathbb{S}} f(\mathbf{x})d\mathbf{x}$	Bestimmtes Integral bezüglich \mathbf{x} über Menge \mathbb{S}

Wahrscheinlichkeits- und Informationstheorie

$a \perp b$	Die Zufallsvariablen a und b sind unabhängig
$a \perp b \mid c$	Sie sind bedingt unabhängig, wenn c zutrifft
$P(a)$	Eine Wahrscheinlichkeitsverteilung über eine diskrete Variable
$p(a)$	Eine Wahrscheinlichkeitsverteilung über eine stetige Variable oder eine Variable, deren Typ nicht angegeben wurde
$a \sim P$	Zufallsvariable a mit der Verteilung P
$\mathbb{E}_{x \sim P}[f(x)]$ oder $\mathbb{E}f(x)$	Erwartung für $f(x)$ bezüglich $P(x)$
$\text{Var}(f(x))$	Varianz von $f(x)$ unter $P(x)$
$\text{Cov}(f(x), g(x))$	Kovarianz von $f(x)$ und $g(x)$ unter $P(x)$
$H(x)$	Shannon-Entropie der Zufallsvariablen x
$D_{\text{KL}}(P \parallel Q)$	Kullback-Leibler-Divergenz von P und Q
$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Normalverteilung über \mathbf{x} mit Mittel $\boldsymbol{\mu}$ und Kovarianz $\boldsymbol{\Sigma}$

Funktionen

$f : \mathbb{A} \rightarrow \mathbb{B}$	Die Funktion f des Definitionsbereichs \mathbb{A} mit dem Wertebereich \mathbb{B}
$f \circ g$	Komposition der Funktionen f und g

$f(\mathbf{x}; \boldsymbol{\theta})$	Eine Funktion \mathbf{x} , parametrisiert durch $\boldsymbol{\theta}$. (Manchmal schreiben wir auch $f(\mathbf{x})$ und lassen das Argument $\boldsymbol{\theta}$ weg, um die Notation zu vereinfachen.)
$\log x$	Natürlicher Logarithmus von x
$\sigma(x)$	Logistische Sigmoidfunktion, $\frac{1}{1 + \exp(-x)}$
$\zeta(x)$	Softplus, $\log(1 + \exp(x))$
$\ \mathbf{x}\ _p$	L^p -Norm von \mathbf{x}
$\ \mathbf{x}\ $	L^2 -Norm von \mathbf{x}
x^+	Positiver Teil von x , d.h. $\max(0, x)$
$\mathbf{1}_{\text{Bedingung}}$	ist 1, wenn die Bedingung zutrifft, anderenfalls 0

Manchmal wenden wir eine Funktion f , deren Argument ein Skalar ist, auf einen Vektor, eine Matrix oder einen Tensor an: $f(\mathbf{x})$, $f(\mathbf{X})$ oder $f(\mathbf{X})$. Dies bezeichnet die elementweise Anwendung von f auf den Vektor, die Matrix oder den Tensor. Beispiel: Ist $\mathbf{C} = \sigma(\mathbf{X})$, dann gilt $C_{i,j,k} = \sigma(X_{i,j,k})$ für alle gültigen Werte von i , j und k .

Datensätze und Verteilungen

p_{data}	Die datengenerierende Verteilung
\hat{p}_{data}	Die empirische Verteilung, die anhand der Trainingsdatenmenge definiert wurde
\mathbb{X}	Eine Sammlung mit Trainingsbeispielen
$\mathbf{x}^{(i)}$	Das i -te Beispiel (Eingabe) eines Datensatzes
$y^{(i)}$ oder $\mathbf{y}^{(i)}$	Der mit $\mathbf{x}^{(i)}$ für das überwachte Lernen verknüpfte Zielwert
\mathbf{X}	Die Matrix $m \times n$ mit dem Eingabebeispiel $\mathbf{x}^{(i)}$ in Zeile $\mathbf{X}_{i,:}$

1

Einleitung

Schon lange träumen Erfinder von einer Maschine, die selbstständig denken kann. Dieser Wunsch lässt sich bis zu den alten Griechen zurückverfolgen. Die Sagengestalten Pygmalion, Dädalus und Hephaistos sind allesamt berühmte Erfinder. Und Galatea, Talos sowie Pandora lassen sich sämtlich als künstliche Lebensformen betrachten (*Ovid und Martin*, 2004; *Sparkes*, 1996; *Tandy*, 1997).

Als man – über 100 Jahre vor dem Bau des ersten programmierbaren Rechners – über die Möglichkeit solcher Computer nachdachte, fragten sich die Menschen, ob diese Maschinen wohl eines Tages intelligent werden würden (*Lovelace*, 1842). Heute ist die **Künstliche Intelligenz** (KI) eine erfolgreiche Disziplin mit einer Vielzahl praktischer Anwendungen und aktiver Forschungsbereiche. Wir verlassen uns auf »intelligente« Software, um Routineaufgaben zu erledigen, Sprache oder Bilder zu erfassen, medizinische Diagnosen zu stellen und uns bei grundlegender wissenschaftlicher Forschung unterstützen zu lassen.

In der Anfangszeit der Künstlichen Intelligenz hat die Forschung schnell Probleme in Angriff genommen und gelöst, die für Menschen schwierig zu lösen, aber relativ unkompliziert für Computer sind – dabei geht es um Probleme, die durch eine Reihe formaler mathematischer Regeln beschrieben werden können. Die wahre Herausforderung an eine Künstliche Intelligenz besteht jedoch darin, Aufgaben zu erledigen, mit denen Menschen keinerlei Probleme haben, obwohl sie schwer in Worte zu fassen sind – solche Dinge also, die wir intuitiv erledigen und die wir nebenbei tun. Gute Beispiele dafür sind das Erkennen der gesprochenen Sprache oder von Gesichtern in einer Fotografie.

In diesem Buch geht es um die Lösung solch intuitiver Aufgabenstellungen. Die Lösung besteht darin, Computern zu ermöglichen, aus Erfahrungen zu lernen und die Welt so erfassen, als ob sie aus hierarchischen Konzepten besteht, wobei jedes Konzept wiederum definiert ist durch die Beziehung zu einfacheren Konzepten. Indem Computer Wissen aus Erfahrung sammeln, ist es nicht notwendig, mit menschlicher Arbeitskraft formal das Wissen genau zu beschreiben, das der Computer zum Lösen solcher Aufgaben braucht. Die Hierarchie der Konzepte ermöglicht dem Computer das Erlernen komplexer Konzepte, indem er diese aus einfacheren zusammensetzt. Wenn wir diese Modelle grafisch darstellen, blicken wir von oben auf viele Schichten, die sich weit in die Tiefe erstrecken. Deshalb nennen wir diese Methode der Künstlichen Intelligenz **Deep Learning**.

Viele der frühen Erfolge der Künstlichen Intelligenz fanden in relativ sterilen und formalen Umgebungen statt. Das hatte den Vorteil, dass die Computer nur wenig Wissen über die echte Welt benötigten. Ein Beispiel ist IBMs Deep Blue. Dieses Schachprogramm besiegte im Jahr 1997 den Weltmeister Garri Kasparow (*Hsu, 2002*). Beim Schach handelt es sich um ein recht einfaches Universum mit nur 64 Feldern und 32 Figuren, deren Bewegungsmöglichkeiten zudem durch klare Regeln eingeschränkt sind. Die Entwicklung einer erfolgreichen Schachstrategie ist eine echte Meisterleistung. Das liegt aber nicht daran, dass es so kompliziert wäre, einem Computer die Regeln und zulässigen Züge beizubringen. Tatsächlich lassen sich die Regeln in eine sehr kurze Liste formaler Anweisungen fassen, die durch den Programmierer im Vorfeld eingegeben werden.

Die Ironie liegt gerade darin, dass die abstrakten und formalen Aufgaben, die von uns Menschen höchste Geistesleistungen erfordern, für den Computer am unteren Ende der Schwierigkeitsskala stehen. Schon lange können Computer auch die besten Großmeister im Schach schlagen. Doch erst seit kurzer Zeit vermögen sie es, auch bei Objekt- und Spracherkennung mit den Menschen gleichzuziehen. Wir alle greifen im Alltag auf einen gewaltigen Erfahrungsschatz zurück, mit dem wir uns in der Welt zurechtfinden. Ein Großteil unseres Wissens ist subjektiv und intuitiv, lässt sich also nur schwer formal ausdrücken. Damit Computer auf intelligente Weise agieren können, benötigen sie genau dieses Wissen und diese Erfahrungen. Eine der größten Herausforderungen auf dem Gebiet der Künstlichen Intelligenz besteht darin, dem Computer dieses informale und nicht eindeutige Wissen zu vermitteln.

In diversen Projekten der Künstlichen Intelligenz wurde versucht, unser Weltwissen fest in formalen Sprachen zu codieren. Mithilfe logischer Inferenzregeln kann ein Computer auf dieser Basis Schlüsse ziehen. Das ist die sogenannte **wissensbasierte** Herangehensweise an die Künstliche

Intelligenz. Allerdings hat keines dieser Projekte den großen Durchbruch gebracht. Ein berühmtes Beispiel ist Cyc (*Lenat und Guha*, 1989). Cyc ist eine Inferenzmaschine samt Datenbank mit Aussagen in der Sprache CycL. Diese Aussagen werden von menschlichen Supervisoren recht umständlich eingegeben. Für uns Menschen ist es ein großes Problem, formale Regeln aufzustellen, die komplex genug sind, um die Welt korrekt zu beschreiben. Das zeigte sich, als Cyc einen Bericht über Fred, der sich morgens rasierte, gründlich missverstand (*Linde*, 1992). Die Inferenzmaschine kam zu dem Schluss, dass der Bericht inkonsistent sei. Warum? Nun, Cyc wusste, dass Menschen keine elektrischen Bauteile enthalten. Aber da Fred einen elektrischen Rasierapparat in der Hand hielt, ging das System davon aus, dass »FredBeimRasieren« zum Teil aus Elektronik bestand. Daher wollte die Inferenzmaschine wissen, ob Fred beim Rasieren noch ein Mensch ist.

Die Schwierigkeiten, denen sich Systeme gegenüber sehen, die sich auf stark codiertes Wissen stützen, legen nahe, dass KI-Systeme in der Lage sein müssen, ihr eigenes Wissen zu erwerben, indem sie aus Rohdaten Muster extrahieren. Diese Fähigkeit wird als **Machine Learning** bezeichnet. Das Aufkommen des Machine Learnings versetzte Computer in die Lage, Probleme anzugehen, für deren Lösung Wissen über die Realität erforderlich ist, und dafür scheinbar subjektive Entscheidungen zu treffen. Ein einfacher Machine-Learning-Algorithmus namens **logistische Regression** kann entscheiden, ob ein Kaiserschnitt angeraten ist (*Mor-Yosef et al.*, 1990). Ein einfacher Machine-Learning-Algorithmus namens **Naive Bayes** kann erwünschte E-Mails und Spam voneinander trennen.

Die Leistung dieser einfachen Machine-Learning-Algorithmen hängt stark von der **Repräsentation** (also der Darstellung oder Aufbereitung) der Ausgangsdaten ab. Ein Beispiel: Wenn logistische Regression verwendet wird, um Empfehlungen zu einem Kaiserschnitt anzugeben, untersucht nicht das KI-System selbst die Patientin. Stattdessen gibt ein Arzt relevante Daten in das System ein, zum Beispiel ob eine Gebärmutternarbe vorhanden ist. Jede in der Repräsentation der Patientin enthaltene Angabe wird als **Merkmals** bezeichnet. Die logistische Regression »lernt«, wie jedes dieser Merkmale der Patientin mit unterschiedlichen Resultaten korreliert. Sie kann jedoch keinerlei Einfluss darauf nehmen, wie die Merkmale definiert werden. Würde der logistischen Regression eine MRT-Aufnahme der Patientin anstelle des formalisierten Arztberichts zur Verfügung gestellt, könnte das System damit keine nützliche Vorhersage machen. Die einzelnen Pixel des MRTs haben kaum eine Korrelation mit möglichen Komplikationen, die während der Entbindung auftreten können.

Diese Abhängigkeit von der Repräsentation ist ein allgemeines Phänomen, das sich durch die gesamte Informatik und sogar unseren Alltag zieht. In der Informatik erfolgt ein Vorgang wie das Durchsuchen einer Datensammlung exponentiell schneller, wenn die Datensammlung auf intelligente Weise strukturiert und indiziert ist. Menschen können problemlos mit arabischen Ziffern rechnen, doch mit römischen Zahlenzeichen fällt es den meisten sehr viel schwerer. Kein Wunder also, dass die Auswahl der Repräsentation oder Darstellung einen gewaltigen Effekt auf die Leistungsfähigkeit von Machine-Learning-Algorithmen hat. Abbildung 1.1 zeigt ein einfaches Beispiel.

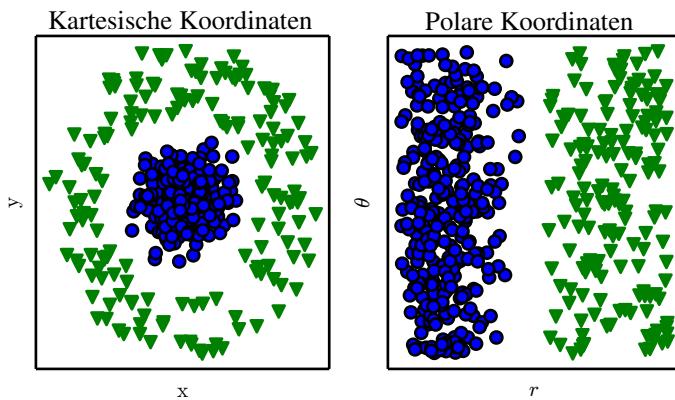


Abbildung 1.1: Beispiel für unterschiedliche Darstellungen: Zwei Datenkategorien sollen durch eine Gerade in einem Punktdiagramm voneinander getrennt werden. Im linken Diagramm werden die Daten mithilfe kartesischer Koordinaten dargestellt. Das macht die Lösung der Aufgabe unmöglich. Im rechten Diagramm sind die Daten mithilfe polarer Koordinaten dargestellt und eine einfache Senkrechte reicht aus, um die Aufgabe zu lösen. (Abbildung gemeinsam mit David Warde-Farley erstellt.)

Viele Aufgaben der Künstlichen Intelligenz können gelöst werden, indem man die richtige Zusammenstellung von Merkmalen konzipiert, die für eine Aufgabe eine Rolle spielen, und diese Merkmale einem einfachen Machine-Learning-Algorithmus zur Verfügung stellt. Um beispielsweise einen Sprecher anhand des Klangs seiner Stimme zu identifizieren, kann sich das Volumen seines Stimmapparats als nützliches Merkmal erweisen. Dieses Merkmal liefert einen guten Hinweis darauf, ob es sich beim Sprecher um einen Mann, eine Frau oder ein Kind handelt.

Für viele Aufgaben besteht jedoch die Schwierigkeit darin zu wissen, welche Merkmale wichtig sind und extrahiert werden müssen. Beispiel: Ein Programm soll Fotografien prüfen und entscheiden, ob darauf ein Pkw zu sehen ist. Wie wir wissen, haben Pkws Räder. Ein Merkmal könnte also das

Vorhandensein von Rädern sein. Aber wie lässt sich ein Rad in Form von Pixelwerten beschreiben? Schwierig, nicht wahr? Ein Rad weist eine einfache geometrische Form auf, aber im Bild gibt es viele andere Elemente, die eine Erkennung erschweren: Schatten, Sonnenblendungen auf Metallteilen des Rads, Kotflügel oder andere Objekte, die Teile des Rads verdecken usw.

Eine Lösung besteht darin, Machine Learning zu verwenden, um nicht nur aus dem Mapping zwischen Repräsentation und Ausgabe, sondern auch aus der Repräsentation (oder Darstellung) selbst Erkenntnisse zu gewinnen. Dieser Ansatz wird als **Representation Learning** bezeichnet. Erlernte Repräsentationen führen häufig zu einer sehr viel besseren Leistung gegenüber individuell angepassten Repräsentationen. Und sie befähigen KI-Systeme dazu, sich bei minimaler menschlicher Intervention schnell an neue Aufgaben anzupassen. Ein Representation-Learning-Algorithmus kann eine gutes Set von Merkmalen für eine einfache Aufgabe in wenigen Minuten erkennen. Bei komplexen Aufgaben kann er einige Stunden oder Monate benötigen. Die händische Konzipierung von Merkmalen für eine komplexe Aufgabe dagegen erfordert jede Menge menschliche Zeit und Mühe – ein Forschungsteam könnte Jahrzehnte damit beschäftigt sein.

Ein Musterbeispiel für einen Representation-Learning-Algorithmus ist der **Autoencoder**. Ein Autoencoder ist eine Kombination aus einer **Encoder**-Funktion, die Eingabedaten in eine andere Repräsentation umwandelt, und einer **Decoder**-Funktion, die diese neue Repräsentation wieder zurück in das Ausgangsformat umwandelt. Autoencoder werden darauf trainiert, so viele Daten wie möglich zu erhalten, wenn eine Eingabe die Encoder- und Decoder-Funktion durchläuft. Gleichzeitig werden sie aber auch darauf trainiert, der neuen Repräsentation einige nützliche Eigenschaften zu verleihen. Je nach Art des Autoencoders liegt das Augenmerk auf unterschiedlichen Eigenschaften.

Beim Entwickeln von Merkmalen oder Algorithmen zum Erlernen von Merkmalen ist es üblicherweise unser Ziel, die **Faktoren der Variation** zu separieren, die die beobachteten Daten erklären. In diesem Kontext bezieht sich das Wort »Faktor« rein auf die unterschiedlichen Einflussquellen. Der Begriff hat nichts mit einer Multiplikation zu tun. Diese Faktoren sind häufig keine direkt beobachteten Größen, sondern entweder unbeobachtete Objekte oder unbeobachtete Kräfte in der physischen Welt, die ihrerseits die einer Beobachtung zugänglichen Größen beeinflussen. Es kann sich auch um Konstrukte des menschlichen Denkens handeln, die nützliche verallgemeinerte Erklärungen oder vermutete Ursachen für die beobachteten Daten liefern. Sie können sich diese Faktoren als Konzepte oder Abstraktionen vorstellen, die dabei helfen, der großen Variabilität der Daten einen Sinn zu

geben. Beim Analysieren einer Sprachaufzeichnung gehören zu den Faktoren der Variation zum Beispiel das Alter und das Geschlecht des Sprechers, der Dialekt und die verwendeten Wörter. Beim Analysieren einer Abbildung eines Fahrzeugs wiederum gehören dessen Position und Farbe, der Betrachtungswinkel und die Helligkeit des Sonnenlichts zu den Faktoren der Variation.

Bei vielen KI-Anwendungen in der Praxis liegt eine große Schwierigkeit darin, dass viele der Faktoren der Variation sich auf jedes einzelne Datenelement auswirken, das wir beobachten können. Die einzelnen Pixel des Fotos eines roten Autos können bei Nacht nahezu schwarz erscheinen. Die Form der Fahrzeugumrisse ist abhängig vom Blickwinkel. Die meisten Anwendungen erfordern es von uns, dass wir die Faktoren der Variation *separieren* und diejenigen verwerfen, die nicht von Belang sind.

Das Extrahieren solch übergeordneter, abstrakter Merkmale aus den Rohdaten kann sich als extrem schwierig erweisen. Viele der Faktoren der Variation wie ein Dialekt können nur bestimmt werden, wenn ein tiefes, nahezu menschliches Verständnis der Daten vorhanden ist. Wenn es genauso schwierig ist, eine Repräsentation zu erzielen wie das ursprüngliche Problem zu lösen, scheint uns das Representation Learning auf den ersten Blick nicht hilfreich zu sein.

Deep Learning löst dieses zentrale Problem beim Representation Learning, indem Repräsentationen eingesetzt werden, die in Form von anderen simpleren Repräsentationen dargestellt werden. Deep Learning befähigt den Computer, komplexe Konzepte aus einfacheren Konzepten zu konstruieren. Abbildung 1.2 zeigt, wie ein Deep-Learning-System das Konzept einer Personenfotografie darstellen kann, indem es einfachere Konzepte kombiniert wie hier Ecken und Umrisse, die wiederum über Kantenverläufe definiert werden.

Ein Musterbeispiel für ein Deep-Learning-Modell ist ein tiefes Feedforward-Netz, auch **mehrschichtiges Perzepron** (engl. *multilayer perceptron*, MLP) genannt. Ein mehrschichtiges Perzepron ist eine mathematische Funktion, die eine Menge von Eingabewerten entsprechenden Ausgabewerten zuordnet. Die Funktion wird aus vielen einfacheren Funktionen zusammengesetzt. Dabei können Sie sich vorstellen, dass jede Anwendung einer anderen mathematischen Funktion eine neue Repräsentation der Eingaben liefert.

Das Konzept des Erlernens der richtigen Repräsentation für die Daten stellt eine Perspektive auf das Deep Learning dar. Eine weitere Perspektive rückt die Tiefe in den Mittelpunkt: Sie ermöglicht dem Computer das Erlernen eines mehrstufigen Computerprogramms. Jede Schicht der

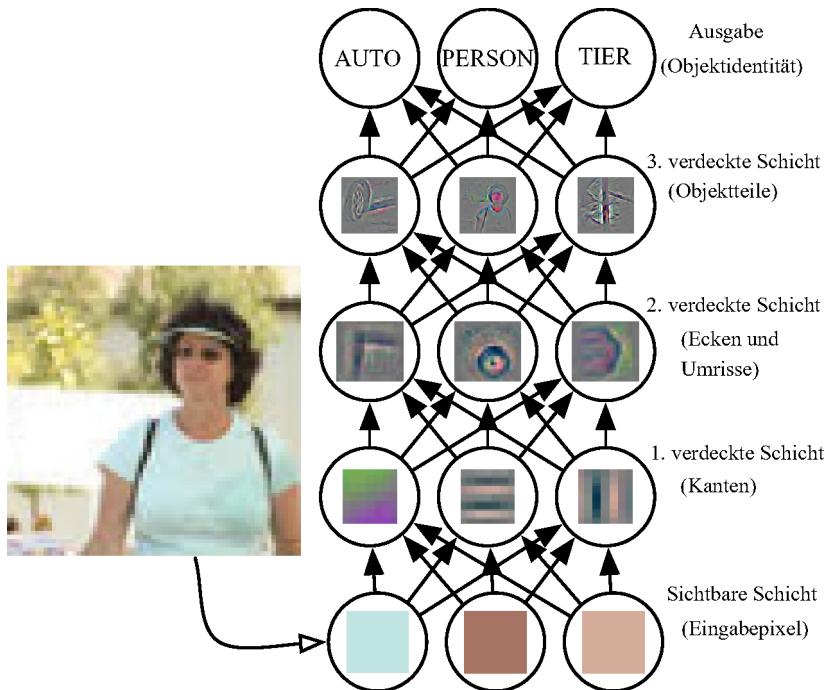


Abbildung 1.2: Darstellung eines Deep-Learning-Modells. Für den Computer ist es nicht einfach, die Bedeutung roher Sensoreingaben zu verstehen. Ein Beispiel ist dieses Foto, das als Ansammlung von Pixelwerten dargestellt wird. Die Funktion zur Zuordnung einer Pixelmenge zu einer Objektidentität ist sehr kompliziert. Das Erlernen oder Bewerten dieser Zuordnung erscheint unmöglich, wenn man das Problem direkt angehen will. Mit Deep Learning wird die Schwierigkeit jedoch vermindert, indem die erwünschte komplizierte Zuordnung in eine Reihe verschachtelter einfacherer Zuordnungen aufgeteilt wird. Jede davon beschreibt eine andere Schicht des Modells. Die Eingabe wird als **sichtbare Schicht** dargestellt. Die Bezeichnung wurde gewählt, weil sie die Variablen enthält, die wir beobachten können. Darauf werden in einer Reihe von **verdeckten Schichten** immer abstraktere Bildmerkmale extrahiert. Diese Schichten werden als »verdeckt« bezeichnet, weil ihre Werte nicht in den Daten enthalten sind, sondern das Modell selbst entscheiden muss, welche Konzepte zum Erklären der Beziehungen in den beobachteten Daten hilfreich sind. Die gezeigten Abbildungen visualisieren die Art von Merkmalen, die durch jede verdeckte Einheit dargestellt werden. Anhand der Pixel können in der ersten Schicht durch Vergleich der Helligkeitswerte benachbarter Pixel die Kanten bestimmt werden. Auf Basis der in der ersten verdeckten Schicht ermittelten Kanten erfolgt in der zweiten verdeckten Schicht die Suche nach Ecken und verlängerten Konturen, die als Aneinanderreihung von Kanten erkannt werden. Nachdem die zweite verdeckte Schicht das Bild anhand von Ecken und Konturen beschrieben hat, sucht die dritte verdeckte Schicht nach kompletten Teilen bestimmter Objekte. Dazu wird auf bestimmte Sammlungen von Konturen und Ecken geprüft. Im Ergebnis steht eine Bildbeschreibung in Form von Objektteilen zur Verfügung, die dann zum Erkennen der Objekte im Foto verwendet werden kann. (Abbildungen mit freundlicher Genehmigung von Zeiler und Fergus (2014))

Repräsentation wird dabei als Speicherzustand des Computers nach der parallelen Ausführung einer weiteren Anweisungsgruppe betrachtet. Netze mit einer größeren Tiefe können mehr Anweisungen nacheinander ausführen. Sequenzielle Anweisungen bieten mehr Leistungsfähigkeit, denn spätere Anweisungen können auf die Ergebnisse früherer Anweisungen zurückgreifen. Bei dieser Perspektive auf das Deep Learning codieren nicht alle Informationen in den Aktivierungen einer Schicht notwendigerweise Faktoren der Variation, die die Eingabe erklären. Die Repräsentation speichert außerdem Informationen über den Zustand, mit deren Hilfe ein Programm ausgeführt werden kann, das die Eingabe auswertet. Diese Zustandsinformationen ähneln einem Zähler oder Pointer in einem klassischen Computerprogramm. Sie haben nichts mit dem eigentlichen Inhalt der Eingabe zu tun, sondern helfen dem Modell, die Verarbeitung zu strukturieren.

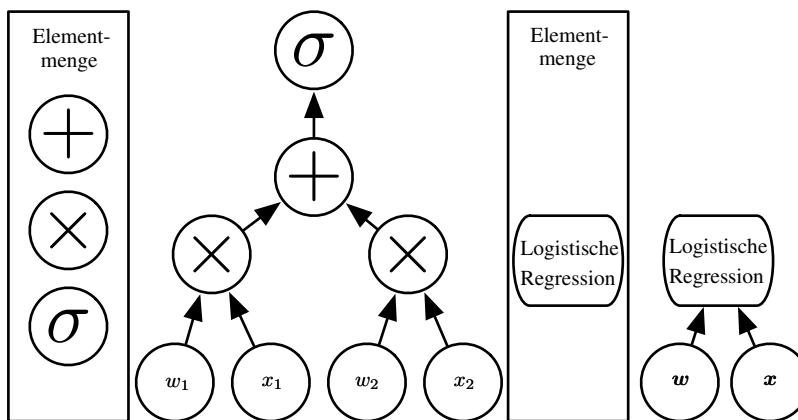


Abbildung 1.3: Abbildung des Berechnungsverlaufs (Graphen) für eine Eingabe, bei dem jeder Knoten auf dem Weg zur Ausgabe eine Berechnung durchführt. Die Tiefe ist der längste Pfad von der Eingabe zur Ausgabe. Sie richtet sich allerdings danach, was als möglicher Berechnungsschritt definiert ist. Die hier dargestellte Berechnung ist die Ausgabe eines logistischen Regressionsmodells $\sigma(\mathbf{w}^T \mathbf{x})$ mit σ als logistischer Sigmoidfunktion. Bei Verwendung von Addition, Multiplikation und logistischen Sigmoidfunktionen als Elemente unserer Programmiersprache hat dieses Modell eine Tiefe von 3. Gilt die logistische Regression als eigenes Element, hat das Modell die Tiefe 1.

Es gibt zwei wesentliche Möglichkeiten zum Messen der Modelltiefe. Bei der ersten wird die Anzahl sequenzieller Anweisungen bestimmt, die zum Bewerten der Architektur ausgeführt werden müssen. Das entspricht quasi dem längsten Pfad in einem Ablaufplan, der die Berechnung der einzelnen Ausgaben des Modells anhand der Eingaben beschreibt. So wie zwei äquivalente Computerprogramme abhängig von der Programmiersprache

unterschiedlich umfangreich sind, kann auch eine Funktion in Ablaufplänen unterschiedlicher Tiefe dargestellt werden, denn letztlich richtet sich die Tiefe danach, was im Ablaufplan als ein Schritt gilt. Abbildung 1.3 zeigt, dass die Auswahl der Sprache zu zwei unterschiedlichen Tiefen für dieselbe Architektur führt.

Ein weiterer Ansatz, der in tiefen probabilistischen Modellen zum Einsatz kommt, setzt die Tiefe eines Modells nicht mit der Tiefe des Berechnungsverlaufs gleich, sondern mit der Tiefe des Graphen, der beschreibt, wie die Konzepte zueinander in Beziehung stehen. In diesem Fall kann die Tiefe des Ablaufplans, mit dem die Repräsentation jedes Konzepts berechnet wird, sehr viel tiefer als der Graph der Konzepte selbst ausfallen. Das liegt daran, dass das Verständnis des Systems für die einfacheren Konzepte durch Bereitstellen von Informationen zu den komplexeren Konzepten verfeinert werden kann. Beispiel: Ein KI-System, das ein Foto eines Gesichts betrachtet, bei dem ein Auge im Schatten liegt, erkennt möglicherweise zunächst nur ein Auge. Nachdem festgestellt wurde, dass das Foto ein Gesicht enthält, kann geschlossen werden, dass wahrscheinlich auch ein zweites Auge zu sehen sein muss. In diesem Fall enthält der Graph nur zwei Schichten: eine für die Augen und eine für Gesichter. Aber der Graph für die Berechnungen enthält $2n$ Schichten, wenn wir unsere Schätzung für jedes Konzept der anderen n Fälle verfeinern.

Da nicht immer eindeutig ist, welche der beiden Betrachtungsweisen – Tiefe des Berechnungsgraphen oder Tiefe des Graphen der probabilistischen Modellierung – von höherer Relevanz ist, und da verschiedene Menschen unterschiedliche Mengen der kleinsten Elemente für den Aufbau ihrer Graphen nutzen, gibt es keinen allgemeingültigen Wert für die Tiefe einer Architektur, genauso wie es nicht einen einzigen richtigen Wert für die Länge eines Computerprogramms gibt. Auch herrscht kein Konsens darüber, wie tief ein Modell sein muss, damit es als »tief« bezeichnet werden darf. Allerdings sind wir auf der sicheren Seite, wenn wir Deep Learning als Untersuchung von Modellen bezeichnen, die eine höhere Anzahl erlernter Funktionen oder Konzepte als im klassischen Machine Learning aufweisen.

Zusammenfassend lässt sich sagen, dass Deep Learning, das Thema dieses Buches, eine Methode der Künstlichen Intelligenz ist. Konkret handelt es sich um einen Unterbereich des Machine Learnings, ein Verfahren, das es Computersystemen ermöglicht, sich mit Erfahrungen und Daten zu verbessern. Wir behaupten, dass Machine Learning der einzige praktikable Ansatz für die Entwicklung von KI-Systemen ist, die in der komplexen realen Welt funktionieren. Deep Learning ist ein spezielles Teilgebiet des Machine Learnings, das hohe Leistungsfähigkeit und Flexibilität erreicht,

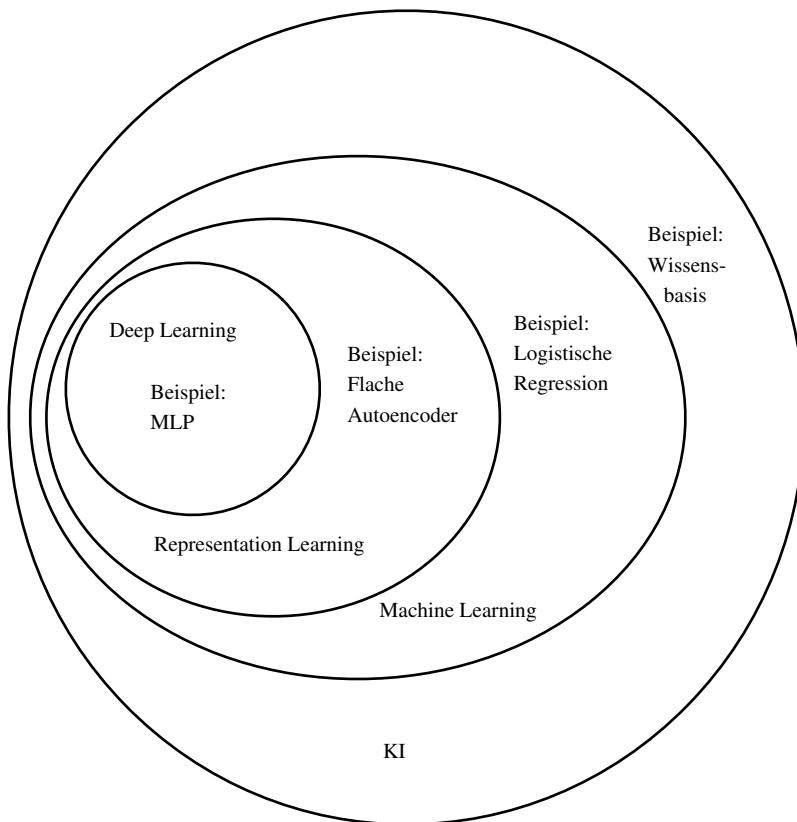


Abbildung 1.4: Dieses Venn-Diagramm zeigt, dass Deep Learning ein Unterbereich des Representation Learnings ist, das wiederum eine Machine-Learning-Variante ist, die für viele – aber nicht alle – KI-Ansätze verwendet wird. Für jeden Kreis im Venn-Diagramm ist beispielhaft eine KI-Technologie genannt.

indem es die Welt als verschachtelte Hierarchie von Konzepten abbildet. Dabei wird jedes Konzept wiederum in Beziehung zu einfacheren Konzepten definiert; abstraktere Repräsentationen werden ebenso mithilfe weniger abstrakterer Darstellungen berechnet. Abbildung 1.4 verdeutlicht die Beziehungen zwischen den verschiedenen KI-Disziplinen. Abbildung 1.5 enthält eine schematische Übersicht der Funktionsweise einzelner KI-Systeme.

1.1 Für wen ist dieses Buch gedacht?

Dieses Buch ist für eine Vielzahl von Lesern nützlich. Bei der Abfassung hatten wir insbesondere zwei Gruppen im Sinn. Eine davon sind Studierende (Bachelor und Master) im Bereich des Machine Learnings, zum Beispiel jene,

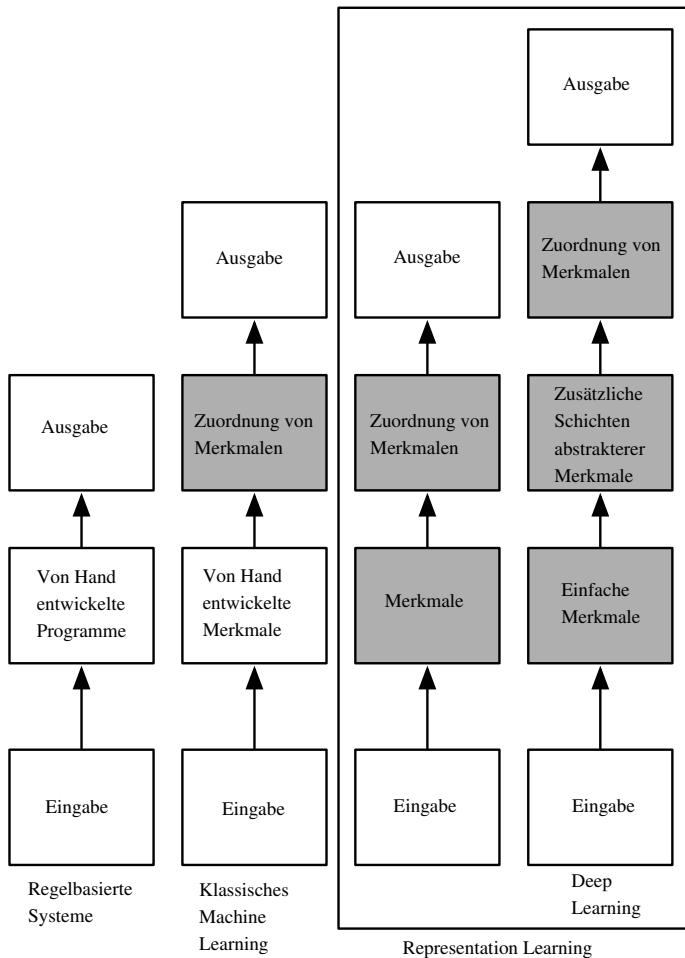


Abbildung 1.5: Diese Ablaufpläne zeigen, wie die einzelnen Bereiche eines KI-Systems in unterschiedlichen KI-Disziplinen untereinander in Beziehung stehen. Schattierte Kästen markieren Komponenten, die aus Daten lernen können.

die eine Forschungslaufbahn mit Schwerpunkt Deep Learning und Künstliche Intelligenz einschlagen. Die andere Zielgruppe umfasst Softwareentwickler, die keine Ausbildung im Bereich Machine Learning oder Statistik genossen haben, aber sich schnell das erforderliche Wissen aneignen möchten, um Deep Learning für ihre Produkte oder Plattformen zu nutzen. Deep Learning hat sich bereits in vielen Softwarebereichen als nützlich erwiesen: Computer Vision, Sprach- und Audioverarbeitung, Verarbeitung natürlicher Sprache (engl. *natural language processing*, NLP), Robotik, Bioinformatik und Chemie, Videospiele, Suchmaschinen, Onlinewerbung und Finanzen.

Dieses Buch ist zur besseren Übersicht in drei Teile gegliedert: Teil I stellt die grundlegenden mathematischen Werkzeuge und Machine-Learning-Konzepte vor. Teil II befasst sich mit den meistverbreiteten Deep-Learning-Algorithmen, die den aktuellen Stand der Technik darstellen. Teil III behandelt neue Konzepte aus der Forschung, die laut allgemeinem Konsens in der Zukunft des Deep Learnings eine wichtige Rolle spielen dürften.

Als Leser können Sie die Teile überspringen (oder überfliegen), mit deren Themen Sie bereits vertraut sind oder die keine Relevanz für Sie haben. Wenn Sie zum Beispiel ein gutes Verständnis der Themen lineare Algebra, Wahrscheinlichkeitsrechnung und der fundamentalen Konzepte des Machine Learnings haben, können Sie Teil I überspringen. Und wenn es Ihnen nur darum geht, ein funktionierendes System zu implementieren, müssen Sie nach Teil II nicht weiterlesen. Als kleine Hilfestellung fasst Abbildung 1.6 die wesentliche Gliederung des Buchs in einem Ablaufplan zusammen.

Wir setzen bei allen Lesern ein fundiertes Informatikwissen voraus. Wir gehen davon aus, dass Sie programmieren können, ein grundlegendes Verständnis in puncto Rechenleistung, Komplexitätstheorie sowie Analysis mitbringen und zumindest die wichtigsten Fachbegriffe aus der Graphentheorie kennen.

1.2 Historische Entwicklungen im Deep Learning

Wissen über die historischen Zusammenhänge ist für ein Verständnis des Deep Learnings hilfreich. Allerdings bietet das Buch keine detaillierte Abhandlung der Historie, sondern befasst sich mit einigen der wesentlichen Entwicklungen:

- Deep Learning blickt auf eine lange und bewegte Historie zurück, allerdings unter vielen verschiedenen Namen. Dabei gab es unterschiedliche philosophische Betrachtungsweisen und das Thema war mal mehr, mal weniger beliebt.
- Deep Learning wurde mit dem Anstieg verfügbarer Trainingsdatensetzmengen nützlicher.
- Deep-Learning-Modelle wurden im Laufe der Zeit immer größer und hielten so Schritt mit der dafür verfügbaren Infrastruktur (Hardware und Software).
- Deep Learning hat dank im Laufe der Zeit gesteigerter Genauigkeit immer komplexere Anwendungsfälle gelöst.

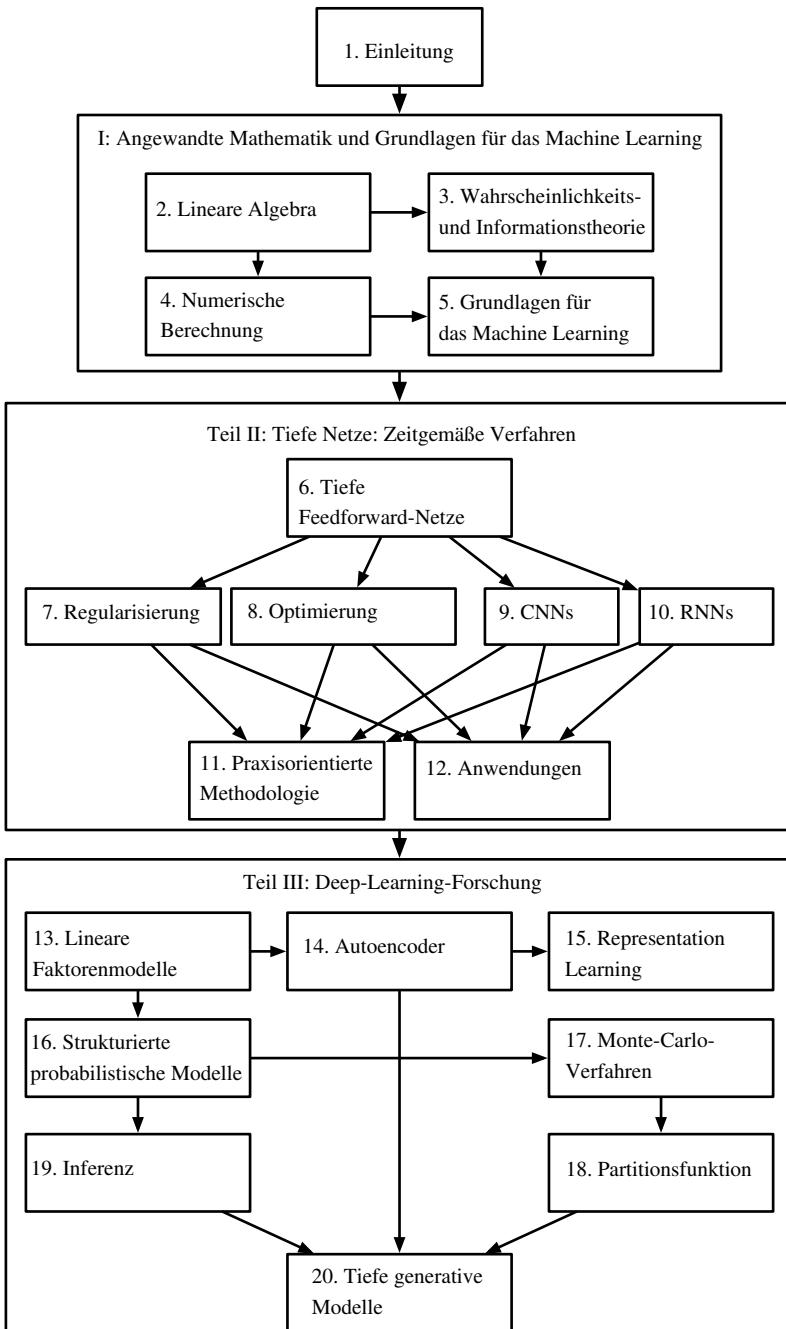


Abbildung 1.6: Wesentliche Gliederung des Buchs. Pfeile, die von einem Kapitel auf ein anderes weisen, geben an, dass Inhalte aus dem zuvor genannten Kapitel für das Verständnis des nachfolgenden erforderlich sind.

1.2.1 Die vielen Namen und wandelnden Schicksale neuronaler Netze

Wir gehen davon aus, dass viele Leser dieses Buchs schon gehört, dass Deep Learning eine spannende neue Technologie ist. Und nun wundern Sie sich vielleicht, dass das Buch einen Abschnitt zur »Historie« einer neuen aufstrebenden Disziplin enthält. Eigentlich gibt es Deep Learning bereits seit den 1940er-Jahren. Deep Learning ist nur *scheinbar* neu, denn bis vor wenigen Jahren fristete es ein weitgehend unbeachtetes Dasein. Zudem wurden viele verschiedene Bezeichnungen dafür verwendet und der Name »Deep Learning« hat sich erst in jüngerer Zeit herausgebildet. Immer wieder wurde dieses Gebiet umbenannt, abhängig davon, welche Forscher und Ansichten gerade besonders einflussreich waren.

Eine umfassende historische Darstellung des Deep Learnings würde den Umfang dieses Buchs sprengen. Für ein besseres Verständnis sind jedoch einige Zusammenhänge förderlich. Vereinfacht ausgedrückt gab es in der Entwicklungsgeschichte des Deep Learnings drei Phasen oder Wellen. Im Zeitraum von etwa 1940 bis 1960 wurde der Begriff **Kybernetik** verwandt, von 1980 bis 1990 die Bezeichnung **Konnektionismus** und seit dem erneuteten Aufstieg im Jahr 2006 schließlich die heute gebräuchliche Titulierung als Deep Learning. Abbildung 1.7 stellt das zeitlich dar.

Einige der ersten Lernalgorithmen, die wir heute kennen, waren als Rechenmodelle für das biologische Lernen gedacht. Es handelte sich um Modelle, die abbilden sollten, wie das Lernen im Gehirn abläuft oder ablaufen könnte. Und so wurde Deep Learning auch als **künstliches neuronales Netz** (engl. *artificial neural network*, ANN) bezeichnet. Die entsprechende Sichtweise auf Deep-Learning-Modelle ist, dass es sich um technische Systeme handelt, die das biologische Gehirn als Vorbild haben (sei es das des Menschen oder eines anderen Lebewesens). Zwar waren bestimmte Arten neuronaler Netze, die für das Machine Learning eingesetzt wurden, manchmal in der Lage, Gehirnfunktionen zu verstehen (*Hinton und Shallice, 1991*), doch generell waren sie nicht als realistische Abbildungen biologischer Funktionen gedacht. Hinter der neuronalen Betrachtungsweise des Deep Learnings stecken in erster Linie zwei Konzepte: einerseits, dass das Gehirn einen Nachweis liefert, dass zum Beispiel intelligentes Verhalten möglich ist und es somit einen geradlinigen Weg gibt, Intelligenz durch eine Rückwärtsentwicklung der Berechnungsprinzipien hinter dem Gehirn zu schaffen und so die Funktionsweise des Gehirns zu kopieren. Eine andere Perspektive ist, dass es extrem interessant sein dürfte, das Gehirn und die Grundlagen der menschlichen Intelligenz zu verstehen, und man somit anhand von Machine-

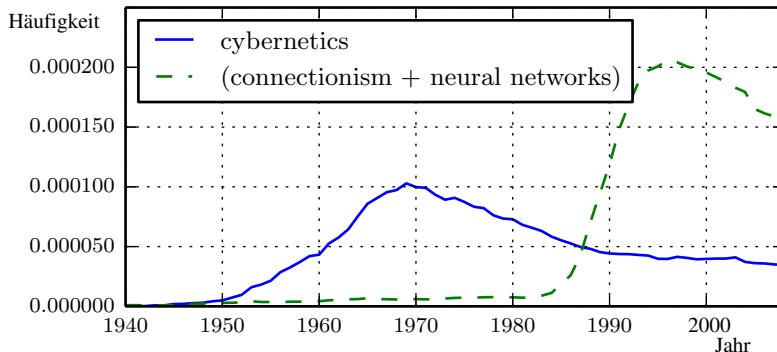


Abbildung 1.7: Zwei der drei historischen Phasen der Forschung an künstlichen neuronalen Netzen anhand der Verwendung der englischen Begriffe »cybernetics« (Kybernetik) und »connectionism« (Konnektionismus) oder »neural networks« (neuronale Netze) (Datenbasis: Google Books). Die dritte Phase ist noch zu jung, um abgebildet zu werden. Die erste Phase begann in den 1940er-Jahren mit der Kybernetik und reichte bis in die 1960er, als die Theorien für biologisches Lernen (*McCulloch und Pitts*, 1943; *Hebb*, 1949) und Implementierungen der ersten Modelle – wie das Perzeptron (*Rosenblatt*, 1958) – aufkamen. Damals wurde das Training eines einzelnen Neurons möglich. Die zweite Phase im Zeitraum 1980–1995 begann mit dem Konnektionismus und der Backpropagation (*Rumelhart et al.*, 1986a) zum Trainieren eines neuronalen Netzes mit einer oder zwei verdeckten Schichten. Die derzeitige dritte Phase, Deep Learning, begann etwa 2006 (*Hinton et al.*, 2006; *Bengio et al.*, 2007; *Ranzato et al.*, 2007a) und erscheint seit 2016 in Buchform. Literatur in Buchform zu den ersten beiden Phasen erschien erst weit nach Beginn der wissenschaftlichen Beschäftigung damit.

Learning-Modellen nicht nur technische Probleme lösen, sondern auch Licht in diese wesentlichen wissenschaftlichen Fragen bringen könnte.

Der heutige Begriff »Deep Learning« geht über die neurowissenschaftliche Perspektive aktueller Machine-Learning-Modelle hinaus und zielt auf das viel allgemeinere Prinzip, die *Komposition von mehreren Schichten* zu lernen, das in Machine-Learning-Frameworks zum Einsatz kommen kann, selbst wenn diese nicht neuronal inspiriert sind.

Bei den ersten Vorläufern der modernen Deep-Learning-Phase handelte es sich um einfache lineare Modelle, die aus neurowissenschaftlicher Sicht von Interesse waren. Diese Modelle waren für eine Menge von n Eingabewerten x_1, \dots, x_n konzipiert, die dann zu einer Ausgabe y führten. Dabei erlernen die Modelle eine Menge von Gewichten w_1, \dots, w_n und berechnen deren Ausgabe $f(\mathbf{x}, \mathbf{w}) = x_1w_1 + \dots + x_nw_n$. Diese erste Forschungswelle zu neuronalen Netzen ist als Kybernetik bekannt (vgl. Abbildung 1.7).

Das McCulloch-Pitts-Neuron (*McCulloch und Pitts*, 1943) ist ein frühes Modell der Hirnfunktion. Dieses lineare Modell kann zwei unterschiedliche Eingabekategorien unterscheiden, indem geprüft wird, ob $f(\mathbf{x}, \mathbf{w})$ positiv oder negativ ist. Damit das Modell der gewünschten Definition der Kategorien entsprechen kann, müssen die Gewichte natürlich korrekt eingestellt werden. Diese Gewichte werden von einem Menschen festgelegt. In den 1950er-Jahren entstand das Perzeptron (*Rosenblatt*, 1958, 1962) als erstes Modell, das die Gewichte erlernen konnte, die die Kategorien anhand von Eingabebeispielen jeder einzelnen Kategorie definieren. Das **adaptive lineare Element** (engl. *ADAptive LINear Element*, ADALINE), das aus etwa derselben Zeit stammt, gab einfach den Wert von $f(\mathbf{x})$ selbst aus, um eine reelle Zahl vorherzusagen (*Widrow und Hoff*, 1960); es konnte ebenfalls lernen, diese Zahlen anhand von Daten vorherzusagen.

Diese einfachen Lernalgorithmen beeinflussten das moderne Bild des Machine Learnings stark. Der Trainingsalgorithmus zum Anpassen der Gewichte im ADALINE war ein Sonderfall eines Algorithmus, der unter der Bezeichnung **stochastisches Gradientenabstiegsverfahren** bekannt ist. Geringfügig modifizierte Versionen des Algorithmus für das stochastische Gradientenabstiegsverfahren sind auch heute noch in Deep-Learning-Modellen vorherrschend.

Die vom Perzeptron und einem ADALINE verwendeten Modelle auf der Basis von $f(\mathbf{x}, \mathbf{w})$ werden als **lineare Modelle** bezeichnet. Sie gehören zu den meistverwendeten Machine-Learning-Modellen, obschon sie häufig anders als die ursprünglichen Modelle *trainiert* werden.

Lineare Modelle unterliegen vielen Einschränkungen. Zu den wesentlichen gehört der Umstand, dass sie die XOR-Funktion nicht erlernen können, für die gilt $f([0, 1], \mathbf{w}) = 1$ und $f([1, 0], \mathbf{w}) = 1$, aber $f([1, 1], \mathbf{w}) = 0$ und $f([0, 0], \mathbf{w}) = 0$. Kritiker, die auf diese Nachteile der linearen Modelle aufmerksam wurden, brachten das biologisch inspirierte Lernen insgesamt in Verruf (*Minsky und Papert*, 1969). Das führte zum ersten großen Einbruch bei der Beliebtheit neuronaler Netze.

Heute gilt die Neurowissenschaft als wichtige Inspirationsquelle für Deep-Learning-Forscher. Allerdings ist sie nicht länger die wichtigste Richtschnur in diesem Bereich.

Der Hauptgrund für den Rückgang der Bedeutung der Neurowissenschaft in der heutigen Deep-Learning-Forschung besteht darin, dass wir einfach nicht über genug Informationen über das Gehirn verfügen und es somit nicht als Vorbild nutzen können. Für ein tiefgreifendes Verständnis der tatsächlich vom Gehirn genutzten Algorithmen müssten wir in der Lage sein,

die Aktivität von (zumindest einigen) Tausenden miteinander vernetzter Neuronen gleichzeitig zu überwachen. Da uns das nicht möglich ist, bleibt uns ein Verständnis selbst einiger der einfachsten und umfassend untersuchten Bereiche des Gehirns verwehrt (*Olshausen und Field, 2005*).

Die Neurowissenschaft hat uns einen Grund zur Hoffnung gegeben, dass ein einzelner Deep-Learning-Algorithmus viele unterschiedliche Aufgaben lösen kann. Neurowissenschaftler haben herausgefunden, dass Frettchen lernen können, mit der Region des Gehirns zu »sehen«, die eigentlich für die auditorische Verarbeitung zuständig ist, wenn ihre Gehirne so neu vernetzt werden, dass die visuellen Signale in diese Region umgeleitet werden (*Von Melchner et al., 2000*). Das deutet darauf hin, dass ein Großteil des Säugetiergehirns möglicherweise einen einzelnen Algorithmus verwendet, um verschiedene Aufgaben im Gehirn zu lösen. Vor dem Aufkommen dieser Hypothese war die Forschung im Bereich des Machine Learnings sehr viel fragmentierter: Forscherteams konzentrierten sich auf unterschiedliche Gebiete wie die Verarbeitung natürlicher Sprache, Computer Vision, die Planung von Bewegungsabläufen und die Spracherkennung. Zwar sind die Anwendungsgruppen auch heute noch getrennt voneinander, aber es ist im Deep Learning mittlerweile üblich, dass Forscherteams sich mit vielen oder gar allen Anwendungsbereichen gleichzeitig befassen.

Wir sind in der Lage, einige grobe Richtlinien aus der Neurowissenschaft abzuleiten. Das grundlegende Konzept hinter einer großen Menge von Berechnungseinheiten, die allein aufgrund der Interaktionen untereinander intelligent werden, ist vom Gehirn inspiriert. Das Neocognitron (*Fukushima, 1980*) führte eine leistungsfähige Modellarchitektur für die Bildverarbeitung ein, die auf der Struktur des Sehapparats von Säugetieren beruht und später zur Basis des modernen CNNs wurde (*LeCun et al., 1998b*) (siehe auch Abschnitt 9.10). Die meisten neuronalen Netze unserer Zeit fußen auf einem Modellneuron namens **ReLU** (engl. *rectified linear unit*, dt. *rektifizierte lineare Einheit*). Das ursprüngliche Cognitron (*Fukushima, 1975*) nutzte eine komplexere Version, die stark von unserem Wissen über die Hirnfunktion inspiriert war. Bei der Entwicklung der vereinfachten modernen Version wurden Konzepte aus vielen Bereichen einbezogen. *Nair und Hinton (2010)* und *Glorot et al. (2011a)* nennen die Neurowissenschaft, während *Jarrett et al. (2009)* technische Einflüsse geltend macht. Die Neurowissenschaft ist zwar eine wichtige Quelle der Inspiration, aber keinesfalls eine starre Richtungsvorgabe. Wir wissen, dass echte Neuronen ganz andere Funktionen als moderne ReLUs berechnen, doch eine realistischere Darstellung von Neuronen hat bisher noch zu keiner Verbesserung beim Machine Learning geführt. Und obwohl die Neurowissenschaft für mehrere *Architekturen* neu-

ronaler Netze erfolgreich Pate stand, wissen wir doch nicht genug über das biologische Lernen, damit uns die Neurowissenschaft als Richtschnur für die *Lernalgorithmen* dienen kann, mit denen wir diese Architekturen trainieren.

Medien betonen häufig die Ähnlichkeit zwischen Deep Learning und dem Gehirn. Es ist zwar wahr, dass Deep-Learning-Forscher dem Gehirn eher einen Einfluss auf ihre Arbeit zuschreiben als Forscher aus anderen Bereichen des Machine Learnings – zum Beispiel Kernel-Maschinen oder bayessche Statistik –, aber man darf nicht den Fehler machen, Deep Learning als Versuche zu sehen, das Gehirn zu simulieren. Das moderne Deep Learning ist von vielen Disziplinen inspiriert, insbesondere von angewandten mathematischen Grundlagen wie lineare Algebra, Wahrscheinlichkeits- und Informationstheorie sowie numerische Optimierung. Während einige Deep-Learning-Forscher die Neurowissenschaft als wichtige Inspirationsquelle nennen, haben andere damit gar nichts am Hut.

Es ist sicher wichtig zu wissen, dass ein Verständnis der Hirnfunktionen auf algorithmischer Ebene nach wie vor sehr aktuell ist. Dieser Forschungszweig wird meist als »Computational Neuroscience« bezeichnet und stellt ein gesondertes Forschungsgebiet dar. Allerdings wechseln Forscher häufig zwischen den beiden Gebieten hin und her. Beim Deep Learning geht es in erster Linie darum, Computersysteme zu bauen, die erfolgreich Aufgaben lösen können, zu deren Bewältigung Intelligenz erforderlich ist. Die Computational Neuroscience dagegen befasst sich in erster Linie damit, exaktere Modelle der tatsächlichen Hirnfunktionen zu bauen.

In den 1980er-Jahren begann die zweite Phase der Forschung an neuronalen Netzen. Verantwortlich dafür war in erster Linie eine als **Konnektionismus** oder **Parallel Distributed Processing** bekannte Bewegung (*Rumelhart et al.*, 1986c; *McClelland et al.*, 1995). Der Konnektionismus entstand im Kontext der Kognitionswissenschaft. Die Kognitionswissenschaft ist ein interdisziplinärer Ansatz zur Erforschung des menschlichen Bewusstseins, bei dem mehrere Analyseebenen kombiniert werden. Anfang der 1980er-Jahre studierten die meisten Kognitionswissenschaftler Modelle des symbolischen Schließens. Obwohl symbolische Modelle beliebt waren, ließen sie sich doch schwer auf eine Art erklären, die zeigt, wie das Gehirn sie mithilfe von Neuronen implementiert. Die Anhänger des Konnektionismus begannen, sich mit Wahrnehmungsmodellen zu befassen, die tatsächlich in neuronalen Implementierungen gegründet werden könnten (*Touretzky und Minton*, 1985); so lebten viele Ideen wieder auf, die auf die Arbeit des Psychologen Donald Hebb aus den 1940er-Jahren zurückgehen (*Hebb*, 1949).

Das zentrale Konzept des Konnektionismus besagt, dass eine große Menge einfacher Berechnungseinheiten in vernetzter Form intelligentes Verhalten an den Tag legen kann. Diese Erkenntnis gilt gleichermaßen für die Neuronen des biologischen Nervensystems und die verborgenen Einheiten in Berechnungsmodellen.

Im Rahmen der Konnektionismus-Bewegung in den 1980er-Jahren entstanden mehrere Schlüsselkonzepte, die bis heute eine tragende Rolle im Deep Learning einnehmen.

Eines dieser Konzepte ist die **verteilte Repräsentation** (*Hinton et al., 1986*). Dabei handelt es sich um die Vorstellung, dass jede Eingabe in einem System durch viele Merkmale dargestellt wird und jedes Merkmal an der Darstellung möglichst vieler Eingaben beteiligt ist. Ein Beispiel: Gegeben ist ein Computer-Vision-System, das zwischen Autos, Lkws und Vögeln in den Farben Rot, Grün und Blau unterscheiden kann. Eine Art, diese Eingaben zu repräsentieren, besteht darin, ein separates Neuron oder eine verborgene Einheit einzusetzen, die für jede der neun Möglichkeiten (roter Lkw, rotes Auto, roter Vogel, grüner Lkw usw.) aktiviert wird. Dazu sind neun verschiedene Neuronen erforderlich, die jedes für sich das Konzept von Farbe und Objektidentität erlernen müssen. Ein besserer Ansatz besteht in einer verteilten Repräsentation mit drei Neuronen für die Farben und drei Neuronen für die Objektidentität. Damit werden nur noch sechs anstelle von neun Neuronen benötigt. Das Neuron für die Farbe Rot kann dieses Merkmal anhand von Bildern erlernen, die Autos, Lkws und Vögel zeigen – nicht nur eine spezielle Objektkategorie. Das Konzept der verteilten Repräsentation zieht sich durch das vorliegende Buch und wird in Kapitel 15 noch genauer beschrieben.

Ein weiterer wesentlicher Erfolg des Konnektionismus ist der Einsatz der Backpropagation zum Trainieren tiefer neuronaler Netze anhand interner Repräsentationen sowie die Verbreitung des Backpropagation-Algorithmus (*Rumelhart et al., 1986a; LeCun, 1987*). Dieser Algorithmus war mal mehr, mal weniger beliebt. Zum Zeitpunkt der Abfassung dieses Buchs ist er der vorherrschende Ansatz für das Training tiefer Modelle.

In den 1990er-Jahren machten Forscher wichtige Fortschritte beim Modellieren von Sequenzen mit neuronalen Netzen. *Hochreiter* (1991) und *Bengio et al.* (1994) identifizierten einige der grundlegenden mathematischen Schwierigkeiten beim Modellieren langer Sequenzen, die in Abschnitt 10.7 beschrieben werden. *Hochreiter und Schmidhuber* (1997) stellten das LSTM-Netz (engl. *long short-term memory*, dt. *langes Kurzzeitgedächtnis*) vor, das einige dieser Schwierigkeiten löst. Heute ist das LSTM ein Quasistandard

für viele Aufgaben bei der Sequenzmodellierung; es wird zum Beispiel bei Google für NLP eingesetzt.

Die zweite Phase der Forschung an neuronalen Netzen dauerte bis Mitte der 1990er an. Unternehmen aus dem Bereich neuronaler Netze und anderer KI-Technologien machten unrealistisch ambitionierte Versprechen auf der Suche nach Mitteln. Als die KI-Forscher diese unberechtigten Versprechungen nicht erfüllten, waren die Investoren enttäuscht. Gleichzeitig gab es Fortschritte in anderen Bereichen des Machine Learnings zu vermelden. Kernel-Maschinen (*Boser et al.*, 1992; *Cortes und Vapnik*, 1995; *Schölkopf et al.*, 1999) und graphische Modelle (*Jordan*, 1998) zeigten gute Ergebnisse bei vielen wichtigen Aufgaben. Diese beiden Faktoren führten zu einem Rückgang der Beliebtheit neuronaler Netze, der bis 2007 anhielt.

In diesem Zeitraum erbrachten neuronale Netze weiterhin beeindruckende Leistungen bei einigen Aufgaben (*LeCun et al.*, 1998b; *Bengio et al.*, 2001). Das Canadian Institute for Advanced Research (CIFAR) half mit seiner Forschungsinitiative Neural Computation and Adaptive Perception (NCAP) dabei, die Forschung an neuronalen Netzen aufrechtzuerhalten. Dieses Programm brachte Machine-Learning-Forschergruppen unter der Leitung von Geoffrey Hinton (University of Toronto), Yoshua Bengio (University of Montreal) und Yann LeCun (New York University) zusammen. Die multidisziplinäre Forschungsinitiative CIFAR NCAP bezog auch Neurowissenschaftler und Experten für menschliches Sehen und Computer Vision mit ein.

Damals glaubte man, dass tiefe Netze nur sehr schwer zu trainieren seien. Mittlerweile wissen wir, dass seit den 1980ern existierende Algorithmen recht gut funktionieren – aber das wurde erst um 2006 offenkundig. Möglicherweise war der Berechnungsaufwand dieser Algorithmen einfach zu hoch, um mit der damals verfügbaren Hardware viel zu experimentieren.

Die dritte Phase der Forschung zu neuronalen Netzen begann 2006 mit einem Paukenschlag: Geoffrey Hinton zeigte, dass ein als Deep-Belief-Netz bezeichnetes neuronales Netz mithilfe eines Verfahrens namens »greedy layer-wise pretraining« (schichtweises Pretraining mit Greedy-Algorithmen) effizient trainiert werden kann (*Hinton et al.*, 2006). Wir gehen in Abschnitt 15.1 näher darauf ein. Die anderen mit CIFAR verbundenen Forscherteams bewiesen schnell, dass dieses Verfahren auch für das Trainieren vieler weiterer tieferer Netze anwendbar war (*Bengio et al.*, 2007; *Ranzato et al.*, 2007a), und halfen systematisch bei der verbesserten Generalisie-

rung von Testbeispielen.¹ Diese Phase der Forschung zu neuronalen Netzen gab der Bezeichnung »Deep Learning« Vorschub und betonte so, dass die Forscher nun in der Lage waren, tiefere neuronale Netze als zuvor zu trainieren. Gleichzeitig wurde die theoretische Bedeutung der Tiefe in den Fokus gerückt (*Bengio und LeCun, 2007; Delalleau und Bengio, 2011; Pascanu et al., 2014a; Montufar et al., 2014*). Zu diesem Zeitpunkt übertrafen tiefe neuronale Netze konkurrierende, auf anderen Machine-Learning-Technologien basierende KI-Systeme ebenso wie händisch konzipierte Systeme. Diese dritte Beliebtheitsphase der neuronalen Netze hält bis zum Zeitpunkt der Abfassung dieses Buchs an. Allerdings hat sich die Bedeutung der Deep-Learning-Forschung im Verlauf der Phase stark gewandelt. Zu Beginn der dritten Phase lag der Schwerpunkt auf dem neuen unüberwachten Lernen und der Möglichkeit tiefer Modelle, auch von kleinen Datensätzen gut zu generalisieren. Heute gilt das Interesse vor allem viel älteren Algorithmen für überwachtes Lernen sowie der Fähigkeit tiefer Modelle, große Datensätze zu nutzen, die mit einem Label (siehe Abschnitt 5.1.3) gekennzeichnet sind.

1.2.2 Wachsende Größe von Datensätzen

Sie fragen sich vielleicht, warum Deep Learning erst in jüngerer Vergangenheit zu einer so wichtigen Technologie geworden ist. Schließlich fanden die ersten Experimente mit künstlichen neuronalen Netzen doch bereits in den 1950er-Jahren statt.

Deep Learning wird in kommerziellen Anwendungen seit den 1990ern erfolgreich eingesetzt, galt aber oftmals eher als Kunst denn als Technologie und hatte den Ruf, nur von Spezialisten beherrscht zu werden. Das hat sich erst kürzlich geändert. Es ist wohl wahr, dass für eine gute Abstimmung eines Deep-Learning-Algorithmus ein gewisses Geschick erforderlich ist. Zum Glück sinkt diese Einstiegshürde mit der zunehmenden Menge an Trainingsdaten immer weiter. Die Lernalgorithmen, die heute bei komplexen Aufgaben mit beinahe menschlichen Leistungen glänzen, unterscheiden sich kaum von den Lernalgorithmen, die in den 1980er-Jahren mit einfachsten Problemen ihre Schwierigkeiten hatten, obwohl die Modelle, die mit diesen Algorithmen trainiert werden, so geändert wurden, dass damit auch das Training sehr tieferer Architekturen einfacher wird. Die wichtigste neue Entwicklung besteht darin, dass wir heute die Ressourcen bereitstellen können, die für den Erfolg der Algorithmen notwendig sind.

¹ Das Wort »Beispiel« wird in diesem Buch oft benutzt, um auf Datenpunkte und Datenbeispiele sowohl als Trainingsdaten als auch als Testdaten zu referenzieren. Dabei ist ein »Beispiel« eine spezifische Repräsentation des zu lösenden Problems.

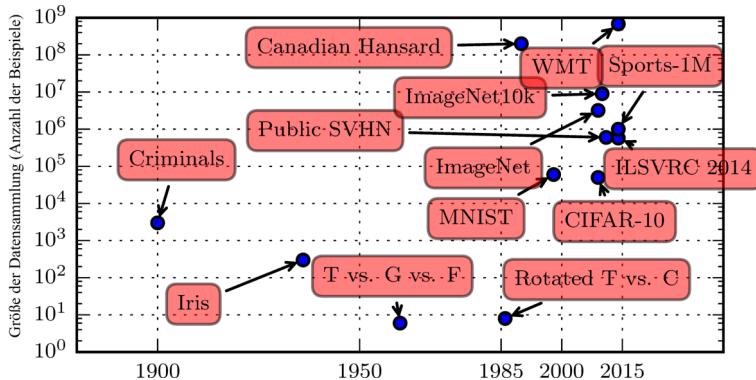


Abbildung 1.8: Im Zeitverlauf wachsende Größe von Datensätzen. Anfang der 1900er-Jahre nutzten Statistiker Hunderte oder gar Tausende von manuell zusammengestellten Messwerten (*Garson*, 1900; *Gosset*, 1908; *Anderson*, 1935; *Fisher*, 1936). In den 1950ern bis hinein in die 1980er-Jahre arbeiteten die Pioniere des biologisch inspirierten Machine Learnings häufig mit kleinen synthetischen Datensätzen, zum Beispiel den niedrig aufgelösten Bitmap-Grafiken von Buchstaben, die für einen geringen Berechnungsaufwand konzipiert waren, um zu zeigen, dass neuronale Netze bestimmte Funktionen erlernen konnten (*Widrow und Hoff*, 1960; *Rumelhart et al.*, 1986b). In den 1980ern und 1990ern verschob sich Machine Learning in Richtung Statistik und setzte auf die Nutzung größerer Datensätze mit Zehntausenden von Beispielen. Ein Vertreter dieser Gattung ist der MNIST-Datensatz (siehe Abbildung 1.9) mit Scans handgeschriebener Ziffern (*LeCun et al.*, 1998b). Im ersten Jahrzehnt der 2000er-Jahre hielten ausgefeilte Datensätze ähnlicher Größe Einzug, bspw. der CIFAR-10-Datensatz (*Krizhevsky und Hinton*, 2009). Gegen Ende des Jahrzehnts und im Laufe der ersten Hälfte der 2010er veränderten wesentlich umfangreichere Datensätze mit Hunderttausenden oder gar zig Millionen Beispielen die Möglichkeiten des Deep Learnings enorm. Zu diesen Datensätzen gehören die öffentlich verfügbaren Hausnummern der Street-View-Datenbank SVHN (*Netzer et al.*, 2011), mehrere Versionen des ImageNet-Datensatzes (*Deng et al.*, 2009, 2010a; *Russakovsky et al.*, 2014a) und der Sports-1M-Datensatz (*Karpathy et al.*, 2014). Oben in der Abbildung wird offensichtlich, dass Datensätze mit übersetzten Satzpaaren wie IBMs Datensatz auf Basis des Canadian Hansard – das sind die offiziellen protokollarischen Aufzeichnungen von Sitzungen verschiedener Parlamente (*Brown et al.*, 1990) – und der Datensatz WMT 2014 mit englischer und französischer Entsprechung (*Schwenk*, 2014) deutlich größer als andere Datensätze sind.

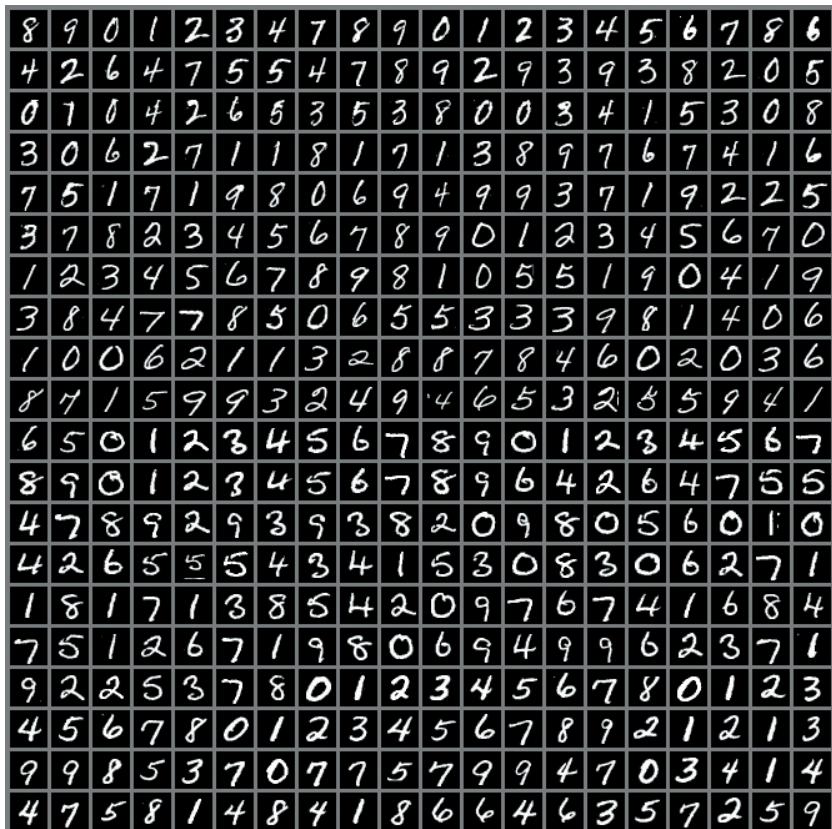


Abbildung 1.9: Beispieleingaben aus dem MNIST-Datensatz. »NIST« steht für National Institute of Standards and Technology (Nationales Institut für Normung und Technologie der USA). Diese Behörde hat die Daten ursprünglich erfasst. Das »M« steht für »modified« (geändert), da die Daten zur einfacheren Verwendung mit Machine-Learning-Algorithmen vorbereitet wurden. Der MNIST-Datensatz besteht aus handgeschriebenen Ziffern und zugehörigen Labeln, die angeben, welche der Ziffern 0 bis 9 ein Bild zeigt. Dieses einfache Klassifizierungsproblem gehört zu den einfachsten und meistverbreiteten Tests in der Deep-Learning-Forschung. Obwohl es mit modernen Methoden recht einfach zu lösen ist, erfreut es sich nach wie vor großer Beliebtheit. Geoffrey Hinton nannte es »die *Drosophila* des Machine Learnings«, denn damit können Machine-Learning-Forscher ihre Algorithmen unter kontrollierten Laborbedingungen ebenso untersuchen, wie Biologen Fruchtfliegen häufig studieren.

Abbildung 1.8 zeigt, wie die Größe der Benchmark-Datensätze im Laufe der Zeit stark angestiegen ist. Diese Entwicklung verdanken wir der zunehmenden Digitalisierung unserer Gesellschaft. Da immer mehr Dinge am Computer erledigt werden, stehen auch immer mehr Datensätze und Aufzeichnungen darüber zur Verfügung. Und da die Computer immer stärker miteinander vernetzt sind, wird es auch einfacher, diese Datensätze zentral zu sammeln und in für Machine-Learning-Anwendungen geeigneten Datensätze zu kuratieren. Das Zeitalter von »Big Data« hat das Machine Learning stark vereinfacht, denn die gewaltige Last der statistischen Schätzung, bei der nach der Untersuchung geringer Datenmengen eine gute Generalisierung für neue Daten erfolgt, wiegt lang nicht mehr so schwer wie zuvor.

Seit dem Jahr 2016 gilt die Faustregel, dass ein überwachter Deep-Learning-Algorithmus im Allgemeinen mit etwa 5 000 mit einem Label gekennzeichneten Testdaten je Kategorie zu einer annehmbaren Leistung findet und sogar die menschliche Leistungsfähigkeit erreicht oder übertrifft, wenn er mit einem Datensatz trainiert wird, der mindestens zehn Millionen mit einem Label gekennzeichnete Testdaten enthält. Der erfolgreiche Einsatz kleinerer Datensätze ist ein wichtiges Forschungsgebiet; dabei geht es besonders darum, wie wir große Mengen von Testdaten ohne Label für unüberwachtes oder halb-überwachtes Lernen nutzen können.

1.2.3 Wachsende Modellgrößen

Ein weiterer wichtiger Grund für den aktuell großen Erfolg neuronaler Netze nach dem Abschwung in den 1980ern ist der Anstieg der Rechenleistung, mit der heute sehr viel größere Modelle betrachtet werden können. Zu den wichtigsten Erkenntnissen des Konnektionismus gehört es, dass Tiere intelligent werden, wenn viele ihrer Neuronen zusammenarbeiten. Ein einzelnes Neuron oder eine kleine Neuronengruppe ist nicht besonders nützlich.

Biologische Neuronen sind nicht gerade eng vernetzt. Wie Abbildung 1.10 zeigt, wiesen unsere Machine-Learning-Modelle eine Vielzahl von Verbindungen pro Neuron auf – so viele, dass über Jahrzehnte sogar die Größenordnung eines Säugetiergehirns erreicht wurde.

Betrachtet man nur die Gesamtzahl der Neuronen, stellt sich heraus, dass neuronale Netze bis vor Kurzem überraschend klein ausfielen (vgl. Abbildung 1.11). Seit Einführung der verdeckten Einheiten hat die Größe künstlicher neuronaler Netze sich etwa alle 2,4 Jahre verdoppelt. Dieses Wachstum wird durch schnellere Computer mit mehr Speicher und die Verfügbarkeit größerer Datensätze beschleunigt. Größere Netze ermöglichen eine höhere Genauigkeit bei komplexeren Aufgaben. Diese Entwicklung dürfte für

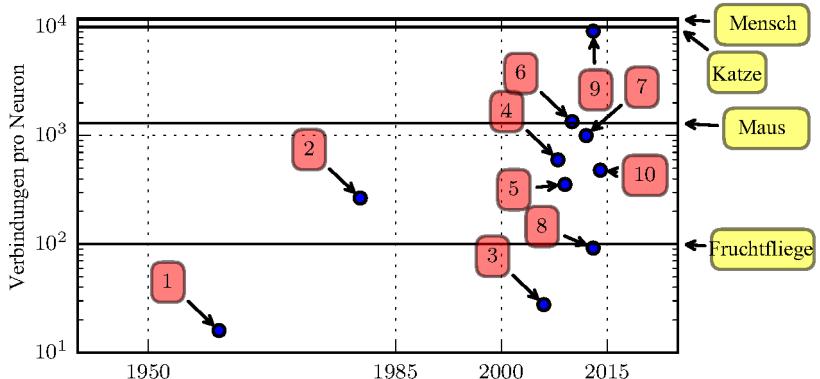


Abbildung 1.10: Anzahl der Verbindungen pro Neuron im Zeitverlauf. Anfänglich war die Anzahl der Verbindungen zwischen Neuronen in künstlichen neuronalen Netzen durch die Hardware begrenzt. Heute handelt es sich eher um eine Designentscheidung. Einige künstliche neuronale Netze weisen nahezu so viele Verbindungen pro Neuron wie eine Katze auf. Und andere neuronale Netze verfügen für gewöhnlich über so viele Verbindungen pro Neuron wie kleinere Säugetiere, zum Beispiel Mäuse. Sogar das menschliche Gehirn zeichnet sich nicht durch eine überwältigende Anzahl von Verbindungen pro Neuron aus. Größe von biologischen neuronalen Netzen laut *Wikipedia* (2015).

1. Adaptives lineares Element (*Widrow und Hoff*, 1960)
2. Neocognitron (*Fukushima*, 1980)
3. GPU-beschleunigtes CNN (*Chellapilla et al.*, 2006)
4. Deep Boltzmann Machine (DBM, dt. *tiefe Boltzmann-Maschine*) (*Salakhutdinov und Hinton*, 2009a)
5. Unüberwachtes CNN (*Jarrett et al.*, 2009)
6. GPU-beschleunigtes mehrschichtiges Perzepron (*Ciresan et al.*, 2010)
7. Verteilter Autoencoder (*Le et al.*, 2012)
8. Multi-GPU-CNN (*Krizhevsky et al.*, 2012)
9. Unüberwachtes COTS-HPC-CNN (*Coates et al.*, 2013)
10. GoogLeNet (*Szegedy et al.*, 2014a)

einige Jahrzehnte anhalten. Sofern keine neuen Technologien eine schnellere Skalierung erlauben, werden künstliche neuronale Netze frühestens in den 2050er-Jahren dieselbe Anzahl von Neuronen wie das menschliche Gehirn aufweisen. Biologische Neuronen repräsentieren möglicherweise kompliziertere Funktionen als aktuelle künstliche Neuronen; es wäre also denkbar, dass biologische neuronale Netze noch größer sein werden, als wir momentan glauben.

In der Rückschau überrascht es nicht wirklich, dass neuronale Netze mit weniger Neuronen als ein Blutegel nicht in der Lage waren, ausgeklügelte

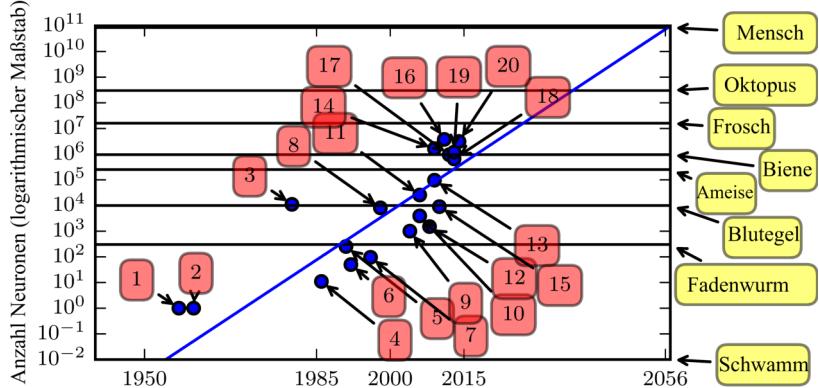


Abbildung 1.11: Im Zeitverlauf wachsende Größe von neuronalen Netzen. Seit Einführung der verdeckten Einheiten hat die Größe künstlicher neuronaler Netze sich etwa alle 2,4 Jahre verdoppelt. Größe von biologischen neuronalen Netzen laut *Wikipedia* (2015).

1. Perzeptron (*Rosenblatt*, 1958, 1962)
2. Adaptives lineares Element (*Widrow und Hoff*, 1960)
3. Neocognitron (*Fukushima*, 1980)
4. Frühes Backpropagation-Netz (*Rumelhart et al.*, 1986b)
5. RNN für die Spracherkennung (*Robinson und Fallside*, 1991)
6. Mehrschichtiges Perzeptron für die Spracherkennung (*Bengio et al.*, 1991)
7. Molekularfeld-Sigmoid-Belief-Netz (*Saul et al.*, 1996)
8. LeNet-5 (*LeCun et al.*, 1998b)
9. Echo-State-Netz (*Jaeger und Haas*, 2004)
10. Deep-Belief-Netz (*Hinton et al.*, 2006)
11. GPU-beschleunigtes CNN (*Chellapilla et al.*, 2006)
12. Deep Boltzmann Machine (DBM) (*Salakhutdinov und Hinton*, 2009a)
13. GPU-beschleunigtes Deep-Belief-Netz (*Raina et al.*, 2009)
14. Unüberwachtes CNN (*Jarrett et al.*, 2009)
15. GPU-beschleunigtes mehrschichtiges Perzeptron (*Ciresan et al.*, 2010)
16. OMP-1-Netz (*Coates und Ng*, 2011)
17. Verteilter Autoencoder (*Le et al.*, 2012)
18. Multi-GPU-CNN (*Krizhevsky et al.*, 2012)
19. Unüberwachtes COTS-HPC-CNN (*Coates et al.*, 2013)
20. GoogLeNet (*Szegedy et al.*, 2014a)

Problemstellungen der Künstlichen Intelligenz zu lösen. Selbst moderne, aus Sicht der Rechensysteme als recht groß zu betrachtende Netze sind doch klein, wenn man sie mit dem Nervensystem selbst relativ einfach gestrickter Wirbeltiere wie Fröschen vergleicht.

Der Zuwachs der Modellgröße im Laufe der Zeit ist schnelleren Prozessoren, der Verfügbarkeit von für viele Zwecke geeigneten GPUs (vgl. Abschnitt 12.1.2), schnelleren Netzanbindungen und einer besseren Softwareinfrastruktur für verteiltes Rechnen geschuldet. Genau dieser Zuwachs ist eine der wichtigsten Entwicklungen in der Geschichte des Deep Learnings. Sie dürfen davon ausgehen, dass er noch eine lange Zeit anhält.

1.2.4 Wachsende Genauigkeit, Komplexität und Auswirkungen auf die reale Welt

Seit den 1980er-Jahren haben die exakte Erkennung bzw. Wahrnehmung und Vorhersage im Deep Learning stetig zugenommen. Außerdem wurde Deep Learning erfolgreich für immer allgemeinere Anwendungen eingesetzt.

Die ersten tiefen Modelle dienten der Erkennung einzelner Objekte in eng beschnittenen, extrem kleinen Bildern (*Rumelhart et al.*, 1986a). Seit damals wurden die Bilder, die mit neuronalen Netzen verarbeitet werden können, langsam größer. Moderne Netze zur Objekterkennung verarbeiten detaillierte, hochauflösende Fotografien, ohne dass ein Beschnitt rund um das Objekt erfolgen muss (*Krizhevsky et al.*, 2012). Frühe Netze konnten auch nur zwei Arten von Objekten erkennen (oder in einigen Fällen das Vorhandensein bzw. Fehlen einer einzelnen Objektart), wohingegen die modernen Netze normalerweise mindestens 1 000 unterschiedliche Objektkategorien erkennen können. Der größte Prüfstein für die Objekterkennung ist die jedes Jahr veranstaltete ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Ein Meilenstein im rasanten Aufstieg des Deep Learnings war der Zeitpunkt, zu dem erstmals ein CNN diese Challenge gewann, und zwar mit großem Abstand. Dabei wurde die damalige Top-5-Fehlerrate von 26,1 auf 15,3 Prozent gesenkt (*Krizhevsky et al.*, 2012). Anders ausgedrückt: Das CNN gab eine sortierte Liste der möglichen Kategorien für jedes einzelne Bild aus, in der die ersten fünf Listeneinträge für alle bis auf 15,3 Prozent der Testmuster die korrekte Kategorie enthielten. Seitdem standen jedes Jahr tiefe CNNs auf dem Podest. Zum Zeitpunkt der Abfassung dieses Buchs haben Fortschritte im Deep Learning die Top-5-Fehlerrate auf 3,6 Prozent gesenkt (vgl. Abbildung 1.12).

Deep Learning hatte auch dramatische Auswirkungen auf die Spracherkennung. Nach stetigen Fortschritten in den 1990ern stagnierten die

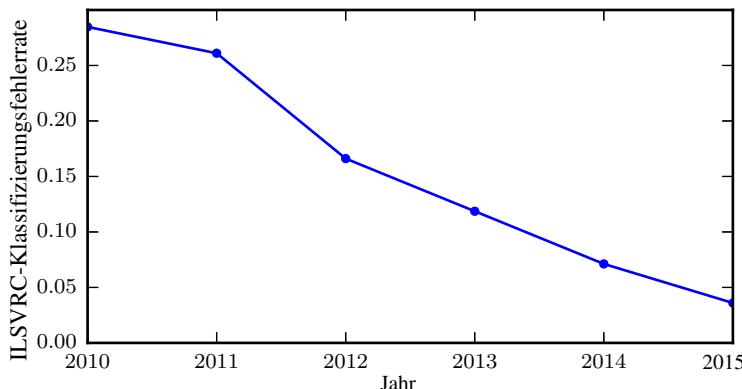


Abbildung 1.12: Abnehmende Fehlerrate im Zeitverlauf. Seit tiefe Netze die Größe erreicht haben, um an der ImageNet Large Scale Visual Recognition Challenge teilnehmen zu können, haben sie diesen Wettbewerb jedes Jahr gewonnen und dabei immer geringere Fehlerraten erzielt. Quelle: *Russakovsky et al. (2014b)* und *He et al. (2015)*.

Fehlerraten für die Spracherkennung um das Jahr 2000. Die Einführung von Deep Learning (*Dahl et al., 2010; Deng et al., 2010b; Seide et al., 2011; Hinton et al., 2012a*) bei der Spracherkennung führte jedoch zu einem plötzlichen Rückgang der Fehlerrate, zum Teil sogar um 50 Prozent. In Abschnitt 12.3 wird das Thema noch genauer behandelt.

Tiefe Netze sind auch für aufsehenerregende Erfolge bei der Fußgängererkennung und Bildsegmentierung verantwortlich (*Sermanet et al., 2013; Farabet et al., 2013; Couprie et al., 2013*). Sie sind auch der Grund für eine übermenschliche Genauigkeit bei der Verkehrszeicheneinstufung (*Ciresan et al., 2012*).

Parallel zur Größe und Genauigkeit tiefer Netze ist auch die Komplexität der Aufgaben, die damit gelöst werden können, gestiegen. *Goodfellow et al. (2014d)* haben gezeigt, dass neuronale Netze lernen können, eine komplette Sequenz von Zeichen zu erkennen, statt nur ein einzelnes Objekt in einem Bild zu identifizieren. Zuvor war man davon ausgegangen, dass diese Art des Lernens die Kennzeichnung der einzelnen Elemente der Sequenz mit einem Label erfordere (*Gülçehre und Bengio, 2013*). RNNs wie das erwähnte LSTM-Sequenzmodell werden mittlerweile eingesetzt, um Beziehungen zwischen *Sequenzen* und anderen *Sequenzen* zu modellieren, und nicht mehr nur für starre Eingaben. Dieses Sequenz-zu-Sequenz-Lernen (engl. *sequence-to-sequence learning*) scheint der Beginn einer weiteren Revolution zu sein,

nämlich der maschinellen Übersetzung (*Sutskever et al.*, 2014; *Bahdanau et al.*, 2015).

Die Entwicklung hin zu steigender Komplexität hat als logische Folge zu der Einführung neuronaler Turing-Maschinen geführt (*Graves et al.*, 2014a). Diese Maschinen lernen, aus Speicherzellen zu lesen und beliebige Inhalte in Speicherzellen zu schreiben. Derartige neuronale Netze können einfache Programme auf der Basis von Beispielen für das erwünschte Verhalten lernen. So ist es möglich, das Sortieren von Zahlen anhand von Listen mit zufälligen und geordneten Sequenzen zu erlernen. Diese Selbstprogrammierung steckt noch in den Kinderschuhen, aber in der Zukunft könnte sie für nahezu beliebige Aufgaben genutzt werden.

Ein weiterer großer Erfolg des Deep Learnings ist seine Anwendung im Bereich des **Reinforcement Learnings** (dt. *bestärkendes* oder *verstärkendes Lernen*). Im Rahmen des Reinforcement Learnings muss ein autonomer Agent durch Versuch und Irrtum lernen, eine Aufgabe zu erledigen. Dabei erhält er keine menschliche Hilfe. DeepMind bewies, dass ein System für Reinforcement Learning auf Deep-Learning-Basis in der Lage ist, Atari-Videospiele zu erlernen und bei vielen Aufgaben menschliches Niveau zu erreichen (*Mnih et al.*, 2015). Deep Learning hat auch die Ergebnisse des Reinforcement Learnings in der Robotik erheblich verbessert (*Finn et al.*, 2015).

Viele dieser Einsatzgebiete für das Deep Learning sind hoch profitabel. Deep Learning wird mittlerweile von vielen großen Technologieunternehmen eingesetzt, darunter Google, Microsoft, Facebook, IBM, Baidu, Apple, Adobe, Netflix, NVIDIA und NEC.

Fortschritte im Bereich des Deep Learnings wiesen auch eine starke Abhängigkeit von Fortschritten bei der Softwareinfrastruktur auf. Softwarebibliotheken wie Theano (*Bergstra et al.*, 2010; *Bastien et al.*, 2012), PyLearn2 (*Goodfellow et al.*, 2013c), Torch (*Collobert et al.*, 2011b), DistBelief (*Dean et al.*, 2012), Caffe (*Jia*, 2013), MXNet (*Chen et al.*, 2015) und TensorFlow (*Abadi et al.*, 2015) haben samt und sonders wichtige Forschungsprojekte und kommerzielle Produkte unterstützt.

Aber Deep Learning hat auch anderen Wissenschaften seinen Stempel aufgedrückt: Moderne CNNs für die Objekterkennung erzeugen ein Modell der visuellen Verarbeitung, das von Neurowissenschaftlern untersucht werden kann (*DiCarlo*, 2013). Deep Learning stellt nützliche Werkzeuge zur Verarbeitung gewaltiger Datenmengen und zum Treffen nützlicher Vorhersagen in wissenschaftlichen Bereichen bereit. Es wurde erfolgreich eingesetzt, um in der Pharmabranche bei der Entwicklung neuer Medikamente die

Interaktion von Molekülen vorherzusagen (*Dahl et al.*, 2014), um nach subatomaren Teilchen zu suchen (*Baldi et al.*, 2014) und um automatisch mikroskopische Aufnahmen auszuwerten und daraus eine 3-D-Karte des menschlichen Gehirns zu erstellen (*Knowles-Barley et al.*, 2014). Wir gehen davon aus, dass Deep Learning künftig in immer mehr Disziplinen genutzt wird.

Zusammenfassend lässt sich sagen, dass Deep Learning ein Bereich des Machine Learnings ist, der stark von unserem Wissen über das menschliche Gehirn, die Statistik und die angewandte Mathematik der letzten Jahrzehnte geprägt ist. In jüngerer Vergangenheit hat Deep Learning einen gewaltigen Schub in Bezug auf Beliebtheit und Nützlichkeit erlebt, der vor allem leistungsfähigeren Computern, größeren Datensätzen und Methoden zum Trainieren tiefer Netze zu verdanken ist. Die kommenden Jahre bieten viele Herausforderungen und Gelegenheiten, Deep Learning weiterhin zu optimieren und Grenzen zu versetzen.

I

Angewandte Mathematik und Grundlagen für das Machine Learning

Dieser Teil stellt die grundlegenden mathematischen Konzepte vor, die Sie für ein Verständnis des Deep Learnings benötigen. Teil I beginnt mit allgemeinen Begriffe aus der angewandten Mathematik, mit denen wir Funktionen vieler Variablen definieren, die Hoch- und Tiefpunkte dieser Funktionen finden und den "Degree of belief"(dt. Grad persönlicher Überzeugung, bayesscher Wahrscheinlichkeitsbegriff) quantitativ bestimmen können.

Anschließend beschreiben wir die grundlegenden Ziele des Machine Learnings. Wir zeigen, wie sich diese Ziele erreichen lassen, indem wir ein Modell spezifizieren, das bestimmte Überzeugungen repräsentiert, eine Kostenfunktion entwickeln, die quantitativ erfasst, wie gut diese Überzeugungen der Wirklichkeit entsprechen, und einen Trainingsalgorithmus verwenden, der die Kostenfunktion minimiert.

Dies ist die elementare Grundlage für eine Vielzahl von Machine-Learning-Algorithmen, einschließlich nicht-tiefer Ansätze. In den weiteren Teilen des Buchs entwickeln wir in diesem Rahmen eigene Deep-Learning-Algorithmen.

2

Lineare Algebra

Die lineare Algebra ist ein Zweig der Mathematik, der auf vielen Gebieten der Wissenschaft und Technik zum Einsatz kommt. Da die lineare Algebra jedoch eine Form der kontinuierlichen Mathematik (und nicht der diskreten Mathematik) darstellt, sind viele Wissenschaftler damit nur wenig vertraut. Ein gutes Verständnis der linearen Algebra ist allerdings eine Grundvoraussetzung für das Verständnis und die Arbeit mit vielen Machine-Learning-Algorithmen und insbesondere mit Deep-Learning-Algorithmen. Wir beginnen unsere Einführung in das Deep Learning daher mit einer kompakten Darstellung der wichtigsten Bereiche der linearen Algebra.

Wenn Sie bereits mit der linearen Algebra vertraut sind, können Sie dieses Kapitel überspringen. Falls Sie bereits über Erfahrungen mit diesen Konzepten verfügen, aber eine detaillierte Referenz zur Vertiefung wesentlicher Formeln benötigen, empfehlen wir *The Matrix Cookbook* (Petersen und Pedersen, 2006). Wenn Sie sich bisher noch nicht mit der linearen Algebra befassen mussten, erhalten Sie in diesem Kapitel das notwendige Rüstzeug für dieses Buch. Allerdings empfehlen wir Ihnen dennoch, sich mithilfe eines entsprechenden Werkes ein tieferes Verständnis anzueignen, zum Beispiel mit *Shilov* (1977).

Dieses Kapitel lässt viele wichtige Themen der linearen Algebra unbehandelt, da sie für ein Verständnis des Deep Learnings nicht erforderlich sind.

2.1 Skalare, Vektoren, Matrizen und Tensoren

Die lineare Algebra befasst sich mit unterschiedlichen mathematischen Objekten:

- **Skalare:** Ein Skalar ist entgegen den meisten anderen Objekten in der linearen Algebra, die ein Tupel aus mehreren Zahlen darstellen, lediglich ein einfacher Zahlenwert. Wir notieren Skalare in Kursivschrift. Skalare erhalten üblicherweise kleingeschriebene Variablennamen. Bei ihrer Vorstellung geben wir an, um welche Art Zahl es sich handelt. So schreiben wir zum Beispiel » $s \in \mathbb{R}$ sei die Steigung der Linie« für die Definition einer reellen Zahl oder » $n \in \mathbb{N}$ sei die Anzahl der Einheiten« für die Definition einer natürlichen Zahl.
- **Vektoren:** Ein Vektor ist ein Tupel aus Zahlen, die in einer bestimmten Reihenfolge angeordnet sind. Jede Zahl wird über einen Index genau bestimmt. Vektoren erhalten üblicherweise kleingeschriebene Namen in Fettschrift, zum Beispiel \mathbf{x} . Die Elemente des Vektors sind in Kursivschrift mit einem tiefgestellten Index angegeben. So ist x_1 das erste Element von \mathbf{x} . Das zweite Element ist x_2 und so weiter. Wir müssen auch angeben, was für Zahlen ein Vektor enthält. Falls alle Elemente in \mathbb{R} enthalten sind und der Vektor n Elemente umfasst, ist der Vektor Teil der Menge, die das kartesische Produkt aus \mathbb{R} und n ist. Wir schreiben dafür kurz \mathbb{R}^n . Um die Elemente eines Vektors explizit anzugeben, notieren wir sie als Spalte in eckigen Klammern:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. \quad (2.1)$$

Stellen Sie sich Vektoren als Erkennungsmerkmale im Raum vor. Jedes Element gibt die Koordinaten entlang einer anderen Achse an.

Manchmal müssen wir eine Menge von Elementen eines Vektors mit einem Index versehen. Dazu definieren wir eine Menge aus diesen Indizes und notieren die Menge in tiefgestellter Schrift. So bilden wir zum Beispiel für die Elemente x_1 , x_3 und x_6 die Menge $S = \{1, 3, 6\}$ und notieren \mathbf{x}_S . Um die Komplementärmenge zu indizieren, verwenden wir das Zeichen $-$. So bezeichnet \mathbf{x}_{-1} den Vektor mit allen

Elementen von \mathbf{x} ausgenommen x_1 . Und \mathbf{x}_{-S} ist der Vektor, der alle Elemente von \mathbf{x} bis auf x_1 , x_3 und x_6 enthält.

- **Matrizen:** Eine Matrix ist eine zweidimensionale Zusammenstellung von Zahlen. Jedes Element einer Matrix wird also über zwei Indizes bestimmt. Matrizen erhalten üblicherweise groß geschriebene Variablennamen in Fettschrift, zum Beispiel \mathbf{A} . Wenn eine reellwertige Matrix \mathbf{A} eine Höhe von m und eine Breite von n aufweist, notieren wir dafür $\mathbf{A} \in \mathbb{R}^{m \times n}$. Für die Elemente einer Matrix verwenden wir üblicherweise ihren Namen in Kursivschrift ohne Fettschrift und führen die Indizes durch Kommata getrennt auf. So ist $A_{1,1}$ der obere linke Eintrag von \mathbf{A} und $A_{m,n}$ ist der untere rechte Eintrag. Wir können alle Zahlen mit der vertikalen Koordinate i benennen, indem wir für die horizontale Koordinate das Zeichen »:« schreiben. So ist $\mathbf{A}_{i,:}$ der horizontale Querschnitt von \mathbf{A} an der vertikalen Koordinate i . Es handelt sich um die i -te **Zeile** von \mathbf{A} . Ebenso steht $\mathbf{A}_{:,i}$ für die i -te **Spalte** von \mathbf{A} . Um einzelne Elemente einer Matrix gezielt zu bestimmen, notieren wir dafür eine Anordnung in eckigen Klammern:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}. \quad (2.2)$$

Manchmal müssen wir matrixwertige Ausdrücke indizieren, die aus mehr als nur einem Buchstaben bestehen. Dafür nutzen wir nach dem Ausdruck eine tiefgestellte Schrift ohne Umwandlung in Kleinbuchstaben. So ist $f(\mathbf{A})_{i,j}$ das Element (i, j) der Matrix, die durch Anwendung der Funktion f auf \mathbf{A} berechnet wurde.

- **Tensoren:** Manchmal benötigen wir eine Zusammenstellung, die mehr als zwei Achsen aufweist. Im Allgemeinen wird eine Zusammenstellung von Zahlen, die in einem regelmäßigen Raster mit einer variablen Anzahl an Achsen angeordnet sind, als Tensor bezeichnet. Wir notieren den Tensor namens » \mathbf{A} « auf diese Weise: \mathbf{A} . Ein in \mathbf{A} an den Koordinaten (i, j, k) enthaltenes Element notieren wir als $A_{i,j,k}$.

Eine wichtige Operation im Zusammenhang mit Matrizen ist das **Transponieren**. Die Transponierte einer Matrix ist das Spiegelbild der Matrix an einer Diagonale, die als **Hauptdiagonale** bezeichnet wird; sie verläuft nach unten und nach rechts und beginnt in der oberen linken Ecke. Abbildung 2.1 stellt diese Operation grafisch dar. Wir notieren die Transponierte einer Matrix \mathbf{A} als \mathbf{A}^\top . Die Definition dafür lautet

$$(\mathbf{A}^\top)_{i,j} = A_{j,i}. \quad (2.3)$$

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} \Rightarrow A^\top = \begin{bmatrix} A_{1,1} & A_{2,1} & A_{3,1} \\ A_{1,2} & A_{2,2} & A_{3,2} \end{bmatrix}$$

Abbildung 2.1: Die Transponierte der Matrix ähnelt einem Spiegelbild, das entlang der Hauptdiagonalen entsteht.

Sie können sich Vektoren als Matrizen mit nur einer Spalte vorstellen. Die Transponierte eines Vektors ist daher eine Matrix mit nur einer Zeile. Manchmal definieren wir einen Vektor, indem wir dessen Elemente im Text als Zeilenmatrix schreiben und dann den Transpositionsoperator verwenden, um diese in einen Spaltenvektor umzuwandeln, zum Beispiel $\mathbf{x} = [x_1, x_2, x_3]^\top$.

Sie können sich einen Skalar als eine Matrix mit nur einem Eintrag vorstellen. Daraus folgt, dass ein Skalar seine eigene Transponierte ist: $a = a^\top$.

Matrizen können addiert werden, sofern sie dieselbe Form aufweisen, indem die jeweiligen Elemente addiert werden: $\mathbf{C} = \mathbf{A} + \mathbf{B}$ mit $C_{i,j} = A_{i,j} + B_{i,j}$.

Wir können auch einen Skalar zu einer Matrix addieren oder eine Matrix mit einem Skalar multiplizieren, indem wir die jeweilige Operation mit allen Elementen einer Matrix durchführen: $\mathbf{D} = a \cdot \mathbf{B} + c$ mit $D_{i,j} = a \cdot B_{i,j} + c$.

Im Kontext von Deep Learning verwenden wir zum Teil auch weniger konventionelle Notationen. Wir erlauben das Addieren einer Matrix und eines Vektors, sodass eine weitere Matrix entsteht: $\mathbf{C} = \mathbf{A} + \mathbf{b}$, mit $C_{i,j} = A_{i,j} + b_j$. Anders formuliert: Der Vektor \mathbf{b} wird zu jeder Zeile der Matrix addiert. Diese Abkürzung macht eine Definition einer Matrix überflüssig, bei der zunächst \mathbf{b} in jede Zeile kopiert wird, bevor die Addition erfolgt. Dieses implizite Kopieren von \mathbf{b} an viele Stellen wird als **Broadcasting** bezeichnet.

2.2 Multiplizieren von Matrizen und Vektoren

Eine der wichtigsten Operationen im Zusammenhang mit Matrizen ist die Matrizenmultiplikation. Das **Matrixprodukt** der Matrizen \mathbf{A} und \mathbf{B} ist eine dritte Matrix \mathbf{C} . Damit das Produkt ermittelt werden kann, muss die Anzahl der Spalten von \mathbf{A} der Anzahl der Zeilen von \mathbf{B} entsprechen. Ist \mathbf{A} von der Form $m \times n$ und \mathbf{B} von der Form $n \times p$, dann ist \mathbf{C} von der Form

$m \times p$. Wir können das Matrixprodukt durch Koppeln der zwei (oder mehr) Matrizen schreiben, zum Beispiel als

$$\mathbf{C} = \mathbf{AB}. \quad (2.4)$$

Das Produkt wird wie folgt definiert:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}. \quad (2.5)$$

Beachten Sie, dass das Standardprodukt zweier Matrizen *nicht* einfach eine Matrix mit den Produkten der einzelnen Elemente ist. Das Ergebnis dieser Operation wird nämlich **elementweises Produkt** oder **Hadamard-Produkt** genannt und so notiert: $\mathbf{A} \odot \mathbf{B}$.

Das **Skalarprodukt** zwischen den beiden Vektoren \mathbf{x} und \mathbf{y} derselben Dimensionalität ist das Matrixprodukt $\mathbf{x}^\top \mathbf{y}$. Sie können sich das Matrixprodukt $\mathbf{C} = \mathbf{AB}$ als Berechnung von $C_{i,j}$ als Skalarprodukt zwischen den Zeilen i von \mathbf{A} und den Spalten j von \mathbf{B} vorstellen.

Matrixprodukt-Operationen bieten viele nützliche Eigenschaften, die eine mathematische Untersuchung von Matrizen vereinfachen. Zum Beispiel ist die Matrizenmultiplikation distributiv:

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}. \quad (2.6)$$

Außerdem ist sie assoziativ:

$$\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}. \quad (2.7)$$

Die Matrizenmultiplikation ist im Gegensatz zur Skalarmultiplikation *nicht* kommutativ (die Bedingung $\mathbf{AB} = \mathbf{BA}$ ist nicht immer erfüllt). Allerdings ist das Skalarprodukt zwischen zwei Vektoren kommutativ:

$$\mathbf{x}^\top \mathbf{y} = \mathbf{y}^\top \mathbf{x}. \quad (2.8)$$

Die Transponierte eines Matrixprodukts weist eine einfache Form auf:

$$(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top. \quad (2.9)$$

Damit können wir Gleichung 2.8 beweisen, indem wir die Tatsache nutzen, dass der Wert eines solchen Produkts ein Skalar und daher gleich der eigenen Transponierten ist:

$$\mathbf{x}^\top \mathbf{y} = (\mathbf{x}^\top \mathbf{y})^\top = \mathbf{y}^\top \mathbf{x}. \quad (2.10)$$

Allerdings befasst sich das vorliegende Buch nicht primär mit der linearen Algebra und wir werden nicht versuchen, eine abschließende Liste der nützlichen Eigenschaften des Matrixprodukts zu erstellen. Machen Sie sich einfach bewusst, dass es davon noch viele mehr gibt.

Wir wissen nun genug über die Notation in der linearen Algebra, um ein System linearer Gleichungen aufzustellen:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad (2.11)$$

Dabei gilt: $\mathbf{A} \in \mathbb{R}^{m \times n}$ ist eine bekannte Matrix, $\mathbf{b} \in \mathbb{R}^m$ ist ein bekannter Vektor und $\mathbf{x} \in \mathbb{R}^n$ ist ein Vektor mit unbekannten Variablen, die wir berechnen möchten. Jedes Element x_i von \mathbf{x} ist eine dieser unbekannten Variablen. Jede Zeile von \mathbf{A} und jedes Element von \mathbf{b} liefert eine weitere Bedingung. Wir können Gleichung 2.11 wie folgt neu umstellen:

$$\mathbf{A}_{1,:}\mathbf{x} = b_1 \quad (2.12)$$

$$\mathbf{A}_{2,:}\mathbf{x} = b_2 \quad (2.13)$$

$$\dots \quad (2.14)$$

$$\mathbf{A}_{m,:}\mathbf{x} = b_m \quad (2.15)$$

oder noch deutlicher so:

$$\mathbf{A}_{1,1}x_1 + \mathbf{A}_{1,2}x_2 + \dots + \mathbf{A}_{1,n}x_n = b_1 \quad (2.16)$$

$$\mathbf{A}_{2,1}x_1 + \mathbf{A}_{2,2}x_2 + \dots + \mathbf{A}_{2,n}x_n = b_2 \quad (2.17)$$

$$\dots \quad (2.18)$$

$$\mathbf{A}_{m,1}x_1 + \mathbf{A}_{m,2}x_2 + \dots + \mathbf{A}_{m,n}x_n = b_m. \quad (2.19)$$

Die Notation für Matrix-Vektor-Produkte ermöglicht eine kompakte Darstellung derartiger Gleichungen.

2.3 Einheits- und Umkehrmatrizen

Die lineare Algebra mit der **Matrixinversion** ist ein leistungsstarkes Werkzeug zur analytischen Lösung von Gleichung 2.11 für viele Werte von \mathbf{A} .

Zum Beschreiben der Matrixinversion müssen wir zunächst das Konzept einer **Einheitsmatrix** definieren. Eine Einheitsmatrix ist eine Matrix, die beim Multiplizieren eines Vektors mit dieser Matrix den Vektor nicht

verändert. Wir schreiben die Einheitsmatrix, die n -dimensionale Vektoren unverändert erhält, als \mathbf{I}_n (von Identität). Formal gilt $\mathbf{I}_n \in \mathbb{R}^{n \times n}$ und

$$\forall \mathbf{x} \in \mathbb{R}^n, \mathbf{I}_n \mathbf{x} = \mathbf{x}. \quad (2.20)$$

Die Struktur der Einheitsmatrix ist einfach: Alle Elemente entlang der Hauptdiagonalen sind 1, alle anderen Elemente sind Null. Abbildung 2.2 zeigt ein Beispiel.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Abbildung 2.2: Beispiel einer Einheitsmatrix, hier \mathbf{I}_3

Die **Inverse der Matrix A** wird A^{-1} geschrieben und definiert als die Matrix, für die gilt

$$A^{-1} A = \mathbf{I}_n. \quad (2.21)$$

Wir können nun Gleichung 2.11 wie folgt lösen:

$$A\mathbf{x} = \mathbf{b} \quad (2.22)$$

$$A^{-1} A\mathbf{x} = A^{-1}\mathbf{b} \quad (2.23)$$

$$\mathbf{I}_n \mathbf{x} = A^{-1}\mathbf{b} \quad (2.24)$$

$$\mathbf{x} = A^{-1}\mathbf{b}. \quad (2.25)$$

Dabei gilt die Voraussetzung, dass A^{-1} bestimmt werden kann. Die Bedingungen für die Existenz von A^{-1} werden im nächsten Abschnitt behandelt.

Wenn A^{-1} existiert, ist die Bestimmung der geschlossenen Form mit verschiedenen Algorithmen möglich. Theoretisch kann dieselbe Inverse einer Matrix verwendet werden, um die Gleichung für unterschiedliche Werte von \mathbf{b} mehrmals zu lösen. A^{-1} dient in erster Linie als theoretisches Werkzeug und sollte für die meisten Softwareanwendungen in der Praxis nicht genutzt werden. Da A^{-1} auf einem digitalen Computer nur mit eingeschränkter Genauigkeit dargestellt werden kann, führen Algorithmen, die den Wert von \mathbf{b} nutzen, normalerweise zu exakteren Schätzungen für \mathbf{x} .

2.4 Lineare Abhangigkeit und lineare Hulle

Damit \mathbf{A}^{-1} existieren kann, muss fur Gleichung 2.11 exakt eine Losung fur jeden Wert von \mathbf{b} vorliegen. Das Gleichungssystem kann auch keine oder unendlich viele Losungen fur einige Werte von \mathbf{b} ergeben. Es ist allerdings nicht moglich, mehr als eine und gleichzeitig weniger als unendlich viele Losungen fur einen bestimmten Wert \mathbf{b} zu erhalten; falls sowohl \mathbf{x} als auch \mathbf{y} Losungen sind, dann ist

$$\mathbf{z} = \alpha\mathbf{x} + (1 - \alpha)\mathbf{y} \quad (2.26)$$

auch eine Losung fur jeden reellen Wert α .

Um die Anzahl der Losungen der Gleichung zu analysieren, stellen Sie sich die Spalten von \mathbf{A} als mogliche Bewegungsrichtungen am **Ursprung** (dem Punkt, der von einem nur aus Nullwerten bestehenden Vektor definiert ist) vor und bestimmen Sie dann, auf wie vielen Wegen \mathbf{b} erreicht werden kann. Bei dieser Betrachtung gibt jedes Element von \mathbf{x} an, wie weit wir uns in eine dieser Richtungen bewegen sollen, und x_i gibt an, wie weit wir uns in Richtung der Spalte i bewegen sollen:

$$\mathbf{Ax} = \sum_i x_i \mathbf{A}_{:,i}. \quad (2.27)$$

Allgemein wird eine derartige Operation als **Linearkombination** bezeichnet. Formal betrachtet ergibt sich eine Linearkombination einer Menge von Vektoren $\{\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(n)}\}$ durch Multiplikation jedes Vektors $\mathbf{v}^{(i)}$ mit dem entsprechenden Skalarkoeffizienten und Aufsummieren der Ergebnisse:

$$\sum_i c_i \mathbf{v}^{(i)}. \quad (2.28)$$

Die **lineare Hulle** (oder der Spann) einer Vektormenge ist die Menge aller Datenpunkte, die durch die Linearkombination der ursprnglichen Vektoren erreicht werden knnen.

Um zu bestimmen, ob fur $\mathbf{Ax} = \mathbf{b}$ eine Losung existiert, muss also geprft werden, ob \mathbf{b} in der linearen Hulle der Spalten von \mathbf{A} liegt. Diese spezielle lineare Hulle wird auch als **Spaltenraum** oder **Bild** von \mathbf{A} bezeichnet.

Damit das System $\mathbf{Ax} = \mathbf{b}$ eine Losung fur alle Werte von $\mathbf{b} \in \mathbb{R}^m$ aufweist, mussen wir also sicherstellen, dass der Spaltenraum von \mathbf{A} Teil von \mathbb{R}^m ist. Befindet sich auch nur ein Punkt in \mathbb{R}^m auerhalb des Spaltenraums, ist dieser Punkt ein moglicher Wert von \mathbf{b} ohne Losung. Die Voraussetzung, dass der Spaltenraum von \mathbf{A} vollstandig Teil von \mathbb{R}^m ist,

impliziert unmittelbar, dass \mathbf{A} mindestens m Spalten aufweisen muss, also $n \geq m$. Ansonsten wäre die Dimensionalität des Spaltenraums kleiner als m . Beispiel: Gegeben ist eine 3×2 -Matrix. Das Ziel \mathbf{b} ist dreidimensional, aber \mathbf{x} ist lediglich zweidimensional, sodass eine Änderung des Wertes von \mathbf{x} es maximal ermöglicht, eine 2-D-Ebene innerhalb von \mathbb{R}^3 zu finden. Die Gleichung hat genau dann eine Lösung, wenn \mathbf{b} in dieser Ebene liegt.

$n \geq m$ ist lediglich eine notwendige Bedingung, damit jeder Punkt eine Lösung hat. Es ist keine hinreichende Bedingung, da möglicherweise einige der Spalten redundant sind. Betrachten Sie eine 2×2 -Matrix, in der beide Spalten identisch sind. Sie weist denselben Spaltenraum wie eine 2×1 -Matrix auf, die nur eine Kopie der replizierten Spalte enthält. Mit anderen Worten, der Spaltenraum ist nach wie vor nur eine Linie und kann nicht die Gesamtheit von \mathbb{R}^2 einschließen – selbst im Falle zweier Spalten.

Formal wird diese Art Redundanz als **lineare Abhängigkeit** bezeichnet. Eine Vektormenge ist **linear unabhängig**, wenn kein Vektor der Menge eine Linearkombination der anderen Vektoren darstellt. Wenn wir der Menge einen Vektor hinzufügen, der eine Linearkombination der anderen Vektoren dieser Menge darstellt, fügt der neue Vektor keine Punkte zur linearen Hülle der Menge hinzu. Damit also der Spaltenraum der Matrix die Gesamtheit von \mathbb{R}^m einschließen kann, muss die Matrix mindestens eine Menge mit m linear unabhängigen Spalten enthalten. Diese Bedingung ist sowohl erforderlich als auch genügend, um mit Gleichung 2.11 eine Lösung für jeden Wert von \mathbf{b} zu bestimmen. Beachten Sie, dass die Voraussetzung lautet, dass die Menge exakt m linear unabhängige Spalten aufweist, nicht mindestens m . Keine Menge m -dimensionaler Vektoren kann mehr als m paarweise linear unabhängige Spalten aufweisen, aber eine Matrix mit mehr als m Spalten kann mehr als eine solche Menge enthalten.

Damit eine Inverse zur Matrix existieren kann, müssen wir zusätzlich sicherstellen, dass Gleichung 2.11 *höchstens* eine Lösung für jeden Wert von \mathbf{b} liefert. Dazu müssen wir dafür sorgen, dass die Matrix maximal m Spalten enthält. Ansonsten gibt es mehrere Möglichkeiten zum Parametrisieren jeder Lösung.

Insgesamt folgt daraus, dass die Matrix **quadratisch** sein muss, also $m = n$ gilt, und dass sämtliche Spalten linear unabhängig sind. Eine Quadratmatrix mit linear abhängigen Spalten wird als **singuläre** Matrix bezeichnet.

Wenn \mathbf{A} nicht quadratisch oder aber quadratisch und gleichzeitig singulär ist, kann die Gleichung noch immer gelöst werden, aber die Lösung kann nicht mehr mittels Matrixinversion bestimmt werden.

Bisher haben wir die Inverse als linksseitige Multiplikation behandelt. Es ist allerdings auch möglich, eine rechtsseitig multiplizierte Inverse zu definieren:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \quad (2.29)$$

Für Quadratmatrizen sind die linke und die rechte Inverse identisch.

2.5 Normen

Manchmal müssen wir die Größe eines Vektors messen. Im Machine Learning verwenden wir dazu eine **Norm** genannte Funktion. Formal ergibt sich die L^p -Norm aus

$$\|\mathbf{x}\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}} \quad (2.30)$$

für $p \in \mathbb{R}, p \geq 1$.

Normen wie die L^p -Norm sind Funktionen, mit denen Vektoren nicht-negativen Werten zugeordnet werden. Anschaulich misst die Norm eines Vektors \mathbf{x} den Abstand vom Ursprung zum Punkt \mathbf{x} . Genauer gesagt ist eine Norm eine beliebige Funktion f , die folgende Eigenschaften erfüllt:

- $f(\mathbf{x}) = 0 \Rightarrow \mathbf{x} = \mathbf{0}$
- $f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y})$ (die **Dreiecksungleichung**)
- $\forall \alpha \in \mathbb{R}, f(\alpha\mathbf{x}) = |\alpha|f(\mathbf{x})$

Die L^2 -Norm mit $p = 2$ ist als **euklidische Norm** bekannt; es handelt sich ganz einfach um den euklidischen Abstand vom Ursprung zum durch \mathbf{x} definierten Punkt. Die L^2 -Norm wird im Machine Learning so häufig verwendet, dass für sie häufig lediglich $\|\mathbf{x}\|$ geschrieben wird (die tiefgestellte 2 fällt weg). Die Größe eines Vektors wird meist mithilfe des Quadrats der L^2 -Norm gemessen, die einfach als $\mathbf{x}^\top \mathbf{x}$ berechnet werden kann.

Mit dem Quadrat der L^2 -Norm lässt sich mathematisch bequemer arbeiten und rechnen als mit der L^2 -Norm selbst. Zum Beispiel ist jede Ableitung des Quadrats der L^2 -Norm bezüglich der einzelnen Elemente von \mathbf{x} nur von dem entsprechenden Element von \mathbf{x} abhängig, wohingegen sämtliche Ableitungen der L^2 -Norm vom gesamten Vektor abhängig sind. In vielen Bereichen wird das Quadrat der L^2 -Norm allerdings nicht gewünscht, da sie in Ursprungsnähe sehr langsam ansteigt. In einigen Machine-Learning-Anwendungen ist es wichtig, zwischen Elementen, die exakt Null sind, und anderen

Elementen, die klein, aber von Null verschieden sind, zu unterscheiden. In diesen Fällen greifen wir auf eine Funktion zurück, die in allen Punkten mit derselben Rate wächst, aber mathematisch einfach zu handhaben ist – die L^1 -Norm. Die L^1 -Norm kann vereinfacht werden zu

$$\|\boldsymbol{x}\|_1 = \sum_i |x_i|. \quad (2.31)$$

Die L^1 -Norm wird im Machine Learning weithin verwendet, wenn die Unterscheidung zwischen Elementen gleich und nicht gleich Null sehr wichtig ist. Jedes Mal, wenn sich ein Element von \boldsymbol{x} um ϵ von 0 entfernt, nimmt die L^1 -Norm um ϵ zu.

Manchmal müssen wir die Größe eines Vektors durch Zählen der von Null verschiedenen Elemente messen. Einige Autoren nennen diese Funktion die » L^0 -Norm«, aber dieser Begriff ist nicht korrekt. Die Anzahl der von 0 verschiedenen Elemente eines Vektors ist keine Norm, denn eine Skalierung des Vektors um α führt nicht zu einer Änderung dieser Anzahl. Häufig wird die L^1 -Norm als Ersatz für die Anzahl der von Null verschiedenen Elemente verwendet.

Eine weitere Norm, die häufig für das Machine Learning genutzt wird, ist die L^∞ -Norm, auch als **Maximumsnorm** bezeichnet. Diese Norm vereinfacht sich auf den Absolutbetrag des Elements mit dem größten Betrag im Vektor,

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i|. \quad (2.32)$$

Manchmal möchten wir auch die Größe einer Matrix messen. Im Kontext des Deep Learnings geschieht dies häufig mit der ansonsten eher obskuren **Frobenius-Norm**

$$\|A\|_F = \sqrt{\sum_{i,j} A_{i,j}^2}, \quad (2.33)$$

und das ist analog zur L^2 -Norm eines Vektors.

Das Skalarprodukt zweier Vektoren kann mithilfe von Normen notiert werden. Insbesondere

$$\boldsymbol{x}^\top \boldsymbol{y} = \|\boldsymbol{x}\|_2 \|\boldsymbol{y}\|_2 \cos \theta, \quad (2.34)$$

wobei θ dem Winkel zwischen \boldsymbol{x} und \boldsymbol{y} entspricht.

2.6 Spezielle Matrizen und Vektoren

Es gibt einige recht nützliche Sonderformen von Matrizen und Vektoren.

Diagonalmatrizen bestehen hauptsächlich aus Nullen und enthalten nur entlang der Hauptdiagonalen von Null verschiedene Einträge. Formal ist eine Matrix \mathbf{D} genau dann diagonal, wenn $D_{i,j} = 0$ für alle $i \neq j$ gilt. Wir haben zuvor bereits ein Beispiel für eine Diagonalmatrix vorgestellt, nämlich die Einheitsmatrix, bei der sämtliche Einträge auf der Diagonalen 1 sind. Wir schreiben $\text{diag}(\mathbf{v})$ für eine quadratische Diagonalmatrix, deren Diagonaleinträge durch die Elemente des Vektors \mathbf{v} bestimmt werden. Diagonalmatrizen sind zum Teil deswegen interessant, weil das Multiplizieren mit einer Diagonalmatrix rechnerisch effizient ist. Für die Berechnung von $\text{diag}(\mathbf{v})\mathbf{x}$ muss nur jedes Element x_i um v_i skaliert werden. Anders ausgedrückt: $\text{diag}(\mathbf{v})\mathbf{x} = \mathbf{v} \odot \mathbf{x}$. Das Umkehren einer quadratischen Diagonalmatrix ist ebenfalls effizient. Die Inverse existiert nur, wenn jeder Diagonaleintrag nicht Null ist, sodass folgt $\text{diag}(\mathbf{v})^{-1} = \text{diag}([1/v_1, \dots, 1/v_n]^\top)$. Oft leiten wir einen allgemeinen Machine-Learning-Algorithmus mithilfe beliebiger Matrizen ab, erhalten aber einen weniger aufwendigen (und deskriptiven) Algorithmus, wenn wir einige Matrizen auf Diagonalmatrizen einschränken.

Nicht alle Diagonalmatrizen müssen quadratisch sein. Es ist möglich, eine rechteckige Diagonalmatrix zu erstellen. Nicht-quadratische Diagonalmatrizen weisen keine Inversen auf, aber sie eignen sich dennoch gut für »günstige« Multiplikationen. Um das Produkt $\mathbf{D}\mathbf{x}$ einer nicht-quadratischen Diagonalmatrix \mathbf{D} zu bestimmen, wird jedes Element von \mathbf{x} skaliert und das Ergebnis entweder um einige Nullen ergänzt (falls \mathbf{D} höher als breit ist) oder um einige der letzten Vektorelemente gekürzt (falls \mathbf{D} breiter als hoch ist).

Eine **symmetrische** Matrix ist jede Matrix, die gleich ihrer eigenen Transponierten ist:

$$\mathbf{A} = \mathbf{A}^\top. \quad (2.35)$$

Symmetrische Matrizen finden sich oft dort, wo die Einträge durch eine Funktion von zwei Argumenten erzeugt werden, bei der die Reihenfolge der Argumente keine Rolle spielt. Ist zum Beispiel \mathbf{A} eine Matrix für Abstandsmessungen und steht $A_{i,j}$ für den Abstand zwischen den Punkten i und j , dann gilt $A_{i,j} = A_{j,i}$, da die Abstandsfunktionen symmetrisch sind.

Ein **Einheitsvektor** ist ein Vektor mit der **Norm 1**:

$$\|\mathbf{x}\|_2 = 1. \quad (2.36)$$

Ein Vektor \mathbf{x} und ein Vektor \mathbf{y} sind **orthogonal** zueinander, wenn gilt $\mathbf{x}^\top \mathbf{y} = 0$. Weisen beide Vektoren eine von Null verschiedene Norm auf, so stehen sie im rechten Winkel (90 Grad) zueinander. In \mathbb{R}^n sind höchstens n Vektoren paarweise orthogonal mit von Null verschiedener Norm. Falls

die Vektoren nicht nur orthogonal sind, sondern auch die Norm 1 aufweisen, bezeichnen wir sie als **orthonormal**.

Eine **orthogonale Matrix** ist eine quadratische Matrix, deren Zeilen und Spalten jeweils paarweise orthonormal sind:

$$\mathbf{A}^\top \mathbf{A} = \mathbf{A} \mathbf{A}^\top = \mathbf{I}. \quad (2.37)$$

Daraus folgt

$$\mathbf{A}^{-1} = \mathbf{A}^\top, \quad (2.38)$$

sodass orthogonale Matrizen aufgrund des geringen Berechnungsaufwands für die Inverse von Interesse sind. Achten Sie besonders sorgfältig auf die Definition von orthogonalen Matrizen. Obschon kontraintuitiv, sind ihre Zeilen nicht nur orthogonal, sondern vollständig orthonormal. Es gibt keine eigenständige Bezeichnung für eine Matrix, deren Zeilen oder Spalten zwar orthogonal, aber nicht orthonormal sind.

2.7 Eigenwertzerlegung

Viele mathematische Objekte sind leichter zu verstehen, wenn man sie in ihre Bestandteile zerlegt oder Eigenschaften bestimmt, die allgemeingültig und nicht ihrer Darstellung geschuldet sind.

Ganze Zahlen können zum Beispiel in Primfaktoren zerlegt werden. Unsere Darstellung der Zahl 12 ändert sich abhängig davon, ob wir sie zur Basis 10 oder binär notieren. Aber es ist eine universelle Wahrheit, dass $12 = 2 \times 2 \times 3$ ist. Aus dieser Darstellung lässt sich auf einige nützliche Eigenschaften schließen: So ist 12 nicht durch 5 teilbar und jedes ganzzahlige Vielfache von 12 ist durch 3 teilbar.

Ebenso wie wir durch die Zerlegung in Primfaktoren mehr über die wahre Natur einer ganzen Zahl erfahren können, erfahren wir durch eine Zerlegung von Matrizen auch etwas über ihre funktionalen Eigenschaften, die in der Matrixdarstellung als Zusammenstellung von Elementen nicht offenkundig sind.

Eines der gängigsten Verfahren zur Zerlegung von Matrizen ist die **Eigenwertzerlegung**; dabei wird eine Matrix in eine Menge von Eigenvektoren und Eigenwerten zerlegt.

Ein **Eigenvektor** einer Quadratmatrix \mathbf{A} ist ein Nicht-Null-Vektor \mathbf{v} der Art, dass die Multiplikation mit \mathbf{A} nur die Streckung von \mathbf{v} ändert:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \quad (2.39)$$

Der Skalar λ wird als **Eigenwert** für diesen Eigenvektor bezeichnet. (Auch der **linke Eigenvektor** $v^\top A = \lambda v^\top$ kann bestimmt werden, aber uns geht es normalerweise um die rechten Eigenvektoren.)

Ist v ein Eigenvektor von A , dann gilt das auch für jeden Vektor mit veränderter Streckung sv für $s \in \mathbb{R}, s \neq 0$. Außerdem weist sv nach wie vor denselben Eigenwert auf. Daher suchen wir meist nur nach Einheits-Eigenvektoren.

Gegeben sei eine Matrix A mit n linear unabhängigen Eigenvektoren $\{v^{(1)}, \dots, v^{(n)}\}$ und den zugehörigen Eigenwerten $\{\lambda_1, \dots, \lambda_n\}$. Wir können alle Eigenvektoren zu einer Matrix V mit einem Eigenvektor pro Spalte verknüpfen: $V = [v^{(1)}, \dots, v^{(n)}]$. Ebenso können wir die Eigenwerte zu einem Vektor $\lambda = [\lambda_1, \dots, \lambda_n]^\top$ verknüpfen. Die **Eigenwertzerlegung** von A ergibt sich dann aus

$$A = V \text{diag}(\lambda) V^{-1}. \quad (2.40)$$

Wir haben gesehen, dass das *Konstruieren* von Matrizen mit bestimmten Eigenwerten und Eigenvektoren uns in die Lage versetzt, den Raum in die gewünschten Richtungen zu strecken. Allerdings möchten wir Matrizen häufig in ihre Eigenwerte und Eigenvektoren **zerlegen**. Damit können wir nämlich bestimmte Eigenschaften der Matrix untersuchen (ähnlich der Zerlegung ganzer Zahlen in Primfaktoren).

Nicht jede Matrix kann in Eigenwerte und Eigenvektoren zerlegt werden. In einigen Fällen ist eine Zerlegung zwar möglich, enthält aber komplexe anstelle von reellen Zahlen. Zum Glück geht es in diesem Buch im Normalfall nur um eine Zerlegung einer bestimmten Klasse von Matrizen, für die eine einfache Zerlegung existiert. Insbesondere lässt sich jede reelle symmetrische Matrix in einen Ausdruck zerlegen, der nur reellwertige Eigenvektoren und Eigenwerte aufweist:

$$A = Q \Lambda Q^\top, \quad (2.41)$$

wobei Q eine orthogonale Matrix aus den Eigenvektoren von A ist und Λ eine Diagonalmatrix. Der Eigenwert $\Lambda_{i,i}$ ist dem Eigenvektor in Spalte i von Q zugeordnet und wird als $Q_{:,i}$ notiert. Da es sich bei Q um eine orthogonale Matrix handelt, können wir uns A als Streckung des Raums um λ_i in der Richtung $v^{(i)}$ vorstellen. Ein Beispiel finden Sie in Abbildung 2.3.

Obwohl jede reelle symmetrische Matrix A garantiert eine Eigenwertzerlegung hat, ist diese Eigenwertzerlegung möglicherweise nicht eindeutig. Falls zwei oder mehr Eigenvektoren denselben Eigenwert aufweisen, sind sämtliche orthogonalen Vektoren, die in ihrer linearen Hülle liegen, ebenfalls Eigenvektoren mit demselben Eigenwert und wir können stattdessen eine

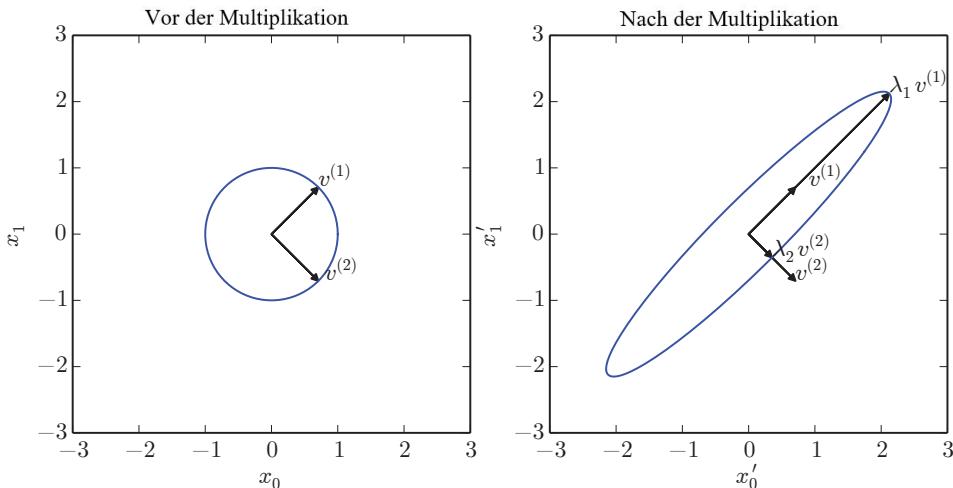


Abbildung 2.3: Ein Beispiel für die Auswirkung von Eigenvektoren und Eigenwerten. Gegeben ist eine Matrix \mathbf{A} mit zwei orthonormalen Eigenvektoren: $\mathbf{v}^{(1)}$ mit dem Eigenwert λ_1 und $\mathbf{v}^{(2)}$ mit dem Eigenwert λ_2 . (Links) Wir zeichnen die Menge aller Einheitsvektoren $\mathbf{u} \in \mathbb{R}^2$ als Einheitskreis. (Rechts) Wir zeichnen die Menge aller Punkte \mathbf{Au} . Die Art, in der \mathbf{A} den Einheitskreis verzerrt, zeigt, dass der Raum in Richtung $\mathbf{v}^{(i)}$ um λ_i gestreckt wird.

beliebige \mathbf{Q} mit diesen Eigenvektoren verwenden. Man ist übereingekommen, die Einträge von $\mathbf{\Lambda}$ in absteigender Reihenfolge zu sortieren. Beachtet man diese Konvention, ist die Eigenwertzerlegung nur dann eindeutig, wenn alle Eigenwerte eindeutig sind.

Die Eigenwertzerlegung einer Matrix offenbart viele nützliche Fakten über die Matrix. Die Matrix ist genau dann singulär, wenn einer der Eigenwerte gleich 0 ist. Die Eigenwertzerlegung einer reellen symmetrischen Matrix kann auch zum Optimieren quadratischer Gleichungen der Form $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A} \mathbf{x}$ mit $\|\mathbf{x}\|_2 = 1$ verwendet werden. Wann immer \mathbf{x} gleich einem Eigenvektor von \mathbf{A} ist, nimmt f den Wert des zugehörigen Eigenwerts an. Der Höchstwert von f im eingeschränkten Bereich ist der größtmögliche Eigenwert; der Mindestwert in diesem Bereich ist der kleinstmögliche Eigenwert.

Eine Matrix, deren Eigenwerte alle positiv sind, heißt **positiv definit**. Eine Matrix, deren Eigenwerte alle positiv oder Null sind, heißt **positiv semidefinit**. Sind dagegen alle Eigenwerte negativ, heißt die Matrix **negativ definit** und für Eigenwerte, die negativ oder Null sind, **negativ semidefinit**. Positiv semidefinite Matrizen sind interessant, da für sie ga-

rantiert Folgendes gilt: $\forall \mathbf{x}, \mathbf{x}^\top \mathbf{A}\mathbf{x} \geq 0$. Positiv definite Matrizen garantieren zusätzlich noch $\mathbf{x}^\top \mathbf{A}\mathbf{x} = 0 \Rightarrow \mathbf{x} = \mathbf{0}$.

2.8 Singulärwertzerlegung

In Abschnitt 2.7 haben Sie erfahren, wie eine Matrix in Eigenvektoren und Eigenwerte zerlegt wird. Die **Singulärwertzerlegung** (engl. *singular value decomposition*, SVD) ist eine weitere Möglichkeit zum Faktorisieren einer Matrix, und zwar in **Singulärvektoren** und **Singulärwerte**. Die SVD enthüllt einige der Informationen aus der Eigenwertzerlegung, ist allerdings deutlich vielseitiger einsetzbar. Jede reelle Matrix ermöglicht eine Singulärwertzerlegung. Das gilt für die Eigenwertzerlegung nicht. Ist eine Matrix zum Beispiel nicht quadratisch, ist die Eigenwertzerlegung nicht definiert und wir müssen auf die Singulärwertzerlegung zurückgreifen.

Denken Sie daran, dass die Eigenwertzerlegung eine Analyse einer Matrix \mathbf{A} erfordert, um eine Matrix \mathbf{V} der Eigenvektoren und einen Vektor der Eigenwerte $\boldsymbol{\lambda}$ zu finden, damit wir \mathbf{A} wie folgt neu notieren können:

$$\mathbf{A} = \mathbf{V} \text{diag}(\boldsymbol{\lambda}) \mathbf{V}^{-1}. \quad (2.42)$$

Die Singulärwertzerlegung verläuft ähnlich, aber hier schreiben wir \mathbf{A} als ein Produkt aus drei Matrizen:

$$\mathbf{A} = \mathbf{U} \mathbf{D} \mathbf{V}^\top. \quad (2.43)$$

Gegeben sei eine $m \times n$ -Matrix \mathbf{A} . Dann ist \mathbf{U} als $m \times m$ -Matrix definiert, \mathbf{D} als $m \times n$ -Matrix und \mathbf{V} als $n \times n$ -Matrix.

Jede dieser Matrizen hat per definitionem eine spezielle Struktur. Die Matrizen \mathbf{U} und \mathbf{V} sind beide als orthogonale Matrizen definiert. Die Matrix \mathbf{D} ist als Diagonalmatrix definiert. Beachten Sie, dass \mathbf{D} nicht unbedingt quadratisch sein muss.

Die Elemente entlang der Diagonalen von \mathbf{D} werden **Singulärwerte** der Matrix \mathbf{A} genannt. Die Spalten von \mathbf{U} werden als **linke Singulärvektoren** bezeichnet, die Spalten von \mathbf{V} als **rechte Singulärvektoren**.

Wir können die Singulärwertzerlegung von \mathbf{A} tatsächlich anhand der Eigenwertzerlegung als Funktionen von \mathbf{A} erklären. Die linken Singulärvektoren von \mathbf{A} sind die Eigenvektoren von $\mathbf{A}\mathbf{A}^\top$ und die rechten Singulärvektoren von \mathbf{A} sind die Eigenvektoren von $\mathbf{A}^\top\mathbf{A}$. Die von Null verschiedenen Singulärwerte von \mathbf{A} sind die Quadratwurzeln der Eigenwerte von $\mathbf{A}^\top\mathbf{A}$. Dasselbe gilt für $\mathbf{A}\mathbf{A}^\top$.

Das wohl nützlichste Merkmal der SVD ist die Möglichkeit, damit eine teilweise generalisierte Matrixinversion für nicht-quadratische Matrizen durchzuführen (siehe nächster Abschnitt).

2.9 Die Moore-Penrose-Pseudoinverse

Für nicht quadratische Matrizen ist die Matrixinversion nicht definiert. Angenommen, wir möchten die linke Inverse \mathbf{B} einer Matrix \mathbf{A} bestimmen, um eine lineare Gleichung

$$\mathbf{Ax} = \mathbf{y} \quad (2.44)$$

durch linksseitige Multiplikation jeder Seite zu lösen mit dem Ergebnis

$$\mathbf{x} = \mathbf{By}. \quad (2.45)$$

Je nach Struktur des Problems gibt es eventuell keine eindeutige Zuordnung von \mathbf{A} zu \mathbf{B} .

Ist \mathbf{A} höher als breit, dann hat die Gleichung möglicherweise keine Lösung. Ist \mathbf{A} breiter als hoch, dann gibt es möglicherweise mehrere Lösungen.

Die **Moore-Penrose-Pseudoinverse** hilft in solchen Fällen weiter. Die Pseudoinverse von \mathbf{A} ist definiert als eine Matrix

$$\mathbf{A}^+ = \lim_{\alpha \searrow 0} (\mathbf{A}^\top \mathbf{A} + \alpha \mathbf{I})^{-1} \mathbf{A}^\top. \quad (2.46)$$

Praktische Algorithmen zum Berechnen der Pseudoinversen basieren allerdings nicht auf dieser Definition, sondern verwenden vielmehr die Formel

$$\mathbf{A}^+ = \mathbf{V} \mathbf{D}^+ \mathbf{U}^\top \quad (2.47)$$

mit \mathbf{U} , \mathbf{D} und \mathbf{V} als Singulärwertzerlegung von \mathbf{A} und der Pseudoinversen \mathbf{D}^+ einer Diagonalmatrix \mathbf{D} als Ergebnis des Kehrwerts ihrer von Null verschiedenen Elemente mit anschließender Transponierung der resultierenden Matrix.

Wenn \mathbf{A} mehr Spalten als Zeilen aufweist, führt die Lösung einer linearen Gleichung anhand der Pseudoinversen zu einer von vielen möglichen Lösungen. Im Besonderen liefert sie die Lösung $\mathbf{x} = \mathbf{A}^+ \mathbf{y}$ mit minimaler euklidischer Norm $\|\mathbf{x}\|_2$ aus allen möglichen Lösungen.

Wenn \mathbf{A} mehr Zeilen als Spalten aufweist, existiert möglicherweise keine Lösung. In diesem Fall ergibt die Pseudoinverse das \mathbf{x} , für das sich \mathbf{Ax} so weit wie möglich an \mathbf{y} annähert (im Sinne der euklidischen Norm $\|\mathbf{Ax} - \mathbf{y}\|_2$).

2.10 Der Spuroperator

Der Spuroperator gibt die Summe aller Diagonaleinträge einer Matrix an:

$$\text{Tr}(\mathbf{A}) = \sum_i A_{i,i}. \quad (2.48)$$

Der Spuroperator ist aus mehreren Gründen nützlich. Einige Operationen, die ohne Rückgriff auf die Summationsnotation nur schwer zu spezifizieren sind, können mittels Matrixprodukten und des Spuroperators spezifiziert werden. Ein Beispiel: Der Spuroperator bietet eine alternative Notation für die Frobenius-Norm einer Matrix:

$$\|A\|_F = \sqrt{\text{Tr}(\mathbf{A}\mathbf{A}^\top)}. \quad (2.49)$$

Wird ein Ausdruck mit dem Spuroperator notiert, ist es möglich, den Ausdruck mithilfe vieler nützlicher Identitäten zu manipulieren. So ist der Spuroperator invariant gegenüber dem Transpositionsoperator:

$$\text{Tr}(\mathbf{A}) = \text{Tr}(\mathbf{A}^\top). \quad (2.50)$$

Die Spur einer Quadratmatrix, die aus vielen Faktoren besteht, ist ebenfalls invariant gegenüber einer Verschiebung des letzten Faktors an die erste Stelle, sofern die Formen der zugehörigen Matrizen eine Definition des resultierenden Produkts erlauben:

$$\text{Tr}(\mathbf{ABC}) = \text{Tr}(\mathbf{CAB}) = \text{Tr}(\mathbf{BCA}) \quad (2.51)$$

oder allgemeiner formuliert,

$$\text{Tr}\left(\prod_{i=1}^n \mathbf{F}^{(i)}\right) = \text{Tr}(\mathbf{F}^{(n)} \prod_{i=1}^{n-1} \mathbf{F}^{(i)}). \quad (2.52)$$

Diese Invarianz gegenüber zyklischer Permutation gilt auch dann, wenn das resultierende Produkt eine andere Form aufweist. Für $\mathbf{A} \in \mathbb{R}^{m \times n}$ und $\mathbf{B} \in \mathbb{R}^{n \times m}$ ergibt sich zum Beispiel

$$\text{Tr}(\mathbf{AB}) = \text{Tr}(\mathbf{BA}), \quad (2.53)$$

obwohl gilt $\mathbf{AB} \in \mathbb{R}^{m \times m}$ und $\mathbf{BA} \in \mathbb{R}^{n \times n}$.

Eine weitere nützliche Tatsache, die Sie im Hinterkopf behalten sollten, ist, dass ein Skalar seine eigene Spur darstellt: $a = \text{Tr}(a)$.

2.11 Die Determinante

Die Determinante einer Quadratmatrix wird $\det(\mathbf{A})$ geschrieben; diese Funktion ordnet Matrizen den reellen Skalaren zu. Die Determinante ist gleich dem Produkt aller Eigenwerte der Matrix. Sie können sich den Absolutbetrag der Determinante als ein Maß dafür vorstellen, wie stark die Multiplikation mit der Matrix den Raum dehnt oder zusammenzieht. Ist die Determinante 0, dann wird der Raum entlang mindestens einer Dimension vollständig zusammengezogen, sodass er sein gesamtes Volumen verliert. Ist die Determinante 1, dann erhält die Transformation das Volumen.

2.12 Beispiel: Hauptkomponentenanalyse

Ein einfacher Machine-Learning-Algorithmus, die **Hauptkomponentenanalyse** (engl. *principal components analysis*, PCA), kann rein mit Kenntnissen der grundlegenden linearen Algebra abgeleitet werden.

Gegeben sei eine Sammlung von m Datenpunkten $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ in \mathbb{R}^n , auf die wir eine verlustbehaftete Komprimierung anwenden möchten. Eine verlustbehaftete Komprimierung bedeutet, dass die Punkte so gespeichert werden, dass sie weniger Speicherplatz belegen; dabei kann es jedoch zu einem gewissen Genauigkeitsverlust kommen. Natürlich möchten wir den Verlust der Genauigkeit möglichst gering halten.

Eine Möglichkeit, diese Punkte zu codieren, besteht darin, sie als niedrig-dimensionalere Versionen ihrer selbst abzubilden. Für jeden Punkt $\mathbf{x}^{(i)} \in \mathbb{R}^n$ suchen wir einen zugehörigen Code-Vektor $\mathbf{c}^{(i)} \in \mathbb{R}^l$. Ist l kleiner als n , wird zum Speichern der Code-Punkte weniger Speicherplatz benötigt als für die Ausgangsdaten. Wir suchen also eine Codierfunktion, die den Code für eine Eingabe $f(\mathbf{x}) = \mathbf{c}$ erzeugt, sowie eine Decodierfunktion, die anhand des Codes $\mathbf{x} \approx g(f(\mathbf{x}))$ die Eingabedaten rekonstruiert.

Die PCA wird über unsere Wahl der Decodierfunktion bestimmt. Da wir den Decoder möglichst einfach gestalten möchten, entscheiden wir uns für die Matrizenmultiplikation, um den Code \mathbb{R}^n zuzuordnen. Es sei $g(\mathbf{c}) = \mathbf{D}\mathbf{c}$, mit $\mathbf{D} \in \mathbb{R}^{n \times l}$ als Matrix, die die Decodierung definiert.

Das Berechnen des optimalen Codes für diesen Decoder könnte sich als großes Problem erweisen. Um das Codierungsproblem einfach zu halten, erfordert die PCA, dass die Spalten von \mathbf{D} orthogonal zueinander sind. (Beachten Sie, dass \mathbf{D} technisch gesehen noch immer keine »orthogonale Matrix« ist, es sei denn, es gilt $l = n$.)

Für die bisherige Problembeschreibung sind viele Lösungen möglich, denn wir können das Maß von $D_{:,i}$ erhöhen, indem wir c_i proportional für alle Punkte verringern. Um dem Problem eine eindeutige Lösung zu geben, schränken wir alle Spalten von D ein, eine Norm gleich 1 zu besitzen.

Um anhand dieses grundlegenden Konzepts einen implementierbaren Algorithmus erstellen zu können, müssen wir herausfinden, wie der optimale Code-Punkt \mathbf{c}^* für jeden Eingabepunkt \mathbf{x} erzeugt wird. Eine Möglichkeit besteht darin, den Abstand zwischen dem Eingabepunkt \mathbf{x} und dessen Rekonstruktion $g(\mathbf{c}^*)$ zu minimieren. Wir können diesen Abstand mithilfe einer Norm messen. Im Hauptkomponentenalgorithmus verwenden wir die L^2 -Norm:

$$\mathbf{c}^* = \arg \min_{\mathbf{c}} \|\mathbf{x} - g(\mathbf{c})\|_2. \quad (2.54)$$

Wir können das Quadrat der L^2 -Norm anstelle der L^2 -Norm selbst nutzen, da beide durch denselben Wert für \mathbf{c} minimiert werden. Beide sind durch denselben Wert für \mathbf{c} minimiert, da die L^2 -Norm nicht-negativ ist und das Quadrieren für nicht-negative Argumente monoton steigt.

$$\mathbf{c}^* = \arg \min_{\mathbf{c}} \|\mathbf{x} - g(\mathbf{c})\|_2^2. \quad (2.55)$$

Die hier minimierte Funktion lässt sich vereinfachen zu

$$(\mathbf{x} - g(\mathbf{c}))^\top (\mathbf{x} - g(\mathbf{c})) \quad (2.56)$$

(durch Definition der L^2 -Norm, Gleichung 2.30)

$$= \mathbf{x}^\top \mathbf{x} - \mathbf{x}^\top g(\mathbf{c}) - g(\mathbf{c})^\top \mathbf{x} + g(\mathbf{c})^\top g(\mathbf{c}) \quad (2.57)$$

(aufgrund der distributiven Eigenschaft)

$$= \mathbf{x}^\top \mathbf{x} - 2\mathbf{x}^\top g(\mathbf{c}) + g(\mathbf{c})^\top g(\mathbf{c}) \quad (2.58)$$

(da der Skalar $g(\mathbf{c})^\top \mathbf{x}$ gleich seiner Transponierten ist).

Wir können nun die hier minimierte Funktion erneut ändern und so den ersten Term ausschließen, da dieser Term nicht abhängig von \mathbf{c} ist:

$$\mathbf{c}^* = \arg \min_{\mathbf{c}} -2\mathbf{x}^\top g(\mathbf{c}) + g(\mathbf{c})^\top g(\mathbf{c}). \quad (2.59)$$

Für den weiteren Verlauf setzen wir die Definition von $g(\mathbf{c})$ ein:

$$\mathbf{c}^* = \arg \min_{\mathbf{c}} -2\mathbf{x}^\top D\mathbf{c} + \mathbf{c}^\top D^\top D\mathbf{c} \quad (2.60)$$

$$= \arg \min_c -2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{I}_l \mathbf{c} \quad (2.61)$$

(\mathbf{D} muss orthogonal und dessen Norm gleich 1 sein)

$$= \arg \min_c -2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{c}. \quad (2.62)$$

Wir können dieses Optimierungsproblem anhand der Vektoranalysis lösen (Abschnitt 4.3 zeigt, wie das geht):

$$\nabla_{\mathbf{c}} (-2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{c}) = \mathbf{0} \quad (2.63)$$

$$-2\mathbf{D}^\top \mathbf{x} + 2\mathbf{c} = \mathbf{0} \quad (2.64)$$

$$\mathbf{c} = \mathbf{D}^\top \mathbf{x}. \quad (2.65)$$

So wird der Algorithmus effizient: Wir können \mathbf{x} optimal rein mit einer Matrix-Vektor-Operation codieren. Zum Codieren eines Vektors verwenden wir die Encoder-Funktion

$$f(\mathbf{x}) = \mathbf{D}^\top \mathbf{x}. \quad (2.66)$$

Durch eine weitere Matrizenmultiplikation können wir auch die PCA-Rekonstruktion definieren:

$$r(\mathbf{x}) = g(f(\mathbf{x})) = \mathbf{D}\mathbf{D}^\top \mathbf{x}. \quad (2.67)$$

Jetzt müssen wir die Codiermatrix \mathbf{D} auswählen. Dazu befassen wir uns nochmals mit dem Minimieren des Abstands L^2 zwischen Eingabe und Rekonstruktion. Da wir dieselbe Matrix \mathbf{D} zum Decodieren aller Punkte einsetzen, dürfen wir die Punkte nicht länger isoliert betrachten. Stattdessen müssen wir die Frobenius-Norm der über alle Dimensionen und Punkte berechneten Fehlermatrix minimieren:

$$\mathbf{D}^* = \arg \min_{\mathbf{D}} \sqrt{\sum_{i,j} \left(x_j^{(i)} - r(\mathbf{x}^{(i)})_j \right)^2} \text{ für } \mathbf{D}^\top \mathbf{D} = \mathbf{I}_l. \quad (2.68)$$

Um einen Algorithmus zur Bestimmung von \mathbf{D}^* abzuleiten, betrachten wir zunächst den Fall $l = 1$. In diesem Fall ist \mathbf{D} lediglich ein einzelner Vektor \mathbf{d} . Einsetzen von Gleichung 2.67 in Gleichung 2.68 und Vereinfachen von \mathbf{D} in \mathbf{d} reduziert das Problem auf

$$\mathbf{d}^* = \arg \min_{\mathbf{d}} \sum_i \|\mathbf{x}^{(i)} - \mathbf{d}\mathbf{d}^\top \mathbf{x}^{(i)}\|_2^2 \text{ für } \|\mathbf{d}\|_2 = 1. \quad (2.69)$$

Die obige Formulierung ist der direkte Weg zum Durchführen der Ersetzung. Aber es gibt eine schönere Schreibweise, bei der der Skalarwert $\mathbf{d}^\top \mathbf{x}^{(i)}$ rechts vom Vektor \mathbf{d} eingesetzt wird. Skalarkoeffizienten werden üblicherweise links vom Vektor, auf den sie wirken, notiert. Daher schreiben wir diese Formeln normalerweise als

$$\mathbf{d}^* = \arg \min_{\mathbf{d}} \sum_i \|\mathbf{x}^{(i)} - \mathbf{d}^\top \mathbf{x}^{(i)} \mathbf{d}\|_2^2 \text{ für } \|\mathbf{d}\|_2 = 1, \quad (2.70)$$

oder, unter Ausnutzung der Tatsache, dass ein Skalar seine eigene Transponierte ist, als

$$\mathbf{d}^* = \arg \min_{\mathbf{d}} \sum_i \|\mathbf{x}^{(i)} - \mathbf{x}^{(i)\top} \mathbf{d} \mathbf{d}\|_2^2 \text{ für } \|\mathbf{d}\|_2 = 1. \quad (2.71)$$

Bitte machen Sie sich mit solchen kosmetischen Umstellungen vertraut.

Es ist nun hilfreich, das Problem als einzelne Entwurfsmatrix mit Beispielen neu zu schreiben und so auf die Darstellung als Summe separater Beispieldatenvektoren zu verzichten. Damit erreichen wir eine kompaktere Notation. Es sei $\mathbf{X} \in \mathbb{R}^{m \times n}$ die durch Stapeln aller Vektoren zur Punktbeschreibung definierte Matrix, sodass gilt $\mathbf{X}_{i,:} = \mathbf{x}^{(i)\top}$. Nun können wir das Problem wie folgt neu schreiben:

$$\mathbf{d}^* = \arg \min_{\mathbf{d}} \|\mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top\|_F^2 \text{ für } \mathbf{d}^\top \mathbf{d} = 1. \quad (2.72)$$

Lassen wir die Bedingung zunächst außer Betracht, dann können wir den Teil mit der Frobenius-Norm wie folgt vereinfachen:

$$\arg \min_{\mathbf{d}} \|\mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top\|_F^2 \quad (2.73)$$

$$= \arg \min_{\mathbf{d}} \text{Tr} \left((\mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top)^\top (\mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top) \right) \quad (2.74)$$

(laut Gleichung 2.49)

$$= \arg \min_{\mathbf{d}} \text{Tr}(\mathbf{X}^\top \mathbf{X} - \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top - \mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} + \mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) \quad (2.75)$$

$$= \arg \min_{\mathbf{d}} \text{Tr}(\mathbf{X}^\top \mathbf{X}) - \text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) - \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X}) + \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) \quad (2.76)$$

$$= \arg \min_{\mathbf{d}} -\text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) - \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X}) + \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) \quad (2.77)$$

(da Terme ohne \mathbf{d} arg min nicht beeinflussen)

$$= \arg \min_{\mathbf{d}} -2 \text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) + \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) \quad (2.78)$$

(da wir die Reihenfolge der Matrizen in einer Spur zyklisch umstellen können, Gleichung 2.52)

$$= \arg \min_d -2 \operatorname{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) + \operatorname{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top d d^\top) \quad (2.79)$$

(durch erneutes Anwenden derselben Eigenschaft).

Jetzt führen wir die Bedingung ein:

$$\arg \min_d -2 \operatorname{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) + \operatorname{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top d d^\top) \text{ für } d^\top d = 1 \quad (2.80)$$

$$= \arg \min_d -2 \operatorname{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) + \operatorname{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) \text{ für } d^\top d = 1 \quad (2.81)$$

(aufgrund der Bedingung)

$$= \arg \min_d -\operatorname{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) \text{ für } d^\top d = 1 \quad (2.82)$$

$$= \arg \max_d \operatorname{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) \text{ für } d^\top d = 1 \quad (2.83)$$

$$= \arg \max_d \operatorname{Tr}(d^\top \mathbf{X}^\top \mathbf{X} d) \text{ für } d^\top d = 1. \quad (2.84)$$

Dieses Optimierungsproblem kann durch Eigenwertzerlegung gelöst werden. Der optimale Vektor \mathbf{d} ist durch den Eigenvektor von $\mathbf{X}^\top \mathbf{X}$, der dem größten Eigenwert entspricht, gegeben.

Diese Herleitung gilt nur für den Fall $l = 1$ und stellt nur die erste Hauptkomponente wieder her. Wenn wir allgemeiner eine Basis der Hauptkomponenten wiederherstellen möchten, ergibt sich die Matrix \mathbf{D} aus den l Eigenvektoren, die den größten Eigenwerten entsprechen. Hierzu können wir einen Induktionsbeweis nutzen. Es empfiehlt sich, den Beweis zu Übungszwecken zu schreiben.

Die lineare Algebra gehört zu den fundamentalen mathematischen Disziplinen, die für ein Verständnis des Deep Learnings erforderlich sind. Ein weiteres mathematisches Kerngebiet, das sich durch alle Aspekte des Machine Learnings zieht, ist die im nächsten Kapitel vorgestellte Wahrscheinlichkeitstheorie.

3

Wahrscheinlichkeits- und Informationstheorie

In diesem Kapitel geht es um Wahrscheinlichkeitstheorie und Informationstheorie.

Die Wahrscheinlichkeitstheorie ist ein mathematischer Rahmen zur Darstellung unsicherer Aussagen. Sie ermöglicht es, die Unsicherheit zu quantifizieren, und hält Axiome zur Ableitung neuer unsicherer Aussagen bereit. In KI-Anwendungen gibt es zwei Hauptanwendungsfälle für die Wahrscheinlichkeitstheorie: Erstens geben die Gesetze der Wahrscheinlichkeit an, wie ein KI-System Schlüsse ziehen sollte; daher konzipieren wir unsere Algorithmen zum Berechnen oder Approximieren diverser Ausdrücke, die auf der Basis der Wahrscheinlichkeitstheorie ermittelt wurden. Zweitens können wir eine Wahrscheinlichkeit und Statistik nutzen, um das Verhalten geplanter KI-Systeme in der Theorie zu analysieren.

Die Wahrscheinlichkeitstheorie ist ein wesentliches Tool für viele wissenschaftliche und technische Disziplinen. Dieses Kapitel soll Lesern, die aus der Softwareentwicklung kommen und wenig Erfahrung mit der Wahrscheinlichkeitstheorie haben, für ein besseres Verständnis des Buchs mit der Materie vertraut machen.

Wo die Wahrscheinlichkeitstheorie es ermöglicht, unsichere Aussagen zu treffen und im Angesicht der Unsicherheit Schlüsse zu ziehen, versetzt uns die Informationstheorie in die Lage, das Maß der Unsicherheit in einer Wahrscheinlichkeitsverteilung zu quantifizieren.

Wenn Sie bereits mit Wahrscheinlichkeits- und Informationstheorie vertraut sind, können Sie dieses Kapitel bis auf Abschnitt 3.14 überspringen.

Jener Abschnitt stellt die Graphen vor, mit denen wir strukturierte probabilistische Modelle für das Machine Learning beschreiben. Wenn Sie sich bisher überhaupt nicht mit diesen Themen befasst haben, sollte dieses Kapitel Sie in die Lage versetzen, Deep-Learning-Forschungsprojekte erfolgreich durchzuführen. Allerdings empfehlen wir das Zurateziehen weiterer Ressourcen, beispielsweise *Jaynes* (2003).

3.1 Warum Wahrscheinlichkeit?

Viele Zweige der Informatik beschäftigen sich in erster Linie mit Größen oder Elementen, die vollständig deterministisch und sicher sind. Ein Programmierer kann normalerweise uneingeschränkt davon ausgehen, dass eine CPU jede Maschinencode-Anweisung fehlerfrei ausführt. Zwar gibt es durchaus Hardwaredefekte, doch treten diese selten genug auf, sodass sie in den meisten Softwareanwendungen vernachlässigt werden können. Bedenkt man, dass viele Informatiker und Softwareentwickler in relativ aufgeräumten und sicheren Umgebungen arbeiten, überrascht es vielleicht, dass die Wahrscheinlichkeitstheorie im Machine Learning weit verbreitet ist.

Beim Machine Learning geht es stets um unsichere Größen und manchmal auch um stochastische (nichtdeterministische) Größen. Unsicherheit und Stochastizität können aus vielen Quellen stammen. Forscher haben schon seit den 1980er-Jahren stichhaltige Argumente für das Quantifizieren von Unsicherheit mithilfe der Wahrscheinlichkeit vorgebracht. Viele der hier dargelegten Argumente wurden *Pearl* (1988) entnommen oder davon inspiriert.

Nahezu alle Aktivitäten erfordern eine gewisse Fähigkeit, auch angesichts der Unsicherheit Schlüsse zu ziehen. Tatsächlich ist es abseits mathematischer Aussagen, die per definitionem wahr sind, schwierig, eine Behauptung aufzustellen, die absolut wahr ist, oder ein Ereignis zu erdenken, das unter allen Umständen garantiert eintreten wird.

Die Unsicherheit entstammt drei möglichen Quellen:

1. Inhärente Stochastizität des modellierten Systems. Zum Beispiel beschreiben die meisten Interpretationen der Quantenmechanik die Dynamik subatomarer Teilchen als probabilistisch. Wir können auch theoretische Szenarios mit postulierter zufälliger Dynamik erstellen, beispielsweise ein hypothetisches Kartenspiel, bei dem wir davon ausgehen, dass die Spielkarten wirklich zufällig gemischt wurden.

2. Unvollständige Beobachtbarkeit. Sogar deterministische Systeme können stochastisch erscheinen, wenn wir nicht sämtliche Variablen, die das Systemverhalten steuern, beobachten können. So wird zum Beispiel beim Ziegenproblem¹ ein Teilnehmer einer Gameshow aufgefordert, sich für eine von drei Türen zu entscheiden, hinter denen sich jeweils ein Preis versteckt. Hinter zwei Türen steht eine Ziege, hinter der dritten ein Auto. Das Ergebnis der Entscheidung für eine Tür ist deterministisch, aus der Perspektive des Spielers jedoch ungewiss.
3. Unvollständige Modellierung. Wenn wir ein Modell verwenden, das einige der von uns beobachteten Daten außer Acht lässt, führen diese verworfenen Daten zu einer Unsicherheit in den Vorhersagen des Modells. Ein Beispiel: Angenommen, wir bauen einen Roboter, der die Position jedes Objekts in seinem Umfeld exakt beobachten kann. Wenn der Roboter den Raum beim Vorhersagen der künftigen Positionen dieser Objekte diskretisiert, entsteht durch die Diskretisierung sofort eine Unsicherheit aufseiten des Roboters bezüglich der exakten Position der Objekte: Jedes einzelne Objekt könnte sich irgendwo innerhalb der diskreten Zelle befinden, in der es beobachtet wurde.

In vielen Fällen ist es praktikabler, eine einfache, aber unsichere Regel anstelle einer komplexen, aber sichereren zu verwenden. Das gilt sogar dann, wenn die Regel deterministisch ist und unser Modellierungssystem genügend Vertrauen für die Einbeziehung einer komplexen Regel bietet. So lässt sich die einfache Regel »Die meisten Vögel fliegen« ohne großen Aufwand entwickeln, wohingegen eine Regel der Art »Vögel fliegen, mit Ausnahme sehr junger Vögel, die das Fliegen noch nicht gelernt haben, kranker oder verletzter Vögel, die nicht mehr fliegen können, flugunfähiger Vogelarten wie Kasuar, Strauß und Kiwi ... « hohen Aufwand bei Entwicklung, Pflege und Kommunikation mit sich bringt. Außerdem ist sie noch immer unausgegoren und anfällig für Fehler.

Es sollte zwar klar sein, dass wir ein Mittel zum Darstellen von und Schlüsseziehen über Unsicherheit benötigen, aber es ist nicht direkt offensichtlich, dass die Wahrscheinlichkeitstheorie sämtliche Hilfsmittel bereithält, die wir für KI-Anwendungen benötigen. Ursprünglich wurde die Wahrscheinlichkeitstheorie zur Untersuchung von Ereignishäufigkeiten entwickelt. Es ist offenkundig, dass damit Ereignisse wie das Ziehen einer bestimmten

¹ Auch als *Monty-Hall-Problem* bekannt. Der Name bezieht sich auf die von Monty Hall moderierte Gameshow *Let's Make a Deal*, der die deutsche Sendung *Geh aufs Ganze!* nachempfunden wurde.

Kartenhand in einem Pokerspiel untersucht werden können. Derartige Ereignisse sind häufig wiederholbar. Wenn wir sagen, dass ein Ergebnis eine Wahrscheinlichkeit von p hat, bedeutet dies, dass bei einer unendlichen Anzahl von Wiederholungen des Experiments (z. B. das Ziehen von Karten) das Verhältnis p der Wiederholungen zu diesem Ergebnis führt. Diese Art des Schlüsseziehens hat auf den ersten Blick keinen Nutzen für nicht wiederholbare Behauptungen. Untersucht ein Arzt einen Patienten und sagt, es bestehe eine 40-prozentige Wahrscheinlichkeit, dass dieser die Grippe habe, steckt eine ganz andere Bedeutung dahinter – wir können nämlich keine unendliche Anzahl Kopien des Patienten anfertigen oder begründet glauben, dass bei solchen Kopien dieselben Symptome auf unterschiedliche Verfassungen zurückgehen. Bei der ärztlichen Diagnose steht die Wahrscheinlichkeit für einen **Grad der Überzeugung** (engl. *degree of belief*), wobei 1 für eine absolute Sicherheit steht, dass der Patient die Grippe hat, während 0 die absolute Sicherheit angibt, dass der Patient keine Grippe hat. Der erstgenannte Wahrscheinlichkeitsbegriff, der sich direkt auf die Häufigkeit bezieht, mit der ein Ereignis auftritt, wird **frequentistischer Wahrscheinlichkeitsbegriff** (oder *objektive Wahrscheinlichkeit*) genannt, der letztere, der eine qualitative Sicherheit bezeichnet, dagegen **bayesscher Wahrscheinlichkeitsbegriff**.

Wenn wir mehrere Eigenschaften auflisten, die nach gesundem Menschenverstand beim Schlußziehen über Unsicherheit unserer Meinung nach vorhanden sein sollten, dann können diese Eigenschaften nur erfüllt werden, wenn wir bayessche Wahrscheinlichkeiten so behandeln, als würden sie sich exakt wie frequentistische Wahrscheinlichkeiten verhalten. Ein Beispiel: Zum Berechnen der Wahrscheinlichkeit, mit der eine Spielerin eine Partie Poker mit einem bestimmten Blatt gewinnt, verwenden wir exakt dieselben Formeln wie zum Berechnen der Wahrscheinlichkeit, dass eine Patientin mit bestimmten Symptomen eine Krankheit hat. Einzelheiten darüber, warum eine kleine Menge von Annahmen zum gesunden Menschenverstand impliziert, dass dieselben Axiome beide Arten von Wahrscheinlichkeit steuern, finden sich in *Ramsey* (1926).

Wahrscheinlichkeit lässt sich als Erweiterung der Logik für den Umgang mit Unsicherheit betrachten. Die Logik bietet eine Reihe von formalen Regeln, mit denen sich bestimmen lässt, welche Annahmen als wahr oder falsch angesehen werden können, wenn vorausgesetzt wird, dass eine weitere Reihe von Annahmen wahr oder falsch ist. Die Wahrscheinlichkeitstheorie bietet eine Reihe formaler Regeln, mit denen sich die Likelihood bestimmen lässt, dass eine Behauptung abhängig von der Likelihood weiterer Annahmen wahr ist.

3.2 Zufallsvariablen

Eine **Zufallsvariable** ist eine Variable, die zufällig unterschiedliche Werte annehmen kann. Im Normalfall sind Zufallsvariablen in diesem Buch als Kleinbuchstaben in der Basisschrift dargestellt und die möglichen Werte in kursiver Schrift: x_1 und x_2 sind zum Beispiel zwei mögliche Werte für die Zufallsvariable x . Bei vektorwertigen Variablen schreiben wir die Zufallsvariable als \mathbf{x} und einen ihrer Werte als x . Eine Zufallsvariable für sich ist lediglich eine Beschreibung der möglichen Zustände; sie muss mit einer Wahrscheinlichkeitsverteilung kombiniert werden, die angibt, wie wahrscheinlich es ist, dass diese Zustände eintreten.

Zufallsvariablen können diskret oder stetig sein. Eine diskrete Zufallsvariable kann eine endliche oder abzählbar unendliche Zahl von Zuständen annehmen. Bei diesen Zuständen muss es sich nicht zwingend um ganze Zahlen handeln; es können ebenso einfach nur benannte Zustände sein, mit denen kein numerischer Wert verknüpft ist. Eine stetige Zufallsvariable ist mit einem reellen Wert verknüpft.

3.3 Wahrscheinlichkeitsverteilungen

Eine **Wahrscheinlichkeitsverteilung** beschreibt, wie wahrscheinlich es ist, dass eine Zufallsvariable oder eine Menge von Zufallsvariablen jeden ihrer möglichen Zustände annimmt. Unsere Beschreibung von Wahrscheinlichkeitsverteilungen richtet sich danach, ob es sich um diskrete oder stetige Variablen handelt.

3.3.1 Diskrete Variablen und Wahrscheinlichkeitsfunktionen

Eine Wahrscheinlichkeitsverteilung über diskrete Variablen kann mithilfe einer **Wahrscheinlichkeitsfunktion** (engl. *probability mass function*, PMF) beschrieben werden. Normalerweise kennzeichnen wir Wahrscheinlichkeitsfunktionen mit dem Großbuchstaben P (engl. *probability*). Häufig verknüpfen wir jede Zufallsvariable mit einer anderen Wahrscheinlichkeitsfunktion; Sie müssen dann anhand der Identität der Zufallsvariable darauf schließen, welche PMF zu verwenden ist (nicht anhand des Namens der Funktion): $P(x)$ ist normalerweise nicht gleichbedeutend mit $P(y)$.

Die Wahrscheinlichkeitsfunktion ordnet einen Zustand einer Zufallsvariable der Wahrscheinlichkeit, dass Zufallsvariablen diesen Zustand annehmen,

zu. Die Wahrscheinlichkeit, dass $x = x$ ist, wird $P(x)$ geschrieben; dabei steht eine Wahrscheinlichkeit von 1 dafür, dass $x = x$ sicher ist, eine Wahrscheinlichkeit von 0 dagegen, dass $x = x$ unmöglich ist. Manchmal schreiben wir zur Angabe der benötigten PMF den Namen der Zufallsvariable aus: $P(x = x)$. Manchmal definieren wir zunächst eine Variable und verwenden anschließend die Notation \sim , um anzugeben, welcher Verteilung sie später folgt: $x \sim P(x)$.

Wahrscheinlichkeitsfunktionen können gleichzeitig auf viele Variablen angewandt werden. Eine solche Wahrscheinlichkeitsverteilung über viele Variablen wird als **multivariate Verteilung** bezeichnet. $P(x = x, y = y)$ benennt die Wahrscheinlichkeit, dass $x = x$ und $y = y$ gleichzeitig zutreffen. Wir können auch kurz $P(x, y)$ schreiben.

Damit es sich um eine PMF für eine Zufallsvariable x handelt, muss eine Funktion P die folgenden Eigenschaften erfüllen:

- Der Definitionsbereich von P muss die Menge aller möglichen Zustände von x sein.
- $\forall x \in X, 0 \leq P(x) \leq 1$. Ein unmögliches Ereignis hat eine Wahrscheinlichkeit von 0; kein Zustand könnte weniger wahrscheinlich sein. Ebenso hat ein garantiert eintretendes Ereignis eine Wahrscheinlichkeit von 1; kein Zustand hat eine höhere Eintrittschance.
- $\sum_{x \in X} P(x) = 1$. Wir bezeichnen diese Eigenschaft als **normalisiert**. Ohne diese Eigenschaft könnten wir Wahrscheinlichkeit größer 1 erhalten, indem wir die Wahrscheinlichkeit für eines von vielen auftretenden Ereignissen berechnen.

Gegeben sei eine einzelne diskrete Zufallsvariable x mit k unterschiedlichen Zuständen. Wir können eine **Gleichverteilung** für x verwenden – also jeden ihrer Zustände gleichermaßen wahrscheinlich machen –, indem wir ihre PMF auf

$$P(x = x_i) = \frac{1}{k} \quad (3.1)$$

für alle i setzen. Es ist offensichtlich, dass damit die Voraussetzungen für eine Wahrscheinlichkeitsfunktion erfüllt sind. Der Wert $\frac{1}{k}$ ist positiv, da k eine positive ganze Zahl ist. Wir sehen auch, dass

$$\sum_i P(x = x_i) = \sum_i \frac{1}{k} = \frac{k}{k} = 1 \quad (3.2)$$

ist, sodass die Verteilung korrekt normalisiert ist.

3.3.2 Stetige Variablen und Wahrscheinlichkeitsdichtefunktionen

Wenn wir mit stetigen Zufallsvariablen arbeiten, beschreiben wir Wahrscheinlichkeitsverteilungen anhand einer **Wahrscheinlichkeitsdichtefunktion** (engl. *probability density function*, PDF) anstelle einer Wahrscheinlichkeitsfunktion. Damit es sich um eine Wahrscheinlichkeitsdichtefunktion handelt, muss eine Funktion p die folgenden Eigenschaften erfüllen:

- Der Definitionsbereich von p muss die Menge aller möglichen Zustände von x sein.
- $\forall x \in \mathbb{X}, p(x) \geq 0$. Beachten Sie, dass $p(x) \leq 1$ nicht gegeben sein muss.
- $\int p(x)dx = 1$.

Eine Wahrscheinlichkeitsdichtefunktion $p(x)$ gibt nicht die Wahrscheinlichkeit eines bestimmten Zustands direkt an; stattdessen ergibt sich die Wahrscheinlichkeit, mit der das Ergebnis innerhalb eines infinitesimalen Bereichs mit dem Volumen δx liegt, aus $p(x)\delta x$.

Wir können die Dichtefunktion integrieren, um die tatsächliche Wahrscheinlichkeit(smasse) einer Menge von Datenpunkten zu bestimmen. Insbesondere wird die Wahrscheinlichkeit, mit der x in einer beliebigen Menge \mathbb{S} liegt, durch das Integral von $p(x)$ über diese Menge angegeben. Im univariaten Fall ergibt sich die Wahrscheinlichkeit, mit der x im Intervall $[a, b]$ liegt, aus $\int_{[a,b]} p(x)dx$.

Ein Beispiel für eine PDF, die einer bestimmten Wahrscheinlichkeitsdichte über eine stetige Zufallsvariable entspricht, können Sie sich eine Gleichverteilung über ein Intervall der reellen Zahlen vorstellen. Dazu setzen wir eine Funktion $u(x; a, b)$ ein, bei der a und b die Endpunkte des Intervalls sind und für die gilt $b > a$. Die Notation »; \cdot ;« bedeutet »parametrisiert durch«; wir betrachten x als das Argument der Funktion und a sowie b als Parameter, die diese Funktion definieren. Damit es keine Wahrscheinlichkeit(smasse) außerhalb des Intervalls gibt, legen wir fest $u(x; a, b) = 0$ für alle $x \notin [a, b]$, in $[a, b], u(x; a, b) = \frac{1}{b-a}$. Daraus folgt, dass die Funktion überall nichtnegativ ist. Zusätzlich ist das Integral gleich 1. Um eine Gleichverteilung von x auf $[a, b]$ anzusehen, schreiben wir häufig $x \sim U(a, b)$.

3.4 Randwahrscheinlichkeit

Manchmal kennen wir die Wahrscheinlichkeitsverteilung über eine Menge von Variablen und möchten die Wahrscheinlichkeitsverteilung für eine Teilmenge davon ermitteln. Die Wahrscheinlichkeitsverteilung für die Teilmenge wird als **Randwahrscheinlichkeitsverteilung** bezeichnet.

Ein Beispiel: Gegeben seien die diskreten Zufallsvariablen x und y . Angenommen, wir kennen $P(x, y)$, dann können wir $P(x)$ mit der **Summenregel** bestimmen:

$$\forall x \in X, P(x = x) = \sum_y P(x = x, y = y). \quad (3.3)$$

Die Bezeichnung »Randwahrscheinlichkeit« stammt aus der Zeit, als Randwahrscheinlichkeiten mit Stift und Papier berechnet wurden. Wenn die Werte von $P(x, y)$ in einem Raster notiert werden (Einzelwerte für x in Zeilen und Einzelwerte für y in Spalten), ist es recht natürlich, die Zeilensumme des Rasters zu berechnen und anschließend $P(x)$ an den Rand des Blatts direkt rechts neben die Zeile zu schreiben.

Für stetige Variablen müssen wir anstelle der Aufsummierung die Integration verwenden:

$$p(x) = \int p(x, y) dy. \quad (3.4)$$

3.5 Bedingte Wahrscheinlichkeit

In vielen Fällen interessiert uns die Wahrscheinlichkeit eines Ereignisses nur dann, wenn zuvor ein anderes Ereignis eingetreten ist. Dies ist die **bedingte Wahrscheinlichkeit**. Wir notieren für die bedingte Wahrscheinlichkeit, dass $y = y$ mit $x = x$ zutrifft, $P(y = y | x = x)$. Diese bedingte Wahrscheinlichkeit lässt sich mit folgender Formel berechnen:

$$P(y = y | x = x) = \frac{P(y = y, x = x)}{P(x = x)}. \quad (3.5)$$

Die bedingte Wahrscheinlichkeit ist nur für $P(x = x) > 0$ definiert. Wir können die bedingte Wahrscheinlichkeit nicht für den Fall, dass ein Ereignis niemals eintrifft, berechnen.

Es ist wichtig, dass Sie die bedingte Wahrscheinlichkeit nicht verwechseln mit der Berechnung dessen, was eintritt, wenn eine bestimmte Handlung durchgeführt wird. Die bedingte Wahrscheinlichkeit, dass eine Person, die Deutsch spricht, auch aus Deutschland stammt, ist relativ groß. Aber wenn

eine zufällig ausgewählte Person Deutschunterricht erhält, ändert sich dadurch nicht ihr Herkunftsland. Das Berechnen der Folgen einer Handlung betrifft den Bereich der **Intervention**. Dieser gehört zur Domäne der **kau-salen Modellierung**, die nicht Thema dieses Buchs ist.

3.6 Die Produktregel der bedingten Wahrscheinlichkeiten

Jede multivariate Verteilung über viele Zufallsvariablen kann in bedingte Verteilungen über nur eine Variable zerlegt werden:

$$P(x^{(1)}, \dots, x^{(n)}) = P(x^{(1)}) \prod_{i=2}^n P(x^{(i)} | x^{(1)}, \dots, x^{(i-1)}). \quad (3.6)$$

Diese Feststellung ist als **Produktregel** der Wahrscheinlichkeiten bekannt. Sie folgt unmittelbar aus der Definition der bedingten Wahrscheinlichkeit in Gleichung 3.5. Wenn wir die Definition zum Beispiel zweimal anwenden, erhalten wir

$$\begin{aligned} P(a, b, c) &= P(a | b, c)P(b, c) \\ P(b, c) &= P(b | c)P(c) \\ P(a, b, c) &= P(a | b, c)P(b | c)P(c). \end{aligned}$$

3.7 Unabhängigkeit und bedingte Unabhängigkeit

Zwei Zufallsvariablen x und y sind **unabhängig**, wenn ihre Wahrscheinlichkeitsverteilung als Produkt aus zwei Faktoren ausgedrückt werden kann, von denen einer nur x einbezieht und der andere nur y :

$$\forall x \in X, y \in Y, p(x = x, y = y) = p(x = x)p(y = y). \quad (3.7)$$

Zwei Zufallsvariablen x und y sind **bedingt unabhängig** für eine Zufallsvariable z , wenn die bedingte Wahrscheinlichkeitsverteilung über x und y dies für jeden Wert von z faktorisiert:

$$\forall x \in X, y \in Y, z \in Z, p(x = x, y = y | z = z) = p(x = x | z = z)p(y = y | z = z). \quad (3.8)$$

Wir können Unabhängigkeit und bedingte Unabhängigkeit kompakt notieren: $x \perp y$ bedeutet, dass x und y unabhängig sind; $x \perp y | z$ bedeutet, dass x und y bedingt unabhängig sind, sofern z gegeben ist.

3.8 Erwartungswert, Varianz und Kovarianz

Der **Erwartungswert** oder **erwartete Wert** einer Funktion $f(x)$ bezüglich einer Wahrscheinlichkeitsverteilung $P(x)$ ist der Durchschnittswert oder Mittelwert, den f annimmt, wenn x aus P abgeleitet wird. Für diskrete Variablen lässt sich dafür die folgende Aufsummierung nutzen:

$$\mathbb{E}_{x \sim P}[f(x)] = \sum_x P(x)f(x), \quad (3.9)$$

für stetige Variablen ein Integral:

$$\mathbb{E}_{x \sim p}[f(x)] = \int p(x)f(x)dx. \quad (3.10)$$

Wenn die Identität der Verteilung aus dem Kontext offensichtlich ist, können wir auch nur den Namen der Zufallsvariablen, für die der Erwartungswert gilt, notieren, zum Beispiel $\mathbb{E}_x[f(x)]$. Wenn offensichtlich ist, für welche Zufallsvariable der Erwartungswert gilt, können wir den Index vollständig weglassen, zum Beispiel $\mathbb{E}[f(x)]$. Generell können wir davon ausgehen, dass $\mathbb{E}[\cdot]$ den Durchschnittswert aller Zufallsvariablenwerte in den Klammern ergibt. Ebenso können wir, wenn keine Unklarheiten bestehen, die eckigen Klammern weglassen.

Erwartungswerte sind linear, beispielsweise

$$\mathbb{E}_x[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}_x[f(x)] + \beta \mathbb{E}_x[g(x)], \quad (3.11)$$

wenn α und β nicht von x abhängig sind.

Die **Varianz** ist ein Maß für die Streuung der Werte einer Funktion einer Zufallsvariablen x , wenn wir Stichproben (engl. *samples*) unterschiedlicher Werte von x aus deren Wahrscheinlichkeitsverteilung ziehen:

$$\text{Var}(f(x)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]. \quad (3.12)$$

Bei einer niedrigen Varianz liegen die Werte für $f(x)$ nah am Erwartungswert. Die Quadratwurzel der Varianz wird als **Standardabweichung** bezeichnet.

Die **Kovarianz** gibt einen Anhaltspunkt, wie sehr zwei Werte linear zusammenhängen und welches Größenverhältnis diese Variablen haben:

$$\text{Cov}(f(x), g(y)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])]. \quad (3.13)$$

Hohe Absolutbeträge der Kovarianz zeigen an, dass die Werte sich stark ändern und beide zugleich weit vom jeweiligen Mittelwert entfernt sind. Ein

positives Vorzeichen der Kovarianz gibt an, dass beide Variablen dazu neigen, gleichzeitig relativ hohe Werte anzunehmen. Ein negatives Vorzeichen bedeutet dagegen, dass eine der Variablen zu einem relativ hohen Wert tendiert, während die andere Variable zu einem relativ niedrigen Wert tendiert und umgekehrt. Andere Maße wie **Korrelation** normalisierte die Verteilung der Variablen, um lediglich zu bestimmen, wie stark der Zusammenhang zwischen den Variablen ist, ohne durch das Größenverhältnis der einzelnen Variablen beeinflusst zu werden.

Kovarianz und Abhängigkeit sind zwar verwandt, aber dennoch klar getrennte Konzepte. Verwandt deshalb, weil zwei unabhängige Variablen eine Kovarianz von Null aufweisen, während zwei Variablen, die eine von Null verschiedene Kovarianz aufweisen, abhängig sind. Unabhängigkeit ist jedoch eine Eigenschaft, die nichts mit der Kovarianz zu tun hat. Damit zwei Variablen eine Kovarianz von Null aufweisen, darf keine lineare Abhängigkeit zwischen ihnen bestehen. Unabhängigkeit ist eine stärkere Voraussetzung als eine Kovarianz von Null, da die Unabhängigkeit zugleich nichtlineare Beziehungen ausschließt. Es ist möglich, dass zwei Variablen abhängig sind und eine Kovarianz von Null aufweisen. Nehmen wir zum Beispiel an, dass wir als Stichprobe eine reelle Zahl x aus einer Gleichverteilung über das Intervall $[-1, 1]$ ziehen. Anschließend ziehen wir als Stichprobe eine Zufallsvariable s . Mit der Wahrscheinlichkeit $\frac{1}{2}$ wählen wir für s den Wert 1. Andernfalls wählen wir für s den Wert -1. Wir können nun durch Zuweisung $y = sx$ eine Zufallsvariable y erzeugen. Es ist offenkundig, dass x und y nicht unabhängig sind, da x die Größe von y vollständig bestimmt. Allerdings ist $\text{Cov}(x, y) = 0$.

Die **Kovarianzmatrix** eines Zufallsvektors $\mathbf{x} \in \mathbb{R}^n$ ist eine $n \times n$ -Matrix, für die gilt

$$\text{Cov}(\mathbf{x})_{i,j} = \text{Cov}(\mathbf{x}_i, \mathbf{x}_j). \quad (3.14)$$

Die Diagonalelemente der Kovarianz ergeben die Varianz:

$$\text{Cov}(\mathbf{x}_i, \mathbf{x}_i) = \text{Var}(\mathbf{x}_i). \quad (3.15)$$

3.9 Häufig genutzte Wahrscheinlichkeitsverteilungen

Es gibt einige einfache Wahrscheinlichkeitsverteilungen, die im Machine Learning vielfach eingesetzt werden.

3.9.1 Bernoulli-Verteilung

Die **Bernoulli-Verteilung** ist eine Verteilung über eine einzelne binäre Zufallsvariable. Sie wird mit dem einzelnen Parameter $\phi \in [0, 1]$ gesteuert, der die Wahrscheinlichkeit dafür angibt, dass die Zufallsvariable gleich 1 ist. Sie hat die folgenden Eigenschaften:

$$P(x = 1) = \phi \quad (3.16)$$

$$P(x = 0) = 1 - \phi \quad (3.17)$$

$$P(x = x) = \phi^x(1 - \phi)^{1-x} \quad (3.18)$$

$$\mathbb{E}_x[x] = \phi \quad (3.19)$$

$$\text{Var}_x(x) = \phi(1 - \phi) \quad (3.20)$$

3.9.2 Multinoulli-Verteilung

Die **Multinoulli-Verteilung** oder **kategoriale Verteilung** ist eine Verteilung über eine einzelne diskrete Variable mit k unterschiedlichen Zuständen, wobei k endlich ist.² Die Multinoulli-Verteilung wird anhand eines Vektors $\mathbf{p} \in [0, 1]^{k-1}$ parametrisiert, wobei p_i die Wahrscheinlichkeit des i -ten Zustands angibt. Die Wahrscheinlichkeit des endgültigen k -ten Zustands ergibt sich aus $1 - \mathbf{1}^\top \mathbf{p}$. Beachten Sie, dass die Bedingung $\mathbf{1}^\top \mathbf{p} \leq 1$ erfüllt sein muss. Multinoulli-Verteilungen werden häufig auf Verteilungen über Objektkategorien bezogen, weswegen wir generell nicht davon ausgehen, dass Zustand 1 dem Zahlenwert 1 entspricht usw. Daher müssen wir den Erwartungswert oder die Varianz der Multinoulli-verteilten Zufallsvariablen normalerweise nicht berechnen.

Die Bernoulli- und Multinoulli-Verteilungen reichen zur Beschreibung beliebiger Verteilungen über ihren Definitionsbereich aus. Damit lassen sich sämtliche Verteilungen über ihre Definitionsbereiche allerdings nicht in erster Linie beschreiben, weil sie besonders leistungsstark sind, sondern weil ihr Definitionsbereich einfach ist – sie modellieren diskrete Variablen, für die sämtliche Zustände aufgezählt werden können. Beim Arbeiten mit stetigen Variablen gibt es überabzählbar viele Zustände, sodass jede Verteilung,

² »Multinoulli« ist ein kürzlich von Gustavo Lacerdo geprägter Begriff, der bekannt wurde durch Murphy (2012). Die Multinoulli-Verteilung ist ein Sonderfall der **Multinomialverteilung**. Eine Multinomialverteilung ist die Verteilung über die Vektoren in $\{0, \dots, n\}^k$ zur Darstellung der Anzahl, mit der jede der k Kategorien bei n Stichproben aus einer Multinoulli-Verteilung besucht wird. Viele Texte verwenden den Begriff »multinomial« für Multinoulli-Verteilungen, ohne klarzustellen, dass es dabei nur um den Fall $n = 1$ geht.

die durch eine geringe Anzahl von Parametern beschrieben wird, strikte Grenzen für die Verteilung setzen muss.

3.9.3 Normalverteilung

Die am häufigsten für reelle Zahlen verwendete Verteilung ist die **Normalverteilung**, auch bekannt als **Gauß-Verteilung**:

$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right). \quad (3.21)$$

Abbildung 3.1 zeigt die Dichtefunktion der Normalverteilung.

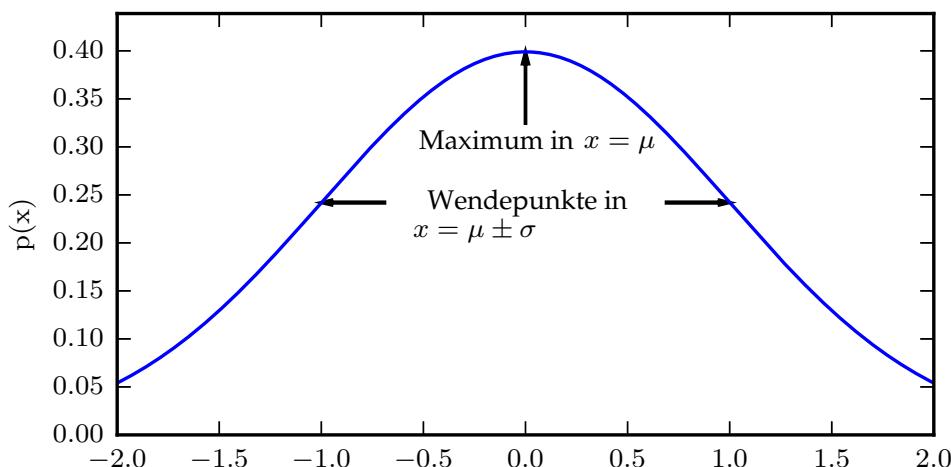


Abbildung 3.1: Die Normalverteilung $\mathcal{N}(x; \mu, \sigma^2)$ zeigt eine klassische »Glockenkurve« mit der x -Koordinate des zentralen höchsten Punkts aus μ und der Breite der Kuppe aus σ . In diesem Beispiel ist die **Standardnormalverteilung** für $\mu = 0$ und $\sigma = 1$ abgebildet.

Die beiden Parameter $\mu \in \mathbb{R}$ und $\sigma \in (0, \infty)$ kontrollieren die Normalverteilung. Der Parameter μ gibt die Koordinate des zentralen Maximalwerts an. Dabei handelt es sich auch um den Mittelwert der Verteilung: $\mathbb{E}[x] = \mu$. Die Standardabweichung der Verteilung ergibt sich aus σ , die Varianz ergibt sich aus σ^2 .

Wenn wir die PDF auswerten, müssen wir σ quadrieren und invertieren. Ist eine häufige Auswertung der PDF mit unterschiedlichen Parameterwerten erforderlich, ist es effizienter, die Verteilung mittels eines Parameters $\beta \in (0, \infty)$ zu parametrisieren, der die **Präzision** (oder die Inverse der Varianz) der Verteilung steuert:

$$\mathcal{N}(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x - \mu)^2\right). \quad (3.22)$$

Normalverteilungen sind eine geeignete Wahl für viele Anwendungen. Wenn noch keine Kenntnis darüber besteht, welche Form eine Verteilung über die reellen Zahlen annehmen soll, ist die Normalverteilung aus zwei Gründen ein guter Ausgangspunkt:

Erstens sind viele der Verteilungen, die wir modellieren möchten, recht nah an der Normalverteilung. Der **zentrale Grenzwertsatz** zeigt, dass die Summe vieler unabhängiger Zufallsvariablen näherungsweise normalverteilt ist. In der Praxis bedeutet dies, dass viele komplizierte Systeme auch dann erfolgreich als normalverteiltes Rauschen modelliert werden können, wenn die Systeme in Teile mit einem strukturierteren Verhalten zerlegt werden können.

Zweitens codiert die Normalverteilung von allen möglichen Wahrscheinlichkeitsverteilungen mit derselben Varianz die maximale Menge an Unsicherheit über die reellen Zahlen. Sie können sich die Normalverteilung daher als die Verteilung vorstellen, die das geringste Vorwissen in ein Modell einbringt. Die vollständige Ausarbeitung und Begründung dieses Konzepts erfordert zusätzliche mathematische Werkzeuge und wird erst in Abschnitt 19.4.2 behandelt.

Die Normalverteilung generalisiert³ sich zu \mathbb{R}^n und wird in diesem Fall als mehrdimensionale oder **multivariate Normalverteilung** bezeichnet. Sie lässt sich mit einer positiv definiten symmetrischen Matrix Σ parametrisieren:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \sqrt{\frac{1}{(2\pi)^n \det(\Sigma)}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right). \quad (3.23)$$

Der Parameter $\boldsymbol{\mu}$ gibt nach wie vor den Mittelwert der Verteilung an, ist aber nun vektorwertig. Der Parameter Σ gibt die Kovarianzmatrix der Verteilung an. Wie im univariaten (eindimensionalen) Fall ist die Kovarianz zur mehrfachen Auswertung der PDF für viele unterschiedliche Parameterwerte rechnerisch keine effiziente Methode zum Parametrisieren der Verteilung, da wir Σ zum Auswerten der PDF invertieren müssen. Wir können allerdings eine **Präzisionsmatrix** β nutzen:

³ Anm. des Übersetzers: Im mathematischen Kontext wird auch von »verallgemeinern« gesprochen. Da »generalisieren« im Deep-Learning-Kontext jedoch gängiger ist, haben wir uns der Einheitlichkeit halber durchgängig für diese Formulierung entschieden.

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\beta}^{-1}) = \sqrt{\frac{\det(\boldsymbol{\beta})}{(2\pi)^n}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\beta}(\mathbf{x} - \boldsymbol{\mu})\right). \quad (3.24)$$

Häufig formen wir die Kovarianzmatrix zu einer Diagonalmatrix um. Eine noch einfachere Version ist die **isotrope** Normalverteilung, deren Kovarianzmatrix ein skalares Vielfaches der Einheitsmatrix ist.

3.9.4 Exponential- und Laplace-Verteilung

Im Deep-Learning-Kontext kommen uns Wahrscheinlichkeitsverteilungen mit einem scharfenen Punkt bei $x = 0$ oft gelegen. Um dieses Ziel zu erreichen, können wir die **Exponentialverteilung** einsetzen:

$$p(x; \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x). \quad (3.25)$$

Die Exponentialverteilung verwendet die Indikatorfunktion $\mathbf{1}_{x \geq 0}$, um allen negativen Werten von x eine Wahrscheinlichkeit von 0 zuzuweisen.

Eine sehr ähnliche Wahrscheinlichkeitsverteilung zum Erzeugen eines spitzen Wahrscheinlichkeitsscheitels in einem beliebigen Punkt μ ist die **Laplace-Verteilung**

$$\text{Laplace}(x; \mu, \gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x - \mu|}{\gamma}\right). \quad (3.26)$$

3.9.5 Dirac-Delta-Verteilung und empirische Verteilung

Manchmal möchten wir festlegen, dass die gesamte Masse einer Wahrscheinlichkeitsverteilung sich um einen einzelnen Punkt sammelt. Dazu definieren wir eine PDF mithilfe der **diracschen Deltafunktion** (engl. *dirac delta function*) $\delta(x)$:

$$p(x) = \delta(x - \mu). \quad (3.27)$$

Die diracsche Deltafunktion ist so definiert, dass sie überall den Wert 0 aufweist, aber doch zu 1 integriert. Die diracsche Deltafunktion ist keine gewöhnliche Funktion, die jedem Wert x eine reellwertige Ausgabe zuordnet. Vielmehr handelt es sich um ein besonderes mathematisches Objekt namens **verallgemeinerte Funktion**, die nach der Integration anhand ihrer Eigenschaften definiert wird. Sie können sich die diracsche Deltafunktion als Grenzpunkt einer Reihe von Funktionen vorstellen, die immer weniger Masse auf alle Punkte mit Ausnahme von 0 legt.

Indem wir $p(x)$ als δ mit der Verschiebung $-\mu$ definieren, ergibt sich ein unendlich schmaler und unendlich hoher Scheitel der Wahrscheinlichkeit(smasse), für den gilt $x = \mu$.

Häufig wird die Dirac-Delta-Verteilung als Komponente einer **empirischen Verteilung**

$$\hat{p}(\mathbf{x}) = \frac{1}{m} \sum_{i=1}^m \delta(\mathbf{x} - \mathbf{x}^{(i)}) \quad (3.28)$$

genutzt, mit der die Wahrscheinlichkeit(smasse) $\frac{1}{m}$ auf jeden der m Datenpunkte $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}$ verteilt wird, die eine vorgegebene Menge von Stichproben bilden. Die Dirac-Delta-Verteilung wird lediglich benötigt, um die empirische Verteilung über stetige Variablen zu definieren. Bei diskreten Variablen ist die Sache einfacher: Eine empirische Verteilung ähnelt einer Multinoulli-Verteilung, bei der jedem möglichen Eingabewert eine Wahrscheinlichkeit zugeordnet ist, die der **empirischen Häufigkeit** des Wertes in der Trainingsdatenmenge entspricht.

Wir können die empirische Verteilung für einen Datensatz von Trainingsbeispielen als Spezifikation der Verteilung betrachten, aus der wir beim Trainieren eines Modells mit diesem Datensatz die Stichproben entnehmen. Ein weiterer wichtiger Aspekt von empirischen Verteilungen besteht darin, dass es die Wahrscheinlichkeitsdichte ist, die die Likelihood der Trainingsdaten maximiert (siehe Abschnitt 5.5).

3.9.6 Kombinierte Verteilungen

Häufig werden Wahrscheinlichkeitsverteilungen auch als Kombination mehrerer einfacher Wahrscheinlichkeitsverteilungen definiert. Eine Möglichkeit zum Kombinieren von Verteilungen besteht im Erstellen einer **Mischverteilung**. Eine Mischverteilung beinhaltet mehrere andere Komponentenverteilungen. Bei jedem Test wird durch das Festlegen einer Komponentenidentität aus einer Multinoulli-Verteilung vorgegeben, welche Komponentenverteilung die Stichprobe erzeugt:

$$P(\mathbf{x}) = \sum_i P(c = i)P(\mathbf{x} | c = i), \quad (3.29)$$

mit $P(c)$ als Multinoulli-Verteilung über die Komponentenidentitäten.

Sie kennen bereits ein Beispiel für eine Mischverteilung, nämlich die empirische Verteilung über reellwertige Variablen. Dabei wird für jedes Trainingsbeispiel eine Dirac-Komponente verwendet.

Das Mischmodell ist eine einfache Strategie zum Kombinieren von Wahrscheinlichkeitsverteilungen für eine üppigere Verteilung. In Kapitel 16 behandeln wir das Erstellen komplexer Wahrscheinlichkeitsverteilungen aus simpleren Verteilungen noch genauer.

Mit dem Mischmodell erhalten wir einen kurzen Einblick in ein Konzept, das später noch enorm wichtig wird – die **latente Variable**. Eine latente Variable ist eine Zufallsvariable, die nicht direkt beobachtbar ist. Ein Beispiel dafür ist die Komponentenidentitätsvariable c des Mischmodells. Latente Variablen können über die multivariate Verteilung mit x in Beziehung stehen, in diesem Fall $P(x, c) = P(x | c)P(c)$. Die Verteilung $P(c)$ über die latente Variable und die Verteilung $P(x | c)$, welche die latente Variable zu den sichtbaren Variablen in Beziehung setzt, bestimmen die Form der Verteilung $P(x)$, obwohl es möglich ist, $P(x)$ ohne Verweis auf die latente Variable zu beschreiben. Latente Variablen werden detaillierter in Abschnitt 16.5 behandelt.

Ein sehr leistungsstarkes und häufig verwendetes Mischmodell ist das **gaußsche Mischmodell** (engl. *Gaussian mixture model*, GMM), dessen Komponenten $p(x | c = i)$ Normalverteilungen sind. Jede Komponente verfügt über einen separat parametrisierten Mittelwert $\mu^{(i)}$ und eine Kovarianz $\Sigma^{(i)}$. Einige Mischungen können mehrere Bedingungen aufweisen. So könnte die Kovarianz mittels der Bedingung $\Sigma^{(i)} = \Sigma, \forall i$ über Komponentengrenzen hinweg gelten. Wie bei einer einzelnen Normalverteilung kann auch die Mischung von Normalverteilungen erfordern, dass die Kovarianzmatrix der einzelnen Komponenten diagonal oder isotrop ist.

Neben den Mittelwerten und Kovarianzen geben die Parameter einer gaußschen Mischverteilung die jeder Komponente i zugewiesene **A-priori-Wahrscheinlichkeit** $\alpha_i = P(c = i)$ an. Der Begriff »a priori« deutet an, dass er die Überzeugung des Modells bezüglich c vor der Beobachtung von x ausdrückt. Dagegen ist $P(c | x)$ eine **A-posteriori-Wahrscheinlichkeit**, da sie nach der Beobachtung von x berechnet wird. Ein GMM ist insofern ein **universeller Approximator** der Dichten, als jede glatte Dichte mit einem bestimmten, von Null verschiedenen Fehlerbetrag durch ein GMM mit einer ausreichend hohen Anzahl von Komponenten approximiert werden kann.

Abbildung 3.2 zeigt Stichproben aus einem GMM.

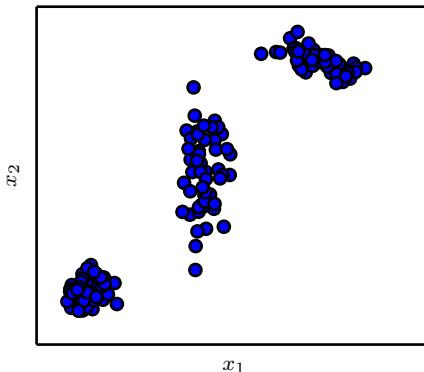


Abbildung 3.2: Stichproben aus einem GMM. In diesem Beispiel gibt es drei Komponenten. (Von links nach rechts:) Die erste Komponente weist eine isotrope Kovarianzmatrix auf, das heißt, das Maß der Varianz ist in allen Richtungen gleich groß. Die zweite Komponente weist eine diagonale Kovarianzmatrix auf; die Varianz wird also in jeder der Achsen separat kontrolliert. Dieses Beispiel weist eine höhere Varianz entlang der x_2 -Achse als entlang der x_1 -Achse auf. Die dritte Komponente weist eine Kovarianzmatrix mit vollem Rang auf, sodass die Varianz in beliebigen Richtungen gesteuert werden kann.

3.10 Nützliche Eigenschaften häufig verwendeter Funktionen

Bei der Arbeit mit Wahrscheinlichkeitsverteilungen treffen Sie oft auf bestimmte Funktionen – und das gilt ganz besonders für Wahrscheinlichkeitsverteilungen in Deep-Learning-Modellen.

Eine dieser Funktionen ist die **logistische Sigmoidfunktion**:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \quad (3.30)$$

Die logistische Sigmoidfunktion wird häufig eingesetzt, um den Parameter ϕ einer Bernoulli-Verteilung zu ermitteln, da ihr Wertebereich $(0, 1)$ beträgt und somit im gültigen Wertebereich des Parameters ϕ liegt. Abbildung 3.3 zeigt eine Darstellung der Sigmoidfunktion. Die Sigmoidfunktion erreicht ihre **Sättigung**, wenn ihr Argument stark positiv oder stark negativ ist, die Funktion also stark abflacht und kleinere Änderungen der Eingabewerte kaum noch zu Änderungen führen.

Eine weitere häufig anzutreffende Funktion ist die **Softplus-Funktion** (*Dugas et al., 2001*):

$$\zeta(x) = \log(1 + \exp(x)). \quad (3.31)$$

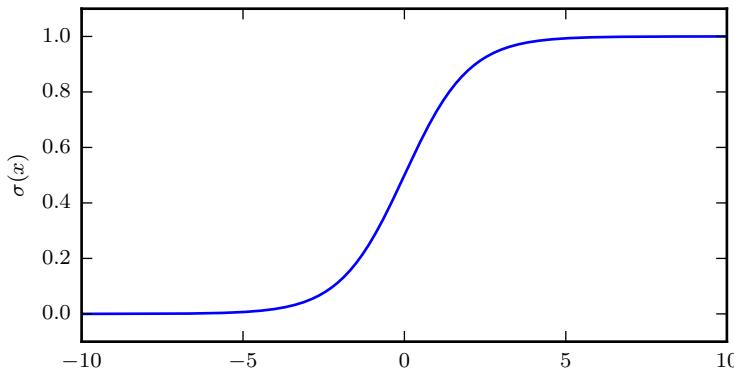


Abbildung 3.3: Die logistische Sigmoidfunktion

Die Softplus-Funktion ist zum Erzeugen des Parameters β oder σ einer Normalverteilung nützlich, da ihr Wertebereich $(0, \infty)$ ist. Sie wird daher auch häufig beim Manipulieren von Ausdrücken für Sigmoidfunktionen verwendet. Der Name der Softplus-Funktion entstand, da es sich um eine geglättete oder »softe« Version von

$$x^+ = \max(0, x) \quad (3.32)$$

handelt. Abbildung 3.4 zeigt die Softplus-Funktion.

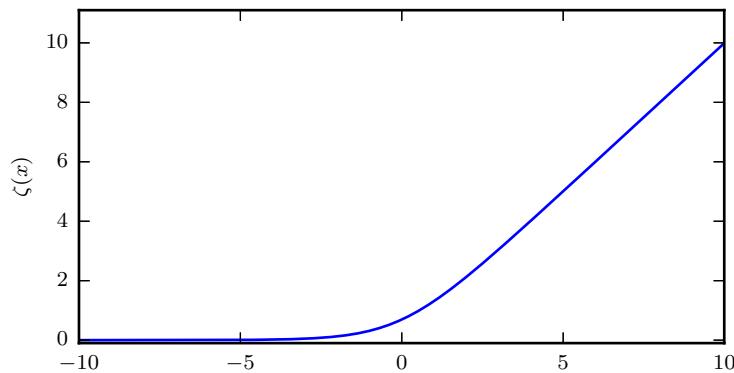


Abbildung 3.4: Die Softplus-Funktion

Die folgenden Eigenschaften sind alle von so großem Nutzen, dass Sie sich diese merken sollten:

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)} \quad (3.33)$$

$$\frac{d}{dx} \sigma(x) = \sigma(x)(1 - \sigma(x)) \quad (3.34)$$

$$1 - \sigma(x) = \sigma(-x) \quad (3.35)$$

$$\log \sigma(x) = -\zeta(-x) \quad (3.36)$$

$$\frac{d}{dx} \zeta(x) = \sigma(x) \quad (3.37)$$

$$\forall x \in (0, 1), \sigma^{-1}(x) = \log \left(\frac{x}{1-x} \right) \quad (3.38)$$

$$\forall x > 0, \zeta^{-1}(x) = \log(\exp(x) - 1) \quad (3.39)$$

$$\zeta(x) = \int_{-\infty}^x \sigma(y) dy \quad (3.40)$$

$$\zeta(x) - \zeta(-x) = x \quad (3.41)$$

Die Funktion $\sigma^{-1}(x)$ heißt in der Statistik **Logit-Funktion**, doch dieser Begriff wird für das Machine Learning selten verwendet.

Gleichung 3.41 liefert eine zusätzliche Begründung für den Namen »Softplus«. Die Softplus-Funktion ist als geglättete Version der **positiven Rampenfunktion** $x^+ = \max\{0, x\}$ gedacht. Die positive Rampenfunktion ist das Gegenstück der **negativen Rampenfunktion**, $x^- = \max\{0, -x\}$. Um eine glatte Funktion zu erhalten, die sich analog zum negativen Gegenstück verhält, können wir $\zeta(-x)$ verwenden. So wie x aus dem positiven und negativen Teil anhand der Identität $x^+ - x^- = x$ ermittelt werden kann, ist es auch möglich, x anhand derselben Beziehung zwischen $\zeta(x)$ und $\zeta(-x)$ wiederherzustellen (vgl. Gleichung 3.41).

3.11 Satz von Bayes

Häufig kennen wir $P(y | x)$ und müssen $P(x | y)$ bestimmen. Zum Glück lässt sich, sofern wir auch $P(x)$ kennen, die gewünschte Größe mit dem **Satz von Bayes** ermitteln:

$$P(x | y) = \frac{P(x)P(y | x)}{P(y)}. \quad (3.42)$$

Obwohl $P(y)$ Teil der Formel ist, reicht es normalerweise aus, $P(y) = \sum_x P(y | x)P(x)$ zu berechnen, sodass keine Kenntnis von $P(y)$ erforderlich ist.

Der Satz von Bayes lässt sich direkt aus der Definition der bedingten Wahrscheinlichkeit ableiten. Dennoch ist es gut, den Namen der Formel zu kennen, da viele Texte nur diese Bezeichnung verwenden. Er wurde nach dem Pfarrer Thomas Bayes benannt, der als Erster einen Sonderfall der Formel entdeckte. Die hier vorgestellte, allgemeine Version wurde unabhängig davon durch Pierre-Simon Laplace entdeckt.

3.12 Technische Einzelheiten stetiger Variablen

Für ein gutes formales Verständnis von stetigen Zufallsvariablen und Wahrscheinlichkeitsdichtefunktionen muss die Wahrscheinlichkeitstheorie in den Begrifflichkeiten eines als **Maßtheorie** bekannten Teilgebiets der Mathematik entwickelt werden. Eine Erklärung der Maßtheorie geht über dieses Lehrbuch hinaus, aber wir möchten zumindest einige der Probleme umreißen, zu deren Lösung die Maßtheorie genutzt wird.

In Abschnitt 3.3.2 haben Sie erfahren, dass sich die Wahrscheinlichkeit eines stetigen vektorwertigen \mathbf{x} in einer Menge \mathbb{S} aus dem Integral von $p(\mathbf{x})$ über die Menge \mathbb{S} ergibt. Für bestimmte Mengen \mathbb{S} kommt es zu einem Paradox. Angenommen, wir konstruieren zwei Mengen \mathbb{S}_1 und \mathbb{S}_2 so, dass gilt $p(\mathbf{x} \in \mathbb{S}_1) + p(\mathbf{x} \in \mathbb{S}_2) > 1$, aber $\mathbb{S}_1 \cap \mathbb{S}_2 = \emptyset$. Diese Mengen werden normalerweise unter Verwendung der unendlichen Genauigkeit reeller Zahlen konstruiert, zum Beispiel durch Erstellen fraktalförmiger Mengen oder von Mengen, die durch Transformation der Menge rationaler Zahlen definiert werden.⁴ Einer der wesentlichen Beiträge der Maßtheorie besteht darin, die Menge der Mengen zu charakterisieren, sodass wir die Wahrscheinlichkeit ohne störende Paradoxe berechnen können. In diesem Buch integrieren wir nur über Mengen mit relativ einfachen Beschreibungen; dieser Aspekt der Maßtheorie ist also nicht wirklich von Belang.

Für unsere Zwecke findet die Maßtheorie eher eine Anwendung beim Beschreiben von Theoremen, die für die meisten Punkte in \mathbb{R}^n zutreffen, jedoch nicht für einige Ausnahmefälle. Die Maßtheorie bietet eine rigorose Möglichkeit, um zu beschreiben, dass eine Menge von Punkten vernachlässigbar klein ist. Eine solche Menge hat das **Maß Null**. Dieses Konzept wird im vorliegenden Buch nicht formal definiert. Für unsere Zwecke reicht es aus, zu verstehen, dass eine Menge mit Maß Null kein Volumen im von uns gemessenen Raum einnimmt. Beispiel: In \mathbb{R}^2 hat eine Linie das Maß Null, ein Polygon dagegen ein positives Maß. Ebenso hat ein einzelner Punkt das Maß Null. Jede Vereinigungsmenge abzählbar vieler Mengen, die jeweils das Maß Null haben, hat ebenfalls das Maß Null. (Zum Beispiel hat die Menge aller rationalen Zahlen das Maß Null.)

Ein weiterer nützlicher Begriff aus der Maßtheorie lautet **fast überall**. Eine Eigenschaft, die fast überall gilt, gilt für den gesamten Raum mit Ausnahmen einer Menge mit Maß Null. Da die Ausnahmen nur einen vernachlässigbaren Raum besetzen, können sie in vielen Anwendungen ohne Folgen ignoriert werden. Einige wichtige Ergebnisse in der Wahrscheinlich-

⁴ Das Banach-Tarski-Paradox ist ein amüsantes Beispiel für derartige Mengen.

keitstheorie gelten für alle diskreten Werte, jedoch nur »fast überall« für stetige Werte.

Ein weiteres technisches Detail der stetigen Variablen hat mit der Handhabung stetiger Zufallsvariablen zu tun, die deterministische Funktionen der jeweils anderen sind. Angenommen, wir haben zwei Zufallsvariablen \mathbf{x} und \mathbf{y} derart, dass $\mathbf{y} = g(\mathbf{x})$ ist (mit g als invertierbarer, stetiger, differenzierbarer Transformation). Man könnte nun erwarten, dass $p_y(\mathbf{y}) = p_x(g^{-1}(\mathbf{y}))$ ist. Aber das ist nicht der Fall.

Hierzu ein einfaches Beispiel anhand der skalaren Zufallsvariablen x und y . Es sei $y = \frac{x}{2}$ und $x \sim U(0, 1)$. Unter Verwendung der Regel $p_y(y) = p_x(2y)$ ist p_y überall 0, mit Ausnahme des Intervalls $[0, \frac{1}{2}]$, in dem es 1 ist. Daraus folgt

$$\int p_y(y) dy = \frac{1}{2}, \quad (3.43)$$

und das verstößt gegen die Definition einer Wahrscheinlichkeitsverteilung. Das ist ein häufiger Fehler. Das Problem bei diesem Ansatz besteht darin, dass er die Raumverzerrung infolge der Funktion g nicht berücksichtigt. Rufen Sie sich in Erinnerung, dass die Wahrscheinlichkeit von \mathbf{x} in einem infinitesimal kleinen Bereich mit dem Volumen $\delta\mathbf{x}$ liegt und sich aus $p(\mathbf{x})\delta\mathbf{x}$ ergibt. Da g den Raum dehnen oder zusammenziehen kann, ist es für das infinitesimale Volumen um \mathbf{x} im Raum \mathbf{x} möglich, im Raum \mathbf{y} ein anderes Volumen aufzuweisen.

Um das Problem zu beheben, sehen wir uns den Skalarfall an. Wir müssen diese Eigenschaft erhalten:

$$|p_y(g(x))dy| = |p_x(x)dx|. \quad (3.44)$$

Durch Auflösen erhalten wir

$$p_y(y) = p_x(g^{-1}(y)) \left| \frac{\partial x}{\partial y} \right| \quad (3.45)$$

oder äquivalent

$$p_x(x) = p_y(g(x)) \left| \frac{\partial g(x)}{\partial x} \right|. \quad (3.46)$$

In höheren Dimensionen generalisiert sich die Ableitung zur Determinante der **Jacobi-Matrix** – die Matrix, für die gilt $J_{i,j} = \frac{\partial x_i}{\partial y_j}$. Somit gilt für reellwertige Vektoren \mathbf{x} und \mathbf{y}

$$p_x(\mathbf{x}) = p_y(g(\mathbf{x})) \left| \det \left(\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} \right) \right|. \quad (3.47)$$

3.13 Informationstheorie

Die Informationstheorie ist ein Teilgebiet der angewandten Mathematik, in dem es darum geht, zu quantifizieren, wie viele Informationen in einem Signal vorliegen. Sie wurde ursprünglich erfunden, um den Nachrichtenversand für diskrete Alphabete über einen rauschbehafteten Kanal zu untersuchen, zum Beispiel in der Funkkommunikation. In diesem Zusammenhang zeigt uns die Informationstheorie, wie optimierte Codes konzipiert werden und die erwartete Länge von Nachrichten, die aus bestimmten Wahrscheinlichkeitsverteilungen mithilfe unterschiedlicher Codierungsverfahren ausgewählt wurden, berechnet werden kann. Im Kontext des Machine Learnings können wir die Informationstheorie auch für stetige Variablen verwenden, wo nicht alle diese Interpretationen zur Nachrichtenlänge gelten. Das Gebiet ist wesentlich für viele Bereiche der Elektrotechnik und Informatik. In diesem Buch verwenden wir in erster Linie einige wenige grundlegende Konzepte aus der Informationstheorie, um Wahrscheinlichkeitsverteilungen zu charakterisieren oder die Ähnlichkeit zwischen Wahrscheinlichkeitsverteilungen zu quantifizieren. Einzelheiten zur Informationstheorie finden Sie in *Cover und Thomas* (2006) und *MacKay* (2003).

Der Informationstheorie liegt im Wesentlichen folgende Vorstellung zugrunde: Zu erfahren, dass ein unwahrscheinliches Ereignis eingetreten ist, stellt einen größeren Informationsgehalt dar, als zu erfahren, dass ein wahrscheinliches Ereignis eingetreten ist. So besitzt die Nachricht »Heute Morgen ist die Sonne aufgegangen« praktisch keinen Informationswert und ihr Versand hätte ebenso gut unterbleiben können. Dagegen ist der Informationswert der Nachricht »Heute Morgen gab es eine Sonnenfinsternis« sehr hoch.

Wir möchten Informationen gerne so quantifizieren, dass diese Vorstellung formalisiert ist.

- Wahrscheinliche Ereignisse sollten einen geringen Informationsgehalt aufweisen und Ereignisse, die garantiert eintreten werden, im Extremfall sogar überhaupt keinen Informationsgehalt.
- Weniger wahrscheinliche Ereignisse sollten einen höheren Informationsgehalt aufweisen.
- Unabhängige Ereignisse sollten einen zusätzlichen (additiven) Informationswert aufweisen. Wird zum Beispiel festgestellt, dass ein Münzwurf zweimal in Folge »Kopf« zeigt, soll der Informationswert doppelt so hoch sein, als wenn der Münzwurf einmal »Kopf« zeigt.

Um diese drei Eigenschaften zu erfüllen, definieren wir den **Informationsgehalt** eines Ereignisses $x = x$ als

$$I(x) = -\log P(x). \quad (3.48)$$

In diesem Buch verwenden wir stets \log für den natürlichen Logarithmus zur Basis e . Unsere Definition für $I(x)$ wird daher in der Einheit **Nit** geschrieben. Ein Nit ist der Maß des Informationsgewinns durch Beobachtung eines Ereignisses mit der Wahrscheinlichkeit $\frac{1}{e}$. Andere Texte verwenden Logarithmen zur Basis 2 und die Einheit **Bit** oder **Shannon**; in Bit gemessene Informationen sind lediglich eine skalierte Darstellung der Messung in Nit.

Ist x stetig, verwenden wir dieselbe Definition für die Information analog; allerdings gehen einige der Eigenschaften für den diskreten Fall verloren. Ein Ereignis mit der Dichte 1 enthält nach wie vor keine Informationen, obwohl es sich nicht um ein garantiert eintretendes Ereignis handelt.

Der Informationsgehalt befasst sich nur mit einem einzelnen Ergebnis. Wir können das Maß der Unsicherheit für eine vollständige Wahrscheinlichkeitsverteilung mithilfe der **Shannon-Entropie** quantifizieren,

$$H(x) = \mathbb{E}_{x \sim P}[I(x)] = -\mathbb{E}_{x \sim P}[\log P(x)], \quad (3.49)$$

auch $H(P)$ notiert. Anders ausgedrückt: Die Shannon-Entropie einer Verteilung ist der erwartete Informationswert für ein aus dieser Verteilung gezogenes Ereignis. Sie legt eine untere Schranke für die Anzahl der Bits (Logarithmus zur Basis 2, ansonsten wird eine andere Einheit verwendet) fest, die im Durchschnitt zum Codieren von aus einer Verteilung P gezogenen Symbolen benötigt wird. Nahezu deterministische Verteilungen (deren Ausgang nahezu sicher ist) weisen eine geringe Entropie auf; Verteilungen, die näher an der Gleichverteilung liegen, dagegen eine hohe Entropie. Abbildung 3.5 stellt dies dar. Ist x stetig, wird die Shannon-Entropie auch **differentielle Entropie** genannt.

Wenn wir zwei unterschiedliche Wahrscheinlichkeitsverteilungen $P(x)$ und $Q(x)$ über dieselbe Zufallsvariable x haben, können wir die Unterschiedlichkeit der beiden Verteilungen mithilfe der **Kullback-Leibler-Divergenz** (KL-Divergenz) ermitteln:

$$D_{\text{KL}}(P \| Q) = \mathbb{E}_{x \sim P} \left[\log \frac{P(x)}{Q(x)} \right] = \mathbb{E}_{x \sim P} [\log P(x) - \log Q(x)]. \quad (3.50)$$

Für diskrete Variablen handelt es sich um das zusätzliche Informationsmaß (gemessen in Bits für einen Logarithmus zur Basis 2; allerdings nutzen

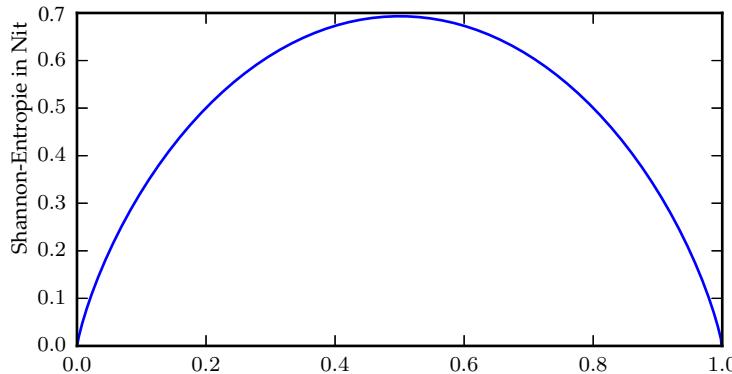


Abbildung 3.5: Shannon-Entropie einer binären Zufallsvariable. Diese Kurve zeigt, dass eher deterministische Verteilungen eine geringe Shannon-Entropie aufweisen, wohingegen zur Gleichverteilung tendierende Verteilungen eine hohe Shannon-Entropie aufweisen. Auf der horizontalen Achse tragen wir p auf, die Wahrscheinlichkeit einer binären Zufallsvariable, die gleich 1 ist. Die Entropie ergibt sich aus $(p - 1) \log(1 - p) - p \log p$. Ist p nahe 0, ist die Verteilung nahezu deterministisch, da die Zufallsvariable fast immer 0 ist. Ist p nahe 1, ist die Verteilung nahezu deterministisch, da die Zufallsvariable fast immer 1 ist. Ist $p = 0,5$, ist die Entropie maximal, da die Verteilung über die beiden Ergebnisse gleichförmig ist.

wir im Machine Learning die Einheit Nit und den natürlichen Logarithmus), das zum Senden einer Nachricht mit aus der Wahrscheinlichkeitsverteilung P gezogenen Symbolen benötigt wird, sofern wir einen Code verwenden, der die Länge der aus der Wahrscheinlichkeitsverteilung Q gezogenen Nachrichten minimieren soll.

Die KL-Divergenz hat viele nützliche Eigenschaften; vor allem ist sie nicht-negativ. Die KL-Divergenz ist genau dann 0, wenn P und Q im Falle diskreter Variablen dieselbe Verteilung aufweisen oder im Falle stetiger Variablen »fast überall« entsprechen. Da die KL-Divergenz nicht-negativ ist und die Unterschiedlichkeit zwischen zwei Verteilungen misst, wird häufig gedacht, dass sie eine Art Distanz zwischen den Verteilungen misst. Es handelt sich aber nicht um ein echtes Distanzmaß, da sie nicht symmetrisch ist: $D_{\text{KL}}(P\|Q) \neq D_{\text{KL}}(Q\|P)$ für einige P und Q . Diese Asymmetrie bedeutet, dass die Wahl der Verwendung von $D_{\text{KL}}(P\|Q)$ oder $D_{\text{KL}}(Q\|P)$ wesentliche Folgen hat. Abbildung 3.6 enthält weitere Informationen.

Eine Größe, die mit der KL-Divergenz eng verwandt ist, ist die **Kreuz-entropie** $H(P, Q) = H(P) + D_{\text{KL}}(P\|Q)$; sie ähnelt der KL-Divergenz, aber es fehlt der Term auf der linken Seite:

$$H(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x). \quad (3.51)$$

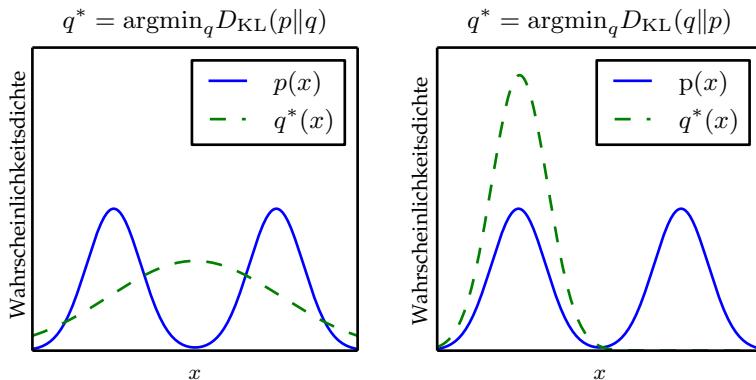


Abbildung 3.6: Die KL-Divergenz ist asymmetrisch. Gegeben ist eine Verteilung $p(x)$, die wir anhand einer weiteren Verteilung $q(x)$ approximieren möchten. Wir können entweder $D_{\text{KL}}(p\|q)$ oder $D_{\text{KL}}(q\|p)$ minimieren. Die Folgen dieser Wahl verdeutlichen wir mit einer Mischung von zwei Normalverteilungen für p und einer einzelnen Normalverteilung für q . Die Wahl der Richtung der KL-Divergenz ist aufgabenabhängig. Für einige Anwendungen wird eine Approximation benötigt, die normalerweise eine hohe Wahrscheinlichkeit beliebig dort platziert, wo die wahre Verteilung eine hohe Wahrscheinlichkeit platziert, während für andere Anwendungen eine Approximation benötigt wird, die eine hohe Wahrscheinlichkeit nur selten dort platziert, wo die wahre Verteilung eine niedrige Verteilung platziert. Die Wahl der Richtung der KL-Divergenz spiegelt wider, welche der beiden Überlegungen für die einzelnen Anwendungen Priorität hat. (*Links*) Die Folgen einer Minimierung von $D_{\text{KL}}(p\|q)$. In diesem Fall wählen wir ein q mit einer hohen Wahrscheinlichkeit dort, wo p eine hohe Wahrscheinlichkeit aufweist. Wenn p mehrere Modi aufweist, führt q zu einer Vermischung der Modi, um die hohe Wahrscheinlichkeit(smasse) auf allen davon zu platzieren. (*Rechts*) Die Folgen einer Minimierung von $D_{\text{KL}}(q\|p)$. In diesem Fall wählen wir ein q mit einer niedrigen Wahrscheinlichkeit dort, wo p eine niedrige Wahrscheinlichkeit aufweist. Wenn p mehrere Modi aufweist, deren Separation ausreichend hoch ist (wie in dieser Abbildung), wird die KL-Divergenz minimiert, indem ein einzelner Modus ausgewählt wird, damit die Wahrscheinlichkeit(smasse) nicht in Bereichen mit einer niedrigen Wahrscheinlichkeit zwischen den Modis platziert wird. Hier zeigen wir das Ergebnis bei Wahl von q zur Betonung des linken Modus. Durch Wahl des rechten Modus hätten wir den identischen Wert für die KL-Divergenz erzielen können. Ist die Separation (durch einen ausreichend starken Bereich mit niedriger Wahrscheinlichkeit) zwischen den Modis nicht groß genug, kann auch diese Ausrichtung der KL-Divergenz zu einer Vermischung der Modi führen.

Das Minimieren der Kreuzentropie für Q entspricht der Minimierung der KL-Divergenz, da Q im verworfenen Term keine Rolle spielt.

Beim Berechnen vieler dieser Größen finden sich oft Ausdrücke der Form $0 \log 0$. Laut Konvention wird im Rahmen der Informationstheorie dieser Ausdruck behandelt als $\lim_{x \rightarrow 0} x \log x = 0$.

3.14 Strukturierte probabilistische Modelle

Machine-Learning-Algorithmen nutzen häufig Wahrscheinlichkeitsverteilungen über sehr viele Zufallsvariablen. Oft beinhalten diese Wahrscheinlichkeitsverteilungen direkte Interaktionen zwischen relativ wenigen Variablen. Das Verwenden einer einzelnen Funktion zum Beschreiben der gesamten multivariaten Verteilung kann sich als sehr ineffizient erweisen (sowohl was den Rechenaufwand als auch die Statistik betrifft).

Statt eine Einzelfunktion zur Repräsentation einer Wahrscheinlichkeitsverteilung zu verwenden, können wir eine Wahrscheinlichkeitsverteilung in viele Faktoren aufteilen, die gemeinsam multipliziert werden. Ein Beispiel: Gegeben sind drei Zufallsvariablen: a, b und c. Angenommen, a beeinflusst den Wert von b und b beeinflusst den Wert von c, aber a und c sind für b unabhängig. Wir können diese Wahrscheinlichkeitsverteilung über alle drei Variablen als Produkt der Wahrscheinlichkeitsverteilungen über zwei Variablen darstellen:

$$p(a, b, c) = p(a)p(b | a)p(c | b). \quad (3.52)$$

Diese Faktorisierungen können die Anzahl der zur Beschreibung der Verteilung benötigten Parameter erheblich reduzieren. Jeder Faktor verwendet eine Anzahl von Parametern, die exponentiell zur Anzahl der Variablen im Faktor ist. Wir können somit die Kosten zur Repräsentation einer Verteilung wesentlich senken, wenn wir eine Faktorisierung in Verteilungen über weniger Variablen finden.

Derartige Faktorisierungen lassen sich mit Graphen darstellen. Hier wird das Wort »Graph« im Sinne der Graphentheorie verwendet: eine Reihe von Knoten, die über Ecken miteinander verbunden sein können. Die Darstellung der Faktorisierung einer Wahrscheinlichkeitsverteilung wird als **strukturiertes probabilistisches Modell** oder **graphisches Modell** bezeichnet.

Es gibt zwei wesentliche Arten von strukturierten probabilistischen Modellen, nämlich gerichtete und ungerichtete (engl. *directed/undirected*

models). Beide Arten von graphischen Modellen verwenden einen Graphen \mathcal{G} , in dem jeder Knoten des Graphen einer Zufallsvariable entspricht; eine Kante, die zwei Zufallsvariablen miteinander verbindet, bedeutet, dass die Wahrscheinlichkeitsverteilung die direkten Interaktionen zwischen diesen beiden Zufallsvariablen darstellen kann.

Gerichtete Modelle nutzen Graphen mit gerichteten Kanten und stellen Faktorisierungen in Form von bedingten Wahrscheinlichkeitsverteilungen dar (wie im obigen Beispiel). Ein gerichtetes Modell zeichnet sich dadurch aus, dass es einen Faktor für jede Zufallsvariable x_i in der Verteilung besitzt, der aus der bedingten Verteilung über x_i mit den übergeordneten Elementen (Eltern) von x_i besteht, geschrieben $Pa_{\mathcal{G}}(x_i)$:

$$p(\mathbf{x}) = \prod_i p(x_i | Pa_{\mathcal{G}}(x_i)). \quad (3.53)$$

Abbildung 3.7 zeigt ein Beispiel eines gerichteten Graphen und die durch diesen dargestellte Faktorisierung der zugehörigen Wahrscheinlichkeitsverteilungen.

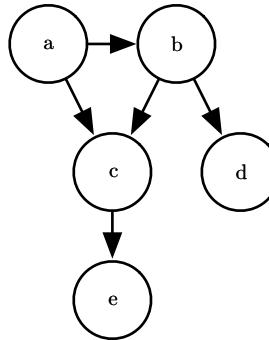


Abbildung 3.7: Ein gerichtetes graphisches Modell über die Zufallsvariablen a, b, c, d und e. Dieser Graph entspricht den Wahrscheinlichkeitsverteilungen, die wie folgt faktorisiert werden können:

$$p(a, b, c, d, e) = p(a)p(b | a)p(c | a, b)p(d | b)p(e | c). \quad (3.54)$$

Das graphische Modell gewährt einen schnellen Einblick in einige Eigenschaften der Verteilung. Zum Beispiel interagieren a und c direkt miteinander, während a und e nur indirekt über c interagieren.

Ungerichtete Modelle nutzen Graphen mit ungerichteten Kanten und stellen Faktorisierungen in einer Reihe von Funktionen dar; anders als im gerichteten Fall handelt es sich dabei normalerweise nicht um Wahrscheinlichkeitsverteilungen beliebiger Art. Jede Menge von Knoten, die in \mathcal{G} alle

miteinander verbunden sind, wird als Clique bezeichnet. Jede Clique $\mathcal{C}^{(i)}$ in einem ungerichteten Modell ist mit einem Faktor $\phi^{(i)}(\mathcal{C}^{(i)})$ verknüpft. Diese Faktoren sind lediglich Funktionen, keine Wahrscheinlichkeitsverteilungen. Die Ausgabe eines Faktors muss nicht-negativ sein, aber es gibt keine Bedingung, dass der Faktor – wie eine Wahrscheinlichkeitsverteilung – sich zu 1 summieren oder integrieren muss.

Die Wahrscheinlichkeit einer Konfiguration von Zufallsvariablen ist **proportional** zum Produkt aller Faktoren – Zuordnungen, die zu höheren Faktorwerten führen, sind wahrscheinlicher. Natürlich gibt es keine Garantie dafür, dass das Produkt sich zu 1 summiert. Wir dividieren daher durch eine Normalisierungskonstante Z , die definiert ist als Summe oder Integral über alle Zustände des Produkts der ϕ -Funktionen, um eine normalisierte Wahrscheinlichkeitsverteilung zu erhalten:

$$p(\mathbf{x}) = \frac{1}{Z} \prod_i \phi^{(i)}(\mathcal{C}^{(i)}). \quad (3.55)$$

Abbildung 3.8 zeigt ein Beispiel eines ungerichteten Graphen und die Faktorisierung der zugehörigen Wahrscheinlichkeitsverteilung.

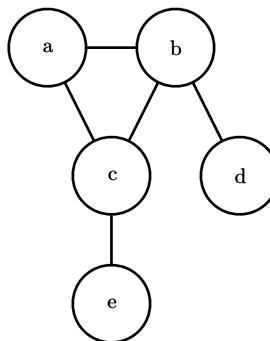


Abbildung 3.8: Ein ungerichtetes graphisches Modell über die Zufallsvariablen a, b, c, d und e. Dieser Graph entspricht den Wahrscheinlichkeitsverteilungen, die faktorisiert werden können als

$$p(a, b, c, d, e) = \frac{1}{Z} \phi^{(1)}(a, b, c) \phi^{(2)}(b, d) \phi^{(3)}(c, e). \quad (3.56)$$

Das graphische Modell gewährt einen schnellen Einblick in einige Eigenschaften der Verteilung. Zum Beispiel interagieren a und c direkt miteinander, während a und e nur indirekt über c interagieren.

Bedenken Sie, dass diese graphischen Darstellungen der Faktorisierungen eine Repräsentationsform zum Abbilden von Wahrscheinlichkeitsverteilungen sind. Es handelt sich nicht um einander ausschließende Familien von

Wahrscheinlichkeitsverteilungen. »Gerichtet« und »ungerichtet« sind keine Eigenschaften einer Wahrscheinlichkeitsverteilung, sondern die Eigenschaften einer bestimmten **Darstellung** einer Wahrscheinlichkeitsverteilung; jede Wahrscheinlichkeitsverteilung kann auf beide Arten dargestellt werden.

In den Teile I und II dieses Buchs setzen wir strukturierte probabilistische Modelle lediglich Darstellungsform ein, um abzubilden, welche probabilistischen Beziehungen für die Darstellung verschiedener Machine-Learning-Algorithmen gewählt werden können. Ein weitergehendes Verständnis strukturierter probabilistischer Modelle wird erst im Rahmen der Diskussion von Forschungsthemen in Teil III benötigt; dort befassen wir uns sehr viel detaillierter mit strukturierten probabilistischen Modellen.

In diesem Kapitel haben Sie einen Überblick über die grundlegenden Konzepte der Wahrscheinlichkeitstheorie erhalten, die für Deep Learning von großer Bedeutung sind. Damit verbleibt noch ein weiteres fundamentales mathematisches Werkzeug, die numerischen Methoden.

4

Numerische Berechnung

Machine-Learning-Algorithmen machen im Normalfall starken Gebrauch von der numerischen Berechnung. Dabei handelt es sich meist um Algorithmen, die mathematische Probleme lösen, indem Schätzungen der Lösung iterativ angepasst werden; es kommt also kein analytisches Formelsystem als symbolischer Ausdruck der korrekten Lösung zum Einsatz. Zu den häufigen Operationen gehören die Optimierung (das Bestimmen eines Argumentenwerts, der eine Funktion minimiert oder maximiert) und das Lösen von linearen Gleichungssystemen. Schon das Auswerten einer mathematischen Funktion auf einem digitalen Computer kann sich als schwierig herausstellen, wenn diese Funktion reelle Zahlen enthält, die mithilfe einer endlichen Speichermenge nicht präzise dargestellt werden können.

4.1 Überlauf und Unterlauf

Die grundlegende Schwierigkeit in Verbindung mit der kontinuierlichen Mathematik auf einem digitalen Computer besteht darin, dass wir unendlich viele reelle Zahlen anhand einer endlichen Anzahl von Bitmustern wiedergeben müssen. Das hat zur Folge, dass für nahezu alle reellen Zahlen gewisse Approximationsfehler beim Darstellen der Zahl im Computer auftreten. Häufig handelt es sich dabei lediglich um einen Rundungsfehler. Ein Rundungsfehler stellt ein Problem dar – vor allem, wenn er über viele Operationen aufsummiert wird – und kann dazu führen, dass theoretisch fehlerfreie Algorithmen in der Praxis nicht funktionieren, sofern sie nicht so konzipiert sind, dass sie die Akkumulation des Rundungsfehlers minimieren.

Eine besonders problematische Form des Rundungsfehlers ist der **Unterlauf**. Der Unterlauf tritt auf, wenn Zahlen, die fast Null sind, auf Null

abgerundet werden. Viele Funktionen verhalten sich qualitativ anders, wenn ihr Argument Null ist und nicht eine kleine positive Zahl. Ein Beispiel: Normalerweise versuchen wir, eine Division durch Null – in einigen Programmumgebungen führt dies zu einem Ausnahmefehler, in anderen wird ein Platzhalter NaN (engl. *not-a-number*) ausgegeben – oder das Bilden des Logarithmus von Null – dieser Vorgang wird gewöhnlich als $-\infty$ betrachtet, was zu einem NaN-Wert für weitere Rechenschritte führt – zu vermeiden.

Mit dem **Überlauf** gibt es einen weiteren, besonders schädlichen numerischen Fehler. Ein Überlauf tritt auf, wenn Zahlen mit großem Betrag als ∞ oder $-\infty$ approximiert werden. Weitere Rechenschritte ändern diese unendlichen Werte normalerweise in NaN-Werte.

Ein Beispiel für eine Funktion, die gegen Unterlauf und Überlauf stabilisiert werden muss, ist die **softmax-Funktion**. Die softmax-Funktion wird häufig zum Vorhersagen der Wahrscheinlichkeiten einer Multinomialverteilung verwendet und ist definiert als

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)}. \quad (4.1)$$

Was geschieht nun, wenn alle x_i gleich einer Konstanten c sind? Analytisch gesehen müssten alle Ergebnisse gleich $\frac{1}{n}$ sein. Numerisch ist das für große Beträge von c möglicherweise nicht der Fall. Ist c stark negativ, dann führt $\exp(c)$ zu einem Unterlauf. Damit wird der Nenner der softmax-Funktion 0 und somit ist das Endergebnis nicht definiert. Wenn c sehr groß und positiv ist, führt $\exp(c)$ zu einem Überlauf, wodurch wiederum der gesamte Ausdruck undefiniert ist. Diese beiden Fälle können abgefangen werden, indem stattdessen $\text{softmax}(\mathbf{z})$ für $\mathbf{z} = \mathbf{x} - \max_i x_i$ bestimmt wird. Einfache Algebra zeigt, dass der Wert der softmax-Funktion analytisch nicht durch Addition oder Subtraktion eines Skalars vom Eingabevektor verändert wird. Durch Subtraktion von $\max_i x_i$ ergibt sich 0 als größtes Argument für \exp , sodass ein Überlauf nicht mehr möglich ist. Und wenn mindestens ein Term im Nenner einen Wert von 1 aufweist, kann es ebenso nicht zu einem Unterlauf im Nenner kommen, der zu einer Division durch Null führen würde.

Doch es gibt noch ein kleines Problem: Der Unterlauf im Zähler kann nach wie vor dazu führen, dass der gesamte Ausdruck Null wird. Wenn wir also $\log \text{softmax}(\mathbf{x})$ implementieren, indem wir zunächst die softmax-Subroutine ausführen und ihr Ergebnis dann an die log-Funktion übergeben, könnte es zum fehlerhaften Ergebnis $-\infty$ kommen. Wir müssen also eine separate Funktion zur numerisch stabilen Berechnung von $\log \text{softmax}$ im-

plementieren. Die Funktion log softmax kann auf dieselbe Weise stabilisiert werden, die wir bereits für die softmax-Funktion verwendet haben.

Größtenteils gehen wir nicht explizit auf sämtliche Details der numerischen Überlegungen ein, die bei der Umsetzung der verschiedenen Algorithmen in diesem Buch eine Rolle spielen. Entwickler von Bibliotheken der unteren Ebene sollten derartige numerische Probleme bei der Implementierung von Deep-Learning-Algorithmen jedoch im Blick behalten. Die meisten Leser dieses Buchs können sich einfach darauf verlassen, dass die Bibliotheken der unteren Ebene für eine stabile Implementierung sorgen. In einigen Fällen ist es möglich, einen neuen Algorithmus zu implementieren und für eine automatische Stabilisierung der neuen Implementierung zu sorgen. Theano (*Bergstra et al., 2010; Bastien et al., 2012*) ist ein Beispiel für ein Softwarepaket, das automatisch viele verbreitete numerisch instabile Ausdrücke im Deep-Learning-Kontext erkennt und stabilisiert.

4.2 Schlechte Konditionierung

Die Konditionierung oder Konditionszahl gibt an, wie sehr eine Funktion von kleinen Änderungen in den Eingangsdaten abhängig ist. Funktionen, die sich bei geringfügigen Störungen der Eingangsdaten stark ändern, können ein Problem bei wissenschaftlichen Berechnungen darstellen, da Rundungsfehler in den Eingangsdaten zu großen Änderungen der Ausgangsdaten führen können.

Betrachten wir die Funktion $f(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$. Bei einer Eigenwertzerlegung von $\mathbf{A} \in \mathbb{R}^{n \times n}$ lautet die **Konditionszahl**

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|. \quad (4.2)$$

Sie gibt das Größenverhältnis des größten und kleinsten Eigenwerts an. Ist diese Zahl zu groß, ist die Matrixinversion für Fehler in den Eingangsdaten besonders anfällig.

Diese Anfälligkeit ist eine intrinsische Eigenschaft der Matrix selbst, nicht etwa das Ergebnis des Rundungsfehlers während der Matrixinversion. Schlecht konditionierte Matrizen verstärken bereits vorhandene Fehler bei der Multiplikation mit der echten Inversen der Matrix. In der Praxis verstärkt sich der Fehler durch numerische Fehler in der Inversion selbst weiter.

4.3 Optimierung auf Gradientenbasis

Die meisten Deep-Learning-Algorithmen beinhalten auch eine Variante der Optimierung. Der Begriff Optimierung bezeichnet die Aufgabe, eine Funktion $f(\mathbf{x})$ durch Verändern von \mathbf{x} zu minimieren oder zu maximieren. Für die meisten Optimierungsprobleme minimieren wir $f(\mathbf{x})$. Die Maximierung kann über einen Minimierungsalgorithmus erreicht werden, indem $-f(\mathbf{x})$ minimiert wird.

Die zu minimierende oder maximierende Funktion wird als **Zielfunktion** oder **Kriterium** bezeichnet. Bei der Minimierung werden auch die Begriffe **Kostenfunktion**, **Verlustfunktion** (engl. *loss function*) oder **Fehlerfunktion** verwendet. In diesem Buch nutzen wir all diese Begriffe. Beachten Sie, dass einige Machine-Learning-Veröffentlichungen jedem davon eine eigene Bedeutung zuweisen.

Häufig notieren wir den Wert, der eine Funktion minimiert oder maximiert, mit einem hochgestellten Sternchen (*). Ein Beispiel: $\mathbf{x}^* = \arg \min f(\mathbf{x})$.

Wir gehen davon aus, dass Sie bereits mit der Analysis vertraut sind, geben aber einen kurzen Überblick über die Verbindung von Konzepten aus der Analysis mit den hier vorgestellten Optimierungen.

Gegeben sei eine Funktion $y = f(x)$, in der x und y reelle Zahlen sind. Die **Ableitung** dieser Funktion wird notiert als $f'(x)$ bzw. $\frac{dy}{dx}$. Die Ableitung $f'(x)$ führt zur Steigung von $f(x)$ im Punkt x . Anders ausgedrückt gibt sie an, wie eine kleine Änderung der Eingangsdaten skaliert werden muss, um die entsprechende Änderung in den Ausgangsdaten zu erhalten: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

Die Ableitung ist daher beim Minimieren einer Funktion nützlich, denn sie gibt an, wie x geändert werden muss, damit eine kleine Verbesserung in y erzielt wird. Ein Beispiel: Wir wissen, dass $f(x - \epsilon \text{sign}(f'(x)))$ kleiner als $f(x)$ ist, sofern ϵ klein genug ist. Daher können wir $f(x)$ durch kleine Verschiebungen von x mit umgekehrtem Vorzeichen in der Ableitung reduzieren. Diese Methode wird als **Gradientenabstiegsverfahren** oder kurz Gradientenverfahren bezeichnet (Cauchy, 1847). Abbildung 4.1 enthält ein Beispiel des Verfahrens.

Für $f'(x) = 0$ bietet die Ableitung keine Informationen über die Bewegungsrichtung. Punkte mit bekannten $f'(x) = 0$ werden **kritische Punkte** oder **stationäre Punkte** genannt. Ein **lokales Minimum** ist ein Punkt, in dem $f(x)$ niedriger als alle Nachbarpunkte ist, sodass $f(x)$ durch infinitesimale Schritte nicht weiter vermindert werden kann. Ein **lokales Maximum**

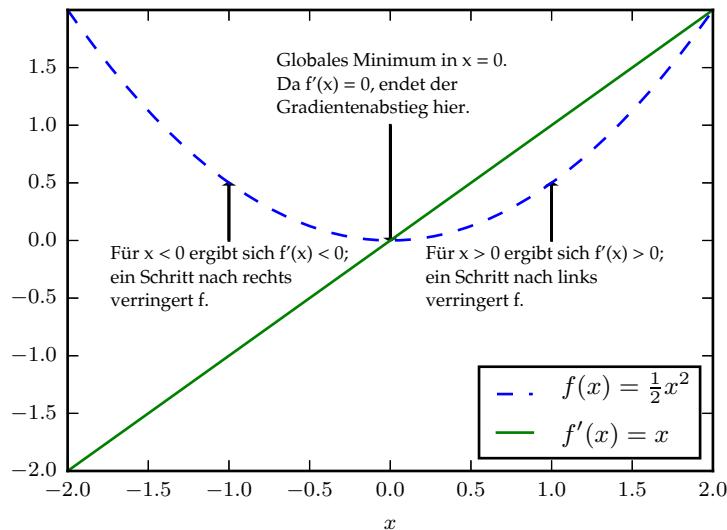


Abbildung 4.1: Gradientenabstiegsverfahren. Diese Abbildung zeigt, wie der Gradientenabstiegsalgorithmus die Ableitungen einer Funktion verwendet und so ein kleinster Wert in absteigender Richtung bestimmt werden kann.

ist ein Punkt, in dem $f(x)$ höher als alle Nachbarpunkte ist, sodass $f(x)$ durch infinitesimale Schritte nicht weiter vergrößert werden kann. Einige kritische Punkte sind weder Maxima noch Minima. Dabei handelt es sich um sogenannte **Sattelpunkte**. Abbildung 4.2 zeigt Beispiele für jede Art kritischer Punkte.

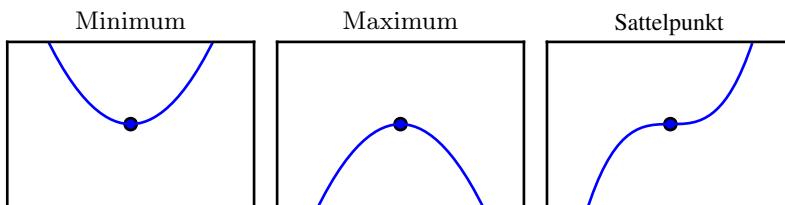


Abbildung 4.2: Beispiele der drei Arten von kritischen Punkten in einer Dimension. Ein kritischer Punkt ist ein Punkt mit Steigung Null. Es kann sich entweder um ein lokales Minimum handeln, das tiefer als die Nachbarpunkte liegt, ein lokales Maximum, das höher als die Nachbarpunkte liegt, oder einen Sattelpunkt, der Nachbarn hat, die höher und tiefer als der Punkt selbst liegen.

Ein Punkt, der den absolut tiefsten Wert für $f(x)$ annimmt, ist ein **globales Minimum**. Es kann entweder nur ein globales Minimum oder mehrere globale Minima der Funktion geben. Außerdem kann es lokale Minima geben, die nicht global optimal sind. Im Deep-Learning-Kontext

optimieren wir Funktionen, die viele lokale, nicht optimale Minima und viele Sattelpunkte, die von sehr ebenen Bereichen umgeben sind, aufweisen können. Das macht die Optimierung schwierig, vor allem wenn die Eingangsdaten der Funktion mehrdimensional sind. Daher begnügen wir uns meist damit, einen Wert für f zu finden, der sehr niedrig ist, aber nicht unbedingt im formalen Sinn den kleinsten Wert darstellen muss. Abbildung 4.3 zeigt ein Beispiel.

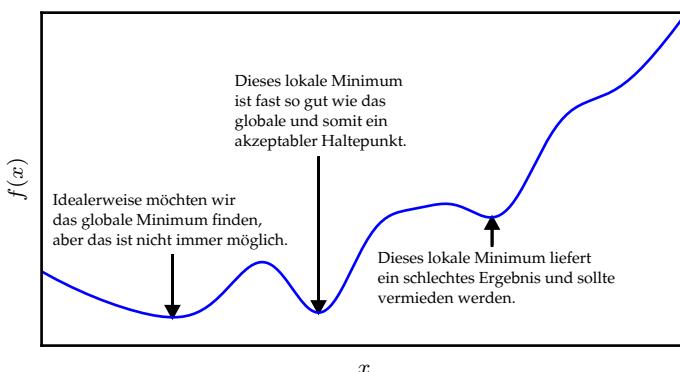


Abbildung 4.3: Approximative Minimierung. Optimierungsalgorithmen finden möglicherweise kein globales Minimum, wenn es viele lokale Minima oder Plateaus gibt. Im Deep-Learning-Kontext akzeptieren wir allgemein solche Lösungen, obwohl sie nicht wirklich das Minimum angeben, sofern sie signifikant niedrigen Werten der Kostenfunktion entsprechen.

Wir minimieren häufig Funktionen, die über mehrere Eingangsdaten verfügen: $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Damit die Minimierung einen Sinn ergibt, darf es dennoch nur eine (skalare) Ausgabe geben.

Für Funktionen mit mehreren Eingangsdaten müssen wir das Konzept der **partiellen Ableitungen** verwenden. Die partielle Ableitung $\frac{\partial}{\partial x_i} f(\mathbf{x})$ misst, wie f sich verändert, wenn nur die Variable x_i im Punkt \mathbf{x} zunimmt. Der **Gradient** generalisiert den Begriff der Ableitung für den Fall, in dem die Ableitung sich auf einen Vektor bezieht: Der Gradient von f ist der Vektor, der alle partiellen Ableitungen enthält; er wird $\nabla_{\mathbf{x}} f(\mathbf{x})$ geschrieben. Das Element i des Gradienten ist die partielle Ableitung von f bezüglich x_i . In mehreren Dimensionen sind kritische Punkte die Punkte, in denen jedes Element des Gradienten gleich Null ist.

Die **Richtungsableitung** in Richtung \mathbf{u} (ein Einheitsvektor) ist die Steigung der Funktion f in Richtung \mathbf{u} . Mit anderen Worten: Mit anderen Worten ist die Richtungsableitung die Ableitung der Funktion $f(\mathbf{x} + \alpha \mathbf{u})$ bezüglich α , berechnet für $\alpha = 0$. Verwenden wir die Kettenregel, dann

können wir sehen, dass $\frac{\partial}{\partial \alpha} f(\mathbf{x} + \alpha \mathbf{u})$ sich zu $\mathbf{u}^\top \nabla_{\mathbf{x}} f(\mathbf{x})$ berechnet, wenn $\alpha = 0$ ist.

Um f zu minimieren, müssen wir die Richtung finden, in der f am schnellsten abnimmt. Dazu nutzen wir die Richtungsableitung:

$$\min_{\mathbf{u}, \mathbf{u}^\top \mathbf{u} = 1} \mathbf{u}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) \quad (4.3)$$

$$= \min_{\mathbf{u}, \mathbf{u}^\top \mathbf{u} = 1} \|\mathbf{u}\|_2 \|\nabla_{\mathbf{x}} f(\mathbf{x})\|_2 \cos \theta. \quad (4.4)$$

Dabei ist θ der Winkel zwischen \mathbf{u} und dem Gradienten. Durch Einsetzen von $\|\mathbf{u}\|_2 = 1$ und Ignorieren der nicht von \mathbf{u} abhängigen Faktoren wird die Gleichung vereinfacht zu $\min_{\mathbf{u}} \cos \theta$. Sie ist minimiert, wenn \mathbf{u} in die dem Gradienten entgegengesetzte Richtung weist. Anders ausgedrückt: Der Gradient zeigt direkt aufwärts, der negative Gradient direkt abwärts. Wir können f vermindern, indem wir in Richtung des negativen Gradienten gehen. Dies wird **Gradientenabstiegsverfahren** (auch »Verfahren des steilsten Abstiegs«) genannt.

Der steilste Abstieg führt zu einem neuen Punkt

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}). \quad (4.5)$$

Dabei ist ϵ die **Lernrate**, ein positiver Skalar, der die Schrittweite bestimmt. Wir können ϵ auf verschiedene Weise wählen. Eine gängige Herangehensweise ist es, eine kleine Konstante für ϵ zu wählen. Manchmal können wir die Schrittweite ermitteln, bei der die Richtungsableitung verschwindet. Eine weitere Herangehensweise ist die Berechnung von $f(\mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}))$ für mehrere Werte von ϵ ; anschließend wird der Wert ausgewählt, der zum kleinsten objektiven Funktionswert führt. Die letztgenannte Vorgehensweise wird als **Liniensuche** bezeichnet.

Der steilste Abstieg konvergiert, wenn jedes Element des Gradienten Null ist (oder, in der Praxis, beinahe Null). In einigen Fällen können wir den iterativen Algorithmus überspringen und den kritischen Punkt direkt anhand der Gleichung $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$ für \mathbf{x} ermitteln.

Das Gradientenabstiegsverfahren kann zwar nur zur Optimierung in kontinuierlichen Räumen genutzt werden, aber das allgemeine Konzept eines wiederholten kleinen Schritts (der ungefähr dem besten kleinen Schritt entspricht) in Richtung einer besseren Konfiguration kann auch auf diskrete Räume ausgeweitet werden. Der Anstieg einer Zielfunktion mit diskreten Parametern wird als **Bergsteigeralgorithmus** bezeichnet (Russel und Norvig, 2003).

4.3.1 Über den Gradienten hinaus: Jacobi- und Hesse-Matrizen

Manchmal müssen wir alle partiellen Ableitungen einer Funktion suchen, deren Eingangs- und Ausgangsdaten Vektoren sind. Eine Matrix, die all diese partiellen Ableitungen enthält, wird **Jacobi-Matrix** genannt. Für eine Funktion $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ ist die Jacobi-Matrix $\mathbf{J} \in \mathbb{R}^{n \times m}$ von f wie folgt definiert: $J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$.

Manchmal interessiert uns auch die Ableitung einer Ableitung. Man spricht hier von einer **zweiten Ableitung**. Ein Beispiel: Für eine Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$ wird die Ableitung bezüglich x_i der Ableitung von f bezüglich x_j als $\frac{\partial^2}{\partial x_i \partial x_j} f$ notiert. In einer einzelnen Dimension können wir für $\frac{d^2}{dx^2} f$ auch $f''(x)$ schreiben. Die zweite Ableitung gibt an, wie die erste Ableitung sich bei variierenden Eingangsdaten ändert. Das ist wichtig, weil wir so herausfinden können, ob ein Gradientenschritt zu einem Maß der Verbesserung führt, das wir allein auf Grundlage des Gradienten erwarten. Sie können sich die zweite Ableitung als Maß für die **Krümmung** oder Drehrichtung vorstellen. Gegeben sei eine quadratische Funktion (viele Funktionen in der Praxis sind nicht quadratisch, können aber zumindest lokal als quadratisch approximiert werden). Hat eine solche Funktion eine zweite Ableitung der Größe Null, weist sie keinerlei Krümmung auf. Es handelt sich um eine vollkommen flache Linie, deren Wert allein mit dem Gradienten vorhergesagt werden kann. Ist der Gradient 1, dann können wir einen Schritt der Weite ϵ entlang des negativen Gradienten machen; dabei nimmt die Kostenfunktion um ϵ ab. Ist die zweite Ableitung negativ, dreht die Funktion abwärts (konkav) und die Kostenfunktion nimmt um mehr als ϵ ab. Und wenn die zweite Ableitung positiv ist, dreht die Funktion aufwärts (konvex), sodass die Kostenfunktion um weniger als ϵ abnehmen kann. Abbildung 4.4 zeigt die verschiedenen Krümmungen und ihren Einfluss auf die Beziehung zwischen dem mittels Gradienten vorhergesagten und dem tatsächlichen Wert der Kostenfunktion.

Bei Funktionen mit mehreren Eingangsdatendimensionen gibt es viele zweite Ableitungen. Diese Ableitungen können in einer **Hesse-Matrix** zusammengefasst werden. Die Hesse-Matrix $\mathbf{H}(f)(\mathbf{x})$ ist wie folgt definiert:

$$\mathbf{H}(f)(\mathbf{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}). \quad (4.6)$$

Äquivalent gilt, dass die Hesse-Matrix die Jacobi-Matrix des Gradienten ist.

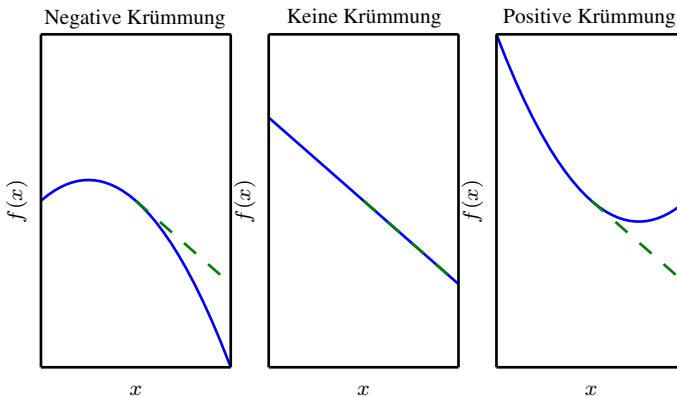


Abbildung 4.4: Die zweite Ableitung gibt die Krümmung einer Funktion an. In der Abbildung werden quadratische Funktionen mit unterschiedlichen Krümmungen gezeigt. Die gestrichelte Linie zeigt den Wert der Kostenfunktion an, den wir rein auf Grundlage der Gradienteninformationen beim Abstieg erwarten. Bei einer negativen Krümmung nimmt die Kostenfunktion schneller ab als anhand des Gradienten erwartet. Ist keinerlei Krümmung vorhanden, sagt der Gradient die Abnahme korrekt vorher. Bei einer positiven Krümmung nimmt die Funktion langsamer ab als erwartet und beginnt schließlich anzusteigen, sodass zu große Schritte die Funktion tatsächlich unbeabsichtigt vergrößern.

Überall wo die zweiten partiellen Ableitungen stetig sind, sind die Differentialoperatoren kommutativ; ihre Reihenfolge kann also vertauscht werden:

$$\frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\mathbf{x}). \quad (4.7)$$

Daraus folgt $H_{i,j} = H_{j,i}$, sodass die Hesse-Matrix in solchen Punkten symmetrisch ist. Die meisten der Funktionen im Deep-Learning-Kontext weisen nahezu überall eine symmetrische Hesse-Matrix auf. Da die Hesse-Matrix reell und symmetrisch ist, können wir sie in eine Menge reeller Eigenwerte und eine orthogonale Basis aus Eigenvektoren zerlegen. Die zweite Ableitung in einer bestimmten, durch einen Einheitsvektor \mathbf{d} vorgegebenen Richtung ergibt sich aus $\mathbf{d}^\top \mathbf{H} \mathbf{d}$. Ist \mathbf{d} ein Eigenvektor von \mathbf{H} , dann ergibt sich die zweite Ableitung in dieser Richtung aus dem entsprechenden Eigenwert. Für andere Richtungen von \mathbf{d} ist die zweite Richtungsableitung ein gewichteter Durchschnittswert aller Eigenwerte mit Gewichten zwischen 0 und 1, wobei Eigenvektoren, die einen kleineren Winkel mit \mathbf{d} aufweisen, ein höheres Gewicht erhalten. Der maximale Eigenwert bestimmt die maximale zweite Ableitung, der minimale Eigenwert bestimmt die minimale zweite Ableitung.

Die zweite (Richtungs-)Ableitung gibt an, wie gut ein Schritt im Gradientenabstiegsverfahren abschneidet. Wir können mit einer Taylorreihe zweiter Ordnung die Funktion $f(\mathbf{x})$ an dem aktuellen Punkt $\mathbf{x}^{(0)}$ approximieren:

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{g} + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{H}(\mathbf{x} - \mathbf{x}^{(0)}). \quad (4.8)$$

Dabei ist \mathbf{g} der Gradient und \mathbf{H} die Hesse-Matrix in $\mathbf{x}^{(0)}$. Bei einer Lernrate von ϵ ergibt sich der neue Punkt \mathbf{x} aus $\mathbf{x}^{(0)} - \epsilon\mathbf{g}$. Durch Einsetzen in unsere Approximation erhalten wir

$$f(\mathbf{x}^{(0)} - \epsilon\mathbf{g}) \approx f(\mathbf{x}^{(0)}) - \epsilon\mathbf{g}^\top \mathbf{g} + \frac{1}{2}\epsilon^2\mathbf{g}^\top \mathbf{H}\mathbf{g}. \quad (4.9)$$

Diese Gleichung enthält drei Terme: Den Originalwert der Funktion, die aus der Steigung der Funktion erwartete Verbesserung und die Korrektur, die wir anwenden müssen, um die Krümmung der Funktion zu berücksichtigen. Ist dieser letzte Term zu groß, kann der Schritt im Gradientenabstiegsverfahren sogar ansteigen. Ist $\mathbf{g}^\top \mathbf{H}\mathbf{g}$ gleich Null oder negativ, sagt die Taylorreihen-Approximation vorher, dass ein unendlicher Anstieg von ϵ einen unendlichen Abstieg von f bedeutet. In der Praxis sind Taylorreihen nur selten für große ϵ dauerhaft genau, sodass in diesem Fall eine heuristischere Wahl für ϵ getroffen werden muss. Ist $\mathbf{g}^\top \mathbf{H}\mathbf{g}$ positiv, führt das Auflösen der optimalen Schrittweite zur stärksten Verminderung der Taylorreihen-Approximation der Funktion zu

$$\epsilon^* = \frac{\mathbf{g}^\top \mathbf{g}}{\mathbf{g}^\top \mathbf{H}\mathbf{g}}. \quad (4.10)$$

Im ungünstigsten Fall (wenn \mathbf{g} mit dem Eigenvektor von \mathbf{H} zusammenfällt, was einem maximalen Eigenwert λ_{\max} entspricht), ergibt sich die optimale Schrittweite aus $\frac{1}{\lambda_{\max}}$. Sofern eine gute Approximation für die minimierte Funktion mithilfe einer quadratischen Funktion möglich ist, bestimmen die Eigenwerte der Hesse-Matrix daher das Maß der Lernrate.

Mit der zweiten Ableitung lässt sich ermitteln, ob ein kritischer Punkt ein lokales Maximum, ein lokales Minimum oder ein Sattelpunkt ist. Sie wissen bereits, dass $f'(x) = 0$ ist. Ist die zweite Ableitung $f''(x) > 0$, nimmt die erste Ableitung $f'(x)$ zu, wenn wir uns nach rechts bewegen, und nimmt ab, wenn wir uns nach links bewegen. Somit gilt $f'(x - \epsilon) < 0$ und $f'(x + \epsilon) > 0$ für ausreichend kleine ϵ . Anders ausgedrückt: Wenn wir uns nach rechts bewegen, beginnt die Steigung rechts nach oben zu weisen; wenn wir uns nach links bewegen, beginnt sie links nach oben zu weisen. Somit können wir für $f'(x) = 0$ und $f''(x) > 0$ schließen, dass x ein lokales Minimum ist. Ebenso können wir für $f'(x) = 0$ und $f''(x) < 0$ schließen, dass x ein

lokales Maximum ist. Man spricht hier von der **Überprüfung der zweiten Ableitung**. Leider ist die Überprüfung für $f''(x) = 0$ ergebnislos, denn in diesem Fall kann es sich bei x um einen Sattelpunkt oder einen Teil eines insgesamt flachen Bereichs handeln.

In mehreren Dimensionen müssen wir alle zweiten Ableitungen der Funktion untersuchen. Anhand der Eigenwertzerlegung der Hesse-Matrix können wir die Überprüfung der zweiten Ableitung für mehrere Dimensionen generalisieren. In einem kritischen Punkt mit $\nabla_x f(x) = 0$ können wir die Eigenwerte der Hesse-Matrix untersuchen, um zu bestimmen, ob es sich bei dem kritischen Punkt um ein lokales Maximum, ein lokales Minimum oder einen Sattelpunkt handelt. Ist die Hesse-Matrix positiv definit (all ihre Eigenwerte sind positiv), handelt es sich um ein lokales Minimum. Das folgt aus der Tatsache, dass die zweite Richtungsableitung in jeder Richtung positiv sein muss, und durch Verweis auf die Überprüfung der eindimensionalen zweiten Ableitung. Ist die Hesse-Matrix dagegen negativ definit (alle ihre Eigenwerte sind negativ), handelt es sich um ein lokales Maximum. In mehreren Dimensionen ist es in der Tat möglich, in bestimmten Fällen einen positiven Beweis für Sattelpunkte zu ermitteln. Wenn mindestens ein Eigenwert positiv und ein weiterer negativ ist, wissen wir, dass x ein lokales Maximum in einem Querschnitt von f und gleichzeitig ein lokales Minimum in einem weiteren Querschnitt ist. Abbildung 4.5 enthält ein Beispiel. Schließlich kann die Überprüfung einer mehrdimensionalen zweiten Ableitung ebenso wie die der eindimensionalen ergebnislos bleiben. Die Überprüfung ist immer dann ergebnislos, wenn alle von Null verschiedenen Eigenwerte dasselbe Vorzeichen aufweisen und mindestens ein Eigenwert Null ist. Das liegt daran, dass die Überprüfung der eindimensionalen zweiten Ableitung im Querschnitt des Null-Eigenwerts ergebnislos ist.

In mehreren Dimensionen gibt es für jede Richtung in einem einzelnen Punkt eine andere zweite Ableitung. Die Konditionszahl der Hesse-Matrix in diesem Punkt gibt an, wie stark sich die zweiten Ableitungen voneinander unterscheiden. Wenn die Hesse-Matrix eine schlechte Konditionszahl aufweist, verläuft das Gradientenabstiegsverfahren schlecht. Das liegt daran, dass die Ableitung in einer Richtung schnell, in der anderen jedoch nur langsam ansteigt. Beim Gradientenabstiegsverfahren bleibt diese Änderung in der Ableitung unbemerkt, sodass nicht klar ist, dass die Untersuchung vorzugsweise in der Richtung erfolgen sollte, in der die Ableitung länger negativ bleibt. Eine schlechte Konditionszahl erschwert auch die Wahl einer guten Schrittweite. Die Schrittweite muss klein genug sein, um ein Übersteigen des Minimums und damit einen Anstieg in Richtungen mit stark positiver Krümmung zu vermeiden. Das bedeutet normalerweise, dass die

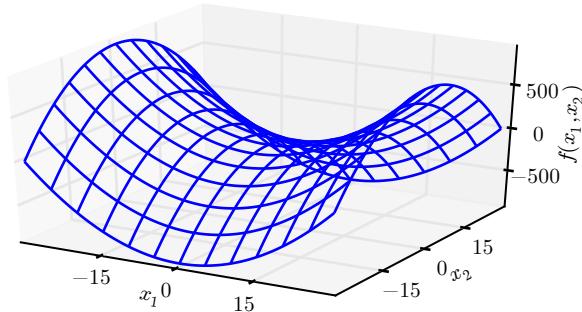


Abbildung 4.5: Ein Sattelpunkt, der gleichzeitig eine positive und negative Krümmung aufweist. Die Funktion hier lautet $f(\mathbf{x}) = x_1^2 - x_2^2$. Entlang der Achse für x_1 ist die Funktion konvex. Diese Achse ist ein Eigenvektor der Hesse-Matrix und hat einen positiven Eigenwert. Entlang der Achse für x_2 ist die Funktion konkav. Diese Richtung ist ein Eigenvektor der Hesse-Matrix mit negativem Eigenwert. Die Bezeichnung »Sattelpunkt« ist von der Form der Funktion abgeleitet, die einem Sattel gleicht. Die Abbildung zeigt das Musterbeispiel einer Funktion mit Sattelpunkt. In mehr als einer Dimension kann es auch ohne einen Eigenwert von 0 einen Sattelpunkt geben – dazu müssen nur sowohl positive als auch negative Eigenwerte vorliegen. Sie können sich einen Sattelpunkt mit Eigenwerten beider Vorzeichen als lokales Maximum in einem Querschnitt und lokales Minimum im anderen Querschnitt vorstellen.

Schrittweite zu klein für merkliche Fortschritte in anderen Richtungen mit geringerer Krümmung ist. Abbildung 4.6 zeigt ein Beispiel.

Zur Lösung des Problems verwenden wir Angaben aus der Hesse-Matrix als Richtschnur für die Suche. Dazu gibt es ein besonders einfaches Verfahren, das **Newton-Verfahren**. Es verwendet eine Entwicklung einer Taylorreihe zweiter Ordnung, um $f(\mathbf{x})$ in der Nähe eines Punktes $\mathbf{x}^{(0)}$ zu approximieren:

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^\top \nabla_{\mathbf{x}} f(\mathbf{x}^{(0)}) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{H}(f)(\mathbf{x}^{(0)}) (\mathbf{x} - \mathbf{x}^{(0)}). \quad (4.11)$$

Wenn wir die kritischen Punkte dieser Funktion bestimmen, erhalten wir

$$\mathbf{x}^* = \mathbf{x}^{(0)} - \mathbf{H}(f)(\mathbf{x}^{(0)})^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}^{(0)}). \quad (4.12)$$

Ist f eine positiv definite quadratische Funktion, besteht das Newton-Verfahren darin, Gleichung 4.12 einmal anzuwenden, um direkt zum Minimum der Funktion zu springen. Ist f nicht wirklich quadratisch, kann aber lokal als positiv definit quadratisch approximiert werden, besteht das Newton-Verfahren in der mehrmaligen Anwendung von Gleichung 4.12. Durch iteratives Aktualisieren der Approximation und Springen zum Minimum der Approximation wird der kritische Punkt sehr viel schneller als mit dem

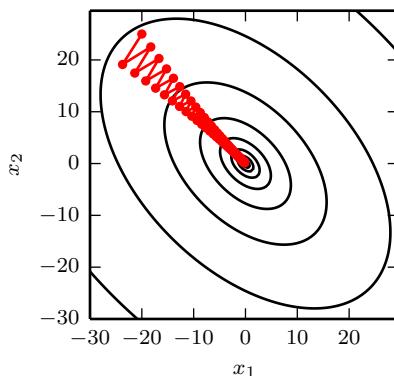


Abbildung 4.6: Das Gradientenabstiegsverfahren kann die Krümmungsinformationen aus der Hesse-Matrix nicht ermitteln. Hier haben wir das Gradientenabstiegsverfahren zum Minimieren einer quadratischen Funktion $f(\mathbf{x})$ verwendet, deren Hesse-Matrix die Konditionszahl 5 aufweist. Dieser Wert bedeutet, dass die Richtung mit der stärksten Krümmung fünf Mal mehr gekrümmmt ist als die Richtung mit der geringsten Krümmung. In diesem Fall liegt die stärkste Krümmung in der Richtung $[1, 1]^\top$ und die geringste Krümmung in der Richtung $[1, -1]^\top$. Die roten Linien zeigen den Pfad für das Gradientenabstiegsverfahren an. Diese sehr lang gestreckte quadratische Funktion erinnert an eine lange Schlucht. Das Gradientenabstiegsverfahren verschwendet seine Zeit damit, wiederholt die Schluchtwände hinabzuklettern, da diese das steilste Merkmal sind. Da die Schrittweite geringfügig zu groß gewählt wurde, schießt sie tendenziell über den Tiefpunkt der Funktion hinaus und steigt daher in der nächsten Iteration die gegenüberliegende Schluchtwand hinab. Der große, positive Eigenwert der Hesse-Matrix, der dem in diese Richtung weisenden Eigenvektor entspricht, deutet darauf hin, dass diese Richtungsableitung rapide anwächst. Daher könnte ein Optimierungsalgorithmus auf Basis der Hesse-Matrix vorhersagen, dass die steilste Richtung in diesem Fall nicht wirklich vielversprechend ist.

Gradientenabstiegsverfahren erreicht. In der Nähe eines lokalen Minimums ist diese Eigenschaft sehr nützlich, doch in der Nähe eines Sattelpunkts möglicherweise von Nachteil. Wie in Abschnitt 8.2.3 gezeigt wird, ist das Newton-Verfahren nur dann geeignet, wenn der naheliegende kritische Punkt ein Minimum ist (alle Eigenwerte der Hesse-Matrix sind positiv). Dagegen strebt das Gradientenabstiegsverfahren Sattelpunkten nur zu, wenn der Gradient auf diese weist.

Optimierungsalgorithmen, die ausschließlich den Gradienten verwenden – z. B. das Gradientenabstiegsverfahren –, werden als **Optimierungsalgorithmen erster Ordnung** bezeichnet. Optimierungsalgorithmen, die auch die Hesse-Matrix verwenden – z.B. das Newton-Verfahren –, werden als

Optimierungsalgorithmen zweiter Ordnung bezeichnet (Nocedal und Wright, 2006).

Die in diesem Buch meist eingesetzten Optimierungsalgorithmen sind auf ein breites Spektrum von Funktionen anwendbar, geben aber nahezu keine Garantien. Deep-Learning-Algorithmen neigen dazu, keine Garantien zu geben, da die für das Deep Learning verwendeten Funktionsfamilien recht kompliziert sind. In vielen anderen Gebieten werden für die Optimierung meist Optimierungsalgorithmen für eine eingeschränkte Funktionsfamilie entwickelt.

Im Deep-Learning-Kontext erhalten wir manchmal gewisse Garantien, indem wir uns selbst auf Funktionen einschränken, die entweder eine **Lipschitz-Stetigkeit** oder Lipschitz-stetige Ableitungen aufweisen. Eine Lipschitz-stetige Funktion ist eine Funktion f , deren Veränderungsrate durch eine **Lipschitz-Konstante \mathcal{L}** eingeschränkt wird:

$$\forall \mathbf{x}, \forall \mathbf{y}, |f(\mathbf{x}) - f(\mathbf{y})| \leq \mathcal{L} \|\mathbf{x} - \mathbf{y}\|_2. \quad (4.13)$$

Diese Eigenschaft ist nützlich, da wir damit unsere Vermutung quantifizieren können, dass eine kleine Änderung der Eingangsdaten durch einen Algorithmus wie das Gradientenabstiegsverfahren zu einer kleinen Änderung in den Ausgangsdaten führt. Die Lipschitz-Stetigkeit ist außerdem eine recht schwache Bedingung; viele Optimierungsprobleme für das Deep Learning können mit relativ geringfügigen Änderungen Lipschitz-stetig gemacht werden.

Das vermutlich erfolgreichste Gebiet spezialisierter Optimierungen ist die **konvexe Optimierung**. Konvexe Optimierungsalgorithmen sind in der Lage, viele weitere Garantien anhand stärkerer Einschränkungen zu geben. Diese Algorithmen lassen sich nur auf konvexe Funktionen anwenden, deren Hesse-Matrix überall positiv semidefinit ist. Derartige Funktionen verhalten sich angemessen, da sie keine Sattelpunkte aufweisen und alle lokalen Minima notwendigerweise globale Minima sind. Allerdings lassen sich die meisten Deep-Learning-Probleme nur schwerlich als konvexe Optimierung ausdrücken. Daher wird die konvexe Optimierung nur als Subroutine in einigen Deep-Learning-Algorithmen eingesetzt. Bei der Untersuchung konvexer Optimierungsalgorithmen erarbeitete Konzepte können beim Beweis der Konvergenz von Deep-Learning-Algorithmen nützlich sein, doch generell ist die konvexe Optimierung im Deep-Learning-Kontext nur von sehr geringer Bedeutung. Boyd und Vandenberghe (2004) oder Rockafellar (1997) gehen näher auf die konvexe Optimierung ein.

4.4 Optimierung unter Nebenbedingungen

Manchmal möchten wir nicht nur eine Funktion $f(\mathbf{x})$ über alle möglichen Werte von \mathbf{x} maximieren oder minimieren, sondern auch alle größten und kleinsten Werte von $f(\mathbf{x})$ für Werte von \mathbf{x} in einer Menge \mathbb{S} ermitteln. Dies wird **Optimierung unter Nebenbedingungen** genannt. Punkte \mathbf{x} , die in der Menge \mathbb{S} liegen, werden in der Terminologie der Optimierung unter Nebenbedingungen als **zulässige** Datenpunkte bezeichnet.

Häufig suchen wir nach einer Lösung, die in gewisser Weise klein ist. Dazu können wir eine Bedingung festlegen, z. B. $\|\mathbf{x}\| \leq 1$.

Ein einfacher Ansatz der Optimierung unter Nebenbedingungen besteht darin, das das Gradientenabstiegsverfahren so zu modifizieren, dass die Bedingung berücksichtigt wird. Wenn wir eine kleine, konstante Schrittweite ϵ verwenden, können wir die Schritte im Gradientenabstiegsverfahren vornehmen und das Ergebnis in \mathbb{S} einspeisen. Im Rahmen einer Liniensuche können wir nur über die Schrittweite ϵ suchen, die zu neuen \mathbf{x} -Datenpunkten führt, die zulässig sind. Alternativ können wir jeden Punkt auf der Linie wieder in den eingeschränkten Bereich projizieren. Wenn möglich, lässt sich dieses Verfahren effizienter gestalten, indem der Gradient in den Tangentenraum des zulässigen Bereichs projiziert wird, bevor der Schritt oder die Liniensuche begonnen wird (*Rosen, 1960*).

Eine elegante Herangehensweise besteht darin, ein anderes Optimierungsproblems ohne Nebenbedingungen zu konzipieren, dessen Lösung für das ursprüngliche Optimierungsproblem unter Nebenbedingungen abgewandelt werden kann. Angenommen, wir möchten $f(\mathbf{x})$ für $\mathbf{x} \in \mathbb{R}^2$ mit der Bedingung minimieren, dass die L^2 -Norm von \mathbf{x} exakt Eins ist, dann können wir auch $g(\theta) = f([\cos \theta, \sin \theta]^\top)$ bezüglich θ minimieren und dann $[\cos \theta, \sin \theta]$ als Lösung für das ursprüngliche Problem angeben. Dieser Ansatz erfordert Kreativität. Die Transformation zwischen Optimierungsproblemen muss für jeden Fall gesondert erdacht werden.

Das **Karush–Kuhn–Tucker**-Verfahren (KKT)¹ bietet eine sehr allgemeine Lösung für die Optimierung unter Nebenbedingungen. Mit dem KKT-Verfahren führen wir eine neue Funktion ein, genannt **Lagrange** oder **allgemeine Lagrange-Funktion**.

Zum Definieren der Lagrange-Funktion müssen wir zunächst \mathbb{S} in Form von Gleichungen und Ungleichungen beschreiben. Wir suchen eine Beschreibung von \mathbb{S} in Form von m -Funktionen $g^{(i)}$ und n -Funktionen $h^{(j)}$, sodass

¹ Das KKT-Verfahren generalisiert das Verfahren der **Lagrange-Multiplikatoren**, die Gleichheitsbedingungen, aber keine Ungleichheitsbedingungen erlauben.

gilt $\mathbb{S} = \{\boldsymbol{x} \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0 \text{ und } \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\}$. Die Gleichungen, in denen $g^{(i)}$ enthalten sind, werden **Gleichheitsbedingungen** genannt, die Ungleichungen mit $h^{(j)}$ **Ungleichheitsbedingungen**.

Wir führen die neuen Variablen λ_i und α_j für jede Bedingung ein; dies sind die KKT-Multiplikatoren. Die allgemeine Lagrange-Funktion wird dann definiert als

$$L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \quad (4.14)$$

Wir können nun ein Minimierungsproblem unter Nebenbedingungen anhand der Optimierung ohne Nebenbedingungen der allgemeinen Lagrange-Funktion lösen. Sofern mindestens ein zulässiger Datenpunkt existiert und $f(\boldsymbol{x})$ nicht den Wert ∞ annehmen darf, hat

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \alpha \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \quad (4.15)$$

denselben optimalen Zielfunktionswert und dieselbe Menge der optimalen Datenpunkte \boldsymbol{x} wie

$$\min_{\boldsymbol{x} \in \mathbb{S}} f(\boldsymbol{x}). \quad (4.16)$$

Das liegt daran, dass jedes Mal, wenn die Bedingungen erfüllt sind,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \alpha \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\boldsymbol{x}) \quad (4.17)$$

ist, während bei jeder Verletzung einer Bedingung

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \alpha \geq 0} L(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty \quad (4.18)$$

ist. Diese Eigenschaften garantieren, dass kein unzulässiger Datenpunkt optimal sein kann und dass das Optimum innerhalb der zulässigen Datenpunkte unverändert bleibt.

Für eine Maximierung unter Nebenbedingungen können wir die allgemeine Lagrange-Funktion $-f(\boldsymbol{x})$ formulieren und so zu diesem Optimierungsproblem gelangen:

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \alpha \geq 0} -f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \quad (4.19)$$

Wir können hieraus auch ein Problem mit Maximierung in der äußeren Schleife machen:

$$\max_{\boldsymbol{x}} \min_{\boldsymbol{\lambda}} \min_{\boldsymbol{\alpha}, \alpha \geq 0} f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) - \sum_j \alpha_j h^{(j)}(\boldsymbol{x}). \quad (4.20)$$

Das Vorzeichen des Terms für die Gleichheitsbedingung ist ohne Bedeutung – wir können nach Belieben Addition oder Subtraktion für die Definition verwenden, da die Optimierung frei zwischen den Vorzeichen für jedes λ_i wählen kann.

Die Ungleichheitsbedingungen sind besonders interessant. Wir sagen, eine Bedingung $h^{(i)}(\mathbf{x})$ ist **aktiv**, wenn $h^{(i)}(\mathbf{x}^*) = 0$ ist. Ist eine Bedingung nicht aktiv, dann bleibt die mit dieser Bedingung gefundene Lösung für das Problem mindestens eine lokale Lösung, wenn die Bedingung entfernt wird. Es ist möglich, dass eine inaktive Bedingung andere Lösungen ausschließt. Beispiel: Ein konkaves Problem mit einem vollständigen Bereich global optimaler Datenpunkte (ein breiter, flacher Bereich mit identischen Kosten) könnte eine Teilmenge des Bereichs aufweisen, das durch Bedingungen ausgeklammert wurde. Und ein nichtkonkaves Problem könnte bessere lokale stationäre Punkte durch eine Bedingung ausschließen, die bei der Konvergenz inaktiv ist. Dennoch ist der bei der Konvergenz ermittelte Punkt unabhängig davon stationär, ob die inaktiven Bedingungen eingeschlossen werden oder nicht. Da ein inaktives $h^{(i)}$ einen negativen Wert aufweist, gilt für die Lösung von $\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ automatisch $\alpha_i = 0$. Wir können also für die Lösung festhalten, dass $\boldsymbol{\alpha} \odot \mathbf{h}(\mathbf{x}) = \mathbf{0}$ ist. Mit anderen Worten: Wir wissen für alle i , dass mindestens eine der Bedingungen $\alpha_i \geq 0$ oder $h^{(i)}(\mathbf{x}) \leq 0$ in der Lösung aktiv sein muss. Um ein gewisses Gespür für dieses Konzept zu entwickeln, können wir sagen, dass die Lösung entweder auf der durch die Ungleichung vorgegebenen Grenze liegt und wir deren KKT-Multiplikator als Parameter für die Lösung \mathbf{x} einbeziehen müssen oder dass die Ungleichung keinen Einfluss auf die Lösung hat und wir dies durch Eliminieren des KKT-Multiplikators zeigen.

Eine einfache Menge von Eigenschaften beschreibt die optimalen Datenpunkte der Optimierungsprobleme unter Nebenbedingungen. Diese Eigenschaften werden als Karush-Kuhn-Tucker-Bedingungen (KKT-Bedingungen) bezeichnet (*Karush, 1939; Kuhn und Tucker, 1951*). Es sind notwendige Bedingungen, aber nicht immer hinreichende Bedingungen, damit ein Punkt optimal sein kann. Die Bedingungen lauten:

- Der Gradient der allgemeinen Lagrange-Funktion ist Null.
- Alle Bedingungen für \mathbf{x} und die KKT-Multiplikatoren sind erfüllt.
- Die Ungleichheitsbedingungen weisen »komplementären Schlupf« auf (Komplementaritätsbedingung): $\boldsymbol{\alpha} \odot \mathbf{h}(\mathbf{x}) = \mathbf{0}$.

Nocedal und Wright (2006) enthält weitere Informationen zum KKT-Verfahren.

4.5 Beispiel: Lineare kleinste Quadrate

Wir suchen das \mathbf{x} , das folgende Funktion minimiert:

$$f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2. \quad (4.21)$$

Spezialisierte Algorithmen der linearen Algebra lösen das Problem effizient. Allerdings können wir auch untersuchen, wie eine Lösung mithilfe der Optimierung auf Gradientenbasis aussieht, und so an einem einfachen Beispiel zeigen, wie diese Verfahren funktionieren.

Zuerst müssen wir den Gradienten bestimmen:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{A}^\top (\mathbf{Ax} - \mathbf{b}) = \mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b}. \quad (4.22)$$

Nun können wir dem Gradienten in Abwärtsrichtung mit kleinen Schritten folgen. Algorithmus 4.1 enthält Einzelheiten.

Algorithmus 4.1 Ein Algorithmus zum Minimieren von $f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2$ bezüglich \mathbf{x} mithilfe des Gradientenabstiegsverfahrens; Ausgangspunkt ist ein beliebiger Wert für \mathbf{x} .

Für Schrittweite (ϵ) und Toleranz (δ) werden kleine, positive Zahlen gewählt.

```
while ||A⊺Ax - A⊺b||2 > δ do
    x ← x - ε (A⊺Ax - A⊺b)
end while
```

Das Problem lässt sich auch mit dem Newton-Verfahren lösen. In diesem Fall ist – da die wahre Funktion quadratisch ist – die quadratische Approximation aus dem Newton-Verfahren exakt und der Algorithmus konvergiert in nur einem Schritt gegen das globale Minimum.

Nun wollen wir dieselbe Funktion erneut minimieren, allerdings unter der Bedingung $\mathbf{x}^\top \mathbf{x} \leq 1$. Dazu führen wir die Lagrange-Funktion ein:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda (\mathbf{x}^\top \mathbf{x} - 1). \quad (4.23)$$

Nun können wir das Problem lösen:

$$\min_{\mathbf{x}} \max_{\lambda, \lambda \geq 0} L(\mathbf{x}, \lambda). \quad (4.24)$$

Die Kleinst-Norm-Lösung für das Kleinste-Quadrat-Problem ohne Nebenbedingungen lässt sich mithilfe der Moore-Penrose-Pseudoinverse

bestimmen: $\mathbf{x} = \mathbf{A}^+ \mathbf{b}$. Ist dieser Datenpunkt zulässig, dann stellt er die Lösung für das Problem unter Nebenbedingungen dar. Ansonsten müssen wir eine Lösung finden, bei der die Bedingung aktiv ist. Durch Differenzieren der Lagrange-Funktion für \mathbf{x} erhalten wir die Gleichung

$$\mathbf{A}^\top \mathbf{A} \mathbf{x} - \mathbf{A}^\top \mathbf{b} + 2\lambda \mathbf{x} = 0. \quad (4.25)$$

Das zeigt uns, dass die Lösung die folgende Form annimmt:

$$\mathbf{x} = (\mathbf{A}^\top \mathbf{A} + 2\lambda \mathbf{I})^{-1} \mathbf{A}^\top \mathbf{b}. \quad (4.26)$$

Die Größe von λ muss so gewählt werden, dass das Ergebnis die Bedingung erfüllt. Wir finden diesen Wert mithilfe des Gradientenanstiegs für λ . Dafür gilt

$$\frac{\partial}{\partial \lambda} L(\mathbf{x}, \lambda) = \mathbf{x}^\top \mathbf{x} - 1. \quad (4.27)$$

Wenn die Norm für \mathbf{x} den Wert 1 überschreitet, ist diese Ableitung positiv; um ihr in steigender Richtung zu folgen und die Lagrange-Funktion bezüglich λ zu erhöhen, erhöhen wir λ . Da der Koeffizient des $\mathbf{x}^\top \mathbf{x}$ -Strafterms gestiegen ist, führt die Lösung der linearen Gleichung für \mathbf{x} nun zu einer Lösung mit einer kleineren Norm. Das Lösen der linearen Gleichung und das Anpassen von λ geht weiter, bis \mathbf{x} die korrekte Norm aufweist und die Ableitung nach λ gleich 0 ist.

Sie kennen nun die mathematischen Grundlagen für die Entwicklung von Machine-Learning-Algorithmen. Im Folgenden widmen wir uns dem Erstellen und Analysieren einiger vollwertiger Lernsysteme.

5

Grundlagen für das Machine Learning

Deep Learning ist eine spezielle Art des Machine Learnings. Daher ist für ein gutes Verständnis des Deep Learnings eine gute Kenntnis der grundlegenden Prinzipien des Machine Learnings erforderlich. In diesem Kapitel lernen Sie die wichtigsten Grundlagen kennen, auf denen der Rest des Buchs aufbaut. Wenn das Thema für Sie neu ist oder Sie tiefer darin einsteigen möchten, empfehlen wir ein Lehrbuch zum Machine Learning, das diese Grundlagen umfassender behandelt, zum Beispiel *Murphy* (2012) oder *Bishop* (2006). Wenn Sie bereits über das erforderlichen Wissensfundament zum Machine Learning verfügen, können Sie direkt in Abschnitt 5.11 weiterlesen. Dort werden einige Aspekte traditioneller Machine-Learning-Verfahren behandelt, die wesentliche Auswirkungen auf die Entwicklung von Deep-Learning-Algorithmen hatten.

Zu Beginn definieren wir, was ein Lernalgorithmus eigentlich ist, und geben mit dem linearen Regressionsalgorithmus ein Beispiel. Anschließend beschreiben wir die unterschiedlichen Herausforderungen bei der Anpassung der Trainingsdaten und der Ermittlung von Mustern, um neue Daten zu generalisieren. Die meisten Machine-Learning-Algorithmen verfügen über Einstellparameter, die außerhalb des eigentlichen Algorithmus festgelegt werden müssen, sogenannte *Hyperparameter*. Wir zeigen, wie diese zusätzlichen Daten eingestellt werden. Machine Learning ist im Grunde genommen eine Art angewandte Statistik, wobei verstärkt Wert auf die Verwendung von Computern zur statistischen Abschätzung komplizierter Funktionen und weniger Wert auf das Beweisen von Konfidenzintervallen dieser Funktionen gelegt wird. Wir befassen uns daher mit zwei wesentlichen statistischen

Ansätzen, den frequentistischen Schätzern und der Bayesschen Inferenz. Die meisten Machine-Learning-Algorithmen lassen sich in zwei Kategorien einteilen: überwachtes Lernen und unüberwachtes Lernen. Wir beschreiben diese Kategorien und liefern für jede Kategorie einige einfache Lernalgorithmen als Beispiel. Die meisten Deep-Learning-Algorithmen beruhen auf einem Optimierungsalgorithmus, der stochastisches Gradientenabstiegsverfahren genannt wird. Wir erklären, wie aus den Komponenten der verschiedenen Algorithmen – darunter ein Optimierungsalgorithmus, eine Kostenfunktion, ein Modell und ein Datensatz – ein Machine-Learning-Algorithmus wird. Abschließend beschreiben wir in Abschnitt 5.11 einige der Faktoren, die die Fähigkeit zur Generalisierung im traditionellen Machine Learning beschränkt haben. Diese Probleme waren Anlass für die Entwicklung der Deep-Learning-Algorithmen, mit denen sie sich überwinden lassen.

5.1 Lernalgorithmen

Ein Machine-Learning-Algorithmus ist ein Algorithmus, der aus Daten lernen kann. Aber was bedeutet eigentlich »lernen«? Mitchell (1997) liefert eine prägnante Definition: »Für ein Computerprogramm lässt sich sagen, es lerne aus Erfahrung E hinsichtlich einer Klasse von Aufgaben T und einer Leistungsbewertung P , wenn seine Leistungsfähigkeit bezüglich der mit dem Maß P bewerteten Aufgaben aus T mit der Erfahrung E steigt.« Wir können uns eine Vielzahl von Erfahrungen E , Aufgaben T und Leistungsbewertungen P (von engl. *experience*, *task*, und *performance measure*) vorstellen und möchten in diesem Buch keine formale Definition der einzelnen Elemente dieser Klassen vornehmen. Stattdessen bieten wir in den folgenden Abschnitten intuitive Beschreibungen und Beispiele für die unterschiedlichen Aufgaben, Leistungsbewertungen und Erfahrungen, mit denen sich Machine-Learning-Algorithmen konstruieren lassen.

5.1.1 Die Aufgabe T

Mit Machine Learning können wir Aufgaben angehen, die mit starren, von Menschen geschriebenen und konzipierten Programmen nicht gelöst werden können. Aus wissenschaftlicher und philosophischer Sicht ist Machine Learning interessant, da ein Entwickeln unseres Verständnisses hierfür gleichzeitig unser Verständnis der Intelligenz erweitert.

In dieser relativ formalen Definition des Wortes »Aufgabe« ist der Prozess oder Vorgang des Lernens selbst nicht die Aufgabe. Lernen ist unser Mittel

zum Erlangen der Fähigkeit, die für das Durchführen der Aufgabe benötigt wird. Ein Beispiel: Wenn wir einem Roboter das Laufen beibringen möchten, ist das Laufen die Aufgabe. Wir können den Roboter dann entweder so programmieren, dass er das Laufen erlernt, oder wir können versuchen, ein Programm zu schreiben, das genau angibt, wie das Laufen funktioniert.

Bei der Beschreibung von Machine-Learning-Aufgaben wird normalerweise angegeben, wie das Machine-Learning-System ein **Beispiel** verarbeiten soll. Ein Beispiel ist eine Sammlung von **Merkmalen** (engl. *Features*), die quantitativ ermittelt wurden, und zwar anhand eines Objekts oder Ereignisses, das durch das Machine-Learning-System verarbeitet werden soll. Das Beispiel wird normalerweise als Vektor $\mathbf{x} \in \mathbb{R}^n$ repräsentiert, wobei jedes Element x_i des Vektors ein weiteres Merkmal ist. Bei einem Bild sind die Merkmale beispielsweise die Werte der einzelnen Pixel.

Viele Aufgaben lassen sich mithilfe von Machine Learning lösen. Zu den üblichen Machine-Learning-Aufgaben gehören diese:

- **Klassifizierung:** Für diese Aufgabe muss ein Computerprogramm bestimmen, in welche von k Kategorien die Eingabe (Eingangsdaten) gehört. Dazu muss der Lernalgorithmus normalerweise eine Funktion $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$ erstellen. Ist $y = f(\mathbf{x})$, weist das Modell eine durch den Vektor \mathbf{x} beschriebene Eingabe einer durch den Zahlencode y bestimmten Kategorie zu. Es gibt noch weitere Varianten von Klassifizierungsaufgaben, bei denen f zum Beispiel eine Wahrscheinlichkeitsverteilung über Klassen ausgibt. Ein Beispiel für eine Klassifizierungsaufgabe ist die Objekterkennung, bei der die Eingabe ein Bild ist (meist in Form einer Reihe von Helligkeitswerten der Pixel) und die Ausgabe ein Zahlencode, der angibt, was für ein Objekt auf dem Bild zu sehen ist. So ist der Roboter Willow Garage PR2 in der Lage, als Kellner zu arbeiten, der unterschiedliche Getränke erkennen und auf Zuruf servieren kann (*Goodfellow et al.*, 2010). Die moderne Objekterkennung ist für Deep Learning prädestiniert (*Krizhevsky et al.*, 2012; *Ioffe und Szegedy*, 2015). Objekterkennung bildet auch die Grundlage für die Gesichtserkennung (*Taigman et al.*, 2014), mit der automatisch bestimmte Personen in Fotosammlungen markiert werden können; außerdem ermöglicht sie eine natürlichere Interaktion zwischen Computer und Mensch.
- **Klassifizierung mit fehlenden Eingangsdaten:** Der Schwierigkeitsgrad steigt, wenn nicht garantiert werden kann, dass dem Computerprogramm für jedes Element des Eingabevektors das passende Maß vorgegeben werden kann. Zum Lösen der Klassifizierungsaufgabe muss

der Lernalgorithmus *genau* eine Funktionszuordnung von der Vektor-eingabe zu einer kategorischen Ausgabe definieren. Wenn nun einige der Eingangsdaten fehlen, kann der Lernalgorithmus nicht einfach eine einzelne Klassifizierungsfunktion bereitstellen, sondern muss eine *Menge* von Funktionen lernen. Jede Funktion entspricht einer Klassifizierung von x für eine andere Teilmenge fehlender Eingangsdaten. Diese Situation tritt bei medizinischen Diagnosen häufig auf, da viele Arten von Tests viel Geld kosten oder invasiv sind. Eine Möglichkeit zur effizienten Definition einer so großen Funktionsmenge besteht darin, eine Wahrscheinlichkeitsverteilung über alle relevanten Variablen zu erlernen und die Klassifizierungsaufgabe anschließend durch Entfernen der fehlenden Variablen zu lösen. Für n Eingangsvariablen können wir insgesamt 2^n unterschiedliche Klassifizierungsfunktionen für alle möglichen Mengen fehlender Eingangsdaten ermitteln und das Computerprogramm muss lediglich eine einzige Funktion zur Beschreibung der Wahrscheinlichkeitsverteilung erlernen. *Goodfellow et al.* (2013b) enthält ein Beispiel für ein tiefes probabilistisches Modell, das auf diese Art angewandt wird. Viele der anderen in diesem Abschnitt beschriebenen Aufgaben lassen sich ebenso auf fehlende Eingangsdaten beziehen. Die Klassifizierung mit fehlenden Eingangsdaten ist lediglich ein Beispiel für die Möglichkeiten des Machine Learnings.

- **Regression:** Für diese Aufgabe muss ein Computerprogramm anhand von Eingangsdaten einen Zahlenwert vorhersagen. Dazu muss der Lernalgorithmus eine Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ausgeben. Die Aufgabe ähnelt vom Ausgabeformat abgesehen der Klassifizierung. Ein Beispiel für eine Regressionsaufgabe ist die Vorhersage eines Erwartungswerts für die Schadenssumme einer versicherten Person (zur Berechnung der Versicherungsprämie) oder der künftigen Kurse von Wertpapieren. Diese Art von Vorhersagen wird auch für den algorithmischen Handel verwendet.
- **Transkription:** Für diese Aufgabe muss das Machine-Learning-System relativ unstrukturierte Daten in eine diskrete Textform überführen. Ein Beispiel dafür ist die optische Zeichenerkennung, bei der ein Computerprogramm das Foto (oder Bild) eines Textes erhält und diesen in bearbeitbare Zeichenfolgen umwandeln soll (z. B. ASCII oder Unicode). Google Street View verarbeitet mit dieser Art von Deep Learning Hausnummern (*Goodfellow et al.*, 2014d). Ein weiteres Beispiel ist die Spracherkennung, bei der ein Computerprogramm aus einer Audio-Wellenform eine Ausgabe mit Zeichenfolgen oder Word-ID-Codes

erstellen soll, die die gesprochenen Wörter und Sätze der Aufnahme enthält. Deep Learning ist ein unverzichtbarer Bestandteil moderner Spracherkennungssysteme, die in vielen Großunternehmen wie Microsoft, IBM und Google eingesetzt werden (*Hinton et al.*, 2012b).

- **Maschinelle Übersetzung:** Für die maschinelle Übersetzung liegen die Eingangsdaten bereits als Sequenz von Symbolen in einer beliebigen Sprache vor. Das Computerprogramm muss diese in eine Sequenz von Symbolen einer anderen Sprache umwandeln. Meist handelt es sich dabei um natürliche Sprachen wie Deutsch, Englisch oder Französisch und die Übersetzung zwischen diesen. Deep Learning spielt auch in diesem Bereich seit Kurzem eine wichtige Rolle (*Sutskever et al.*, 2014; *Bahdanau et al.*, 2015).
- **Strukturierte Ausgabe:** Aufgaben hinsichtlich einer strukturierten Ausgabe umfassen alle Aufgaben, deren Ausgabe ein Vektor (oder eine andere Datenstruktur mit mehreren Werten) ist, in der wichtige Beziehungen zwischen den einzelnen Elementen vorliegen. Diese Kategorie ist sehr breit aufgestellt und schließt unter anderem die bereits genannten Transkriptions- und Übersetzungsaufgaben sowie viele weitere Aufgaben ein. Ein Beispiel dafür ist das *Parsing*. Dieser Begriff steht für eine Analyse und Zergliederung der Eingangsdaten, zum Beispiel um einen Satz aus einer natürlichen Sprache in eine Baumstruktur zu unterteilen, die seine grammatische Struktur beschreibt. Dabei werden einzelne Knoten des Baums als Verben, Nomen, Adverbien usw. klassifiziert. *Collobert* (2011) enthält ein Beispiel für eine Parsing-Aufgabe mit Deep Learning. Ein weiteres Beispiel ist die pixelweise Einteilung von Bildern, bei der das Computerprogramm jedes Pixel in eine bestimmte Kategorie einstuft. Auf diese Weise kann Deep Learning den Straßenverlauf in Luftbildern bestimmen (*Mnih und Hinton*, 2010). Die Ausgabeform muss nicht unbedingt so stark mit der Struktur der Eingangsdaten übereinstimmen, wie das in diesen Anmerlungsaufgaben der Fall ist. Geht es beispielsweise um Bildunterschriften, untersucht das Computerprogramm ein Bild und gibt dann eine Beschreibung des Bildinhalts in natürlicher Sprache aus (*Kiros et al.*, 2014a,b; *Mao et al.*, 2015; *Vinyals et al.*, 2015b; *Donahue et al.*, 2014; *Karpathy und Li*, 2015; *Fang et al.*, 2015; *Xu et al.*, 2015). Diese Aufgaben werden *strukturierte Ausgabeaufgaben* genannt, da das Programm mehrere Werte, die in einer engen Beziehung zueinander stehen, ausgeben muss. So müssen die Wörter für die Bildunterschrift einen gültigen Satz ergeben.

- **Erkennung von Anomalien:** Für diese Aufgabe durchsucht das Computerprogramm eine Menge von Ereignissen oder Objekten und markiert einige davon als ungewöhnlich oder atypisch. Ein Beispiel für eine Erkennung von Anomalien ist die Erkennung von Kreditkartenbetrug. Durch Modellieren Ihrer Kaufgewohnheiten kann das Kreditkartenunternehmen einen Kartenmissbrauch erkennen. Stiehlt ein Dieb Ihre Kreditkarte oder die Daten, spiegeln dessen Einkäufe oft eine andere Wahrscheinlichkeitsverteilung wider als Ihre eigenen Käufe. Das Kreditkartenunternehmen kann den Betrug verhindern, indem es das Konto sperrt, sobald die Karte für ungewöhnliche Einkäufe verwendet wird. *Chandola et al.* (2009) enthält eine Untersuchung zu Verfahren für die Erkennung von Anomalien.
- **Synthese und Stichprobenverfahren (Sampling):** Für diese Aufgabe muss der Machine-Learning-Algorithmus neue Beispiele erzeugen, die den Trainingsdaten ähneln. Synthese und Stichprobenverfahren via Machine Learning sind zum Beispiel in Medienanwendungen nützlich, wenn das manuelle Erstellen umfangreicher Inhalte viel Geld kostet, langwierig ist oder zu lange dauert. So können Videospiele z.B. automatisch Texturen großer Objekte oder Landschaften generieren, damit nicht ein Grafiker jeden einzelnen Pixel manuell bearbeiten muss (*Luo et al.*, 2013). In manchen Fällen soll abhängig von den Eingangsdaten eine bestimmte Art von Ausgabe aus dem Stichprobenverfahren oder der Synthese entstehen. Bei der Sprachsynthese bestehen die Eingangsdaten zum Beispiel aus einem geschriebenen Satz, den das Programm als Audio-Wellenform ausgeben muss, die die gesprochene Version des Satzes enthält. Auch dies ist eine Art strukturierte Ausgabeaufgabe, jedoch mit der zusätzlichen Qualifikation, dass es nicht die eine korrekte Ausgabe für die Eingangsdaten gibt; tatsächlich ist eine große Streuung in der Ausgabe sogar wünschenswert, damit diese natürlicher und realistischer wirkt.
- **Ersetzen fehlender Werte (auch Imputation):** Für diese Aufgabe steht dem Machine-Learning-Algorithmus ein neues Beispiel $\mathbf{x} \in \mathbb{R}^n$ zur Verfügung, in dem einige Einträge x_i von \mathbf{x} fehlen. Der Algorithmus muss dann die Werte der fehlenden Einträge vorhersagen.
- **Denoising (Entrauschen):** Für diese Aufgabe muss der Machine-Learning-Algorithmus aus den Eingangsdaten eines *beschädigten oder fehlerhaften Beispiels* $\tilde{\mathbf{x}} \in \mathbb{R}^n$, für das die Fehlerursache unbekannt ist, ein *sauberes Beispiel* $\mathbf{x} \in \mathbb{R}^n$ erzeugen. Der Klassifikator muss das

saubere Beispiel \mathbf{x} anhand der beschädigten Version $\tilde{\mathbf{x}}$ voraussagen, d. h., die bedingte Wahrscheinlichkeitsverteilung $p(\mathbf{x} | \tilde{\mathbf{x}})$ vorhersagen.

- **Dichteschätzung oder Wahrscheinlichkeitsschätzung:** Für die Dichteschätzung muss der Machine-Learning-Algorithmus eine Funktion $p_{\text{model}} : \mathbb{R}^n \rightarrow \mathbb{R}$ erlernen, in der $p_{\text{model}}(\mathbf{x})$ als Wahrscheinlichkeitsfunktion (bei stetigem \mathbf{x}) oder als Wahrscheinlichkeitsdichtefunktion (bei diskretem \mathbf{x}) für den Raum, aus dem die Beispiele stammen, interpretiert werden kann. Um diese Aufgabe gut zu erledigen (wir gehen bei der Behandlung der Leistungsbewertung P noch darauf ein, was das genau heißt), muss der Algorithmus die Struktur der beobachteten Daten erlernen. Er muss wissen, wo die Beispiele eng beieinander liegen und wo ihr Auftreten eher unwahrscheinlich ist. Für die meisten der oben beschriebenen Aufgaben muss der Lernalgorithmus die Struktur der Wahrscheinlichkeitsverteilungen zumindest implizit erfassen. Für die Dichteschätzung muss die Verteilung explizit erfasst werden. Dann können wir prinzipiell Berechnungen für die Verteilung durchführen, mit der sich auch die anderen Aufgaben lösen lassen. Beispiel: Wenn wir anhand der Dichteschätzung eine Wahrscheinlichkeitsverteilung $p(\mathbf{x})$ ermittelt haben, können wir diese für die Imputation fehlender Werte verwenden. Fehlt ein Wert x_i und sind alle anderen Werte \mathbf{x}_{-i} bekannt, wissen wir, dass sich die Verteilung aus $p(x_i | \mathbf{x}_{-i})$ ergibt. In der Praxis ermöglicht die Dichteschätzung nicht immer das Lösen aller verbundenen Aufgaben, da die erforderlichen Operationen für $p(\mathbf{x})$ häufig nicht effizient berechenbar (engl. *intractable*) sind.

Natürlich gibt es noch viele weitere Aufgaben und Aufgabentypen. Die obige Aufstellung ist nur als Beispiel dafür gedacht, wozu Machine Learning in der Lage ist. Es handelt sich nicht um eine abschließende Aufzählung oder starre Taxonomie.

5.1.2 Die Leistungsbewertung P

Um die Fähigkeiten eines Machine-Learning-Algorithmus zu bewerten, benötigen wir ein Maß zur quantitativen Leistungsbewertung. Diese Leistungsbewertung P wird normalerweise speziell für die durch das System ausgeführte Aufgabe T festgelegt.

Für Aufgaben wie die Klassifizierung, die Klassifizierung mit fehlenden Eingangsdaten und die Transkription messen wir meist die **Korrektklassifikationsrate** des Modells. Als Korrektklassifikationsrate wird der Anteil der

Beispiele bezeichnet, für die das Modell die korrekte Ausgabe erzeugt. Wir erhalten entsprechende Angaben auch durch Ermitteln der **Fehlerquote** (auch Fehlerrate genannt), das ist der Anteil der Beispiele, für die das Modell eine inkorrekte Ausgabe erzeugt. Die Fehlerquote wiederum wird häufig als erwarteter 0-1-Verlust bezeichnet. Der 0-1-Verlust für ein bestimmtes Beispiel ist 0, wenn es korrekt klassifiziert wird; andernfalls ist er 1.

Bei Aufgaben wie der Dichteschätzung ist die Messung der Korrektklassifikationsrate, Fehlerquote oder anderer Maße für den 0-1-Verlust nicht nützlich. Stattdessen müssen wir ein anderes Leistungsmaß verwenden, das dem Modell für jedes Beispiel eine stetigwertige Punktzahl gibt. Eine verbreitete Herangehensweise besteht darin, die durchschnittliche Log-Wahrscheinlichkeit (engl. *log-probability*) anzugeben, die das Modell einigen Beispielen zuweist.

Meist interessieren wir uns dafür, wie gut der Machine-Learning-Algorithmus mit Daten funktioniert, die ihm nicht bekannt waren, da sich daraus sein Erfolg in der Realität ableiten lässt. Wir untersuchen also diese Leistungsbewertung anhand einer **Testmenge** von Daten, die nicht in der Trainingsdatenmenge für das Machine-Learning-System enthalten war.

Die Auswahl der Leistungsbewertung wirkt vielleicht ganz einfach und objektiv, aber häufig ist es schwierig, eine Leistungsbewertung auszuwählen, die das gewünschte Systemverhalten gut widerspiegelt.

Das kann daran liegen, dass bereits die Entscheidung über die zu messende Größe schwerfällt. Ein Beispiel: Ist bei der Bewertung einer Transkriptionsaufgabe das Transkribieren vollständiger Sequenzen wichtiger für die Korrektklassifikationsrate oder sollten wir einen Ansatz verfolgen, bei dem für jede beliebige, korrekt erkannte kurze Sequenz eine gewisse Punktzahl vergeben wird? Sollten bei einer Regressionsaufgabe höhere Strafen für häufige, mittelschwere Fehler oder für seltene, schwere Fehler vergeben werden? Die Antwort hängt von der Anwendung ab.

Es kann aber auch vorkommen, dass wir die zu messende Größe exakt benennen, aber nicht problemlos beobachten können. Das ist häufig im Rahmen der Dichteschätzung der Fall. Viele der besten probabilistischen Modelle repräsentieren Wahrscheinlichkeitsverteilungen nur implizit. Der tatsächliche Wahrscheinlichkeitswert für einen bestimmten Punkt im Raum ist in vielen derartigen Modellen nicht effizient berechenbar. In solchen Fällen müssen wir ein alternatives Kriterium bestimmen, das mit der Zielsetzung übereinstimmt – oder aber eine gute Approximation für das gesuchte Kriterium festlegen.

5.1.3 Die Erfahrung E

Machine-Learning-Algorithmen lassen sich abhängig von der Art der Erfahrungen, denen sie in der Lernphase ausgesetzt sind, grob in **unüberwachte** und **überwachte** Algorithmen einteilen.

Für die meisten Lernalgorithmen in diesem Buch gilt, dass sie einen vollständigen **Datensatz** nutzen. Ein Datensatz ist eine Sammlung mit vielen Beispielen (vgl. die Definition in Abschnitt 5.1.1). Manchmal werden Beispiele auch **Datenpunkte** genannt.

Einer der ältesten Datensätze für Statistiker und Machine-Learning-Forscher ist der Iris-Datensatz (Fisher, 1936). Es handelt sich um eine Sammlung von Messdaten für unterschiedliche Teile von 150 Schwertlilien (engl. *irises*). Jede einzelne Pflanze stellt ein Beispiel dar. Die Merkmale der einzelnen Beispiele sind die Messwerte der Pflanzenbestandteile: Länge des Kelchblatts, Breite des Kelchblatts, Länge des Blütenblatts und Breite des Blütenblatts. Der Datensatz zeigt auch auf, zu welcher von drei Arten die Pflanze gehört.

Algorithmen für unüberwachtes Lernen arbeiten mit einem Datensatz, der viele Merkmale enthält, um nützliche Eigenschaften über die Struktur dieses Datensatzes zu erlernen. Im Deep-Learning-Kontext interessieren wir uns meist für das Erlernen der gesamten Wahrscheinlichkeitsverteilung, aus der der Datensatz erzeugt wurde – entweder explizit wie in der Dichteschätzung oder implizit wie bei Synthese oder Denoising. Einige weitere Algorithmen für unüberwachtes Lernen haben andere Aufgaben wie das Clustering, bei dem der Datensatz in Cluster (Gruppen) ähnlicher Beispiele aufgeteilt wird.

Algorithmen für überwachtes Lernen arbeiten mit einem Datensatz, der ebenfalls Merkmale enthält. Allerdings ist jedes Beispiel zudem mit einem **Label** oder einem **Zielwert** (engl. *target*) versehen. Im Iris-Datensatz ist für jede Schwertlilie zum Beispiel die Art angegeben. Ein Algorithmus für überwachtes Lernen kann den Iris-Datensatz studieren und lernen, Schwertlilien aufgrund ihrer Messwerte in drei verschiedene Arten zu unterteilen.

Grob ausgedrückt besteht unüberwachtes Lernen darin, mehrere Beispiele eines Zufallsvektors \mathbf{x} zu beobachten und zu versuchen, die Wahrscheinlichkeitsverteilung $p(\mathbf{x})$ oder andere wichtige Eigenschaften der Verteilung implizit oder explizit zu erlernen, während beim überwachten Lernen mehrere Beispiele eines Zufallsvektors \mathbf{x} und eines zugehörigen Werts oder Vektors \mathbf{y} beobachtet werden, um zu lernen, \mathbf{y} anhand von \mathbf{x} vorherzusagen, meist

durch Schätzung von $p(\mathbf{y} \mid \mathbf{x})$. Der Begriff **überwachtes Lernen** wird verwendet, da der Zielwert \mathbf{y} von einem Supervisor oder Lehrer bereitgestellt wird, der dem Machine-Learning-System so zeigt, was es tun soll. Beim unüberwachten Lernen gibt es keinen Supervisor oder Lehrer und der Algorithmus muss ohne Hilfestellung einen Sinn in den Daten finden.

Unüberwachtes Lernen und überwachtes Lernen sind keine formal definierten Begriffe. Die Grenze zwischen den beiden Kategorien verläuft oft fließend. Viele Machine-Learning-Technologien können beide Aufgaben erfüllen. So besagt die Produktregel für Wahrscheinlichkeiten, dass die multivariate Verteilung für einen Vektor $\mathbf{x} \in \mathbb{R}^n$ wie folgt zerlegt werden kann:

$$p(\mathbf{x}) = \prod_{i=1}^n p(x_i \mid x_1, \dots, x_{i-1}). \quad (5.1)$$

Diese Zerlegung bedeutet, dass wir das auf den ersten Blick unüberwachte Problem zur Modellierung von $p(\mathbf{x})$ auch in n überwachte Lernprobleme unterteilen können. Alternativ können wir das unüberwachte Lernproblem zum Erlernen von $p(y \mid \mathbf{x})$ mithilfe klassischer Technologien für das unüberwachte Lernen so lösen, dass die multivariate Verteilung $p(\mathbf{x}, y)$ gelernt und anschließend abgeleitet wird

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}. \quad (5.2)$$

Obwohl unüberwachtes Lernen und überwachtes Lernen keine gänzlich formalen oder trennscharfen Konzepte sind, ermöglichen sie uns doch eine grobe Einteilung einiger Dinge, die mit Machine-Learning-Algorithmen machbar sind. Traditionell werden Regressions-, Klassifizierungs- und strukturierte Ausgabeprobleme dem überwachten Lernen zugeordnet. Die Dichteschätzung zur Unterstützung anderer Aufgaben wird dagegen meist als unüberwachtes Lernen angesehen.

Es sind noch weitere Varianten von Lernparadigmen möglich. So gibt es das halb-überwachte Lernen, bei dem einige, aber nicht alle Beispiele einen Zielwert enthalten. Beim Multi-Instanz-Lernen wird für eine gesamte Beispielsammlung angegeben, ob sie Beispiele einer Klasse enthält oder nicht enthält, ohne dass die einzelnen Elemente der Sammlung mit einem Label gekennzeichnet werden. Ein jüngeres Beispiel für das Multi-Instanz-Lernen mit tiefen Modellen finden Sie in *Kotzias et al. (2015)*.

Einige Machine-Learning-Algorithmen arbeiten nicht nur mit einem starren Datensatz. So interagieren Algorithmen für das **Reinforcement Learning** mit einer Umgebung und es kommt zu einer Feedbackschleife

zwischen dem Lernsystem und seinen Erfahrungen. Derartige Algorithmen sprengen den Rahmen dieses Buchs. *Sutton und Barto* (1998) oder *Bertsekas und Tsitsiklis* (1996) bieten weitere Informationen zum Reinforcement Learning und in *Mnih et al.* (2013) wird der Deep-Learning-Ansatz für das Reinforcement Learning näher betrachtet.

Die meisten Machine-Learning-Algorithmen nutzen einfach einen Datensatz. Dabei lässt sich ein Datensatz auf vielerlei Art beschreiben. In allen Fällen handelt es sich um eine Sammlung von Beispielen, die wiederum Sammlungen von Merkmalen darstellen.

Eine gängige Beschreibung eines Datensatzes ist die **Entwurfsmatrix**. Dabei handelt es sich um eine Matrix, die in jeder Zeile mehrere Beispiele enthält. Jede Spalte der Matrix entspricht einem anderen Merkmal. Zum Beispiel enthält der Iris-Datensatz 150 Beispiele mit jeweils vier Merkmalen. Er lässt sich daher mit einer Entwurfsmatrix $\mathbf{X} \in \mathbb{R}^{150 \times 4}$ wiedergeben, in der $X_{i,1}$ die Länge des Kelchblatts der Pflanze i angibt, $X_{i,2}$ die Breite des Kelchblatts der Pflanze i usw. Für die meisten Lernalgorithmen in diesem Buch geben wir an, wie sie mit Entwurfsmatrix-Datensätzen arbeiten.

Um einen Datensatz als Entwurfsmatrix zu beschreiben, muss es natürlich möglich sein, jedes der Beispiele als Vektor zu beschreiben. Außerdem müssen alle Vektoren dieselbe Größe aufweisen. Das ist allerdings nicht immer möglich. In einer Fotosammlung, die Bilder mit unterschiedlichen Höhen und Breiten aufweist, ist die Anzahl der Pixel pro Foto unterschiedlich hoch; daher können nicht alle Fotos mit Vektoren derselben Länge beschrieben werden. In Abschnitt 9.7 und Kapitel 10 zeigen wir, wie die unterschiedlichen Typen solch heterogener Daten behandelt werden. In derartigen Fällen wird der Datensatz nicht als Matrix mit m Zeilen beschrieben, sondern als Menge mit m Elementen: $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}\}$. Diese Schreibweise besagt nicht, dass zwei beliebige Beispielvektoren $\mathbf{x}^{(i)}$ und $\mathbf{x}^{(j)}$ die gleiche Größe haben.

Beim überwachten Lernen enthält das Beispiel neben der Merkmalssammlung auch ein Label oder einen Zielwert. Um zum Beispiel einen Lernalgorithmus zur Objekterkennung in Fotografien einzusetzen, müssen wir angeben, welches Objekt in allen Fotos zu sehen ist. Dazu können wir einen Zahlencode verwenden, vielleicht 0 für eine Person, 1 für ein Fahrzeug, 2 für eine Katze usw. Für Datensätze mit einer Entwurfsmatrix der Merkmalsbeobachtungen \mathbf{X} geben wir auch einen Vektor mit Labels \mathbf{y} an, wobei y_i das Label für das Beispiel i enthält.

Es kann auch vorkommen, dass das Label mehr als nur eine Zahl beinhaltet. Damit ein Spracherkennungssystem komplexe Sätze transkribieren

kann, muss das Label für jeden Beispielsatz aus einer Sequenz von Wörtern bestehen.

Wie bei der Unterscheidung zwischen überwachtem und unüberwachtem Lernen gibt es auch bei Datensätzen und Erfahrungen keine starren Begrifflichkeiten. Die hier beschriebenen Strukturen decken zwar die meisten Fälle ab, aber in der Praxis können Sie jederzeit neue Strukturen für neue Anwendungen konzipieren.

5.1.4 Beispiel: Lineare Regression

Laut unserer Definition eines Machine-Learning-Algorithmus handelt es sich dabei um einen Algorithmus, der in der Lage ist, die Leistungsfähigkeit eines Computerprogramms bei einer bestimmten Aufgabe durch Erfahrung zu verbessern. Diese recht abstrakte Definition können wir am Beispiel eines einfachen Machine-Learning-Algorithmus, der **linearen Regression**, konkretisieren. Wir werden bei der Einführung weiterer Machine-Learning-Konzepte, die das Verständnis für das Verhalten des Algorithmus vertiefen, immer wieder auf dieses Beispiel zurückgreifen.

Wie der Name andeutet, löst die lineare Regression ein Regressionsproblem. Anders ausgedrückt: Das System soll einen Vektor $\mathbf{x} \in \mathbb{R}^n$ entgegennehmen (Eingangsdaten) und den Wert eines Skalars $y \in \mathbb{R}$ vorhersagen (Ausgangsdaten). Die lineare Regression gibt eine lineare Funktion der Eingangsdaten aus. Sei \hat{y} der Wert, den unser Modell für y vorhersagt. Wir definieren die Ausgabe als

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \quad (5.3)$$

dabei ist $\mathbf{w} \in \mathbb{R}^n$ ein **Parameter**-Vektor.

Parameter sind Werte, die das Verhalten des Systems steuern. In diesem Fall ist w_i der Koeffizient, den wir mit dem Merkmal x_i multiplizieren, bevor wir die Beiträge aller Merkmale aufsummieren. Sie können sich \mathbf{w} als Menge von **Gewicht** vorstellen, die bestimmen, wie sich jedes der Merkmale auf die Vorhersage auswirkt. Erhält ein Merkmal x_i ein positives Gewicht w_i , dann steigt durch Erhöhung des Werts für dieses Merkmal auch der Wert der Vorhersage \hat{y} . Erhält ein Merkmal ein negatives Gewicht, dann sinkt durch Erhöhung des Werts für dieses Merkmal der Wert der Vorhersage. Ist das Gewicht eines Merkmals groß, hat es eine starke Auswirkung auf die Vorhersage. Bei einem Gewicht von Null hat das Merkmal keinerlei Auswirkung auf die Vorhersage.

Für unsere Aufgabe T gilt somit folgende Definition: Sage y für \mathbf{x} durch Ausgabe von $\hat{y} = \mathbf{w}^\top \mathbf{x}$ vorher. Nun müssen wir die Leistungsbewertung P definieren.

Angenommen, wir verfügen über eine Entwurfsmatrix mit m Beispieleingaben, die nicht für das Training, sondern lediglich zur Bewertung der Modellleistung verwendet wird. Außerdem verfügen wir über einen Vektor mit den Zielwerten für die Regression, die für jedes dieser Beispiele den korrekten Wert für y angeben. Da der Datensatz nur zur Bewertung verwendet wird, handelt es sich um unsere Testdatenmenge. Die Entwurfsmatrix mit den Eingangsdaten bezeichnet man als $\mathbf{X}^{(\text{test})}$ und den Vektor mit den Zielwerten für die Regression als $\mathbf{y}^{(\text{test})}$.

Eine Möglichkeit zur Messung der Modellleistung besteht in der Berechnung des **mittleren quadratischen Fehlers**, kurz: MQF (engl. *mean squared error*, MSE), des Modells für die Testdatenmenge. Ergibt $\hat{\mathbf{y}}^{(\text{test})}$ die Vorhersagen des Modells für die Testdatenmenge, so ergibt sich der mittlere quadratische Fehler aus

$$\text{MQF}_{\text{test}} = \frac{1}{m} \sum_i (\hat{\mathbf{y}}^{(\text{test})} - \mathbf{y}^{(\text{test})})_i^2. \quad (5.4)$$

Intuitiv wird klar, dass dieses Fehlermaß auf 0 fällt, wenn $\hat{\mathbf{y}}^{(\text{test})} = \mathbf{y}^{(\text{test})}$ ist. Außerdem ist offensichtlich, dass

$$\text{MQF}_{\text{test}} = \frac{1}{m} \|\hat{\mathbf{y}}^{(\text{test})} - \mathbf{y}^{(\text{test})}\|_2^2 \quad (5.5)$$

ist, der Fehler also zunimmt, sobald der euklidische Abstand zwischen Vorhersagen und Zielwerten zunimmt.

Zum Konstruieren eines Machine-Learning-Algorithmus müssen wir einen Algorithmus entwickeln, der das Gewicht \mathbf{w} so verbessert, dass MQF_{test} reduziert wird, wenn der Algorithmus Erfahrungen aus der Beobachtung einer Trainingsdatenmenge $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$ sammelt. Intuitiv (die Begründung folgt in Abschnitt 5.5.1) können wir hierzu den mittleren quadratischen Fehler für die Trainingsdatenmenge $\text{MQF}_{\text{train}}$ minimieren.

Dazu bestimmen wir, wo der Gradient $\mathbf{0}$ ist:

$$\nabla_{\mathbf{w}} \text{MQF}_{\text{train}} = 0 \quad (5.6)$$

$$\Rightarrow \nabla_{\mathbf{w}} \frac{1}{m} \|\hat{\mathbf{y}}^{(\text{train})} - \mathbf{y}^{(\text{train})}\|_2^2 = 0 \quad (5.7)$$

$$\Rightarrow \frac{1}{m} \nabla_{\mathbf{w}} \|\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2 = 0 \quad (5.8)$$

$$\Rightarrow \nabla_{\mathbf{w}} (\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})})^\top (\mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})}) = 0 \quad (5.9)$$

$$\Rightarrow \nabla_w (\mathbf{w}^\top \mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \mathbf{w} - 2\mathbf{w}^\top \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} + \mathbf{y}^{(\text{train})\top} \mathbf{y}^{(\text{train})}) = 0 \quad (5.10)$$

$$\Rightarrow 2\mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \mathbf{w} - 2\mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} = 0 \quad (5.11)$$

$$\Rightarrow \mathbf{w} = (\mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})})^{-1} \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} \quad (5.12)$$

Das Gleichungssystem, dessen Lösung sich aus Gleichung 5.12 ergibt, nennt man auch die **Normalgleichungen**. Die Berechnung von Gleichung 5.12 stellt einen einfachen Lernalgorithmus dar. Abbildung 5.1 zeigt den Lernalgorithmus der linearen Regression in Aktion.

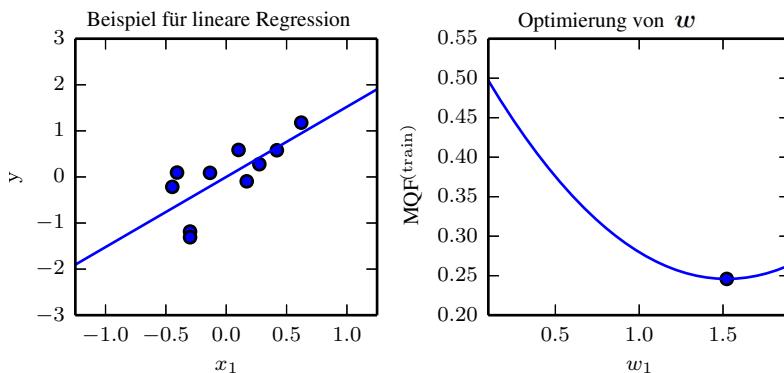


Abbildung 5.1: Ein lineares Regressionsproblem mit einer Trainingsdatenmenge, die zehn Datenpunkte mit jeweils einem Merkmal umfasst. Da es nur ein Merkmal gibt, enthält der Gewichtungsvektor \mathbf{w} auch nur einen zu lernenden Parameter w_1 . (*Links*) Beachten Sie, dass die lineare Regression lernt, w_1 so zu wählen, dass die Linie $y = w_1 x$ möglichst durch sämtliche Trainingspunkte verläuft. (*Rechts*) Der eingezeichnete Punkt zeigt den durch die Normalgleichungen ermittelten Wert von w_1 an, der den mittleren quadratischen Fehler auf der Trainingsdatenmenge minimiert.

Beachten Sie, dass der Begriff **lineare Regression** häufig für ein etwas raffinierteres Modell mit einem zusätzlichen Parameter verwendet wird – der Achsenschnittpunktsterm b . In diesem Modell ist

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b, \quad (5.13)$$

sodass die Zuordnung der Parameter zu den Vorhersagen nach wie vor eine lineare Funktion ist, die von den Merkmalen zu den Vorhersagen jedoch eine affine Funktion. Aus der Erweiterung zu affinen Funktionen ergibt sich, dass der Plot der Modellvorhersagen noch wie eine Linie aussieht, aber nicht mehr durch den Ursprung verlaufen muss. Anstatt den Verzerrungsparameter b zu

addieren, können wir weiterhin das Modell mit nur den Gewichten verwenden und \boldsymbol{x} um einen Zusatzeintrag ergänzen, der stets 1 ist. Das Gewicht des zusätzlichen Eintrags 1 spielt die Rolle des Verzerrungsparameters. Im Rahmen dieses Buchs verwenden wir den Begriff »linear« häufig mit Verweis auf affine Funktionen.

Der Achsenabschnitt b wird auch als **Verzerrungsparameter** der affinen Transformation bezeichnet. Diese Bezeichnung leitet sich aus der Sichtweise ab, dass ohne jegliche Eingabe die Ausgabe der Transformation zu b verschoben ist. Der Begriff darf nicht mit einer statistischen Verzerrung (auch *Bias*) verwechselt werden, bei der die erwartete Schätzgröße eines statistischen Schätzalgorithmus nicht der wahren Größe entspricht.

Natürlich handelt es sich bei der linearen Regression um einen sehr einfachen und eingeschränkten Lernalgorithmus, an dem sich allerdings gut zeigen lässt, wie ein Lernalgorithmus funktioniert. In den folgenden Abschnitten lernen Sie einige der Grundlagen für die Konstruktion von Lernalgorithmen kennen und erfahren, wie diese Grundlagen in komplexeren Lernalgorithmen verwendet werden können.

5.2 Kapazität, Überanpassung und Unteranpassung

Die zentrale Herausforderung beim Machine Learning besteht darin, dass der Algorithmus gut mit *neuen, bisher unbekannten* Eingangsdaten zurechtkommen muss – nicht nur mit jenen, für die das Modell trainiert wurde. Die Fähigkeit, mit bisher unbeobachteten Eingangsdaten gut zuretzukommen, wird als **Generalisierung** bezeichnet.

Beim Trainieren eines Machine-Learning-Modells steht uns normalerweise eine Trainingsdatenmenge zur Verfügung. Wir können eine bestimmte Fehlergröße für die Trainingsdatenmenge berechnen, den sogenannten **Trainingsfehler**. Das Ziel besteht darin, diesen Trainingsfehler zu reduzieren. Bisher haben wir ein simples Optimierungsproblem beschrieben. Der Unterschied zwischen Machine Learning und Optimierung besteht darin, dass wir auch einen geringen **Generalisierungsfehler** (auch **Testfehler** genannt) anstreben. Der Generalisierungsfehler ist definiert als Erwartungswert des Fehlers für neue Eingangsdaten. Der Erwartungswert wird hier über mehrere unterschiedliche mögliche Eingaben betrachtet, die aus der Verteilung jener Eingangsdaten entnommen wurden, die wir für den praktischen Einsatz des Systems erwarten.

Wir schätzen den Generalisierungsfehler eines Machine-Learning-Modells üblicherweise durch Ermitteln der Leistung auf einer **Testdatenmenge** mit Beispielen, die unabhängig von der Trainingsdatenmenge gesammelt wurden.

In unserem Beispiel zur linearen Regression haben wir das Modell durch Minimierung des Trainingsfehlers

$$\frac{1}{m^{(\text{train})}} \|\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\|_2^2 \quad (5.14)$$

trainiert, aber tatsächlich interessiert uns der Testfehler $\frac{1}{m^{(\text{test})}} \|\mathbf{X}^{(\text{test})}\mathbf{w} - \mathbf{y}^{(\text{test})}\|_2^2$.

Wie aber können wir die Leistung auf der Testdatenmenge beeinflussen, wenn wir doch nur die Trainingsdatenmenge beobachten können? Hier hält das Gebiet der **statistischen Lerntheorie** einige Antworten bereit. Werden Trainings- und Testdatenmenge tatsächlich willkürlich gesammelt, haben wir kaum Möglichkeiten. Wenn wir jedoch einige Annahmen darüber treffen dürfen, wie diese beiden Mengen erfasst wurden, dann öffnet sich ein Weg.

Die Trainings- und Testdaten werden von einer Wahrscheinlichkeitsverteilung über den Datensatz erzeugt, der als **datengenerierender Prozess** bezeichnet wird. Für gewöhnlich treffen wir eine Reihe von Annahmen, die in ihrer Gesamtheit **u.i.v.-Annahmen** genannt werden. Diese Annahmen besagen, dass die Beispiele in jedem Datensatz **unabhängig** voneinander sind und dass die Trainingsdatenmenge und die Testdatenmenge **identisch verteilt** sind, also aus der gegenseitig identischen Wahrscheinlichkeitsverteilung stammen. Diese Annahme ermöglicht es, den datengenerierenden Prozess mit einer Wahrscheinlichkeitsverteilung über ein einzelnes Beispiel zu beschreiben. Dieselbe Verteilung wird anschließend zum Erzeugen jedes einzelnen Trainingsbeispiels und jedes einzelnen Testbeispiels genutzt. Wir nennen die zugrunde liegende Verteilung die **datengenerierende Verteilung**, geschrieben p_{data} . Dieser probabilistischer Rahmen und die u.i.v.-Annahmen versetzen uns in die Lage, die Beziehung zwischen Trainingsfehler und Testfehler mathematisch zu untersuchen.

Eine unmittelbare Verbindung, die wir zwischen Trainingsfehler und Testfehler beobachten können, ist, dass der erwartete Trainingsfehler eines zufällig ausgewählten Modells gleich dem erwarteten Testfehler des Modells ist. Gegeben sei eine Wahrscheinlichkeitsverteilung $p(\mathbf{x}, y)$. Durch wiederholtes Ziehen von Stichproben (engl. *sampling*) erstellen wir daraus die Trainingsdatenmenge und die Testdatenmenge. Für einen festen Wert \mathbf{w} entspricht der erwartete Fehler auf der Trainingsdatenmenge exakt dem

erwarteten Fehler der Testdatenmenge, da beide Erwartungswerte aus derselben Stichprobenentnahme stammen. Der einzige Unterschied zwischen den beiden Bedingungen liegt im Namen, den wir dem ausgewählten Datensatz zuweisen.

Wenn wir einen Machine-Learning-Algorithmus verwenden, fixieren wir die Parameter vor dem Zusammenstellen der beiden Datensätze natürlich nicht. Wir bestimmen die Trainingsdatenmenge und verwenden sie zum Auswählen der Parameter für eine Reduzierung des Fehlers für die Trainingsdatenmenge; erst dann bestimmen wir die Testdatenmenge. Bei diesem Verfahren ist der erwartete Testfehler größer oder gleich dem Erwartungswert für den Trainingsfehler. Die Faktoren, die die Leistung eines Machine-Learning-Algorithmus beeinflussen, sind seine Fähigkeit,

1. den Trainingsfehler klein zu halten und
2. den Abstand zwischen Trainingsfehler und Testfehler klein zu halten.

Diese beiden Gesichtspunkte entsprechen den beiden zentralen Herausforderungen im Machine Learning: **Unteranpassung** und **Überanpassung**. Unteranpassung tritt auf, wenn das Modell nicht in der Lage ist, einen ausreichend kleinen Fehlerwert für die Trainingsdatenmenge zu erzielen. Überanpassung tritt auf, wenn der Abstand zwischen Trainingsfehler und Testfehler zu groß ist.

Wir können steuern, ob ein Modell eher zur Überanpassung oder zur Unteranpassung neigt, indem wir dessen **Kapazität** anpassen. Salopp ausgedrückt bezeichnet die Kapazität eines Modells seine Fähigkeit, sich einer Vielzahl von Funktionen anzupassen. Modelle mit einer niedrigen Kapazität haben möglicherweise Probleme mit der Anpassung an die Trainingsdatenmenge. Bei Modellen mit einer hohen Kapazität kann es zur Überanpassung kommen, weil diese Eigenschaften der Trainingsdatenmenge speichern, die nicht zu einer besseren Leistung für die Testdatenmenge beitragen.

Eine Möglichkeit zum Anpassen der Kapazität eines Lernalgorithmus besteht darin, seinen **Hypothesenraum** zu bestimmen, also die Menge der Funktionen, die der Lernalgorithmus als Lösung auswählen darf. So entspricht der Hypothesenraum für den linearen Regressionsalgorithmus der Menge aller linearen Funktionen seiner Eingangsdaten. Wir können die lineare Regression so generalisieren, dass zusätzlich zu den linearen Funktionen auch Polynome in ihren Hypothesenraum fallen. Dadurch erhöhen wir die Kapazität des Modells.

Ein Polynom vom Grad 1 ergibt ein bereits bekanntes lineares Regressionsmodell mit der Vorhersage

$$\hat{y} = b + wx. \quad (5.15)$$

Durch Einführen des zusätzlichen Merkmals x^2 für das lineare Regressionsmodell erlernen wir ein Modell, das als eine Funktion von x quadratisch ist:

$$\hat{y} = b + w_1x + w_2x^2. \quad (5.16)$$

Obwohl das Modell eine quadratische Funktion für die *Eingangsdaten* nutzt, ist die Ausgabe nach wie vor eine lineare Funktionen der *Parameter*, sodass wir noch immer die Normalgleichungen zum Trainieren des Modells in geschlossener Form verwenden können. Wir können nun weitere Potenzen von x als zusätzliche Merkmale hinzufügen, um zum Beispiel ein Polynom vom Grad 9 zu erhalten:

$$\hat{y} = b + \sum_{i=1}^9 w_i x^i. \quad (5.17)$$

Machine-Learning-Algorithmen arbeiten normalerweise am besten, wenn ihre Kapazität der tatsächlichen Komplexität der vorliegenden Aufgabe und der Menge der verfügbaren Trainingsdaten angemessen ist. Modelle mit unzureichender Kapazität können keine komplexen Aufgaben lösen. Modelle mit hoher Kapazität können komplexe Aufgaben lösen, doch wenn ihre Kapazität höher als für die vorliegende Aufgabe notwendig ist, neigen sie zur Überanpassung.

Abbildung 5.2 verdeutlicht das. Wir untersuchen eine lineare, quadratische und unabhängige Näherung vom Grad 9 für die Anpassung eines Problems, dessen zugrunde liegende Funktion quadratisch ist. Die lineare Funktion kann die Krümmung im tatsächlich zugrunde liegenden Problem nicht erfassen, sodass es zu einer Unteranpassung kommt. Die Näherung vom Grad 9 kann die korrekte Funktion darstellen, ist aber auch in der Lage, unendlich viele weitere Funktionen darzustellen, die exakt durch die Trainingspunkte verlaufen, da wir mehr Parameter als Trainingsbeispiele haben. Angesichts derart vieler so unterschiedlicher Lösungen haben wir kaum eine Chance, eine Lösung auszuwählen, die gut generalisiert.

In diesem Beispiel passt das quadratische Modell perfekt zur wahren Struktur der Aufgabe und generalisiert daher gut auf neue Daten.

Bisher haben wir lediglich eine Möglichkeit zum Ändern der Kapazität des Modells beschrieben, nämlich das Ändern der Anzahl von Merkmalen auf

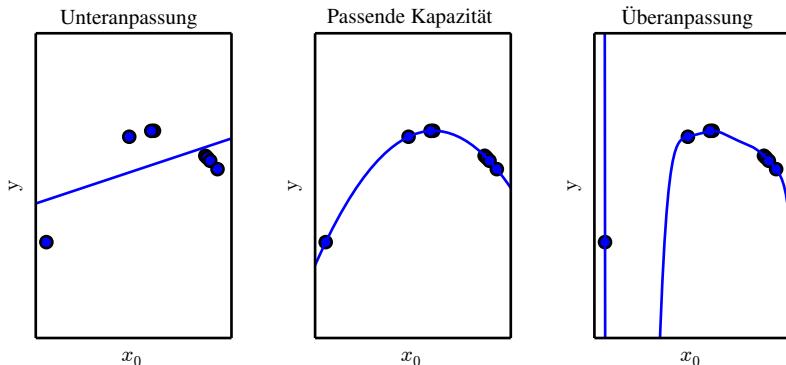


Abbildung 5.2: Wir passen drei Modelle für diese Beispieltrainingsdatenmenge an. Die Trainingsdaten wurden synthetisch durch zufälliges Festlegen von x -Werten und deterministische y -Auswahl mittels Auswertung einer quadratischen Funktion erzeugt. (*Links*) Eine auf die Daten angepasste lineare Funktion weist eine Unteranpassung auf – die in den Daten vorliegende Krümmung wird nicht erkannt. (*Mitte*) Eine auf die Daten angepasste quadratische Funktion generalisiert gut auf nicht betrachtete Punkte. Es gibt keine nennenswerte Über- oder Unteranpassung. (*Rechts*) Ein auf die Daten angepasstes Polynom vom Grad 9 weist eine Überanpassung auf. Wir haben hier die Moore-Penrose-Pseudoinverse zum Lösen der unterbestimmten Normalgleichungen verwendet. Die Lösung verläuft exakt durch alle Trainingspunkte, aber unglücklicherweise wurde die korrekte Struktur nicht extrahiert. Es gibt im Gegensatz zur tatsächlich zugrunde liegenden Funktion ein tiefes Tal zwischen zwei Trainingspunkten. Außerdem ist auf der linken Seite der Daten ein steiler Anstieg zu sehen, obwohl die wahre Funktion in diesen Bereich fällt.

der Eingangsseite bei gleichzeitigem Hinzufügen neuer Parameter für diese Merkmale. Es gibt aber viele weitere Wege zum Ändern der Modellkapazität. Die Kapazität wird nicht allein durch die Wahl des Modells bestimmt. Das Modell legt fest, aus welcher Funktionsfamilie der Lernalgorithmus auswählen kann, wenn Parameter zum Reduzieren eines Trainingsziels variiert werden. Das wird auch als **repräsentative Kapazität** des Modells bezeichnet. Häufig stellt die Suche nach der besten Funktion innerhalb der Familie ein schwieriges Optimierungsproblem dar. In der Praxis sucht der Lernalgorithmus nicht nach der besten Funktion, sondern lediglich nach einer Funktion, die den Trainingsalgorithmus deutlich reduziert. Diese zusätzlichen Einschränkungen – zum Beispiel die Unvollkommenheit des Optimierungsalgorithmus – führen dazu, dass die **tatsächliche Kapazität** des Lernalgorithmus möglicherweise geringer als die repräsentative Kapazität der Modellfamilie ist.

Unsere modernen Vorstellungen zur Optimierung der Generalisierung von Machine-Learning-Modellen beruhen auf Überlegungen, die schon Philosophen wie Ptolemäus angestellt haben. Viele Gelehrte nutzen ein Prinzip der Sparsamkeit, das allgemein unter der Bezeichnung **Ockhams Rasiermesser** bekannt wurde (ca. 1287–1347). Es besagt, dass von mehreren Hypothesen, die einen Sachverhalt gleich gut erklären, die »einfachste« den anderen vorzuziehen ist. Im 20. Jahrhundert wurde dieser Gedanke von den Gründern der statistischen Lerntheorie formalisiert und präzisiert (*Vapnik und Chervonenkis*, 1971; *Vapnik*, 1982; *Blumer et al.*, 1989; *Vapnik*, 1995).

Die statistische Lerntheorie bietet mehrere Möglichkeiten zum Quantifizieren der Modellkapazität. Die bekannteste davon ist die **Vapnik-Chervonenkis-Dimension**, kurz VC-Dimension. Die VC-Dimension bestimmt die Kapazität eines binären Klassifikators. Sie ist definiert als der größtmögliche Wert für ein m , für das eine Trainingsdatenmenge mit m verschiedenen x -Punkten existiert, die der Klassifikator beliebig kennzeichnen kann.

Durch das Quantifizieren der Modellkapazität lassen sich mit der statistischen Lerntheorie quantitative Vorhersagen machen. Die wichtigsten Ergebnisse der statistischen Lerntheorie zeigen, dass die Diskrepanz zwischen Trainingsfehler und Generalisierungsfehler nach oben durch eine Größe begrenzt wird, die mit der Modellkapazität wächst, aber sinkt, wenn die Anzahl der Trainingsbeispiele zunimmt (*Vapnik und Chervonenkis*, 1971; *Vapnik*, 1982; *Blumer et al.*, 1989; *Vapnik*, 1995). Diese Schranken belegen die Funktion von Machine-Learning-Algorithmen auf theoretischer Ebene; allerdings werden sie im praktischen Einsatz von Deep-Learning-Algorithmen nur selten eingesetzt. Das liegt zum einen daran, dass die Schranken häufig nicht fest sind, und zum anderen daran, dass die Kapazität von Deep-Learning-Algorithmen sich oft nur schwer bestimmen lässt. Die Bestimmung der Kapazität eines Deep-Learning-Modells erweist sich als besonders schwierig, da die Effektivkapazität durch die Fähigkeiten des Optimierungsalgorithmus beschränkt wird und wir nur über ein geringes theoretisches Verständnis der allgemeinen, nichtkonvexen Optimierungsprobleme im Deep-Learning-Kontext verfügen.

Wir müssen bedenken, dass einfachere Funktionen eine höhere Wahrscheinlichkeit der Generalisierung aufweisen (also eine kleinere Lücke zwischen Trainings- und Testfehler), wir aber dennoch eine ausreichend komplexe Hypothese aufstellen müssen, um einen niedrigen Trainingsfehler zu erreichen. Normalerweise nimmt der Trainingsfehler ab, bis er sich mit zunehmender Modellkapazität dem kleinstmöglichen Fehlerwert annähert (wenn wir davon ausgehen, dass die Fehlergröße einen Mindestwert aufweist).

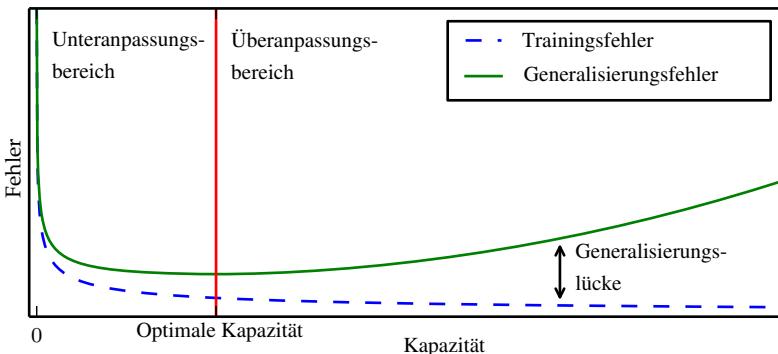


Abbildung 5.3: Typische Beziehung zwischen Kapazität und Fehler. Trainings- und Testfehler verhalten sich unterschiedlich. Am linken Ende des Graphen sind Trainingsfehler und Generalisierungsfehler beide hoch. Das ist der **Unteranpassung** geschuldet. Mit zunehmender Kapazität nimmt der Trainingsfehler ab, aber die Lücke zwischen Trainings- und Generalisierungsfehler zu. Irgendwann übersteigt die Größe der Lücke die Abnahme des Trainingsfehlers und wir geraten in den **Überanpassungsbereich**, in dem die Kapazität zu hoch ist; die **optimale Kapazität** ist überschritten.

Der Generalisierungsfehler ist normalerweise eine U-förmige Funktion der Modellkapazität. Dieser Umstand ist in Abbildung 5.3 dargestellt.

Um den Extremfall einer beliebig hohen Kapazität zu erreichen, führen wir das Konzept der **nichtparametrischen Modelle** ein. Bisher haben wir uns nur mit parametrischen Modellen wie der linearen Regression beschäftigt. Parametrische Modelle lernen eine Funktion, die durch einen Parametervektor beschrieben wird, dessen Größe eingeschränkt und festgelegt ist, bevor überhaupt Daten beobachtet werden. Nichtparametrische Modelle kennen solche Einschränkungen nicht.

Manchmal sind nichtparametrische Modelle lediglich theoretische Abstraktionen (beispielsweise ein Algorithmus, der alle möglichen Wahrscheinlichkeitsverteilungen durchsucht), die nicht praxistauglich sind. Allerdings können wir auch umsetzbare nichtparametrische Modelle konzipieren, deren Komplexität sich nach der Größe der Trainingsdatenmenge richtet. Ein Beispiel dafür ist die **Nearest-Neighbor-Regression** (Nächste-Nachbarn-Regression). Anders als die lineare Regression, die über einen Gewichtungsvektor fester Länge verfügt, speichert das Nearest-Neighbor-Regressionsmodell einfach die Werte \mathbf{X} und \mathbf{y} aus der Trainingsdatenmenge. Soll nun ein Testpunkt \mathbf{x} klassifiziert werden, sucht das Modell den nächstgelegenen (engl. *nearest*) Eintrag in der Trainingsdatenmenge und gibt den zugehörigen Wert \mathbf{y} zurück.

rigen Zielwert für die Regression zurück. Oder anders ausgedrückt: $\hat{y} = y_i$ mit $i = \arg \min ||\mathbf{X}_{i,:} - \mathbf{x}||_2^2$. Der Algorithmus kann auch für andere Abstandsmetriken als die L^2 -Norm generalisiert werden, zum Beispiel erlernte Abstandsmetriken (Goldberger et al., 2005). Sofern der Algorithmus Verbindungen aufheben darf, indem er den Durchschnittswert für die y_i aller $\mathbf{X}_{i,:}$, die mit dem Nächsten verbunden sind, bildet, kann der Algorithmus den kleinstmöglichen Trainingsfehler für jeden Regressionsdatensatz erreichen (wobei dieser Fehler größer als 0 sein kann, wenn zwei identische Eingaben mit unterschiedlichen Ausgaben verknüpft sind).

Wir können sogar einen nichtparametrischen Lernalgorithmus erstellen, indem wir einen parametrischen Lernalgorithmus in einem anderen Algorithmus »verpacken«. Der äußere Algorithmus hat die Aufgabe, bei Bedarf die Anzahl der Parameter zu erhöhen. Ein Beispiel ist eine äußere Lernschleife, die zusätzlich zum polynomialen Wachstum der Eingangsdaten den Grad des mittels linearer Regression erlernten Polynoms verändert.

Das ideale Modell ist ein Orakel, das die echte Wahrscheinlichkeitsverteilung zum Erzeugen der Daten einfach kennt. Doch auch dieses Modell würde bei vielen Problemen einen Fehler aufweisen, denn die Verteilung kann ja nach wie vor ein gewisses Rauschen enthalten. Beim überwachten Lernen kann die Zuordnung zwischen \mathbf{x} und y inhärent stochastisch sein, oder y kann eine deterministische Funktion sein, die andere Variablen über die aus \mathbf{x} einbezieht. Der von einem Orakel bei Vorhersagen anhand der echten Verteilung $p(\mathbf{x}, y)$ verursachte Fehler wird **Bayes-Fehler** genannt.

Trainings- und Generalisierungsfehler variieren mit schwankender Größe der Trainingsdatenmenge. Der erwartete Generalisierungsfehler kann bei zunehmender Anzahl von Trainingsbeispielen niemals zunehmen. Für nichtparametrische Modelle führen mehr Daten so lange zu einer besseren Generalisierung, bis der bestmögliche Fehler erreicht ist. Jedes feste parametrische Modell mit einer Kapazität unter dem Optimum nähert sich einem Fehlerwert an, der den Bayes-Fehler übersteigt. Abbildung 5.4 zeigt dies. Beachten Sie, dass es trotz optimaler Modellkapazität durchaus möglich ist, dass ein großer Abstand zwischen Trainings- und Generalisierungsfehler vorliegt. In diesem Fall können wir die Lücke eventuell durch Erfassen weiterer Trainingsbeispiele verkleinern.

5.2.1 Das No-Free-Lunch-Theorem

Die Lerntheorie behauptet, dass ein Machine-Learning-Algorithmus auf Basis einer endlichen Trainingsdatenmenge mit Beispielen gut generalisieren kann. Das scheint gegen einige Grundregeln der Logik zu verstößen. Induktives

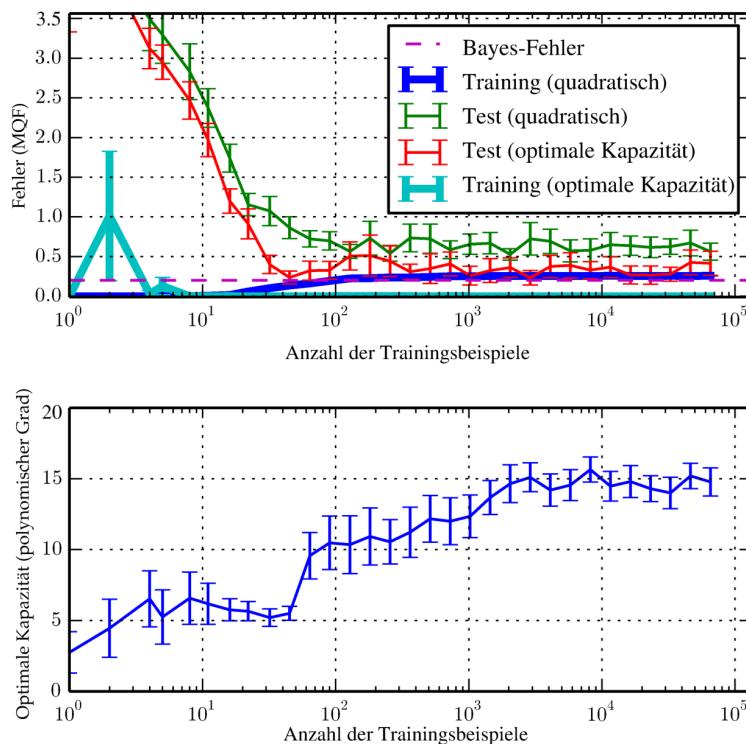


Abbildung 5.4: Die Auswirkung der Größe der Trainingsdatenmenge auf den Trainings- und Testfehler sowie die optimale Modellkapazität. Wir haben ein synthetisches Regressionsproblem durch Einbringen eines moderaten Rauschens auf einem Polynom vom Grad 5 konstruiert, eine einfache Testdatenmenge und anschließend unterschiedlich große Trainingsdatenmengen erzeugt. Für jede Größe haben wir 40 unterschiedliche Trainingsdatenmengen erzeugt, um Fehlerbalken mit 95-Prozent-Konfidenzintervallen darzustellen. (*Oben*) Der MQF der Trainings- und Testdatenmenge für zwei unterschiedliche Modelle: ein quadratisches Modell und ein Modell, dessen Grad für einen möglichst geringen Testfehler gewählt wurde. Beide sind in geschlossener Form für den Zweck geeignet. Beim quadratischen Modell nimmt der Trainingsfehler mit zunehmender Trainingsdatenmenge zu. Das liegt daran, dass die Anpassung größerer Datensätze schwieriger ist. Gleichzeitig nimmt der Testfehler ab, da weniger inkorrekte Hypothesen mit den Trainingsdaten übereinstimmen. Die Kapazität des quadratischen Modells ist zum Lösen der Aufgabe zu klein, sodass sich sein Testfehler einem hohen Wert annähert. Der Testfehler nähert sich bei optimaler Kapazität dem Bayes-Fehler an. Der Trainingsfehler kann unter den Bayes-Fehler fallen, da der Trainingsalgorithmus sich bestimmte Instanzen der Trainingsdatenmenge merken kann. Wenn die Trainingsgröße gegen unendlich geht, muss der Trainingsfehler jedes Modells mit fester Kapazität (hier also das quadratische Modell) mindestens so groß wie der Bayes-Fehler sein. (*Unten*) Mit zunehmender Größe der Trainingsdatenmenge steigt die optimale Kapazität (hier als Grad des optimalen polynomialen Regressors dargestellt). Die optimale Kapazität erreicht einen konstanten Wert, sobald die Komplexität hoch genug zum Lösen der Aufgabe ist.

Schließen oder das Ableiten allgemeiner Regeln von einer eingeschränkten Anzahl Beispiele ist logisch nicht valide. Um auf logische Weise eine Regel abzuleiten, die jedes Element einer Menge beschreibt, muss man über Informationen zu jedem Element der Menge verfügen.

Zum Teil umgeht Machine Learning dieses Problem durch die Nutzung rein probabilistischer Regeln und den Verzicht auf die vollständig gewissen Regeln aus dem rein logischen Schließen. Machine Learning verspricht, Regeln zu finden, die *vermutlich* für die *meisten* Elemente der betrachteten Menge korrekt sind.

Leider lässt sich das Problem selbst hierdurch nicht zur Gänze lösen. Das »**No-Free-Lunch-Theorem**« für das Machine Learning (Wolpert, 1996) besagt, dass jeder Klassifizierungsalgorithmus im Durchschnitt über alle möglichen datengenerierenden Verteilungen dieselbe Fehlerquote aufweist, wenn bisher unbeobachtete Punkte klassifiziert werden. Anders ausgedrückt: In gewisser Weise ist kein Machine-Learning-Algorithmus universell gesehen besser als ein anderer. Der raffinierteste Algorithmus, den wir erdenken können, weist dieselbe durchschnittliche Leistung (bezüglich aller möglichen Aufgaben) auf wie die einfache Vorhersage, dass jeder Punkt zur selben Klasse gehört.

Zum Glück gilt dieses Ergebnis nur für den Durchschnittswert *aller* möglichen datengenerierenden Verteilungen. Wenn wir Annahmen über die Art der Wahrscheinlichkeitsverteilungen, mit denen wir in realen Anwendungen zu tun haben, treffen, können wir Lernalgorithmen entwickeln, die für diese Verteilungen gut funktionieren.

Das bedeutet, dass die Machine-Learning-Forschung nicht das Ziel hat, einen universellen Lernalgorithmus oder den absolut besten Lernalgorithmus zu finden. Vielmehr besteht das Ziel in einem Verständnis dafür, welche Art von Verteilungen für die »echte Welt«, auf die ein KI-Agent trifft, von Bedeutung sind, und welche Art von Machine-Learning-Algorithmen mit den Daten der Art von datengenerierenden Verteilungen, die uns wichtig sind, gut zureckkommt.

5.2.2 Regularisierung

Das No-Free-Lunch-Theorem impliziert, dass wir unsere Machine-Learning-Algorithmen auf eine gute Funktion bei einer bestimmten Aufgabe trimmen müssen. Dazu bauen wir eine Reihe von Vorgaben oder Präferenzen in den Lernalgorithmus ein. Wenn diese Präferenzen zu Lernproblemen passen, die der Algorithmus lösen soll, funktioniert er besser.

Bisher haben wir uns lediglich mit einem Verfahren zum Ändern eines Lernalgorithmus genauer befasst, nämlich dem Steigern oder Reduzieren der repräsentativen Kapazität eines Modells durch Hinzufügen oder Entfernen von Funktionen aus dem Hypothesenraum der Lösungen, unter denen der Lernalgorithmus auswählen kann. Wir haben ein spezifisches Beispiel für die Veränderung des Grads eines Polynoms im Rahmen eines Regressionsproblems behandelt. Das war eine stark vereinfachte Betrachtungsweise.

Das Verhalten unseres Algorithmus wird nicht nur von der Größe der Funktionsmenge im Hypothesenraum beeinflusst, sondern auch von der Identität der einzelnen Funktionen. Der bisher betrachtete Lernalgorithmus, die lineare Regression, verfügt über einen Hypothesenraum, der aus der Menge der linearen Funktionen der Eingangsdaten besteht. Diese linearen Funktionen können sich als nützlich erweisen, wenn die Beziehung zwischen Eingaben und Ausgaben in der Problemstellung nahezu linear ist. Bei stark nichtlinearen Problemen nimmt ihr Nutzen ab. So kommt die lineare Regression nicht gut zurecht, wenn wir $\sin(x)$ aus x vorhersagen möchten. Die Leistung eines Algorithmus wird also durch die Auswahl der Funktionen, aus denen Lösungen gezogen werden können, und die Anzahl dieser Funktionen gesteuert.

Wir können im Lernalgorithmus auch eine Präferenz für eine Lösung gegenüber einer anderen Lösung im Hypothesenraum vorsehen. Beide Funktionen sind dann geeignet, aber die eine wird bevorzugt verwendet. Die nicht bevorzugte Lösung wird nur gewählt, wenn sie deutlich besser zu den Trainingsdaten passt als die bevorzugte Lösung.

Wir können zum Beispiel das Trainingskriterium für die lineare Regression so ändern, dass **Weight Decay** (schrittweise Verringerung der Gewichte) einbezogen wird. Für die lineare Regression mit Weight Decay minimieren wir eine Summe aus dem mittleren quadratischen Fehler des Trainings und einem Kriterium $J(\mathbf{w})$, das eine Präferenz für Gewichte mit kleinerem Quadrat der L^2 -Norm ausdrückt. Damit ist

$$J(\mathbf{w}) = \text{MQF}_{\text{train}} + \lambda \mathbf{w}^\top \mathbf{w}, \quad (5.18)$$

wobei λ im Vorfeld festgelegt wird und die Stärke unserer Präferenz für kleinere Gewichte steuert. Für $\lambda = 0$ ist die Präferenz aufgehoben, doch je größer λ wird, desto kleiner werden die Gewichte. Durch Minimieren von $J(\mathbf{w})$ werden Gewichte gewählt, die zu einem Kompromiss zwischen der Anpassung und der geringen Größe der Trainingsdaten führen. So erhalten wir Lösungen mit einer geringeren Steigung oder einem Gewicht auf weniger Merkmalen. Weight Decay kann die Tendenz eines Modells zu Überanpassung und Unteranpassung regeln; wir können zum Beispiel

ein polynomiales Regressionsmodell von hohem Grad mit unterschiedlichen Werten für λ trainieren. Abbildung 5.5 enthält die Ergebnisse.

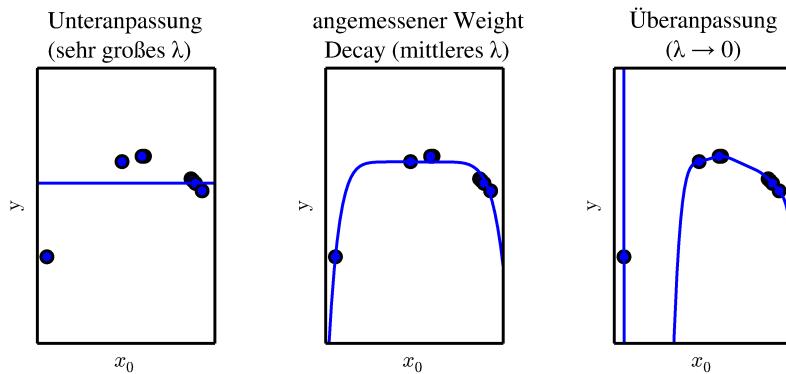


Abbildung 5.5: Wir passen ein polynomiales Regressionsmodell von hohem Grad an unsere Beispieltrainingsdatenmenge aus Abbildung 5.2 an. Die tatsächliche Funktion ist quadratisch, aber hier nutzen wir nur Modelle vom Grad 9. Wir variieren die Höhe des Weight Decays, um eine Überanpassung dieser hochgradigen Modelle zu verhindern. (*Links*) Für ein sehr großes λ können wir das Modell zwingen, eine Funktion ohne jede Steigung zu erlernen. Es kommt zur Unteranpassung, da nur eine konstante Funktion dargestellt werden kann. (*Mitte*) Für einen mittelgroßen Wert von λ stellt der Lernalgorithmus eine Kurve mit einer allgemein korrekten Form her. Obwohl das Modell Funktionen mit deutlich komplexeren Verläufen darstellen kann, hat der Weight Decay dazu geführt, dass eine einfachere, durch kleinere Koeffizienten beschriebene Funktion verwendet wird. (*Rechts*) Nähert sich der Weight Decay 0 an (wird also die Moore-Penrose-Pseudoinverse zum Lösen des unterbestimmten Problems mit minimaler Regularisierung verwendet), kommt es zu einer dramatischen Überanpassung des Polynoms vom Grad 9 (vgl. Abbildung 5.2).

Allgemein ausgedrückt können wir ein Modell regularisieren, das eine Funktion $f(\mathbf{x}; \boldsymbol{\theta})$ erlernt, indem wir einen Strafterm – den sogenannten **Regularisierer** – zur Kostenfunktion hinzufügen. Bei Weight Decay ist der Regularisierer $\Omega(\mathbf{w}) = \mathbf{w}^\top \mathbf{w}$. In Kapitel 7 wird gezeigt, dass viele weitere Regularisierer denkbar sind.

Das Bevorzugen einer Funktion gegenüber einer anderen ist eine allgemeinere Möglichkeit, die Modellkapazität zu steuern, als es beim Ein- oder Ausschließen von Elementen aus dem Hypothesenraum der Fall ist. Sie können sich das Ausschließen einer Funktion aus einem Hypothesenraum auch so vorstellen, als gäbe es eine unendlich starke Präferenz gegen diese Funktion.

In unserem Weight-Decay-Beispiel haben wir eine Präferenz für lineare Funktionen mit möglichst kleinen Gewichten explizit mithilfe eines zusätzli-

chen Terms in dem von uns minimierten Kriterium ausgedrückt. Es gibt noch viele weitere – implizite wie explizite – Möglichkeiten zum Festlegen der Präferenzen für unterschiedliche Lösungen. Diese unterschiedlichen Ansätze werden unter dem Begriff **Regularisierung** zusammengefasst. *Regularisierung ist jede Änderung, die wir an einem Lernalgorithmus vornehmen, die zum Ziel hat, dessen Generalisierungsfehler – nicht aber dessen Trainingsfehler – zu reduzieren.* Die Regularisierung gehört zu den zentralen Säulen im Machine Learning; von gleicher Bedeutung ist nur noch die Optimierung.

Das No-Free-Lunch-Theorem hat verdeutlicht, dass es **den** besten Machine-Learning-Algorithmus ebenso wenig gibt wie **die** beste Form der Regularisierung. Stattdessen müssen wir uns für eine Form der Regularisierung entscheiden, die für die zu lösende Aufgabe am besten geeignet ist. Die Philosophie des Deep Learnings im Allgemeinen und dieses Buchs im Besonderen besagt, dass eine große Anzahl von Aufgaben (wie all die intellektuellen Leistungen, zu denen Menschen fähig sind) wirkungsvoll durch sehr allgemeine Formen der Regularisierung gelöst werden kann.

5.3 Hyperparameter und Validierungsdaten

Die meisten Machine-Learning-Algorithmen verwenden Hyperparameter; das sind Einstellparameter, mit denen wir das Verhalten des Algorithmus steuern können. Die Werte der Hyperparameter werden nicht vom Lernalgorithmus selbst angepasst (obwohl wir einen verschachtelten Lernprozess entwerfen könnten, in dem ein Lernalgorithmus die besten Hyperparameter für einen anderen Lernalgorithmus erlernt).

Das Beispiel für die polynomiale Regression aus Abbildung 5.2 enthält nur einen Hyperparameter, nämlich den Grad des Polynoms. Dieser agiert als Hyperparameter: für die **Kapazität**. Der λ -Wert, der die Stärke des Weight Decay steuert, ist ein weiteres Beispiel für einen Hyperparameter.

Manchmal wird als Einstellparameter ein Hyperparameter gewählt, der vom Lernalgorithmus nicht erlernt werden kann, weil es zu schwierig ist, die Einstellung zu optimieren. Meistens soll aber der Einstellparameter ein Hyperparameter sein, denn es ist nicht sinnvoll, den Hyperparameter anhand der Trainingsdatenmenge zu erlernen. Das gilt für alle Hyperparameter, die die Modellkapazität steuern. Würden solche Hyperparameter anhand der Trainingsdatenmenge erlernt, würden sie stets die größtmögliche Modellkapazität auswählen, was eine Überanpassung zur Folge hätte (vgl. Abbildung 5.3). Es ist beispielsweise immer einfacher, die Trainingsdatenmenge mithilfe eines höhergradigen Polynoms und einer Weight-Decay-

Einstellung $\lambda = 0$ anzupassen, als mit einem Polynom von niedrigerem Grad und einer positiven Weight-Decay-Einstellung.

Zum Lösen des Problems benötigen wir eine **Validierungsdatenmenge** mit Beispielen, die der Trainingsalgorithmus nicht beobachtet.

Wir haben bereits behandelt, wie eine separate Testdatenmenge mit Beispielen aus derselben Verteilung wie die Trainingsdatenmenge dazu dienen kann, den Generalisierungsfehler eines Klassifikators zu schätzen, nachdem der Lernprozess abgeschlossen ist. Dabei dürfen die Testbeispiele keinesfalls zum Treffen irgendeiner Wahl über das Modell verwendet werden – auch nicht für die Hyperparameter. Somit darf kein Beispiel aus der Testdatenmenge in der Validierungsdatenmenge enthalten sein. Wir konstruieren die Validierungsdatenmenge also immer aus den *Trainingsdaten*. Entsprechend teilen wir die Trainingsdaten in zwei disjunkte Teilmengen auf. Eine dieser Teilmengen wird zum Lernen der Parameter verwendet, die andere als Validierungsdatenmenge zum Abschätzen des Generalisierungsfehlers während oder nach dem Training, sodass die Hyperparameter entsprechend angepasst werden können. Die Teilmenge zum Erlernen der Parameter wird normalerweise weiterhin Trainingsdatenmenge genannt, obwohl es dadurch zu einem Missverständnis bezüglich der größeren Datenmenge kommen kann, die für den gesamten Trainingsprozess genutzt wird. Die Teilmenge zur Auswahl der Hyperparameter wird als Validierungsdatenmenge oder kurz Validierungsdaten bezeichnet. Üblicherweise werden etwa 80 Prozent der Trainingsdaten für das Training und die restlichen 20 Prozent für die Validierung verwendet. Da die Validierungsdatenmenge zum »Trainieren« der Hyperparameter dient, unterschätzt der Validierungsdatenfehler den Generalisierungsfehler, allerdings meist um eine geringere Höhe, als dies beim Trainingsfehler der Fall ist. Nachdem die Hyperparameter-Optimierung abgeschlossen ist, kann der Generalisierungsfehler anhand der Testdatenmenge geschätzt werden.

In der Praxis führt die über viele Jahre wiederholte Verwendung derselben Testdatenmenge für eine Bewertung der Leistung unterschiedlicher Algorithmen dazu, dass für die Testdatenmenge selbst optimistische Bewertungen abgegeben werden; das gilt insbesondere, wenn wir all die Versuche der Wissenschaftsgemeinde berücksichtigen, die anerkannte Bestleistung auf dieser Testdatenmenge zu überflügeln. So können Benchmarks veraltet sein und dann nicht mehr die tatsächliche Leistung eines trainierten Systems unter Realbedingungen widerspiegeln. Zum Glück wendet die Wissenschaftsgemeinde sich gerne neuen (und meist ambitionierteren und größeren) Benchmark-Datensätzen zu.

5.3.1 Kreuzvalidierung

Das Aufteilen des Datensatzes in eine feste Trainingsdatenmenge und eine feste Testdatenmenge kann bei kleinen Testdatenmengen zu Problemen führen, denn eine kleine Testdatenmenge impliziert eine statistische Unsicherheit für den geschätzten durchschnittlichen Testfehler. Dann lässt sich schwerlich behaupten, dass Algorithmus A eine gegebene Aufgabe besser erledigt als Algorithmus B .

Bei Datensätzen mit Hunderttausenden Beispielen ist das kein ernstes Problem. Bei zu kleinen Datensätzen bieten sich Alternativen, mit denen alle Beispiele für die Schätzung des mittleren Testfehlers verwendet werden können, wenn auch auf Kosten eines höheren Berechnungsaufwands. Diese Verfahren beruhen auf der Idee, Training und Test anhand unterschiedlicher, zufällig ausgewählter Teilmengen oder Bereiche des Originaldatensatzes zu wiederholen. Meist wird die in Algorithmus 5.1 dargestellte k -fache Kreuzvalidierung genutzt. Dabei wird ein Teil des Datensatzes durch Aufteilen der Sammlung in k Teilmengen ohne Überschneidung gebildet. Der Testfehler kann nun durch Bilden des durchschnittlichen Testfehlers über k Versuche geschätzt werden. Im Versuch i wird die i -te Teilmenge der Daten als Testdatenmenge verwendet, während die restlichen Daten als Trainingsdatenmenge dienen. Dabei gibt es allerdings keine erwartungstreuen Schätzer der Varianz solcher Durchschnittfehlerschätzer (*Bengio und Grandvalet*, 2004); daher werden normalerweise Approximationen verwendet.

5.4 Schätzer, Verzerrung und Varianz

Die Statistik hält viele Werkzeuge bereit, mit denen das Ziel von Machine-Learning umgesetzt werden kann, nämlich eine Aufgabe nicht nur für die Trainingsdatenmenge zu lösen, sondern auch zu generalisieren. Grundlegende Konzepte wie Parameterschätzung, Verzerrung und Varianz sind hilfreich bei der formalen Charakterisierung der Begriffe Generalisierung, Unteranpassung und Überanpassung.

5.4.1 Punktschätzung

Die Punktschätzung ist der Versuch, die eine »beste« Vorhersage einer betrachteten Größe zu treffen. Allgemein formuliert kann es sich bei der betrachteten Größe um einen einzelnen Parameter oder einen Parametervektor in einem parametrischen Modell handeln, zum Beispiel die Gewichtungen

Algorithmus 5.1 Algorithmus für die k -fache Kreuzvalidierung. Zum Schätzen des Generalisierungsfehlers eines Lernalgorithmus A bei einem Datensatz \mathbb{D} , der zu klein für eine einfache Aufteilung in Training/Test oder Training/Gültigkeit zur Genauigkeitsabschätzung des Generalisierungsfehlers ist, da der Mittelwert eines Verlusts L für eine kleine Trainingsdatenmenge möglicherweise eine zu hohe Varianz aufweist. Die Elemente des Datensatzes \mathbb{D} sind die abstrakten Beispiele $\mathbf{z}^{(i)}$ (für das i -te Beispiel), die bei überwachtem Lernen für ein (Eingabe, Zielwert)-Paar $\mathbf{z}^{(i)} = (\mathbf{x}^{(i)}, y^{(i)})$ und bei unüberwachtem Lernen für eine Eingabe $\mathbf{z}^{(i)} = \mathbf{x}^{(i)}$ stehen könnten. Der Algorithmus gibt den Fehlervektor \mathbf{e} für jedes Beispiel in \mathbb{D} zurück, dessen Mittel der geschätzte Generalisierungsfehler ist. Anhand der Fehler für individuelle Beispiele kann ein Konfidenzintervall für den Mittelwert berechnet werden (Gleichung 5.47). Obwohl diese Konfidenzintervalle nach Einsatz der Kreuzvalidierung nicht gut passen, werden sie allgemein dennoch verwendet, um zu zeigen, dass Algorithmus A besser ist als Algorithmus B , sofern das Konfidenzintervall des Fehlers von Algorithmus A unter dem Konfidenzintervall von Algorithmus B liegt und dieses nicht schneidet.

Define KFoldXV(\mathbb{D}, A, L, k):

Require: \mathbb{D} , der verfügbare Datensatz mit Elementen $\mathbf{z}^{(i)}$

Require: A , der Lernalgorithmus als Funktion, deren Ein- und Ausgangsdaten für eine erlernte Funktion ein Datensatz ist

Require: L , die Verlustfunktion als Funktion einer erlernten Funktion f mit einem Beispiel $\mathbf{z}^{(i)} \in \mathbb{D}$ für einen Skalar $\in \mathbb{R}$

Require: k , die Anzahl der Teilmengen

Teile \mathbb{D} in k einander ausschließende Teilmengen \mathbb{D}_i auf, deren Vereinigungsmenge \mathbb{D} ist

for i von 1 bis k **do**

$f_i = A(\mathbb{D} \setminus \mathbb{D}_i)$

for $\mathbf{z}^{(j)}$ in \mathbb{D}_i **do**

$e_j = L(f_i, \mathbf{z}^{(j)})$

end for

end for

Return \mathbf{e}

des Beispiels zur linearen Regression aus Abschnitt 5.1.4, aber auch eine komplettete Funktion.

Um die Schätzwerte der Parameter von den tatsächlichen Werten zu unterscheiden, notieren wir den Punktschätzwert des Parameters θ als $\hat{\theta}$.

Sei $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ eine Menge aus m unabhängig und identisch verteilten (u.i.v.) Datenpunkten. Ein **Punktschätzer** oder **statistischer Wert** ist jede Funktion der Daten:

$$\hat{\boldsymbol{\theta}}_m = g(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}). \quad (5.19)$$

Die Definition fordert nicht, dass g einen Wert zurückgibt, der dem tatsächlichen $\boldsymbol{\theta}$ nahe kommt; ebenso wenig muss der Wertebereich von g der Menge erlaubter Werte von $\boldsymbol{\theta}$ entsprechen. Diese Definition eines Punktschätzers ist sehr allgemein gehalten und erlaubt extreme Flexibilität beim Entwerfen eines Schätzers. Zwar ist damit nahezu jede Funktion der Definition nach ein Schätzer, aber ein guter Schätzer ist eine Funktion, deren Ausgabe dem tatsächlich zugrunde liegenden $\boldsymbol{\theta}$ zur Erzeugung der Trainingsdaten nahezu entspricht.

Vorerst schließen wir uns der frequentistischen Sichtweise der Statistik an und gehen davon aus, dass der wahre Parameterwert $\boldsymbol{\theta}$ fest, aber unbekannt ist, während der Punktschätzwert $\hat{\boldsymbol{\theta}}$ eine Funktion der Daten ist. Da die Daten einem Zufallsprozess entspringen, ist jede Funktion der Daten zufällig. Folglich ist $\hat{\boldsymbol{\theta}}$ eine Zufallsvariable.

Die Punktschätzung kann sich auch auf die Schätzung der Beziehung zwischen Eingangsdaten und Zielvariablen beziehen. Wir bezeichnen solche Arten von Punktschätzwerten als Funktionsschätzer.

Funktionsschätzung: Manchmal möchten wir eine Funktionsschätzung (oder Funktionsapproximation) vornehmen. Dabei versuchen wir, eine Variable \mathbf{y} anhand eines Eingabevektors \mathbf{x} vorherzusagen. Wir nehmen an, dass es eine Funktion $f(\mathbf{x})$ gibt, die die approximative Beziehung zwischen \mathbf{y} und \mathbf{x} beschreibt. Beispielsweise können wir annehmen, dass $\mathbf{y} = f(\mathbf{x}) + \epsilon$ ist, wobei ϵ für den Teil von \mathbf{y} steht, der sich aus \mathbf{x} nicht vorhersagen lässt. In der Funktionsschätzung sind wir daran interessiert, f mit einem Modell oder einem Schätzwert \hat{f} zu approximieren. Dabei ist die Funktionsschätzung letztlich eine Schätzung eines Parameters $\boldsymbol{\theta}$; der Funktionsschätzer \hat{f} ist einfach ein Punktschätzer im Funktionsraum. Die Beispiele für die lineare Regression (siehe Abschnitt 5.1.4) und die polynomiale Regression (siehe Abschnitt 5.2) zeigen jeweils Szenarios auf, die man auch als Schätzung eines Parameters \mathbf{w} bzw. einer Funktion \hat{f} mit Zuordnung von \mathbf{x} zu y betrachten könnte.

Wir widmen uns nun den meistuntersuchten Eigenschaften von Punktschätzern und zeigen, was wir daraus über diese Schätzer lernen können.

5.4.2 Verzerrung

Die Verzerrung eines Schätzers wird definiert als

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}, \quad (5.20)$$

wobei der Erwartungswert aus den Daten gebildet wird (wenn man sie als Stichproben einer Zufallsvariablen ansieht) und $\boldsymbol{\theta}$ der tatsächlich zugrunde liegende Wert von $\boldsymbol{\theta}$ ist, mit dem die datengenerierende Verteilung definiert wird. Ein Schätzer $\hat{\boldsymbol{\theta}}_m$ wird **erwartungstreu** (oder unverzerrt) genannt, wenn $\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$ ist, was $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$ impliziert. Ein Schätzer $\hat{\boldsymbol{\theta}}_m$ wird **asymptotisch erwartungstreu** genannt, wenn $\lim_{m \rightarrow \infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$ ist, was $\lim_{m \rightarrow \infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$ impliziert.

Beispiel: Bernoulli-Verteilung Gegeben sei eine Menge von Stichproben $\{x^{(1)}, \dots, x^{(m)}\}$, die im Rahmen einer Bernoulli-Verteilung mit dem Mittelwert θ unabhängig und identisch verteilt sind:

$$P(x^{(i)}; \theta) = \theta^{x^{(i)}} (1 - \theta)^{(1-x^{(i)})}. \quad (5.21)$$

Ein gebräuchlicher Schätzer für den Parameter θ dieser Verteilung ist der Mittelwert der Trainingsbeispiele:

$$\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)}. \quad (5.22)$$

Um zu bestimmen, ob dieser Schätzer eine Verzerrung aufweist, können wir Gleichung 5.22 in Gleichung 5.20 einsetzen:

$$\text{bias}(\hat{\theta}_m) = \mathbb{E}[\hat{\theta}_m] - \theta \quad (5.23)$$

$$= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right] - \theta \quad (5.24)$$

$$= \frac{1}{m} \sum_{i=1}^m \mathbb{E}[x^{(i)}] - \theta \quad (5.25)$$

$$= \frac{1}{m} \sum_{i=1}^m \sum_{x^{(i)}=0}^1 \left(x^{(i)} \theta^{x^{(i)}} (1 - \theta)^{(1-x^{(i)})} \right) - \theta \quad (5.26)$$

$$= \frac{1}{m} \sum_{i=1}^m (\theta) - \theta \quad (5.27)$$

$$= \theta - \theta = 0 \quad (5.28)$$

Da $\text{bias}(\hat{\theta}) = 0$ ist, sagen wir, dass unser Schätzer $\hat{\theta}$ erwartungstreu ist.

Beispiel: Schätzer für den Mittelwert einer Normalverteilung Betrachten wir nun eine Menge von Stichproben $\{x^{(1)}, \dots, x^{(m)}\}$, die im Rahmen einer Normalverteilung $p(x^{(i)}) = \mathcal{N}(x^{(i)}; \mu, \sigma^2)$ unabhängig und identisch verteilt sind, wobei gilt $i \in \{1, \dots, m\}$. Wie Sie wissen, wird die Wahrscheinlichkeitsdichtefunktion der Normalverteilung ausgedrückt durch

$$p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x^{(i)} - \mu)^2}{\sigma^2}\right). \quad (5.29)$$

Ein gebräuchlicher Schätzer für den Mittelwert der Normalverteilung wird als **Stichprobenmittelwert** bezeichnet:

$$\hat{\mu}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)} \quad (5.30)$$

Um die Verzerrung des Stichprobenmittelwerts zu bestimmen, berechnen wir wiederum dessen Erwartungswert:

$$\text{bias}(\hat{\mu}_m) = \mathbb{E}[\hat{\mu}_m] - \mu \quad (5.31)$$

$$= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right] - \mu \quad (5.32)$$

$$= \left(\frac{1}{m} \sum_{i=1}^m \mathbb{E}[x^{(i)}]\right) - \mu \quad (5.33)$$

$$= \left(\frac{1}{m} \sum_{i=1}^m \mu\right) - \mu \quad (5.34)$$

$$= \mu - \mu = 0 \quad (5.35)$$

Somit stellen wir fest, dass der Stichprobenmittelwert ein erwartungstreuer Schätzer für den Parameter des Mittelwerts der Normalverteilung ist.

Beispiel: Schätzer für die Varianz einer Normalverteilung Für dieses Beispiel vergleichen wir zwei unterschiedliche Schätzer für den Varianzparameter σ^2 einer Normalverteilung miteinander. Wir möchten wissen, ob einer der Schätzer eine Verzerrung aufweist.

Der erste betrachtete Schätzer σ^2 wird **Stichprobenvarianz** genannt

$$\hat{\sigma}_m^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2, \quad (5.36)$$

wobei $\hat{\mu}_m$ der Stichprobenmittelwert ist. Formal ausgedrückt möchten wir folgenden Term berechnen:

$$\text{bias}(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2. \quad (5.37)$$

Wir berechnen zunächst den Term $\mathbb{E}[\hat{\sigma}_m^2]$:

$$\mathbb{E}[\hat{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2\right] \quad (5.38)$$

$$= \frac{m-1}{m} \sigma^2 \quad (5.39)$$

Anhand von Gleichung 5.37 folgern wir, dass $-\sigma^2/m$ die Verzerrung von $\hat{\sigma}_m^2$ ist. Somit ist die Stichprobenvarianz ein nicht erwartungstreuer (verzerrter) Schätzer.

Der erwartungstreue Schätzer der Stichprobenvarianz

$$\tilde{\sigma}_m^2 = \frac{1}{m-1} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2 \quad (5.40)$$

bietet einen weiteren Ansatz. Wie der Name bereits ausdrückt, ist dieser Schätzer erwartungstreu. Somit gilt $\mathbb{E}[\tilde{\sigma}_m^2] = \sigma^2$:

$$\mathbb{E}[\tilde{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m-1} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2\right] \quad (5.41)$$

$$= \frac{m}{m-1} \mathbb{E}[\hat{\sigma}_m^2] \quad (5.42)$$

$$= \frac{m}{m-1} \left(\frac{m-1}{m} \sigma^2\right) \quad (5.43)$$

$$= \sigma^2. \quad (5.44)$$

Wir verfügen über zwei Schätzer, von denen einer erwartungstreu ist, der andere nicht. Während erwartungstreue Schätzer offenkundig vorzuziehen sind, handelt es sich nicht immer um die »optimalen« Schätzer. Sie werden feststellen, dass wir häufig Schätzer mit einer Verzerrung nutzen, die andere wichtige Eigenschaften aufweisen.

5.4.3 Varianz und Standardfehler

Eine weitere Eigenschaft des Schätzers, die wir in Betracht ziehen sollten, gibt an, welche Variation wir von ihm als Funktion der Datenprobe erwarten können. Neben dem Erwartungswert des Schätzers zum Ermitteln der

Verzerrung können wir auch seine Varianz berechnen. Die **Varianz** eines Schätzers ist ganz einfach die Varianz

$$\text{Var}(\hat{\theta}) \quad (5.45)$$

für die der Trainingsdatenmenge entsprechende Zufallsvariable. Die Quadratwurzel der Varianz wird **Standardfehler** genannt, bezeichnet $\text{SE}(\hat{\theta})$ (von engl. *standard error*).

Die Varianz oder der Standardfehler eines Schätzers ist ein Maß für die erwartete Streuung des aus den Daten der Stichprobe berechneten Schätzwerts, wobei wiederholt neue zufällige Stichproben aus den vorhandenen Daten gezogen werden (engl. *resampling*). Neben einer geringen Verzerrung des Schätzers wünschen wir uns auch eine relativ geringe Varianz.

Beim Berechnen eines statistischen Werts mittels einer endlichen Anzahl von Stichproben ist unser Schätzwert des tatsächlich zugrunde liegenden Parameters insofern unsicher, als wir andere Stichproben aus derselben Verteilung hätten wählen können, deren statistischen Werte anders ausgesehen hätten. Die erwartete Streuung eines Schätzers ist eine Fehlerquelle, die wir bemessen möchten.

Der Standardfehler des Mittelwerts ergibt sich aus

$$\text{SE}(\hat{\mu}_m) = \sqrt{\text{Var}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}, \quad (5.46)$$

wobei σ^2 die tatsächliche Varianz der Stichproben x^i ist. Der Standardfehler wird häufig anhand eines Schätzwerts σ geschätzt. Leider sind weder die Quadratwurzel der Stichprobenvarianz noch die Quadratwurzel des erwartungstreuen Schätzers der Varianz erwartungstreue Schätzwerte für die Standardabweichung. Beide Ansätze neigen zum Unterschätzen der tatsächlichen Standardabweichung, werden jedoch in der Praxis nach wie vor verwendet. Die Quadratwurzel des erwartungstreuen Schätzers der Varianz neigt weniger zur Unterschätzung. Für große m ist die Approximation durchaus sinnvoll.

Der Standardfehler des Mittelwerts ist in Machine-Learning-Experimenten sehr hilfreich. Wir schätzen häufig den Generalisierungsfehler, indem wir den Stichprobenmittelwert des Fehlers für die Testdatenmenge berechnen. Die Anzahl der Objekte in der Testdatenmenge bestimmt die Korrektklassifikationsrate dieses Schätzwerts. Durch Ausnutzung des zentralen Grenzwertsatzes, der besagt, dass der Mittelwert in einer Normalverteilung näherungsweise verteilt ist, können wir den Standardfehler verwenden,

um die Wahrscheinlichkeit zu berechnen, mit der der tatsächliche Erwartungswert in ein frei gewähltes Intervall fällt. So beträgt das 95-Prozent-Konfidenzintervall um den Mittelwert $\hat{\mu}_m$

$$(\hat{\mu}_m - 1,96\text{SE}(\hat{\mu}_m), \hat{\mu}_m + 1,96\text{SE}(\hat{\mu}_m)), \quad (5.47)$$

für die Normalverteilung mit dem Erwartungswert $\hat{\mu}_m$ und der Varianz $\text{SE}(\hat{\mu}_m)^2$. In Machine-Learning-Experimenten sagen wir gerne, dass Algorithmus A besser ist als Algorithmus B , sofern die obere Schranke des 95-Prozent-Konfidenzintervalls des Fehlers für Algorithmus A kleiner als die untere Schranke des 95-Prozent-Konfidenzintervalls des Fehlers für Algorithmus B ist.

Beispiel: Bernoulli-Verteilung Wir betrachten erneut eine Menge von Stichproben $\{x^{(1)}, \dots, x^{(m)}\}$, die unabhängig und identisch aus einer Bernoulli-Verteilung gezogen wurden (beachten Sie, dass $P(x^{(i)}; \theta) = \theta^{x^{(i)}}(1 - \theta)^{1-x^{(i)}}$ ist). Dieses Mal möchten wir die Varianz des Schätzers $\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)}$ berechnen.

$$\text{Var}(\hat{\theta}_m) = \text{Var}\left(\frac{1}{m} \sum_{i=1}^m x^{(i)}\right) \quad (5.48)$$

$$= \frac{1}{m^2} \sum_{i=1}^m \text{Var}(x^{(i)}) \quad (5.49)$$

$$= \frac{1}{m^2} \sum_{i=1}^m \theta(1 - \theta) \quad (5.50)$$

$$= \frac{1}{m^2} m\theta(1 - \theta) \quad (5.51)$$

$$= \frac{1}{m} \theta(1 - \theta) \quad (5.52)$$

Die Varianz des Schätzers nimmt als Funktion von m (der Anzahl Objekte im Datensatz) ab. Dies ist eine häufige Eigenschaft gebräuchlicher Schätzer, auf die wir im Rahmen des Themas Konsistenz zurückkommen werden (siehe Abschnitt 5.4.5).

5.4.4 Abstimmen von Verzerrung und Varianz zum Minimieren des mittleren quadratischen Fehlers

Verzerrung und Varianz messen zwei unterschiedliche Fehlerquellen eines Schätzers. Die Verzerrung misst die erwartete Abweichung vom tatsächlichen

Wert der Funktion oder des Parameters. Die Varianz gibt dagegen ein Maß für die Abweichung vom Erwartungswert des Schätzers an, die jede einzelne Probenahme der Daten wahrscheinlich verursacht.

Was geschieht, wenn wir zwischen zwei Schätzern wählen können, von denen einer eine höhere Verzerrung aufweist, der andere eine höhere Varianz? Wie wählen wir zwischen diesen beiden? Ein Beispiel: Wir suchen die Approximation der Funktion aus Abbildung 5.2 und haben lediglich die Wahl zwischen einem Modell mit einer großen Verzerrung und einem mit einer großen Varianz. Wie treffen wir eine Wahl zwischen diesen beiden?

In solchen Fällen ist als Kompromiss die Kreuzvalidierung gängig. Empirisch betrachtet ist die Kreuzvalidierung bei vielen Aufgaben in der Realität sehr erfolgreich. Wir können auch den **mittleren quadratischen Fehler** (MQF) der Schätzwerte vergleichen:

$$\text{MQF} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] \quad (5.53)$$

$$= \text{Bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m) \quad (5.54)$$

Der MQF misst die insgesamt erwartete Abweichung – im Sinne eines quadratischen Fehlers – zwischen dem Schätzer und dem tatsächlichen Wert des Parameters θ . Aus Gleichung 5.54 ist klar, dass die Berechnung des MQF sowohl Verzerrung als auch Varianz einbezieht. Wünschenswerte Schätzer weisen einen kleinen MQF auf; diese Schätzer helfen dabei, Verzerrung und Varianz einigermaßen im Zaum zu halten.

Die Beziehung zwischen Verzerrung und Varianz ist eng mit den Machine-Learning-Konzepten in puncto Kapazität, Unteranpassung und Überanpassung verknüpft. Wenn der Generalisierungsfehler als MQF gemessen wird (in dem Verzerrung und Varianz aussagekräftige Komponenten des Generalisierungsfehlers sind), führt eine Erhöhung der Kapazität häufig zu einer Erhöhung der Varianz bei gleichzeitiger Verringerung der Verzerrung. Abbildung 5.6 verdeutlicht dies: Auch hier tritt als Funktion der Kapazität eine U-förmige Kurve für den Generalisierungsfehler auf.

5.4.5 Konsistenz

Bisher haben wir die Eigenschaften unterschiedlicher Schätzer für eine Trainingsdatenmenge fester Größe betrachtet. Meist interessiert uns aber auch das Verhalten des Schätzers bei einer wachsenden Trainingsdatenmenge. So ist es uns normalerweise wichtig, dass mit einer wachsenden Anzahl von Datenpunkten m in unserem Datensatz die Punktschätzwerte gegen den wah-

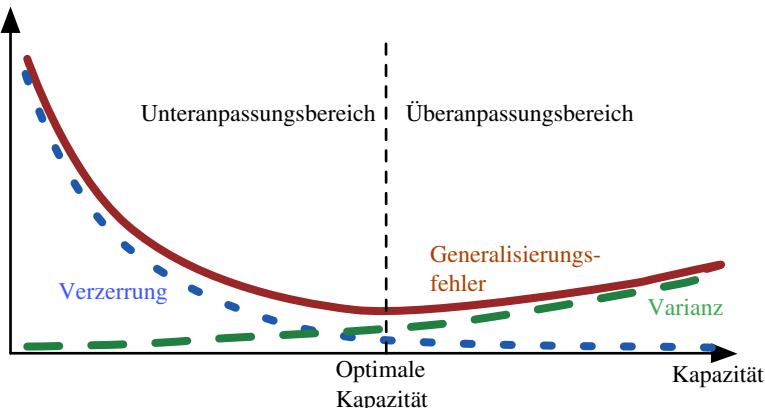


Abbildung 5.6: Mit zunehmender Kapazität (x -Achse) neigt die Verzerrung (gepunktet) zur Abnahme und die Varianz (gestrichelt) zur Zunahme, wodurch eine weitere U-förmige Kurve für den Generalisierungsfehler entsteht (dicke Linie). Wenn wir die Kapazität entlang einer Achse verändern, stoßen wir auf eine optimale Kapazität: Unterhalb des Optimums kommt es zur Unteranpassung, darüber zur Überanpassung. Diese Beziehung ähnelt der zwischen Kapazität, Unteranpassung und Überanpassung, die wir in Abschnitt 5.2 und Abbildung 5.3 betrachtet haben.

ren Wert der entsprechenden Parameter konvergieren. Formal ausgedrückt lautet die Forderung:

$$\text{plim}_{m \rightarrow \infty} \hat{\theta}_m = \theta. \quad (5.55)$$

Das Symbol plim gibt die Konvergenz der Wahrscheinlichkeit an; es bedeutet, dass für jedes $\epsilon > 0$, $P(|\hat{\theta}_m - \theta| > \epsilon) \rightarrow 0$ für $m \rightarrow \infty$ gilt. Die durch Gleichung 5.55 beschriebene Bedingung wird als **Konsistenz** bezeichnet. Manchmal wird sie auch *schwache Konsistenz* genannt, um sie von der *starken Konsistenz* abzugrenzen, die sich auf die **fast sichere Konvergenz** von $\hat{\theta}$ zu θ bezieht. Die **fast sichere Konvergenz** einer Sequenz von Zufallsvariablen $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots$ für einen Wert \mathbf{x} tritt auf, wenn $p(\lim_{m \rightarrow \infty} \mathbf{x}^{(m)} = \mathbf{x}) = 1$ ist.

Konsistenz stellt sicher, dass die vom Schätzer induzierte Verzerrung mit steigendem Stichprobenumfang abnimmt. Allerdings ist der Umkehrschluss nicht korrekt: Eine asymptotische Erwartungstreue (auch asymptotische Unverzerrtheit genannt) impliziert keine Konsistenz. Ein Beispiel: Der Mittelwert des Parameters μ einer Normalverteilung $\mathcal{N}(x; \mu, \sigma^2)$ wird geschätzt, wobei der Datensatz aus m Stichproben besteht: $\{x^{(1)}, \dots, x^{(m)}\}$. Wir können die erste Stichprobe $x^{(1)}$ des Datensatzes als erwartungstreuen Schätzer nutzen: $\hat{\theta} = x^{(1)}$. In dem Fall gilt $\mathbb{E}(\hat{\theta}_m) = \theta$, sodass der Schätzer unabhängig der Anzahl betrachteter Datenpunkte stets erwartungstreu ist. Das

impliziert natürlich, dass der Schätzwert asymptotisch erwartungstreu ist. Allerdings handelt es sich hier nicht um einen konsistenten Schätzer, da $\hat{\theta}_m \rightarrow \theta$ nicht zutrifft für $m \rightarrow \infty$.

5.5 Maximum-Likelihood-Schätzung

Wir haben einige Definitionen für gängige Schätzer vorgestellt und ihre Eigenschaften analysiert. Aber woher stammen diese Schätzer? Anstatt einfach zu raten, ob irgendeine Funktion einen guten Schätzer abgibt und im Anschluss deren Verzerrung und Varianz zu überprüfen, möchten wir ein Rezept haben, das uns bei der Ableitung bestimmter Funktionen hilft, die sich für unterschiedliche Modelle als gute Schätzer erweisen.

Zum Glück gibt es ein solches Rezept; es nennt sich Maximum Likelihood.

Gegeben sei eine Menge mit m Beispielen $\mathbb{X} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$, die unabhängig aus der wahren, aber unbekannten datengenerierenden Verteilung $p_{\text{data}}(\mathbf{x})$ entnommen wurden.

Es sei $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ eine parametrische Familie von Wahrscheinlichkeitsverteilungen über denselben Raum, der von $\boldsymbol{\theta}$ indiziert wird. Anders ausgedrückt: $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ ordnet jede Konfiguration \mathbf{x} einer reellen Zahl zur Schätzung der tatsächlichen Wahrscheinlichkeit $p_{\text{data}}(\mathbf{x})$ zu.

Der Maximum-Likelihood-Schätzer für $\boldsymbol{\theta}$ wird dann definiert als

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}), \quad (5.56)$$

$$= \arg \max_{\boldsymbol{\theta}} \prod_{i=1}^m p_{\text{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \quad (5.57)$$

Dieses Produkt über viele Wahrscheinlichkeiten kann sich aus diversen Gründen als unpraktisch erweisen. Es neigt zum Beispiel zu einem numerischen Unterlauf. Für ein praktikableres, aber gleichwertiges Optimierungsproblem stellen wir fest, dass der Logarithmus der Likelihood den Wert $\arg \max$ nicht verändert, aber das Produkt gefällig in eine Summe umwandelt:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log p_{\text{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \quad (5.58)$$

Da $\arg \max$ sich beim Skalieren der Kostenfunktion nicht verändert, können wir m aufteilen, um eine Version des Kriteriums zu erhalten, die als Erwar-

tungswert bezüglich der durch die Trainingsdaten definierten empirischen Verteilung \hat{p}_{data} ausgedrückt wird:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}). \quad (5.59)$$

Eine Möglichkeit zur Interpretation der Maximum-Likelihood-Schätzung besteht darin zu sagen, dass sie die Unähnlichkeit zwischen der empirischen Verteilung \hat{p}_{data} , die durch die Trainingsdatenmenge definiert wird, und der Modellverteilung minimiert, wobei der Grad der Unähnlichkeit zwischen den beiden durch die KL-Divergenz gemessen wird. Die KL-Divergenz ergibt sich aus

$$D_{\text{KL}}(\hat{p}_{\text{data}} \| p_{\text{model}}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} [\log \hat{p}_{\text{data}}(\mathbf{x}) - \log p_{\text{model}}(\mathbf{x})]. \quad (5.60)$$

Der linke Term ist eine reine Funktion des datengenerierenden Prozesses, nicht des Modells. Wenn wir daher das Modell trainieren, um die KL-Divergenz zu minimieren, müssen wir lediglich

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} [\log p_{\text{model}}(\mathbf{x})] \quad (5.61)$$

minimieren, was natürlich der Maximierung aus Gleichung 5.59 entspricht.

Das Minimieren der KL-Divergenz entspricht exakt dem Minimieren der Kreuzentropie zwischen den Verteilungen. Viele Autoren nutzen den Begriff »Kreuzentropie« ausschließlich für die negative Log-Likelihood einer Bernoulli- oder softmax-Verteilung, aber das ist nicht korrekt. Jeder Verlust, der aus einer negativen Log-Likelihood besteht, ist eine Kreuzentropie zwischen der empirischen Verteilung, die durch die Trainingsdatenmenge definiert wird, und der Wahrscheinlichkeitsverteilung, die durch das Modell definiert wird. Ein Beispiel: Der mittlere quadratische Fehler ist die Kreuzentropie zwischen der empirischen Verteilung und einem gaußschen Modell.

Die Maximum Likelihood erweist sich also als Versuch, die Modellverteilung an die empirische Verteilung \hat{p}_{data} anzugeleichen. Idealerweise wäre es die wahre datengenerierende Verteilung p_{data} , aber wir haben keinen direkten Zugang zu dieser Verteilung.

Obwohl das optimale $\boldsymbol{\theta}$ ungeachtet der Maximierung der Likelihood oder der Minimierung der KL-Divergenz identisch ist, unterscheiden sich die Werte der Zielfunktion doch. Im Softwarekontext sagen wir häufig, dass beide eine Kostenfunktion minimieren. Daher wird die Maximum Likelihood zur Minimierung der negativen Log-Likelihood (NLL) bzw. entsprechend zur Minimierung der Kreuzentropie. Die Betrachtung der Maximum Likelihood

als kleinste KL-Divergenz ist in diesem Fall hilfreich, da der kleinste Wert der KL-Divergenz, nämlich 0, bekannt ist. Die negative Log-Likelihood kann tatsächlich für ein reellwertiges \boldsymbol{x} negativ werden.

5.5.1 Bedingte Log-Likelihood und mittlerer quadratischer Fehler

Der Maximum-Likelihood-Schätzer kann leicht zum Schätzwert einer bedingten Wahrscheinlichkeit $P(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ generalisiert werden, um \mathbf{y} aus \mathbf{x} vorherzusagen. Dies ist tatsächlich der häufigste Fall, denn er bildet die Grundlage für den Großteil des überwachten Lernens. Wenn \mathbf{X} sämtliche Eingaben darstellt und \mathbf{Y} sämtliche beobachteten Ziele, dann ist der bedingte Maximum-Likelihood-Schätzer

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} P(\mathbf{Y} \mid \mathbf{X}; \boldsymbol{\theta}). \quad (5.62)$$

Wenn die Beispiele als u.i.v. betrachtet werden, dann wird folgende Zerlegung möglich:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}; \boldsymbol{\theta}). \quad (5.63)$$

Beispiel: Lineare Regression als Maximum Likelihood Die in Abschnitt 5.1.4 vorgestellte lineare Regression kann als Maximum-Likelihood-Verfahren erklärt werden. Bisher haben wir die lineare Regression als Algorithmus betrachtet, der lernt, aus einem Eingabewert \boldsymbol{x} einen Ausgabewert \hat{y} zu erzeugen. Die Zuordnung von \boldsymbol{x} zu \hat{y} wird so gewählt, dass der mittlere quadratische Fehler minimiert wird – ein Kriterium, das wir mehr oder weniger willkürlich festgelegt hatten. Betrachten wir nun die lineare Regression vom Standpunkt der Maximum-Likelihood-Schätzung: Statt eine einzelne Vorhersage \hat{y} hervorzu bringen, stellen wir uns das Modell nun so vor, dass eine bedingte Verteilung $p(y \mid \boldsymbol{x})$ entsteht. Wir können uns vorstellen, dass für eine unendlich große Trainingsdatenmenge mehrere Trainingsbeispiele mit identischem Eingabewert \boldsymbol{x} , aber unterschiedlichen Werten y existieren. Das Ziel des Lernalgorithmus besteht nun darin, die Verteilung $p(y \mid \boldsymbol{x})$ an all diese unterschiedlichen y -Werte anzupassen, die alle mit \boldsymbol{x} kompatibel sind. Um denselben linearen Regressionsalgorithmus wie bisher zu erhalten, definieren wir $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; \hat{y}(\boldsymbol{x}; \mathbf{w}), \sigma^2)$. Die Funktion $\hat{y}(\boldsymbol{x}; \mathbf{w})$ ergibt die Vorhersage des Mittelwerts der Normalverteilung. In diesem Beispiel gehen wir davon aus, dass die Varianz eine feste, vom Benutzer festgelegte Konstante σ^2 ist. Sie werden sehen, dass die Wahl der funktionalen Form

$p(y | \mathbf{x})$ dazu führt, dass die Maximum-Likelihood-Schätzung den von uns zuvor entwickelten Lernalgorithmus erzeugt. Da die Beispiele als u.i.v. betrachtet werden, ergibt sich die bedingte Log-Likelihood (Gleichung 5.63) aus

$$\sum_{i=1}^m \log p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta}) \quad (5.64)$$

$$= -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^m \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}, \quad (5.65)$$

wobei $\hat{y}^{(i)}$ die Ausgabe der linearen Regression für die i -te Eingabe $\mathbf{x}^{(i)}$ ist und m die Anzahl der Trainingsbeispiele. Beim Vergleich der Log-Likelihood mit dem mittleren quadratischen Fehler

$$\text{MQF}_{\text{train}} = \frac{1}{m} \sum_{i=1}^m \|\hat{y}^{(i)} - y^{(i)}\|^2, \quad (5.66)$$

sehen wir sofort, dass durch Maximieren der Log-Likelihood für \mathbf{w} derselbe Schätzwert für den Parameter \mathbf{w} entsteht wie beim Minimieren des mittleren quadratischen Fehlers. Die beiden Kriterien weisen unterschiedliche Werte auf, aber dieselbe Stelle des Optimums. Das begründet den Einsatz des MQF als Verfahren zur Maximum-Likelihood-Schätzung. Wie wir noch sehen werden, zeichnet sich der Maximum-Likelihood-Schätzer durch mehrere wünschenswerte Eigenschaften aus.

5.5.2 Eigenschaften der Maximum Likelihood

Der wichtigste Vorteil des Maximum-Likelihood-Schätzers besteht darin, dass er sich mit wachsender Zahl der Beispiele $m \rightarrow \infty$, asymptotisch als bester Schätzer in puncto Konvergenz bei zunehmendem m erweist.

Unter angemessenen Bedingungen ist der Maximum-Likelihood-Schätzer konsistent (siehe Abschnitt 5.4.5), d. h. bei Streben der Anzahl der Trainingsbeispiele gegen unendlich konvergiert der Maximum-Likelihood-Schätzer eines Parameters zum wahren Parameterwert. Diese Bedingungen sind die folgenden:

- Die wahre Verteilung p_{data} muss innerhalb der Modellfamilie $p_{\text{model}}(\cdot; \boldsymbol{\theta})$ liegen. Ansonsten kann kein Schätzer p_{data} wiederherstellen.

- Die wahre Verteilung p_{data} muss exakt einem Wert von θ entsprechen. Ansonsten kann die Maximum Likelihood die korrekte p_{data} wiederherstellen, aber nicht ermitteln, welcher Wert von θ für den datengenerierenden Prozess verwendet wurde.

Es gibt weitere induktive Prinzipien neben dem Maximum-Likelihood-Schätzer, von denen viele ebenfalls konsistente Schätzer sind. Konsistente Schätzer können sich jedoch hinsichtlich ihrer **statistischen Effizienz** unterscheiden. Das bedeutet, dass ein konsistenter Schätzer einen geringeren Generalisierungsfehler für eine feste Anzahl von Stichproben m erzielen kann bzw. weniger Beispiele benötigt, um ein bestimmtes Niveau des Generalisierungsfehlers zu erzielen.

Statistische Effizienz wird normalerweise für den **parametrischen Fall** (wie bei der linearen Regression) untersucht, um den Wert eines Parameters (sofern es möglich ist, den tatsächlichen Parameter zu bestimmen) zu schätzen, nicht etwa den Wert einer Funktion. Der erwartete mittlere quadratische Fehler bietet eine Möglichkeit zur Messung der Annäherung an den tatsächlichen Parameter. Dazu wird das Quadrat der Differenz zwischen dem geschätzten und dem tatsächlichen Parameterwert berechnet, wobei der Erwartungswert über m Trainingsstichproben aus der datengenerierenden Verteilung gebildet wird. Der parametrische mittlere quadratische Fehler nimmt bei zunehmendem m ab. Ist m groß, zeigt die Cramér-Rao-Ungleichung (*Rao*, 1945; *Cramér*, 1946), dass kein konsistenter Schätzer einen niedrigeren MQF als der Maximum-Likelihood-Schätzer aufweist.

Aus diesen Gründen (Konsistenz und Effizienz) wird die Maximum Likelihood häufig als bevorzugter Schätzer für das Machine Learning betrachtet. Wenn die Anzahl der Beispiele klein genug für eine Überanpassung ist, können Regularisierungsverfahren wie Weight Decay eingesetzt werden, um eine verzerrte Version der Maximum Likelihood zu erhalten, die eine geringere Varianz aufweist, wenn die Trainingsdaten eingeschränkt sind.

5.6 Bayessche Statistik

Bisher haben wir die **frequentistische Statistik** und Ansätze für die Schätzung eines einzelnen Werts θ behandelt, um auf diesem einzelnen Schätzwert weitere Vorhersagen aufzubauen. Eine andere Herangehensweise betrachtet alle möglichen Werte von θ bei der Vorhersage. Damit begeben wir uns in den Bereich der **bayesschen Statistik**.

Wie wir in Abschnitt 5.4.1 betrachtet haben, besagt die frequentistische Sichtweise, dass der tatsächliche Parameterwert θ zwar unbekannt, aber fest ist, während der Punktschätzwert $\hat{\theta}$ eine Zufallsvariable ist, da es sich um eine Funktion des (als zufällig geltenden) Datensatzes handelt.

Die bayessche Sichtweise zur Statistik sieht ganz anders aus und benutzt Wahrscheinlichkeiten, um den Grad der Gewissheit in Kenntnisständen darzustellen. Der Datensatz wird direkt beobachtet und ist somit nicht zufällig. Andererseits ist der tatsächliche Parameter θ unbekannt oder unsicher und wird somit als Zufallsvariable dargestellt.

Vor der Beobachtung der Daten stellen wir unsere Kenntnis über θ mithilfe der **A-priori-Wahrscheinlichkeitsverteilung** $p(\theta)$ dar (kurz als *A-priori-Wahrscheinlichkeit* bezeichnet). Generell wählen wir im Machine Learning eine relativ umfassende A-priori-Verteilung aus (also eine solche mit hoher Entropie), um einen hohen Grad der Ungewissheit für den Wert θ vor dem Beobachten der Daten anzugeben. So könnte man a priori annehmen, dass θ in einem endlichen Bereich oder Raum mit einer Gleichverteilung liegt. Viele A-priori-Wahrscheinlichkeiten zeigen eine Präferenz für »einfachere« Lösungen (z. B. Koeffizienten kleinerer Größenordnung oder eine Funktion, die eher konstant ist).

Nehmen wir an, wir haben eine Menge von Stichproben: $\{x^{(1)}, \dots, x^{(m)}\}$. Wir können die Auswirkung der Daten auf unsere Überzeugung bezüglich θ wiederherstellen, indem wir die Daten-Likelihood $p(x^{(1)}, \dots, x^{(m)} | \theta)$ mit der A-priori-Wahrscheinlichkeit über den Satz von Bayes kombinieren:

$$p(\theta | x^{(1)}, \dots, x^{(m)}) = \frac{p(x^{(1)}, \dots, x^{(m)} | \theta)p(\theta)}{p(x^{(1)}, \dots, x^{(m)})} \quad (5.67)$$

In den Fällen, in denen die bayessche Schätzung normalerweise genutzt wird, beginnt die A-priori-Wahrscheinlichkeit als relativ gleichförmige oder Normalverteilung mit hoher Entropie; die Beobachtung der Daten führt meist zu einem Entropieverlust der A-posteriori-Wahrscheinlichkeit und einer Konzentration auf einige hoch wahrscheinliche Werte des Parameters.

Bezüglich der Maximum-Likelihood-Schätzung weist die bayessche Schätzung zwei wichtige Unterschiede auf: Erstens macht der bayessche Ansatz Vorhersagen aufgrund einer vollständigen Verteilung über θ , anders als die Maximum Likelihood, bei der die Vorhersagen auf einem Punktschätzwert θ basieren. So ergibt sich zum Beispiel nach der Beobachtung von m Beispielen die vorhergesagte Verteilung über die nächste Stichprobe $x^{(m+1)}$ aus

$$p(x^{(m+1)} | x^{(1)}, \dots, x^{(m)}) = \int p(x^{(m+1)} | \theta)p(\theta | x^{(1)}, \dots, x^{(m)}) d\theta. \quad (5.68)$$

Hier trägt jeder Wert von θ mit positiver Wahrscheinlichkeitsdichte zur Vorhersage des nächsten Beispiels bei, wobei die Verteilung anhand der Dichte der A-posteriori-Wahrscheinlichkeit selbst gewichtet wird. Nachdem $\{x^{(1)}, \dots, x^{(m)}\}$ beobachtet wurde, fließt – sofern es noch immer eine große Ungewissheit bezüglich des Wertes für θ gibt – die Ungewissheit direkt in weitere Vorhersagen ein.

In Abschnitt 5.4 haben wir gezeigt, wie der frequentistische Ansatz die Ungewissheit für einen bestimmten Punktschätzwert θ durch Berechnen seiner Varianz berücksichtigt. Die Varianz des Schätzers ist eine Beurteilung, wie der Schätzwert sich bei anderen Probenahmen für die beobachteten Daten ändert. Die bayessche Antwort auf die Frage nach dem Umgang mit der Ungewissheit im Schätzer besteht darin, einfach darüber zu integrieren, was meist einen guten Schutz gegen eine Überanpassung darstellt. Dieses Integral ist natürlich nur eine Anwendung der Wahrscheinlichkeitsgesetze, wodurch der bayessche Ansatz leicht belegt werden kann, während die frequentistische Maschine zur Konstruktion eines Schätzers auf der Ad-hoc-Entscheidung beruht, sämtliche Kenntnisse aus dem Datensatz als einzelnen Punktschätzwert zusammenzufassen.

Der zweite wichtige Unterschied zwischen dem bayesschen Ansatz zur Schätzung und dem Maximum-Likelihood-Ansatz ist bedingt durch die Verteilung der bayesschen A-priori-Verteilung. Die A-priori-Wahrscheinlichkeit führt zu einer Verschiebung der Dichte der Wahrscheinlichkeitsmasse in Bereiche des Parameterraums, die a priori bevorzugt werden. In der Praxis drückt die A-priori-Wahrscheinlichkeit häufig eine Präferenz für einfachere oder glattere Modelle aus. Kritiker des bayesschen Ansatzes bezeichnen die A-priori-Wahrscheinlichkeit als Quelle einer subjektiven menschlichen Einschätzung, die die Vorhersagen beeinflusst.

Bayessche Verfahren generalisieren im Allgemeinen deutlich besser, wenn nur eingeschränkte Trainingsdaten vorliegen. Mit steigender Anzahl der Trainingsbeispiele kommt es jedoch meist zu einem höheren Berechnungsaufwand.

Beispiel: Bayessche lineare Regression In diesem Beispiel betrachten wir den bayesschen Ansatz zur Schätzung für das Erlernen der Parameter der linearen Regression. Bei der linearen Regression erlernen wir eine lineare Zuordnung eines Eingabevektors $x \in \mathbb{R}^n$ zur Vorhersage des Werts eines Skalars $y \in \mathbb{R}$. Die Vorhersage wird mit dem Vektor $w \in \mathbb{R}^n$ parametrisiert:

$$\hat{y} = w^\top x. \quad (5.69)$$

Für eine Menge von m Trainingsstichproben ($\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})}$) können wir die Vorhersage für y über die gesamte Trainingsdatenmenge so ausdrücken:

$$\hat{\mathbf{y}}^{(\text{train})} = \mathbf{X}^{(\text{train})}\mathbf{w}. \quad (5.70)$$

Ausgedrückt als bedingte Normalverteilung für $\mathbf{y}^{(\text{train})}$ erhalten wir

$$p(\mathbf{y}^{(\text{train})} | \mathbf{X}^{(\text{train})}, \mathbf{w}) = \mathcal{N}(\mathbf{y}^{(\text{train})}; \mathbf{X}^{(\text{train})}\mathbf{w}, \mathbf{I}) \quad (5.71)$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{y}^{(\text{train})} - \mathbf{X}^{(\text{train})}\mathbf{w})^\top(\mathbf{y}^{(\text{train})} - \mathbf{X}^{(\text{train})}\mathbf{w})\right), \quad (5.72)$$

wobei wir der üblichen MQF-Formulierung folgen und davon ausgehen, dass die Varianz der Normalverteilung für y Eins ist. Im Folgenden erleichtern wir uns die Notation, indem wir $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$ einfach mit (\mathbf{X}, \mathbf{y}) bezeichnen.

Zum Bestimmen der A-priori-Verteilung über den Modellparametervektor \mathbf{w} müssen wir zunächst eine A-posteriori-Verteilung angeben. Die A-priori-Wahrscheinlichkeit sollte unsere naive Überzeugung bezüglich des Werts dieser Parameter ausdrücken. Es ist nicht immer leicht oder natürlich, unsere vorherige Überzeugung als Modellparameter anzugeben. In der Praxis gehen wir daher meist von einer relativ breiten Verteilung aus, die einen hohen Grad der Ungewissheit über $\boldsymbol{\theta}$ zum Ausdruck bringt. Für reellwertige Parameter wird üblicherweise eine Normalverteilung als A-priori-Verteilung verwendet:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0) \propto \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right), \quad (5.73)$$

wobei $\boldsymbol{\mu}_0$ und $\boldsymbol{\Lambda}_0$ den Mittelwertvektor der A-priori-Verteilung bzw. die Kovarianz angeben.¹

Nachdem die A-priori-Wahrscheinlichkeit festgelegt ist, können wir im weiteren Verlauf die **A-posteriori**-Verteilung über die Modellparameter bestimmen:

$$p(\mathbf{w} | \mathbf{X}, \mathbf{y}) \propto p(\mathbf{y} | \mathbf{X}, \mathbf{w})p(\mathbf{w}) \quad (5.74)$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w})\right) \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right) \quad (5.75)$$

$$\propto \exp\left(-\frac{1}{2} \left(-2\mathbf{y}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \boldsymbol{\Lambda}_0^{-1}\mathbf{w} - 2\boldsymbol{\mu}_0^\top \boldsymbol{\Lambda}_0^{-1}\mathbf{w} \right) \right). \quad (5.76)$$

¹ Sofern es keinen Grund für eine spezielle Kovarianzstruktur gibt, gehen wir üblicherweise von einer diagonalen Kovarianzmatrix aus: $\boldsymbol{\Lambda}_0 = \text{diag}(\boldsymbol{\lambda}_0)$.

Nun definieren wir $\Lambda_m = (\mathbf{X}^\top \mathbf{X} + \Lambda_0^{-1})^{-1}$ und $\mu_m = \Lambda_m (\mathbf{X}^\top \mathbf{y} + \Lambda_0^{-1} \mu_0)$. Anhand dieser neuen Variable stellen wir fest, dass die A-posteriori-Wahrscheinlichkeit als Normalverteilung neu notiert werden kann:

$$p(\mathbf{w} | \mathbf{X}, \mathbf{y}) \propto \exp\left(-\frac{1}{2}(\mathbf{w} - \mu_m)^\top \Lambda_m^{-1}(\mathbf{w} - \mu_m) + \frac{1}{2}\mu_m^\top \Lambda_m^{-1} \mu_m\right) \quad (5.77)$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{w} - \mu_m)^\top \Lambda_m^{-1}(\mathbf{w} - \mu_m)\right). \quad (5.78)$$

Alle Terme, die den Parametervektor \mathbf{w} nicht enthalten, wurden weggelassen; sie werden durch die Tatsache impliziert, dass die Verteilung für die Integration gegen 1 normalisiert werden muss. Gleichung 3.23 zeigt das Normalisieren einer mehrdimensionalen Normalverteilung.

Durch Untersuchen dieser A-posteriori-Verteilung gelangen wir zu einem gewissen Gespür für die Auswirkung der bayesschen Inferenz. In den meisten Fällen bestimmen wir $\mu_0 = \mathbf{0}$. Für $\Lambda_0 = \frac{1}{\alpha} \mathbf{I}$ ergibt μ_m denselben Schätzwert für \mathbf{w} wie die frequentistische lineare Regression mit einem Weight-Decay-Strafterm von $\alpha \mathbf{w}^\top \mathbf{w}$. Ein Unterschied besteht darin, dass der bayessche Schätzwert für $\alpha = 0$ nicht definiert ist – wir können den bayesschen Lernprozess nicht mit einer unendlich breiten A-priori-Wahrscheinlichkeit für \mathbf{w} starten. Der gewichtigere Unterschied ist, dass der bayessche Schätzwert eine Kovarianzmatrix erzeugt, die zeigt, wie wahrscheinlich die verschiedenen Werte von \mathbf{w} sind – nicht nur den Schätzwert μ_m .

5.6.1 Maximum-a-posteriori-Methode (MAP)

Der grundlegende Ansatz besteht darin, Vorhersagen mithilfe der vollständigen bayesschen A-posteriori-Verteilung für den Parameter θ zu treffen. Allerdings ist es häufig wünschenswert, einen einzelnen Punktschätzwert zu nutzen. Ein häufiger Grund für diesen Wunsch liegt darin, dass die meisten Operationen, bei denen die bayessche A-posteriori-Wahrscheinlichkeit auf die interessantesten Modelle angewandt wird, nicht effizient durchführbar (engl. *intractable*) sind, wohingegen ein Punktschätzwert eine effizient durchführbare Approximation (engl. *tractable approximation*) darstellt. Statt einfach den Maximum-Likelihood-Schätzwert zu nutzen, können wir den bayesschen Ansatz doch noch zu unserem Vorteil nutzen, indem wir der A-priori-Wahrscheinlichkeit erlauben, die Wahl des Punktschätzwerts zu beeinflussen. Eine rationale Möglichkeit besteht in der Auswahl des MAP-Punktschätzwerts (**Maximum-a-posteriori-Methode**). Der MAP-Schätzwert wählt den Punkt der maximalen A-posteriori-Wahrscheinlichkeit aus (oder der

maximalen Wahrscheinlichkeitsdichte für den häufigeren Fall eines stetigen θ :

$$\boldsymbol{\theta}_{\text{MAP}} = \arg \max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathbf{x}) = \arg \max_{\boldsymbol{\theta}} \log p(\mathbf{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}). \quad (5.79)$$

Auf der rechten Seite stehen mit $\log p(\mathbf{x} \mid \boldsymbol{\theta})$ der Log-Likelihood-Standardterm und $\log p(\boldsymbol{\theta})$ für die A-priori-Verteilung.

Ein Beispiel: Gegeben sei ein lineares Regressionsmodell mit einer normalverteilten A-priori-Wahrscheinlichkeit für die Gewichte \mathbf{w} . Ergibt sich diese A-priori-Wahrscheinlichkeit aus $\mathcal{N}(\mathbf{w}; \mathbf{0}, \frac{1}{\lambda} \mathbf{I}^2)$, dann ist der Term der Log-a-priori-Wahrscheinlichkeit in Gleichung 5.79 proportional zum bekannten Weight-Decay-Strafterm $\lambda \mathbf{w}^\top \mathbf{w}$ zuzüglich eines Terms, der nicht von \mathbf{w} abhängig ist und den Lernprozess nicht beeinflusst. Die bayessche MAP-Inferenz mit einer normalverteilten A-priori-Wahrscheinlichkeit der Gewichte entspricht somit dem Weight Decay.

Wie die vollständige bayessche Inferenz bietet auch die bayessche MAP-Inferenz den Vorteil, dass aus der A-priori-Wahrscheinlichkeit stammende, in den Trainingsdaten nicht enthaltene Informationen verfügbar gemacht werden. Diese zusätzlichen Informationen helfen dabei, die Varianz des MAP-Punktschätzwerts zu reduzieren (gegenüber dem ML-Schätzwert). Allerdings steigt dadurch die Verzerrung an.

Viele regularisierte Abschätzungsverfahren wie das Maximum Likelihood Learning mit Weight Decay können als MAP-Approximation für die bayessche Inferenz betrachtet werden. Diese Ansicht gilt, wenn die Regularisierung aus der Ergänzung eines zusätzlichen Terms zur Zielfunktion besteht, die $\log p(\boldsymbol{\theta})$ entspricht. Nicht alle Regularisierungsstrafterme entsprechen der bayesschen MAP-Inferenz. Zum Beispiel sind einige Regularisierungsterme möglicherweise nicht der Logarithmus einer Wahrscheinlichkeitsverteilung. Andere Regularisierungsterme sind abhängig von den Daten – ein Umstand, der für eine A-priori-Wahrscheinlichkeitsverteilung keinesfalls zulässig ist.

Die bayessche MAP-Inferenz bietet einen geradlinigen Weg zur Entwicklung komplexer und dennoch lesbarer Regularisierungsterme. Beispielsweise lässt sich ein komplexerer Strafterm aus einer Mischung von Normalverteilungen anstelle nur einer Normalverteilung als A-priori-Verteilung ableiten (*Nowlan und Hinton, 1992*).

5.7 Algorithmen für überwachtes Lernen

In Abschnitt 5.1.3 haben Sie gelernt, dass Algorithmen für überwachtes Lernen grob formuliert Lernalgorithmen sind, die lernen Eingaben mit

Ausgaben zu verknüpfen, wenn eine Trainingsdatenmenge gegeben ist mit Eingaben \mathbf{x} und Ausgaben \mathbf{y} . Häufig lassen sich die Ausgaben \mathbf{y} nur schwer automatisch erfassen und müssen daher von einem menschlichen Supervisor bereitgestellt werden. Der Begriff ist aber auch dann korrekt, wenn die Zielwerte der Trainingsdatenmenge automatisch erfasst wurden.

5.7.1 Probabilistisches überwachtes Lernen

Die meisten Algorithmen für überwachtes Lernen in diesem Buch basieren auf dem Schätzen einer Wahrscheinlichkeitsverteilung $p(y | \mathbf{x})$. Dazu ermitteln wir mithilfe der Maximum-Likelihood-Schätzung den besten Parametervektor $\boldsymbol{\theta}$ für eine parametrische Familie von Verteilungen $p(y | \mathbf{x}; \boldsymbol{\theta})$.

Sie haben bereits gesehen, dass die lineare Regression der folgenden Familie entspricht:

$$p(y | \mathbf{x}; \boldsymbol{\theta}) = \mathcal{N}(y; \boldsymbol{\theta}^\top \mathbf{x}, \mathbf{I}). \quad (5.80)$$

Wir können die lineare Regression für einen solchen Fall der Klassifizierung generalisieren, indem wir eine weitere Familie von Wahrscheinlichkeitsverteilungen definieren. Wenn wir über zwei Klassen – Klasse 0 und Klasse 1 – verfügen, müssen wir lediglich die Wahrscheinlichkeit für eine dieser Klassen angeben. Die Wahrscheinlichkeit der Klasse 1 bestimmt die Wahrscheinlichkeit der Klasse 0, denn die beiden Werte müssen zusammen 1 ergeben.

Die von uns für die lineare Regression verwendete Normalverteilung über reellwertige Zahlen ist durch einen Mittelwert parametrisiert. Jeder Wert, den wir für diesen Mittelwert vorgeben, ist gültig. Eine Verteilung über eine binäre Variable ist ein wenig komplexer, da ihr Mittelwert stets zwischen 0 und 1 liegen muss. Eine Möglichkeit zum Lösen dieses Problems bietet die logistische Sigmoidfunktion, mit der die Ausgabe der linearen Funktion in das Intervall (0,1) gezwungen und als Wahrscheinlichkeit interpretiert wird:

$$p(y = 1 | \mathbf{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \mathbf{x}). \quad (5.81)$$

Dieser Ansatz wird **logistische Regression** genannt (zugegeben eine seltsame Bezeichnung, da wir das Modell doch zur Klassifizierung und nicht zur Regression nutzen).

Im Falle der linearen Regression können wir die optimalen Gewichte durch Lösen der Normalgleichungen ermitteln. Die logistische Regression ist etwas schwieriger. Es gibt keine geschlossene Lösung (engl. *closed form solution*) für die optimalen Gewichte. Stattdessen müssen wir durch Maximieren der Log-Likelihood danach suchen. Dazu können wir die negative Log-Likelihood im Gradientenabstiegsverfahren minimieren.

Dasselbe Verfahren kann praktisch für jede Aufgabenstellung im überwachten Lernen genutzt werden, indem wir eine parametrische Familie bedingter Wahrscheinlichkeitsverteilungen für die passenden Eingangs- und Ausgangsvariablen notieren.

5.7.2 Support Vector Machines

Zu den einflussreichsten Ansätzen im überwachten Lernen gehört die Support Vector Machine (SVM, dt. *Stützvektormaschine*) (*Boser et al.*, 1992; *Cortes und Vapnik*, 1995). Dieses Modell ähnelt der logistischen Regression insofern, als eine lineare Funktion $\mathbf{w}^\top \mathbf{x} + b$ dahinter steht. Anders als die logistische Regression gibt die Support Vector Machine keine Wahrscheinlichkeiten aus, sondern lediglich eine Klassenidentität. Die SVM sagt vorher, dass die positive Klasse vorliegt, wenn $\mathbf{w}^\top \mathbf{x} + b$ positiv ist. Ebenso sagt sie voraus, dass die negative Klasse vorliegt, wenn $\mathbf{w}^\top \mathbf{x} + b$ negativ ist.

Eine wesentliche Innovation im Zusammenhang mit Support Vector Machines ist der **Kernel-Trick**. Er besteht in der Feststellung, dass viele Machine-Learning-Algorithmen gänzlich in Form von Skalarprodukten zwischen Beispielen notiert werden können. Es lässt sich zum Beispiel zeigen, dass die lineare Funktion der Support Vector Machine auch als

$$\mathbf{w}^\top \mathbf{x} + b = b + \sum_{i=1}^m \alpha_i \mathbf{x}^\top \mathbf{x}^{(i)} \quad (5.82)$$

geschrieben werden kann, wobei $\mathbf{x}^{(i)}$ ein Trainingsbeispiel ist und α ein Vektor der Koeffizienten. Wenn wir den Lernalgorithmus so neu schreiben, können wir für \mathbf{x} die Ausgabe einer gegebenen Merkmalsfunktion $\phi(\mathbf{x})$ und das Skalarprodukt mit einer Funktion $k(\mathbf{x}, \mathbf{x}^{(i)}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}^{(i)})$, **Kernel** genannt, einsetzen. Der Operator \cdot steht für ein Skalarprodukt (auch inneres Produkt genannt) analog zu $\phi(\mathbf{x})^\top \phi(\mathbf{x}^{(i)})$. Für einige Merkmalsräume nutzen wir nicht wirklich das innere Produkt des Vektors. In einigen unendlich-dimensionalen Räumen müssen wir andere Arten von inneren Produkten verwenden, zum Beispiel solche, die auf Integration und nicht auf Aufsummierung beruhen. Eine vollständige Entwicklung dieser Arten von inneren Produkten sprengt den Rahmen dieses Buchs.

Nachdem wir Skalarprodukte durch Kernel-Evaluationen ersetzt haben, können wir mit folgender Funktion Vorhersagen machen:

$$f(\mathbf{x}) = b + \sum_i \alpha_i k(\mathbf{x}, \mathbf{x}^{(i)}). \quad (5.83)$$

Diese Funktion ist bezüglich \mathbf{x} nichtlinear, aber die Beziehung zwischen $\phi(\mathbf{x})$ und $f(\mathbf{x})$ ist linear. Auch die Beziehung zwischen α und $f(\mathbf{x})$ ist linear. Die Funktion auf Kernel-Basis ist exakt äquivalent zur Vorverarbeitung der Daten durch Anwenden von $\phi(\mathbf{x})$ auf alle Eingangsdaten mit anschließendem Erlernen eines linearen Modells im neu transformierten Raum.

Der Kernel-Trick ist aus zwei Gründen so leistungsstark. Erstens ermöglicht er das Erlernen von Modellen, die als Funktion von \mathbf{x} nichtlinear sind, und zwar mithilfe konvexer Optimierungsverfahren, die garantiert effizient konvergieren. Das ist möglich, weil wir ϕ als unveränderlich betrachten und nur α optimieren, sodass der Optimierungsalgorithmus die Entscheidungsfunktion in einem anderen Raum als linear ansehen kann. Zweitens erlaubt die Kernel-Funktion k häufig eine Implementierung, die deutlich einfacher zu berechnen ist als die Konstruktion zweier Vektoren $\phi(\mathbf{x})$ mit anschließender Berechnung des Skalarprodukts.

In einigen Fällen kann $\phi(\mathbf{x})$ sogar unendlich-dimensional sein, was zu einem unendlich hohen Berechnungsaufwand für den naiven, expliziten Ansatz führen würde. In vielen Fällen ist $k(\mathbf{x}, \mathbf{x}')$ selbst dann eine nichtlineare effizient berechenbare Funktion von \mathbf{x} , wenn $\phi(\mathbf{x})$ nicht effizient berechenbar ist. Als Beispiel für einen unendlich-dimensionalen Merkmalsraum mit einem effizient berechenbaren Kernel konstruieren wir eine Merkmalszuordnung $\phi(x)$ über die nicht-negativen ganzen Zahlen x . Angenommen, diese Zuordnung gibt einen Vektor mit x Einsen gefolgt von unendlich vielen Nullen zurück. Wir können eine Kernel-Funktion $k(x, x^{(i)}) = \min(x, x^{(i)})$ schreiben, die exakt äquivalent zum entsprechenden unendlich-dimensionalen Skalarprodukt ist.

Der meistgenutzte Kernel ist der **gaußsche Kernel**

$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; 0, \sigma^2 \mathbf{I}), \quad (5.84)$$

wobei $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ die Standardnormalverteilung ist. Dieser Kernel wird auch als Kernel mit **radialen Basisfunktionen** oder RBF-Kernel bezeichnet (auch RBF-Kern, von engl. *radial basis function kernel*), da der Wert entlang der Linien von \mathbf{v} ausgehend von \mathbf{u} im Raum abnimmt. Der gaußsche Kernel entspricht einem Skalarprodukt in einem unendlich-dimensionalen Raum, aber die Ableitung des Raums ist weniger geradlinig als in unserem Beispiel des min-Kernels für die ganzen Zahlen.

Sie können sich den gaußschen Kernel als eine Art **Template Matching** vorstellen. Ein Trainingsbeispiel \mathbf{x} mit dem Trainings-Label y wird zum Template (Vorlage) für die Klasse y . Befindet sich ein Testpunkt \mathbf{x}' laut euklidischem Abstand nahe \mathbf{x} , zeigt der gaußsche Kernel eine starke Reaktion, die anzeigt, dass \mathbf{x}' dem Template \mathbf{x} sehr ähnlich ist. Das Modell

weist dem zugehörigen Trainings-Label y dann ein sehr hohes Gewicht zu. Insgesamt kombiniert die Vorhersage viele solcher nach Ähnlichkeit der zugehörigen Trainingsbeispiele gewichteten Trainings-Label.

Support Vector Machines sind nicht die einzigen Algorithmen, die sich mit dem Kernel-Trick verbessern lassen. Viele andere lineare Modelle sind dafür ebenfalls offen. Die Kategorie der Algorithmen, die den Kernel-Trick nutzen, wird als **Kernel-Maschinen** oder **Kernel-Methoden** bezeichnet (*Williams und Rasmussen, 1996; Schölkopf et al., 1999*).

Ein großer Nachteil von Kernel-Maschinen besteht darin, dass der Berechnungsaufwand für die Entscheidungsfunktion linear zur Anzahl der Trainingsbeispiele zunimmt, da das i -te Beispiel den Term $\alpha_i k(\mathbf{x}, \mathbf{x}^{(i)})$ zur Entscheidungsfunktion beiträgt. Support Vector Machines können das Problem abschwächen, indem ein Vektor $\boldsymbol{\alpha}$ erlernt wird, der größtenteils aus Nullen besteht. Für das Klassifizieren eines neuen Beispiels muss dann lediglich die Kernel-Funktion für die Trainingsbeispiele mit von Null verschiedenen α_i bestimmt werden. Diese Trainingsbeispiele werden als **Stützvektoren** (engl. *support vectors*) bezeichnet.

Kernel-Maschinen führen auch bei großen Datensätzen zu einem hohen Berechnungsaufwand im Training. Wir kommen in Abschnitt 5.9 darauf zurück. Kernel-Maschinen mit generischen Kernels haben Probleme mit der Fähigkeit zur Generalisierung. Warum das so ist, erfahren Sie in Abschnitt 5.11. Die moderne Ausprägung des Deep Learnings hatte das Ziel, diese Einschränkungen von Kernel-Maschinen zu überwinden. Der aktuelle Aufschwung des Deep Learnings begann, als *Hinton et al. (2006)* zeigte, dass ein neuronales Netz der RBF-Kernel-SVM beim MNIST-Benchmark überlegen war.

5.7.3 Andere einfache Algorithmen für überwachtes Lernen

Sie haben bereits einen anderen nichtprobabilistischen Algorithmus für überwachtes Lernen kennengelernt: die Nearest-Neighbor-Regression. Allgemein ausgedrückt bezeichnet k -Nearest-Neighbor eine Familie von Verfahren zur Klassifizierung oder Regression. Als nichtparametrischer Lernalgorithmus ist k -Nearest-Neighbor nicht auf eine feste Anzahl von Parametern eingeschränkt. Wir betrachten den k -Nearest-Neighbor-Algorithmus meist als parameterlosen Algorithmus, der eine einfache Funktion der Trainingsdaten implementiert. Tatsächlich gibt es noch nicht einmal eine Trainingsphase oder einen Lernprozess. Stattdessen suchen wir zum Testzeitpunkt (wenn wir eine Ausgabe y für eine neue Testeingabe \mathbf{x} erzeugen möchten) in

den Trainingsdaten \mathbf{X} die k -Nearest-Neighbors \mathbf{x} . Dann geben wir den Durchschnittswert der entsprechenden y Werte aus der Trainingsdatenmenge zurück. Das funktioniert für praktisch jede Art überwachtes Lernen, bei dem sich ein Durchschnittswert für y Werte bilden lässt. Im Fall der Klassifizierung können wir einen Durchschnittswert über One-hot-Code-Vektoren \mathbf{c} mit $c_y = 1$ und $c_i = 0$ für alle anderen Werte von i bilden. Anschließend können wir den Durchschnittswert über diese One-hot-Codes als Wahrscheinlichkeitsverteilung über Klassen auswerten. Als nichtparametrischer Lernalgorithmus kann k -Nearest-Neighbor eine sehr hohe Kapazität erreichen. Ein Beispiel: Angenommen, wir haben eine Mehrfachklassifizierungsaufgabe und messen die Leistung mit 0-1-Verlust. In diesem Fall konvergiert 1-Nearest-Neighbor gegen den doppelten Bayes-Fehler, wenn die Anzahl der Trainingsbeispiele gegen unendlich geht. Der über den Bayes-Fehler hinausgehende Fehler resultiert aus der Wahl eines einzelnen Nachbarn durch das zufällige Trennen der Verbindungen zwischen gleich weit voneinander entfernten Nachbarn. Bei unendlich vielen Trainingsdaten weisen sämtliche Testpunkte \mathbf{x} unendlich viele Nachbarpunkte in den Trainingsdaten auf, bei denen das Distanzmaß gegen Null geht. Wenn wir zulassen, dass all diese Nachbarn in die Entscheidung des Algorithmus einfließen (und nicht einfach einen davon zufällig auswählen), konvergiert das Verfahren gegen die Bayes-Fehlerquote. Die hohe Kapazität von k -Nearest-Neighbor ermöglicht eine hohe Korrektklassifikationsrate, sofern eine große Trainingsdatenmenge vorliegt. Dabei entsteht jedoch ein hoher Berechnungsaufwand. Außerdem leidet die Fähigkeit zur Generalisierung stark, wenn nur eine kleine, endliche Trainingsdatenmenge vorhanden ist. Eine Schwäche von k -Nearest-Neighbor ist die Unfähigkeit zu erlernen, dass ein Merkmal aussagekräftiger als ein anderes ist. Stellen wir uns eine Regressionsaufgabe mit $\mathbf{x} \in \mathbb{R}^{100}$ aus einer isotropen Normalverteilung und nur einer einzelnen, für die Ausgabe relevanten Variable x_1 vor. Gehen wir weiter davon aus, dass dieses Merkmal die Ausgabe lediglich direkt codiert, sodass in allen Fällen gilt $y = x_1$. Die Nearest-Neighbor-Regression kann dieses einfache Muster nicht erkennen. Der Nearest-Neighbor der meisten Punkte \mathbf{x} wird anhand der großen Anzahl der Merkmale x_2 bis x_{100} ermittelt, nicht anhand des einzelnen Merkmals x_1 . Die Ausgabe für kleine Trainingsdatenmengen ist daher gänzlich zufällig.

Ein weiterer Lernalgorithmus, der den Eingaberaum in Bereiche aufteilt und für jeden Bereich eigene Parameter aufweist, ist der **Entscheidungsbaum** (Breiman et al., 1984) oder eine seiner vielen Varianten. Abbildung 5.7 zeigt, dass jeder Knoten des Entscheidungsbaums zu einem Bereich im Eingaberaum gehört. Die Innenknoten teilen diesen Bereich in Teilbereiche für alle Kinder des Knotens auf (die normalerweise an den Achsen ausgerichtet

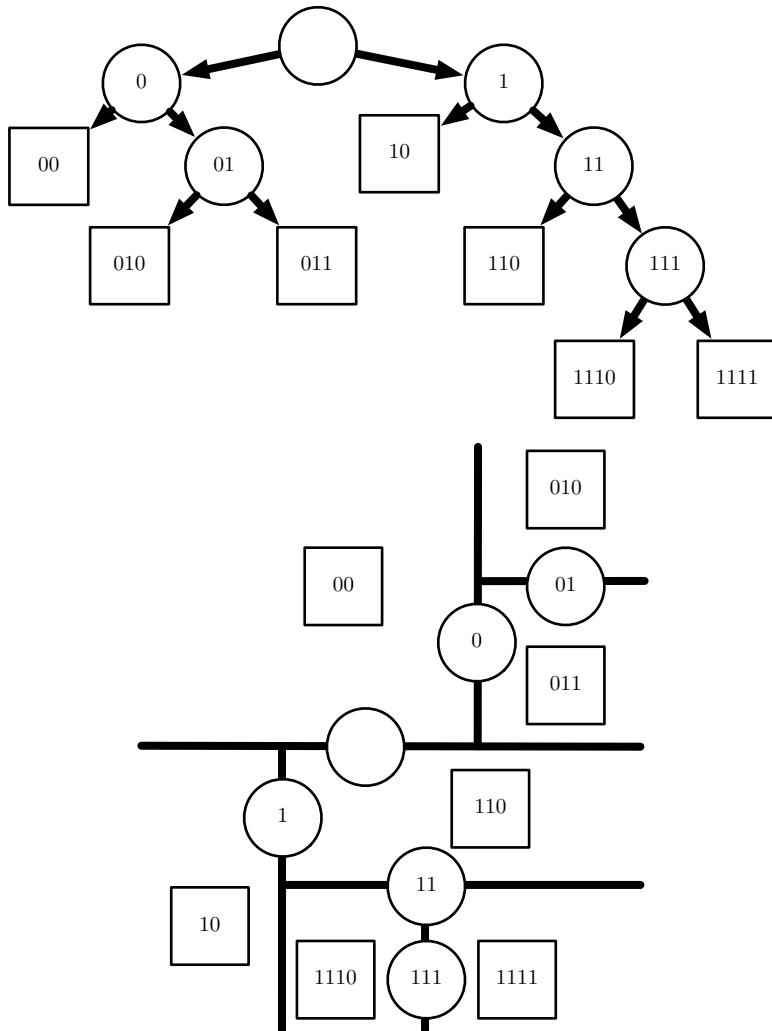


Abbildung 5.7: Diagramme zur Funktionsweise von Entscheidungsbäumen. (*Oben*) Jeder Knoten des Baums entscheidet sich dafür, das Eingabebeispiel entweder an den Kindknoten links (0) oder aber den Kindknoten rechts (1) weiterzugeben. Innenknoten sind als Kreise dargestellt, Blattknoten als Quadrate. Für jeden Knoten ist eine binäre Stringkennung angegeben, die seiner Position im Baum entspricht. Diese Kennung besteht aus der Kennung des Elternknotens ergänzt um das Auswahlbit (0 = links oder oben, 1 = rechts oder unten). (*Unten*) Der Baum unterteilt den Raum in Bereiche. Die 2-D-Darstellung zeigt, wie ein Entscheidungsbaum \mathbb{R}^2 aufteilen könnte. Die Knoten des Baums sind in dieser Ebene dargestellt. Jeder Innenknoten wird entlang der Trennlinie gezeichnet, die von diesem zur Kategorisierung von Beispielen verwendet wird. Die Blattknoten befinden sich in der Mitte der Bereiche der von ihnen erhaltenen Beispiele. Das Ergebnis ist eine stückweise konstante Funktion mit einem Stück pro Blatt. Für die Definition eines Blatts ist jeweils mindestens ein Trainingsbeispiel erforderlich. Der Entscheidungsbau kann also keine Funktion erlernen, die mehr lokale Maxima aufweist als Trainingsbeispiele.

werden). Der Raum wird auf diese Art in einander nicht überschneidende Bereiche unterteilt; zwischen Blattknoten und Eingabebereichen liegt eine Eins-zu-eins-Beziehung vor. Jeder Blattknoten ordnet üblicherweise jeden Punkt in seinem Eingabebereich derselben Ausgabe zu. Entscheidungsbäume werden normalerweise mit spezialisierten Algorithmen trainiert, die nicht in diesem Buch behandelt werden. Der Lernalgorithmus kann als nichtparametrisch betrachtet werden, wenn er einen Baum beliebiger Größe erlernen darf. Allerdings werden Entscheidungsbäume meist durch Größeneinschränkungen regularisiert, sodass es sich in der Praxis um parametrische Modelle handelt. In der gängigen Anwendungsform sind Entscheidungsbäume mit an den Achsen ausgerichteten Unterteilungen und konstanten Ausgaben innerhalb eines Knotens selbst für Aufgaben, die mittels logistischer Regression leicht zu lösen sind, denkbar ungeeignet. Ein Beispiel: Gegeben sei ein Zwei-Klassen-Problem. Die positive Klasse tritt auf, wenn $x_2 > x_1$ ist, dann ist die Entscheidungsgrenze nicht an einer Achse ausgerichtet. Der Entscheidungsbaum muss die Entscheidungsgrenze daher über viele Knoten approximieren. Hierzu kommt eine Treppenfunktion (engl. *step function*) zum Einsatz, die immer wieder in den Achsrichtungen über der tatsächlichen Entscheidungsfunktion hin- und herspringt.

Sie haben erfahren, dass Nearest-Neighbor-Näherungen und Entscheidungsbäume vielen Einschränkungen unterliegen. Nichtsdestotrotz handelt es sich dabei um nützliche Lernalgorithmen, wenn die Berechnungsressourcen eingeschränkt sind. Wir können außerdem ein Gespür für ausgefeilte Lernalgorithmen entwickeln, indem wir die Gemeinsamkeiten und Unterschiede zwischen ausgefeilten Algorithmen und k -Nearest-Neighbors oder Entscheidungsbäumen genauer untersuchen.

In Murphy (2012), Bishop (2006), Hastie et al. (2001) und anderen Lehrbüchern zum Thema Machine Learning finden Sie weitere Informationen zu klassischen Algorithmen für das überwachte Lernen.

5.8 Algorithmen für unüberwachtes Lernen

In Abschnitt 5.1.3 haben Sie erfahren, dass unüberwachte Algorithmen lediglich »Merkmale«, aber kein Überwachungssignal erfahren. Die Unterteilung in überwachte und unüberwachte Algorithmen ist nicht formal oder starr definiert, da es keine objektive Prüfung zur Einstufung eines Werts als Merkmal oder als durch einen Supervisor bestimmten Zielwert gibt. Informell bezeichnet unüberwachtes Lernen die meisten Versuche der Informationsgewinnung aus einer Verteilung, in der Menschen keine Bei-

spiele mit Bezeichnungen versehen müssen. Der Begriff wird meist mit der Dichteschätzung in Verbindung gebracht, dem Erlernen des Ziehens von Stichproben aus einer Verteilung, dem Denoising (Entrauschen) von Daten einer Verteilung, dem Suchen eines Vielfachen, in dessen Nähe die Daten liegen oder dem Clustering von Daten in Gruppen verwandter Beispiele.

Eine klassische Aufgabe für unüberwachtes Lernen ist die Suche nach der »besten« Repräsentation der Daten. »Beste« kann hier unterschiedliche Dinge bedeuten, doch allgemein ausgedrückt suchen wir eine Repräsentation, die möglichst viele Informationen über x enthält und dabei einen Strafterm oder eine Bedingung berücksichtigt, der oder die für eine *einfachere* oder zugänglichere Darstellung sorgt, als es x selbst ist.

Es gibt viele Möglichkeiten zum Definieren einer einfacheren Repräsentation. Zu den drei häufigsten gehören geringer-dimensionale Repräsentationen, dünnbesetzte Repräsentationen (engl. *sparse representations*) und unabhängige Repräsentationen. Geringdimensionale Repräsentationen versuchen, möglichst viele Informationen über x in einer kleineren Repräsentation darzustellen. Dünnbesetzte Repräsentationen (*Barlow*, 1989; *Olshausen und Field*, 1996; *Hinton und Ghahramani*, 1997) betten den Datensatz in eine Repräsentation ein, deren Einträge für die meisten Eingangsdaten hauptsächlich Nullen sind. Die Nutzung von dünnbesetzten Repräsentationen erfordert im Allgemeinen eine erhöhte Dimensionalität, damit die überwiegend zu Nullen werdende Repräsentation nicht zu viele Informationen verwirft. Das führt zu einer Gesamtstruktur der Repräsentation, die dazu tendiert, die Daten entlang der Achsen des Darstellungsraums zu verteilen. Unabhängige Repräsentationen versuchen, die Quellen von Variationen hinter der Datenverteilung so zu *separieren*, dass die Dimensionen der Repräsentation statistisch unabhängig sind.

Natürlich schließen diese drei Kriterien einander nicht aus. Geringdimensionale Repräsentationen weisen häufig Elemente auf, die weniger oder schwächere Abhängigkeiten haben als die ursprünglichen, hochdimensionalen Daten. Das liegt daran, dass zum Verringern der Größe einer Repräsentation Redundanzen gesucht und entfernt werden können. Durch das Bestimmen und Entfernen von mehr Redundanz kann der Algorithmus zur Dimensionsreduktion eine höhere Komprimierung erreichen und dabei weniger Informationen verwerfen.

Der Begriff der Repräsentation ist eines der zentralen Themen beim Deep Learning und somit auch eines der zentralen Themen dieses Buchs. In diesem Abschnitt entwickeln wir einige einfache Beispiele für Representation-Learning-Algorithmen. Gemeinsam zeigen diese Beispielalgorithmen,

wie die genannten drei Kriterien praktisch umgesetzt werden können. Die weiteren Kapitel stellen größtenteils zusätzliche Representation-Learning-Algorithmen vor, mit denen diese Kriterien auf unterschiedliche Weise weiterentwickelt oder neue Kriterien eingeführt werden.

5.8.1 Hauptkomponentenanalyse

In Abschnitt 2.12 haben Sie erfahren, dass der Algorithmus zur Hauptkomponentenanalyse (engl. *principal components analysis*, PCA) ein Mittel zur Datenkomprimierung darstellt. Wir können die PCA auch als Algorithmus für unüberwachtes Lernen betrachten, der eine Datenrepräsentation erlernt. Diese Repräsentation beruht auf zwei der oben genannten Kriterien für eine einfache Darstellung. Die PCA erlernt eine Repräsentation mit einer gegenüber den ursprünglichen Eingangsdaten geringeren Dimensionalität. Sie erlernt auch eine Repräsentation, deren Elemente keine lineare Korrelation untereinander aufweisen. Dies ist ein erster Schritt zum Lernkriterium einer Repräsentation, deren Elemente statistisch unabhängig sind. Für eine vollständige Unabhängigkeit muss der Representation-Learning-Algorithmus auch die nichtlinearen Beziehungen zwischen Variablen entfernen.

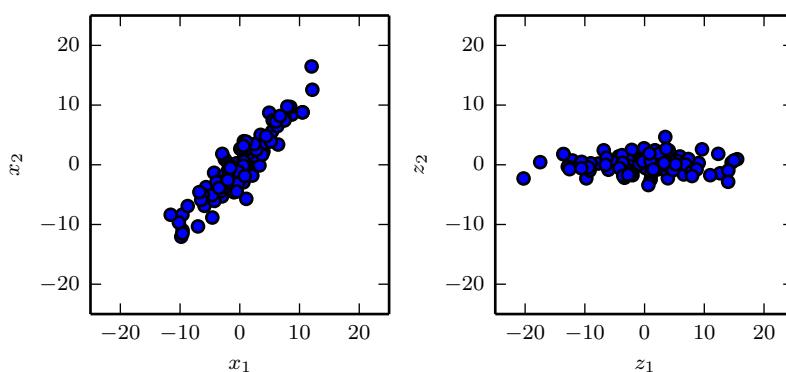


Abbildung 5.8: Die PCA erlernt eine lineare Projektion, die die Richtung der größten Varianz an den Achsen des neuen Raums ausrichtet. (*Links*) Die Originaldaten bestehen aus Stichproben aus \mathbf{x} . In diesem Raum tritt die Varianz möglicherweise nicht in einer Achsrichtung auf. (*Rechts*) Die transformierten Daten $\mathbf{z} = \mathbf{x}^\top \mathbf{W}$ zeigen nun die größte Variation entlang der Achse z_1 . Die Richtung der zweithäufigsten Varianz folgt z_2 .

Die PCA erlernt eine orthogonale, lineare Transformation der Daten, die einen Eingabewert \mathbf{x} auf eine Repräsentation \mathbf{z} abbildet (vgl. Abbildung 5.8). In Abschnitt 2.12 wurde gezeigt, dass wir eine eindimensionale Repräsentation erlernen können, die die ursprünglichen Daten (im Sinne

des mittleren quadratischen Fehlers) bestmöglich wiederherstellen kann, und dass diese Repräsentation tatsächlich der ersten Hauptkomponente der Daten entspricht. Somit können wir die PCA als einfaches und effektives Verfahren zur Dimensionsreduktion einsetzen, das möglichst viele Informationen der Daten erhält (wiederum genessen am Rekonstruktionsfehler nach der Methode der kleinsten Quadrate). Im Folgenden untersuchen wir, wie die PCA-Repräsentation die Repräsentation \mathbf{X} der Originaldaten dekorreliert.

Gegeben sei die $m \times n$ -Entwurfsmatrix \mathbf{X} . Wir gehen davon aus, dass die Daten einen Erwartungswert von Null aufweisen: $\mathbb{E}[\mathbf{x}] = \mathbf{0}$. Andernfalls können die Daten in einem vorbereitenden Schritt problemlos durch Subtrahieren des Erwartungswerts von allen Beispielen zentriert werden.

Die erwartungstreue Stichproben-Kovarianzmatrix für \mathbf{X} ergibt sich aus

$$\text{Var}[\mathbf{x}] = \frac{1}{m-1} \mathbf{X}^\top \mathbf{X}. \quad (5.85)$$

Die PCA findet eine Repräsentation (durch lineare Transformation) $\mathbf{z} = \mathbf{W}^\top \mathbf{x}$, wobei $\text{Var}[\mathbf{z}]$ diagonal ist.

In Abschnitt 2.12 wurde gezeigt, dass die Hauptkomponenten einer Entwurfsmatrix \mathbf{X} sich aus den Eigenvektoren von $\mathbf{X}^\top \mathbf{X}$ ergeben. Entsprechend gilt

$$\mathbf{X}^\top \mathbf{X} = \mathbf{W} \Lambda \mathbf{W}^\top. \quad (5.86)$$

In diesem Abschnitt beschäftigen wir uns mit einer alternativen Ableitung der Hauptkomponenten. Die Hauptkomponenten können auch mittels Singulärwertzerlegung (engl. *singular value decomposition*, SVD) bestimmt werden. Insbesondere handelt es sich um die rechten Singulärvektoren von \mathbf{X} . Nachweis: \mathbf{W} seien die rechten Singulärvektoren in der Zerlegung $\mathbf{X} = \mathbf{U} \Sigma \mathbf{W}^\top$. Wir stellen nun die ursprüngliche Eigenvektorgleichung mit \mathbf{W} als Basis aus Eigenvektoren wieder her:

$$\mathbf{X}^\top \mathbf{X} = (\mathbf{U} \Sigma \mathbf{W}^\top)^\top \mathbf{U} \Sigma \mathbf{W}^\top = \mathbf{W} \Sigma^2 \mathbf{W}^\top. \quad (5.87)$$

Die SVD ist hilfreich, um zu zeigen, dass die PCA eine Diagonale $\text{Var}[\mathbf{z}]$ ergibt. Mithilfe der SVD von \mathbf{X} können wir die Varianz von \mathbf{X} wie folgt ausdrücken:

$$\text{Var}[\mathbf{x}] = \frac{1}{m-1} \mathbf{X}^\top \mathbf{X} \quad (5.88)$$

$$= \frac{1}{m-1} (\mathbf{U} \Sigma \mathbf{W}^\top)^\top \mathbf{U} \Sigma \mathbf{W}^\top \quad (5.89)$$

$$= \frac{1}{m-1} \mathbf{W} \Sigma^\top \mathbf{U}^\top \mathbf{U} \Sigma \mathbf{W}^\top \quad (5.90)$$

$$= \frac{1}{m-1} \mathbf{W} \boldsymbol{\Sigma}^2 \mathbf{W}^\top, \quad (5.91)$$

wobei wir den Umstand nutzen, dass $\mathbf{U}^\top \mathbf{U} = \mathbf{I}$ ist, da die \mathbf{U} -Matrix der Singulärwertzerlegung als orthogonal definiert ist. Das beweist, dass die Kovarianz von \mathbf{z} wie gefordert diagonal ist:

$$\text{Var}[\mathbf{z}] = \frac{1}{m-1} \mathbf{Z}^\top \mathbf{Z} \quad (5.92)$$

$$= \frac{1}{m-1} \mathbf{W}^\top \mathbf{X}^\top \mathbf{X} \mathbf{W} \quad (5.93)$$

$$= \frac{1}{m-1} \mathbf{W}^\top \mathbf{W} \boldsymbol{\Sigma}^2 \mathbf{W}^\top \mathbf{W} \quad (5.94)$$

$$= \frac{1}{m-1} \boldsymbol{\Sigma}^2, \quad (5.95)$$

wobei wir dieses Mal den Umstand nutzen, dass $\mathbf{W}^\top \mathbf{W} = \mathbf{I}$ ist, wiederum gemäß der Definition der SVD.

Die obige Analyse zeigt, dass die resultierende Repräsentation bei der Projektion der Daten \mathbf{x} auf \mathbf{z} über die lineare Transformation \mathbf{W} eine diagonale Kovarianzmatrix aufweist (gemäß $\boldsymbol{\Sigma}^2$), die sofort impliziert, dass die einzelnen Elemente von \mathbf{z} paarweise unkorreliert sind.

Die Fähigkeit der PCA zur Transformation von Daten in eine Repräsentation, deren Elemente paarweise unkorreliert sind, stellt eine sehr wichtige Eigenschaft der PCA dar. Es ist ein einfaches Beispiel einer Repräsentation, die versucht, *die unbekannten Faktoren der Variation zu separieren*, die den Daten zugrunde liegen. Im Falle der PCA erfolgt diese Separation durch das Bestimmen einer Rotation des Eingaberaums (beschrieben durch \mathbf{W}), die dafür sorgt, dass die Hauptachsen der Varianz mit der Basis des neuen Darstellungsraums für \mathbf{z} zusammenfallen.

Obwohl die Korrelation eine wichtige Abhängigkeitskategorie zwischen den Datenelementen darstellt, interessieren wir uns aber auch für das Erlernen von Repräsentationen, die komplexere Abhängigkeiten zwischen Merkmalen separieren können. Zu diesem Zweck müssen wir einfache lineare Transformationen hinter uns lassen.

5.8.2 *k*-Means-Clustering

Ein weiteres Beispiel für einen einfachen Representation-Learning-Algorithmus ist *k*-Means-Clustering. Der *k*-Means-Clustering-Algorithmus teilt die Trainingsdatenmenge in k verschiedene Beispielcluster auf, die nahe

beieinander liegen. Der Algorithmus stellt quasi einen k -dimensionalen One-hot-Code-Vektor \mathbf{h} zur Darstellung einer Eingabe \mathbf{x} bereit. Wenn \mathbf{x} zum Cluster i gehört, ist $h_i = 1$ und alle anderen Einträge der Repräsentation \mathbf{h} sind Null.

Der One-hot-Code aus dem k -Means-Clustering ist ein Beispiel für eine dünnbesetzte Repräsentation, da die Mehrheit ihrer Einträge Nullen für jede Eingabe sind. In einem späteren Abschnitt entwickeln wir andere Algorithmen, die flexiblere dünnbesetzte Repräsentationen erlernen, in denen mehr als ein Eintrag für jede Eingabe \mathbf{x} von Null verschieden sein kann. One-hot-Codes sind ein extremes Beispiel für dünnbesetzte Repräsentationen, in denen viele Vorteile einer verteilten Repräsentation verloren gehen. Der One-hot-Code bietet nach wie vor einige statistische Vorteile (er vermittelt auf natürliche Weise die Vorstellung, dass alle Beispiele im selben Cluster einander ähnlich sind), und er bietet den rechnerischen Vorteil, dass die gesamte Repräsentation in nur einer ganzen Zahl erfasst werden kann.

Der k -Means-Algorithmus initialisiert zunächst k verschiedene Zentroide $\{\boldsymbol{\mu}^{(1)}, \dots, \boldsymbol{\mu}^{(k)}\}$ für unterschiedliche Werte und springt dann bis zum Erreichen der Konvergenz zwischen zwei Schritten hin und her. In einem Schritt wird jedes Trainingsbeispiel dem Cluster i zugewiesen, wobei i der Index des nächstgelegenen Zentroiden $\boldsymbol{\mu}^{(i)}$ ist. Im anderen Schritt wird jeder Zentroid $\boldsymbol{\mu}^{(i)}$ anhand des Mittelwerts aller Trainingsbeispiele $\mathbf{x}^{(j)}$, die dem Cluster i zugewiesen sind, angepasst.

Eine Schwierigkeit beim Clustering besteht darin, dass das Clustering-Problem an sich schlecht gestellt ist, da es kein einzelnes Kriterium gibt, das bestimmt, wie gut das Clustering der Daten der Realität entspricht. Wir können die Eigenschaften des Clusterings messen, zum Beispiel den mittleren euklidischen Abstand vom Cluster-Zentroiden zu den Elementen des Clusters. Das gibt uns die Möglichkeit zu bestimmen, wie gut wir die Trainingsdaten aus den Cluster-Zuweisungen wiederherstellen können. Wir wissen nicht, wie gut die Cluster-Zuweisungen den Eigenschaften in der Realität entsprechen. Es ist sogar möglich, dass viele verschiedene Clusterings einer Eigenschaft in der Realität gleich gut entsprechen. Während wir darauf hoffen, ein Clustering zu finden, das in Beziehung zu einem bestimmten Merkmal steht, finden wir vielleicht ein ebenso gültiges Clustering, das für unsere Aufgabe keinerlei Relevanz hat. Ein Beispiel: Wir nutzen zwei Clustering-Algorithmen für denselben Datensatz mit Bildern von roten Lkws, roten Pkws, grauen Lkws und grauen Pkws. Wenn wir jeden Clustering-Algorithmus nach zwei Clustern suchen lassen, findet der eine möglicherweise einen Cluster mit Pkws und einen zweiten Cluster mit Lkws, während der andere einen Cluster mit roten und einen zweiten Cluster mit grauen Fahrzeugen findet.

Wir könnten sogar einen dritten Clustering-Algorithmus auf das Problem ansetzen, der die Anzahl der Cluster frei bestimmen darf. Dieser teilt die Muster vielleicht in vier Cluster ein: rote Pkws, rote Lkws, graue Pkws und graue Lkws. Dieses neue Clustering hat zwar die Informationen über beide Attribute erfasst, aber dabei sind die Informationen zur Ähnlichkeit verloren gegangen. So, wie rote Pkws von den grauen Lkws getrennt sind, befinden sie sich auch in einem anderen Cluster als graue Pkws. Die Ausgabe des Clustering-Algorithmus gibt nicht an, dass rote Pkws den grauen Pkws ähnlicher sind als den grauen Lkws. Wir wissen nur, dass sie sich von den beiden anderen unterscheiden.

Diese Probleme verdeutlichen einige der Gründe, aus denen wir möglicherweise eine verteilte Repräsentation einer One-hot-Repräsentation vorziehen. Eine verteilte Repräsentation kann für jedes Fahrzeug zwei Attribute nutzen – eines für die Farbe und ein zweites für den Typ (Pkw oder Lkw). Es ist noch immer nicht offensichtlich, wie die optimale verteilte Repräsentation aussieht (wie kann der Lernalgorithmus wissen, ob wir stärker an der Unterteilung in Farbe und Typ als an der in Hersteller und Alter interessiert sind?), aber die Verfügbarkeit von vielen Attributen reduziert den »Druck« für den Algorithmus, das für uns wichtige Attribut zu erraten, und gibt uns die Möglichkeit, die Ähnlichkeit zwischen Objekten feinkörnig zu untersuchen, indem wir viele Attribute miteinander vergleichen, anstatt zu prüfen, ob ein Attribut zutrifft.

5.9 Stochastisches Gradientenabstiegsverfahren

Ein sehr wichtiger Algorithmus bildet das Fundament für den Großteil des Deep Learnings: das **stochastische Gradientenabstiegsverfahren** (engl. *stochastic gradient descent*, SGD), kurz stochastisches Gradientenverfahren. Das stochastische Gradientenabstiegsverfahren ist eine Erweiterung des Algorithmus für das Gradientenabstiegsverfahren, den wir in Abschnitt 4.3 vorgestellt haben.

Ein wiederkehrendes Problem im Machine Learning besteht darin, dass für eine gute Generalisierung große Trainingsdatenmengen erforderlich sind, die allerdings einen höheren Rechenaufwand bedeuten.

Die Kostenfunktion eines Machine-Learning-Algorithmus lässt sich häufig als Summe einer auf Trainingsbeispiele angewendeten Verlustfunktion

darstellen. Zum Beispiel können wir die negative Log-Likelihood der Trainingsdaten wie folgt notieren:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} L(\mathbf{x}, y, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}), \quad (5.96)$$

wobei L der Verlust $L(\mathbf{x}, y, \boldsymbol{\theta}) = -\log p(y | \mathbf{x}; \boldsymbol{\theta})$ pro Beispiel ist.

Für diese additive Kostenfunktionen müssen wir im Gradientenabstiegsverfahren folgenden Term berechnen:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \quad (5.97)$$

Der Berechnungsaufwand dieser Operation ist $O(m)$. Mit wachsender Größe der Trainingsdatenmenge hin zu Milliarden von Beispielen dauert jeder einzelne Gradientenschritt unverhältnismäßig lange.

Die Erkenntnis des SGDs ist, dass der Gradient ein Erwartungswert ist. Der Erwartungswert kann anhand einer kleinen Menge von Stichproben näherungsweise geschätzt werden. So können wir in jedem Schritt des Algorithmus eine kleine Teilmenge, **Mini-Batch** genannt, von Beispielen $\mathbb{B} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m')}\}$ heranziehen, die gleichförmig aus der Trainingsdatenmenge gezogen wird. Die Mini-Batch-Größe m' wird üblicherweise relativ klein gewählt, meist zwischen einem und wenigen Hundert Beispielen. Es ist entscheidend, m' auch bei wachsender Größe der Trainingsdatenmenge m unverändert zu lassen. Wir können eine Trainingsdatenmenge mit Milliarden von Beispielen auf der Grundlage von mit wenigen Hundert Beispielen berechneten Updates anpassen.

Der Schätzwert des Gradienten wird mit den Beispielen aus dem Mini-Batch \mathbb{B} gebildet als

$$\mathbf{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \quad (5.98)$$

Der Algorithmus für das stochastische Gradientenabstiegsverfahren folgt dann dem geschätzten Gradienten abwärts:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \mathbf{g}, \quad (5.99)$$

wobei ϵ die Lernrate ist.

Das Gradientenabstiegsverfahren wird häufig als langsam oder unzulässig betrachtet. Bisher zog man mit der Nutzung des Gradientenabstiegsverfahrens in nichtkonvexen Optimierungsproblemen eher Spott oder

zumindest ungläubige Blicke auf sich. Heute wissen wir, dass die in Teil II beschriebenen Machine-Learning-Modelle nach einem Training mit dem Gradientenabstiegsverfahren sehr gut funktionieren. Der Optimierungsalgorithmus erreicht vielleicht nicht in jedem Fall ein lokales Minimum in annehmbarer Zeit, aber er findet häufig einen sehr niedrigen Wert für die Kostenfunktion in so kurzer Zeit, dass er nützlich ist.

Das Gradientenabstiegsverfahren kennt außerhalb des Deep-Learning-Kontextes viele wichtige Anwendungsfälle. Es ist die wesentliche Methode zum Trainieren großer linearer Modelle anhand sehr großer Datensätze. Für Modelle mit unveränderlicher Größe ist der Aufwand für jedes SGD-Update unabhängig von der Größe m der Trainingsdatenmenge. In der Praxis nutzen wir mit zunehmender Größe der Trainingsdatenmenge häufig ein größeres Modell – aber das geschieht ohne Zwang. Die Anzahl der für das Erreichen von Konvergenz erforderlichen Updates nimmt normalerweise mit der Größe der Trainingsdatenmenge zu. Wenn m jedoch gegen unendlich geht, konvergiert das Modell schließlich gegen seinen bestmöglichen Testfehler, bevor das SGD jedes Beispiel aus der Trainingsdatenmenge untersucht hat. Eine weitere Erhöhung von m verlängert die zum Erreichen des bestmöglichen Testfehlers für das Modell benötigte Zeit nicht. So gesehen lässt sich argumentieren, dass die asymptotischen Kosten des Trainings eines Modells mittels SGD $O(1)$ als Funktion von m beträgt.

Vor dem Aufkommen von Deep Learning wurde für das Erlernen von nichtlinearen Modellen in erster Linie der Kernel-Trick in Verbindung mit einem linearen Modell verwendet. Viele Kernel-Lernalgorithmen setzen das Konstruieren einer $m \times m$ -Matrix $G_{i,j} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ voraus. Dabei entsteht ein Berechnungsaufwand in Höhe von $O(m^2)$, der natürlich bei Datensätzen mit Milliarden von Beispielen unerwünscht ist. In der Wissenschaft wurde Deep Learning ab 2006 interessant, da damit beim Trainieren von mittelgroßen Datensätzen mit Zehntausenden von Beispielen besser für neue Beispiele generalisiert werden konnte als mit anderen Algorithmen. Schon bald wuchs das Interesse an Deep Learning in der Wirtschaft, denn es bot eine skalierbare Möglichkeit zum Trainieren nichtlinearer Modelle anhand großer Datensätze.

Das stochastische Gradientenabstiegsverfahren und viele Verbesserungen daran werden in Kapitel 8 genauer beschrieben.

5.10 Entwickeln eines Machine-Learning-Algorithmus

Nahezu alle Deep-Learning-Algorithmen lassen sich als spezielle Instanzen eines recht einfachen Rezepts beschreiben: Vermenge eine Spezifikation eines Datensatzes, eine Kostenfunktion, ein Optimierungsverfahren und ein Modell miteinander.

Im Falle des Algorithmus zur linearen Regression werden beispielsweise ein Datensatz bestehend aus \mathbf{X} und \mathbf{y} , die Kostenfunktion

$$J(\mathbf{w}, b) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}), \quad (5.100)$$

die Modellspezifikation $p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; \mathbf{x}^\top \mathbf{w} + b, 1)$ und – in den meisten Fällen – der Optimierungsalgorithmus, der durch Beantwortung der Frage (mittels Normalgleichungen), ob der Gradient der Kosten Null beträgt, definiert wird, kombiniert.

Ist dieser Umstand einmal erkannt, können wir jede dieser Komponenten, die größtenteils von den anderen unabhängig ist, austauschen und so eine Vielzahl von Algorithmen entwickeln.

Die Kostenfunktion umfasst normalerweise mindestens einen Term, der eine statistische Schätzung im Lernprozess verursacht. Die gängigste Kostenfunktion ist die negative Log-Likelihood, bei der das Minimieren der Kostenfunktion zur Maximum-Likelihood-Schätzung führt.

Die Kostenfunktion kann auch weitere Terme enthalten, zum Beispiel Regularisierungsterme. Wir können zum Beispiel Weight Decay zur Kostenfunktion der linearen Regression hinzufügen, sodass wir

$$J(\mathbf{w}, b) = \lambda \|\mathbf{w}\|_2^2 - \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) \quad (5.101)$$

erhalten. Damit ist noch immer eine Optimierung der geschlossenen Form möglich.

Wenn wir das Modell nichtlinear auslegen, lassen sich die meisten Kostenfunktionen nicht mehr in geschlossener Form optimieren. Wir müssen also ein iteratives numerisches Optimierungsverfahren auswählen, zum Beispiel das Gradientenabstiegsverfahren.

Das Rezept für die Konstruktion eines Lernalgorithmus durch Kombinieren von Modellen, Kosten und Optimierungsalgorithmen ist für überwachtes und unüberwachtes Lernen geeignet. Das Beispiel der linearen Regression zeigt, wie überwachtes Lernen unterstützt wird. Für unüberwachtes Lernen muss ein Datensatz definiert werden, der nur \mathbf{X} enthält; außerdem müssen

wir angemessene Kosten und ein Modell für den unüberwachten Fall angeben. Wir können beispielsweise den ersten PCA-Vektor ermitteln, indem wir unsere Verlustfunktion als

$$J(\mathbf{w}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \|\mathbf{x} - r(\mathbf{x}; \mathbf{w})\|_2^2 \quad (5.102)$$

definieren und für das Modell \mathbf{w} mit Norm 1 und der Rekonstruktionsfunktion $r(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} \mathbf{w}$ festlegen.

In manchen Fällen ist die Kostenfunktion eine Funktion, die sich der tatsächlichen Berechnung aus Gründen des Aufwands verschließt. Wir können sie dann dennoch näherungsweise minimieren, indem wir auf eine iterative numerische Optimierung zurückgreifen – sofern wir in irgendeiner Art ihre Gradienten approximieren können.

Die meisten Machine-Learning-Algorithmen nutzen dieses Rezept, auch wenn es nicht immer offenkundig ist. Falls ein Machine-Learning-Algorithmus einzigartig oder von Hand gemacht erscheint, reicht im Allgemeinen der Einsatz eines Optimierers für Sonderfälle zu seinem Verständnis aus. Einige Modelle wie Entscheidungsbäume und k -Means benötigen Optimierer für Sonderfälle, da ihre Kostenfunktionen flache Bereiche aufweisen, die sich der Minimierung durch Optimierer auf Gradientenbasis entziehen. Wenn wir wissen, dass sich die meisten Machine-Learning-Algorithmen mit diesem Rezept beschreiben lassen, können wir unterschiedliche Algorithmen als Teil einer Taxonomie von Methoden zur Erledigung ähnlicher Aufgaben erkennen, die aus ähnlichen Gründen funktionieren; es handelt sich also nicht um eine lange Liste von Algorithmen, die jeweils separate Daseinsberechtigungen haben.

5.11 Probleme, an denen Deep Learning wächst

Die in diesem Kapitel beschriebenen, einfachen Machine-Learning-Algorithmen funktionieren für eine Vielzahl wichtiger Problemstellungen. Allerdings haben sie beim Lösen der zentralen KI-Probleme versagt, beispielsweise bei der Sprach- oder Objekterkennung.

Die Entwicklung des Deep Learnings wurde zum Teil durch die mangelnde Fähigkeit zur Generalisierung klassischer Algorithmen in Bezug auf KI-Aufgaben vorangetrieben.

In diesem Abschnitt geht es darum, wie die Herausforderung zur Generalisierung für neue Beispiele exponentiell schwieriger wird, wenn es um hochdimensionale Daten geht. Außerdem wird gezeigt, inwiefern die Mechanismen zum Erreichen der Generalisierung im klassischen Machine Learning

daran scheitern, komplizierte Funktionen in hochdimensionalen Räumen zu erlernen. Solche Räume verursachen häufig einen hohen Berechnungsaufwand. Deep Learning wurde entwickelt, um diese und andere Hindernisse zu überwinden.

5.11.1 Der Fluch der Dimensionalität

Viele Machine-Learning-Probleme werden extrem schwierig, wenn die Anzahl der Dimensionen in den Daten hoch ist. Dieses Phänomen wird als **Fluch der Dimensionalität** bezeichnet. Dabei wächst die Anzahl der möglichen eindeutigen Konfigurationen als Menge von Variablen mit zunehmender Anzahl der Variablen exponentiell an.

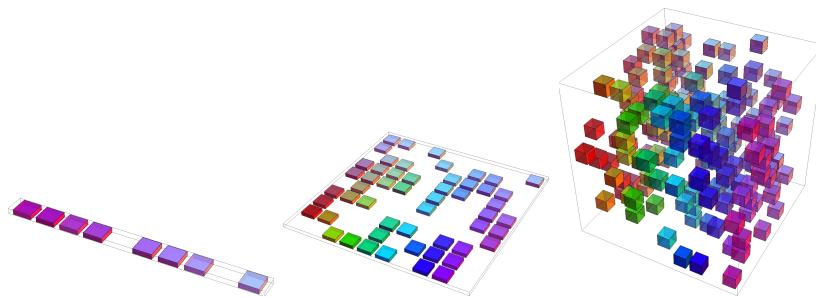


Abbildung 5.9: Mit zunehmender Anzahl der relevanten Dimensionen der Daten (von links nach rechts) wächst die Zahl der zu betrachtenden Konfigurationen exponentiell. (*Links*) In diesem eindimensionalen Beispiel gibt es eine Variable, die lediglich zwischen zehn zu betrachtenden Bereichen unterscheiden soll. Sofern genügend Beispiele für jeden dieser Bereiche vorliegen (jeder Bereich wird in der Abbildung als ein Kästchen dargestellt), haben Lernalgorithmen kein Problem mit einer korrekten Generalisierung. Ein geradliniger Weg zur Generalisierung besteht im Abschätzen des Werts für die Zielfunktion in jedem der Bereiche (und eventuell dem Interpolieren zwischen benachbarten Bereichen). (*Mitte*) In zwei Dimensionen gestaltet sich die Unterscheidung zehn verschiedener Werte für jede Variable schon schwieriger. Wir müssen jetzt bis zu $10 \times 10 = 100$ Bereiche im Blick behalten und benötigen mindestens ebenso viele Beispiele, um all diese Bereiche abzudecken. (*Rechts*) Bei drei Dimensionen ergeben sich $10^3 = 1\,000$ Bereiche und mindestens ebenso viele Beispiele. Für d Dimensionen und v zu unterscheidende Werte pro Achse müssten wir also $O(v^d)$ Bereiche und Beispiele benötigen. Das verdeutlicht den Fluch der Dimensionalität. Die Abbildung wurde freundlicherweise von Nicolas Chapados zur Verfügung gestellt.

Der Fluch der Dimensionalität tritt in vielen Bereichen der Informatik und ganz besonders im Machine Learning auf.

Eine Herausforderung, vor die der Fluch der Dimensionalität uns stellt, ist statistischer Natur. Wie Abbildung 5.9 zeigt, kommt es zu dieser statistischen Herausforderung, weil die Anzahl der möglichen Konfigurationen \mathbf{x} viel größer als die Anzahl der Trainingsbeispiele ist. Betrachten wir zum besseren Verständnis einen im Raster aufgebauten Eingaberaum (wie in der Abbildung). Wir können einen geringdimensionalen Raum mit einer kleinen Anzahl von Rasterzellen beschreiben, die größtenteils mit den Daten gefüllt sind.

Beim Generalisieren auf einen neuen Datenpunkt lässt sich meist durch eine einfache Untersuchung der Trainingsbeispiele der Zelle, in der auch die neue Eingabe liegt, ein Aktionsplan entwickeln. Wenn wir beispielsweise die Wahrscheinlichkeitsdichte in einem Punkt \mathbf{x} schätzen, können wir einfach die Anzahl der Trainingsbeispiele in derselben Einheitsvolumenzelle wie \mathbf{x} durch die Gesamtzahl der Trainingsbeispiele teilen und ausgeben. Zum Klassifizieren eines Beispiels können wir die häufigste Klasse der Trainingsbeispiele in der entsprechenden Zelle verwenden. Im Rahmen einer Regression können wir die für die Beispiele einer Zelle beobachteten Zielwerte mitteln. Aber was geschieht im Falle von Zellen, für die keine Beispiele vorliegen? Da in hochdimensionalen Räumen die Anzahl der Konfigurationen riesig ist – viel größer als die Anzahl der Beispiele –, gibt es für die meisten Rasterzellen keine Trainingsbeispiele. Wie können wir nun eine relevante Aussage über die neuen Konfigurationen machen? Viele klassische Machine-Learning-Algorithmen gehen einfach davon aus, dass die Ausgabe am neuen Punkt in etwa der Ausgabe des nächstgelegenen Trainingspunkts entsprechen sollte.

5.11.2 Lokale Konstanz und Regularisierung durch Glattheit

Für eine gute Fähigkeit zur Generalisierung müssen Machine-Learning-Algorithmen von vorausgehenden Annahmen geleitet werden, die vorgeben, welche Art von Funktion sie lernen sollen. Sie haben solche Annahmen als explizite Überzeugungen in Form von Wahrscheinlichkeitsverteilungen über die Modellparameter kennengelernt. Einfach ausgedrückt könnte man sagen, dass diese Annahmen die *Funktion* selbst direkt und die Parameter nur indirekt beeinflussen als Ergebnis der Beziehung zwischen den Parametern und der Funktion. Außerdem beschreiben wir einfach ausgedrückt diese Annahmen, indem wir implizit sagen, dass Algorithmen ausgewählt werden, die manche Klassen von Funktionen gegenüber anderen bevorzugen, auch wenn diese Tendenz nicht als Wahrscheinlichkeitsverteilung dargestellt wird

(oder werden kann), die unseren Grad der Überzeugung (engl. *degree of belief*) bezüglich diverser Funktionen angibt.

Zu den meistverwendeten dieser impliziten »Annahmen« gehört die **Glattheitsannahme**, auch **Annahme der lokalen Konstanz** genannt. Diese Annahme besagt, dass die zu erlernende Funktion in einem kleinen Bereich keine größeren Änderungen aufweist.

Viele einfache Algorithmen verlassen sich für eine gute Generalisierung ausschließlich auf diese Annahme, wodurch sie an den statistischen Herausforderungen, die zur Lösung von KI-Aufgaben notwendig sind, scheitern. Im Rahmen dieses Buchs beschreiben wir, wie Deep Learning zusätzliche (explizite und implizite) Annahmen einführt, um den Generalisierungsfehler aufwendiger Aufgaben zu reduzieren. Hier erläutern wir, warum die Glattheitsannahme allein diesen Aufgaben nicht gewachsen ist.

Es gibt viele Möglichkeiten, implizit oder explizit eine vorherige Überzeugung auszudrücken, dass die erlernte Funktion glatt oder lokal konstant sein sollte. Diese unterschiedlichen Methoden zielen darauf ab, den Lernprozess zum Erlernen einer Funktion f^* anzuregen, die die Bedingung

$$f^*(\mathbf{x}) \approx f^*(\mathbf{x} + \epsilon) \quad (5.103)$$

für die meisten Konfigurationen \mathbf{x} und kleine Änderungen von ϵ erfüllt. Mit anderen Worten: Wenn wir eine gute Antwort zu einer Eingabe \mathbf{x} kennen (falls \mathbf{x} zum Beispiel ein mit Label gekennzeichnetes Trainingsbeispiel ist), dann trifft diese Antwort vermutlich auch in der Umgebung von \mathbf{x} recht gut zu. Wenn wir mehrere gute Antworten in einer Umgebung kennen, können wir diese (durch eine Art Mittelwertbildung oder Interpolation) kombinieren, um eine Antwort zu finden, die mit so vielen davon wie möglich übereinstimmt.

Ein Extrembeispiel der lokalen Konstanz stellen die Lernalgorithmen der k -Nearest-Neighbor-Familie dar. Diese Näherungen sind buchstäblich in jedem Bereich konstant, der alle Punkte \mathbf{x} mit derselben Menge der k nächstgelegenen Nachbarn in der Trainingsdatenmenge enthält. Für $k = 1$ darf die Anzahl der unterscheidbaren Bereiche die Anzahl der Trainingsbeispiele nicht übersteigen.

Während der k -Nearest-Neighbor-Algorithmus die Ausgabe der nahegelegenen Trainingsbeispiele kopiert, interpolieren die meisten Kernel-Maschinen zwischen den Ausgaben der Trainingsdatenmenge für nahegelegene Trainingsbeispiele. Eine wichtige Kernel-Klasse ist die Familie der **lokalen Kernel**, in der $k(\mathbf{u}, \mathbf{v})$ groß ist, wenn $\mathbf{u} = \mathbf{v}$ ist, und abnimmt, wenn \mathbf{u} und \mathbf{v} sich weiter voneinander entfernen. Sie können sich einen lokalen Kernel

als Ähnlichkeitsfunktion vorstellen, der ein Template Matching durchführt, indem gemessen wird, wie stark ein Testbeispiel \boldsymbol{x} jedem Trainingsbeispiel $\boldsymbol{x}^{(i)}$ ähnelt. Ein Großteil der modernen Motivation hinter dem Deep Learning entsteht aus den Untersuchungen der Einschränkungen eines lokalen Template Matchings und der Art, wie tiefe Modelle dort erfolgreich sein können, wo das lokale Template Matching fehlschlägt (*Bengio et al.*, 2006b).

Entscheidungsbäume unterliegen ebenfalls den Beschränkungen des ausschließlichen Lernens auf Basis von Glattheit, da sie den Eingaberaum in so viele Bereiche unterteilen, wie es Blätter gibt, und für jeden Bereich einen separaten Parameter (oder für Erweiterungen der Entscheidungsbäume auch viele Parameter) verwenden. Wenn die Zielfunktion einen Baum mit mindestens n Blättern für eine exakte Darstellung benötigt, sind mindestens n Trainingsbeispiele zur Anpassung des Baums erforderlich. Ein Vielfaches von n wird benötigt, um ein gewisses Niveau statistischer Konfidenz in die vorhergesagte Ausgabe zu erreichen.

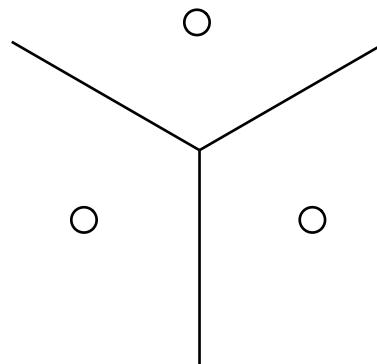


Abbildung 5.10: Darstellung der Unterteilung des Eingaberaums in Bereiche durch den Nearest-Neighbor-Algorithmus. Ein Beispiel (hier als Kreis dargestellt) in jedem Bereich definiert die Bereichsgrenze (hier in Form von Linien dargestellt). Der y -Wert eines jeden Beispiels definiert die Sollausgabe für alle Punkte im zugehörigen Bereich. Die durch das Nearest-Neighbor-Matching definierten Bereiche bilden ein geometrisches Muster, das als Voronoi-Diagramm bezeichnet wird. Die Anzahl der angrenzenden Bereiche kann schneller als die Anzahl der Trainingsbeispiele wachsen. Diese Abbildung stellt speziell das Verhalten des Nearest-Neighbor-Algorithmus dar. Andere Machine-Learning-Algorithmen, die sich zur Generalisierung ausschließlich auf die lokale Glattheitsannahme stützen, legen ein ähnliches Verhalten an den Tag: Jedes Trainingsbeispiel informiert den Klassifikator lediglich darüber, wie die Generalisierung in der direkten Umgebung erfolgt.

Grundsätzlich werden zur Unterscheidung von $O(k)$ Bereichen im Eingaberaum bei all diesen Verfahren $O(k)$ Beispiele benötigt. Normalerweise gibt es $O(k)$ Parameter, wobei $O(1)$ Parameter jedem der $O(k)$ Bereiche

zugeordnet sind. Das Nearest-Neighbor-Szenario, in dem jedes Trainingsbeispiel zum Definieren höchstens eines Bereichs verwendet werden kann, ist in Abbildung 5.10 dargestellt.

Gibt es eine Möglichkeit zur Repräsentation einer komplizierten Funktion, bei der viel mehr Bereiche unterschieden werden müssen, als Trainingsbeispiele vorliegen? Offensichtlich versetzt die Annahme der Glattheit der zugrunde liegenden Funktionen den Klassifikator nicht in diese Lage. Stellen Sie sich die Zielfunktion zum Beispiel als eine Art Schachbrett vor. Ein Schachbrett enthält viele Variationen, aber seine Struktur ist simpel. Was geschieht nun, wenn die Anzahl der Trainingsbeispiele deutlich kleiner als die Anzahl der schwarzen und weißen Felder auf dem Spielbrett ist? Wenn ausschließlich eine lokale Generalisierung und die Glattheitsannahme oder Annahme der lokalen Konstanz Anwendung finden, kann der Klassifikator garantiert die Farbe eines neuen Punktes richtig erraten, wenn dieser neuer Punkt im selben Feld wie ein Trainingsbeispiel liegt. Allerdings gilt diese Garantie nicht, wenn die Felder keine Trainingsbeispiele enthalten. Mit dieser Annahme allein gibt ein Beispiel nur an, welche Farbe das zugehörige Feld aufweist. Um nun die Farben aller Felder des Schachbretts zu ermitteln, muss auf jedem Feld mindestens ein Beispiel liegen.

Die Glattheitsannahme und die entsprechenden nichtparametrischen Lernalgorithmen arbeiten extrem gut, sofern der Lernalgorithmus über genügend Beispiele zur Beobachtung der Hochpunkte auf den meisten Scheiteln und Tiefpunkten in den meisten Tälern der tatsächlich zugrunde liegenden, zu erlernenden Funktion verfügt. Diese Aussage ist allgemein wahr, wenn die zu erlernende Funktion glatt genug ist und in ausreichend wenig Dimensionen variiert. In hohen Dimensionen kann selbst eine sehr glatte Funktion in jeder Dimension andere glatte Änderungen aufweisen. Verhält sich die Funktion dann noch in unterschiedlichen Bereichen anders, kann es extrem schwierig sein, sie mit einer Menge von Trainingsbeispielen zu beschreiben. Ist die Funktion komplex (und möchten wir eine – im Vergleich zur Anzahl der Beispiele – große Anzahl von Bereichen unterscheiden), scheint es kaum Hoffnung auf eine gute Fähigkeit zur Generalisierung zu geben.

Zum Glück lassen sich die beiden Probleme der effizienten Repräsentation einer komplizierten Funktion und der guten Generalisierungsfähigkeit der geschätzten Funktion für neue Eingaben lösen. Der Schlüssel liegt darin, dass eine sehr große Anzahl von Bereichen, beispielsweise $O(2^k)$, mit $O(k)$ Beispielen definiert werden kann, sofern wir einige Abhängigkeiten zwischen den Bereichen schaffen, indem wir zusätzliche Annahmen über die zugrunde liegende datengenerierende Verteilung treffen. Auf diese Weise können wir nichtlokal generalisieren (*Bengio und Monperrus, 2005; Bengio et al., 2006c*).

Viele unterschiedliche Deep-Learning-Algorithmen treffen implizite oder explizite Annahmen, die für eine Vielzahl von KI-Aufgaben angebracht sind, um sich diese Vorteile zunutze zu machen.

Andere Ansätze im Machine Learning treffen häufig stärkere Annahmen, die auf die jeweilige Aufgabe zugeschnitten sind. Wir können zum Beispiel die Schachbrettaufgabe problemlos lösen, indem wir annehmen, dass die Zielfunktion periodisch ist. Derart starke, aufgabenspezifische Annahmen sind in neuronalen Netzen nicht an der Tagesordnung, da sie die Fähigkeit zur Generalisierung für eine sehr viel größere Strukturvariation behindern. KI-Aufgaben weisen eine Struktur auf, die viel zu komplex ist, als dass man sie durch einfache, manuelle Vorgaben wie die zur Periodizität einschränken sollte. Daher suchen wir nach Lernalgorithmen, die sehr viel allgemeinere Annahmen nutzen. Beim Deep Learning gehen wir grundsätzlich davon aus, dass die Daten durch eine *Zusammensetzung von Faktoren* oder Merkmalen erzeugt wurden, eventuell gar in mehreren Hierarchiestufen.

Viele andere ähnlich allgemeine Annahmen können zur Verbesserung von Deep-Learning-Algorithmen beitragen. Diese scheinbar schwachen Annahmen erlauben einen exponentiellen Gewinn in der Beziehung zwischen der Anzahl der Beispiele und der Anzahl der unterscheidbaren Bereiche. Wir gehen in den Abschnitten 6.4.1, 15.4 und 15.5 näher auf diese exponentiellen Verbesserungen ein. Die exponentiellen Vorteile, die durch den Einsatz von tiefen verteilten Repräsentation entstehen, heben die exponentiellen Herausforderungen, die sich aus dem Fluch der Dimensionalität ergeben, auf.

5.11.3 Manifold Learning

Das Konzept der Mannigfaltigkeit ist für viele Bereiche des Machine Learnings von großer Bedeutung.

Eine **Mannigfaltigkeit** (engl. *manifold*) ist ein verbundener Bereich. Mathematisch handelt es sich um eine Menge von Punkten mit einer Umgebung, die jeden Punkt umgibt. Für jeden einzelnen dieser Punkte erscheint die Mannigfaltigkeit lokal als euklidischer Raum. Im Alltag erscheint uns die Erdoberfläche als 2-D-Ebene, obwohl es sich tatsächlich um eine sphärische Mannigfaltigkeit im 3-D-Raum handelt.

Das Konzept der Umgebung um jeden der Punkte deutet an, dass es Transformationen gibt, mit denen man sich in der Mannigfaltigkeit von einer Position zu einer benachbarten Position bewegen kann. Im Beispiel der

Erdoberfläche können wir uns in der Mannigfaltigkeit nach Norden, Süden, Osten oder Westen bewegen.

Entgegen der streng mathematischen Bedeutung des Begriffs »Mannigfaltigkeit« wird dieser im Machine Learning eher frei verwendet, um eine verbundene Menge von Punkten zu beschreiben, die durch Betrachtung einer kleinen Anzahl von Freiheitsgraden oder Dimensionen in einem höherdimensionalen Raum eine gute Approximation ermöglichen. Jede Dimension entspricht einer lokalen Richtung der Variation. Abbildung 5.11 zeigt ein Beispiel für Trainingsdaten, die in der Nähe einer eindimensionalen Mannigfaltigkeit liegen, die im zweidimensionalen Raum eingebettet ist. Im Machine-Learning-Kontext erlauben wir einen Wechsel der Dimensionalität der Mannigfaltigkeit zwischen einzelnen Punkten. Das geschieht häufig dann, wenn die Mannigfaltigkeit sich selbst schneidet. So ist die Ziffer Acht eine Mannigfaltigkeit, die an den meisten Stellen nur eine Dimension aufweist – lediglich im Schnittpunkt in der Mitte sind es zwei Dimensionen.

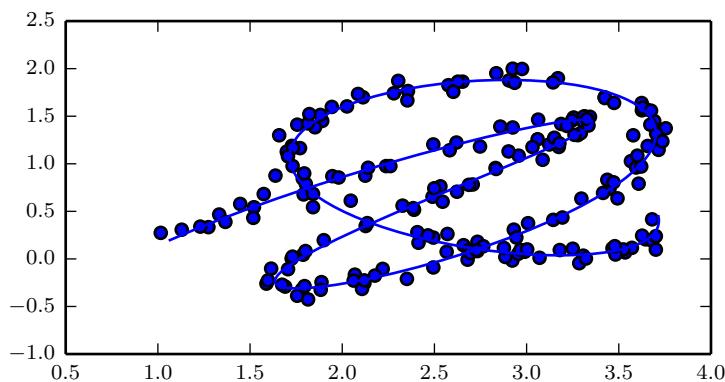


Abbildung 5.11: Datenauszug aus einer Verteilung in einem zweidimensionalen Raum, der sich tatsächlich wie eine verdrehte Schnur in der Nähe einer eindimensionalen Mannigfaltigkeit konzentriert. Die durchgezogene Linie markiert die zugrunde liegende Mannigfaltigkeit, auf die der Klassifikator schließen soll.

Viele Machine-Learning-Probleme erscheinen uns hoffnungslos, wenn wir erwarten, dass der Machine-Learning-Algorithmus Funktionen mit interessanten Variationen für die Gesamtmenge \mathbb{R}^n erlernen soll. **Manifold-Learning**-Algorithmen überwinden dieses Hindernis mithilfe der Annahme, dass \mathbb{R}^n größtenteils aus ungültigen Eingaben besteht und die interessanten Eingaben sich auf eine Reihe von Mannigfaltigkeiten konzentrieren, die nur eine kleine Teilmenge von Punkten enthalten, und dass die interessanten Variationen der Ausgabe der erlernten Funktion nur in den Richtungen, die in der Mannigfaltigkeit liegen, bzw. beim Wechsel zwischen Mannigfaltigkeiten stattfinden.

tigkeiten auftreten. Manifold Learning wurde für den Fall stetigwertiger Daten im Rahmen des unüberwachten Lernens eingeführt, auch wenn diese Vorstellung einer Wahrscheinlichkeitskonzentration sogar auf diskrete Daten und überwachtes Lernen übertragen werden kann: Die wesentliche Annahme bleibt, dass die Wahrscheinlichkeitsmasse stark konzentriert ist.

Allerdings ist die Annahme, dass die Daten sich entlang einer geringdimensionalen Mannigfaltigkeit anordnen, nicht immer richtig oder nützlich. Wir behaupten, dass im KI-Kontext, beispielsweise bei der Verarbeitung von Bildern, Tönen oder Texten, die Annahme der Mannigfaltigkeit zumindest näherungsweise richtig ist. Der Beweis zugunsten dieser Annahme besteht aus zwei Beobachtungskategorien.

Die erste Beobachtung zugunsten der **Mannigfaltigkeit-Hypothese** (auch Manifold-Hypothese, engl. *manifold hypothesis*) zeigt, dass die Wahrscheinlichkeitsverteilung für Bilder, Zeichenfolgen und Töne, die unter Realbedingungen vorkommen, stark konzentriert ist. Ein gleichförmiges Rauschen ähnelt in der Praxis niemals den strukturierten Eingaben aus diesen Definitionsbereichen. Abbildung 5.12 zeigt, wie gleichförmig entnommene Datenpunkte stattdessen dem Bildrauschen ähneln, das auf einem Fernseher ohne angeschlossene Antenne angezeigt wird. Ebenso stellt sich die Frage, wie hoch wohl die Wahrscheinlichkeit ist, einen sinnvollen deutschen Text zu erhalten, wenn man ein Dokument durch zufälliges und gleichförmiges Auswählen einzelner Buchstaben und Zeichen erzeugt. Die Antwort lautet offensichtlich: beinahe Null. Das liegt daran, dass die meisten längeren Sequenzen von Buchstaben keiner natürlichen Sprachfolge entsprechen. Die Verteilung der Sequenzen natürlicher Sprache nimmt nur einen geringen Teil des Gesamtraums der möglichen Sequenzen von Buchstaben ein.

Natürlich sind konzentrierte Wahrscheinlichkeitsverteilungen nicht ausreichend, um zu beweisen, dass die Daten in einer angemessen kleinen Anzahl von Mannigfaltigkeiten liegen. Wir müssen auch zeigen, dass die vorgefundenen Beispiele über andere Beispiele miteinander verbunden sind und dass jedes Beispiel von anderen, sehr ähnlichen Beispielen umgeben ist, die sich durch Transformationen innerhalb und zwischen den Mannigfaltigkeiten ergeben. Das zweite Argument, das für die Mannigfaltigkeit-Hypothese spricht, ist die Tatsache, dass wir uns derartige Umgebungen und Transformation zumindest theoretisch vorstellen können. Im Falle von Bildern können wir uns gewiss viele mögliche Transformationen vorstellen, mit denen sich eine Mannigfaltigkeit im Bildraum finden lässt: Wir können die Lichtquellen stärker oder weniger stark strahlen lassen, Objekte im Bild Stück für Stück drehen oder verschieben, die Farben von Objektoberflächen nach und nach anpassen und so weiter. Für die meisten Anwendungen ziehen

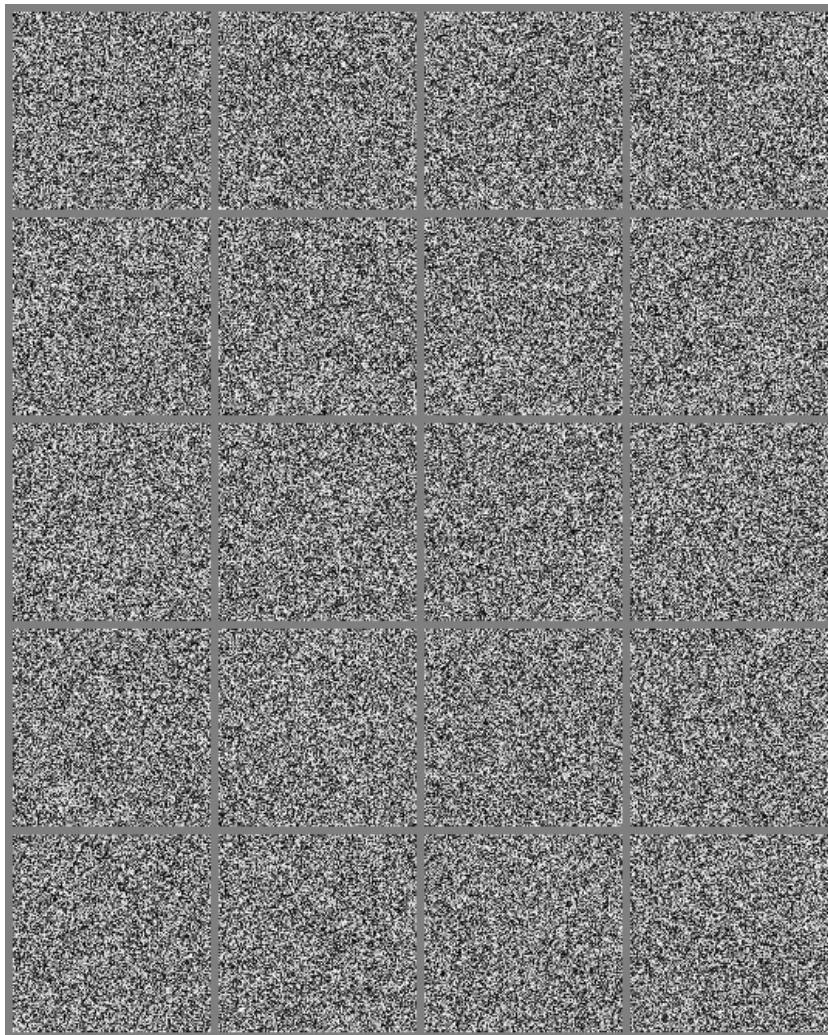


Abbildung 5.12: Ein gleichförmige und zufällige Stichprobengenerierung von Bildern (bei dem jedes Pixel zufällig anhand einer Gleichverteilung ausgewählt wird) fordert das Auftreten verrauschter Bilder. Obwohl eine von Null verschiedene Wahrscheinlichkeit vorliegt, ein Bild eines Gesichts oder eines anderen häufig in KI-Anwendungen anzutreffenden Objekts zu erzeugen, beobachten wir dies in der Praxis niemals. Das legt nahe, dass die Bilder im Rahmen von KI-Anwendungen einen vernachlässigbar geringen Anteil des gesamten Bildraums ausmachen.

wir dafür wahrscheinlich mehrere Mannigfaltigkeiten hinzufügen. Ein Beispiel: Die Mannigfaltigkeit der Bilder des menschlichen Gesichts ist nicht mit der Mannigfaltigkeit der Bilder von Katzengesichtern verbunden.

Diese Gedankenexperimente vermitteln einige intuitive Begründungen für die Mannigfaltigkeit-Hypothese. Rigorosere Experimente (*Cayton*, 2005; *Narayanan und Mitter*, 2010; *Schölkopf et al.*, 1998; *Roweis und Saul*, 2000; *Tenenbaum et al.*, 2000; *Brand*, 2003; *Belkin und Niyogi*, 2003; *Donoho und Grimes*, 2003; *Weinberger und Saul*, 2004) unterstützen die Hypothese ganz klar für große Klassen von Datensätzen auf dem Gebiet der KI.

Wenn die Daten in einer geringdimensionalen Mannigfaltigkeit liegen, ist es für Machine-Learning-Algorithmen ganz natürlich, sie in Form von Koordinaten der Mannigfaltigkeit darzustellen, und nicht als Koordinaten der Menge \mathbb{R}^n . Im Alltag sind Straßen ein Beispiel für eindimensionale Mannigfaltigkeiten, die in den dreidimensionalen Raum eingebettet sind. Wenn wir Anschriften mitteilen, dann verwenden wir dazu Hausnummern entlang dieser eindimensionalen Straßen, keine Koordinaten im 3-D-Raum. Das Extrahieren dieser Mannigfaltigkeitskoordinaten ist nicht leicht, aber bietet Verbesserungspotenzial für viele Machine-Learning-Algorithmen. Dieses allgemeine Prinzip findet in vielen Bereichen Anwendung. Abbildung 5.13 zeigt die Mannigfaltigkeitsstruktur eines Datensatzes mit Gesichtern. Am Ende des Buchs werden wir die Verfahren entwickelt haben, die zum Erlernen einer solchen Mannigfaltigkeitsstruktur erforderlich sind. In Abbildung 20.6 wird gezeigt, wie ein Machine-Learning-Algorithmus dieses Ziel erfolgreich erreichen kann.

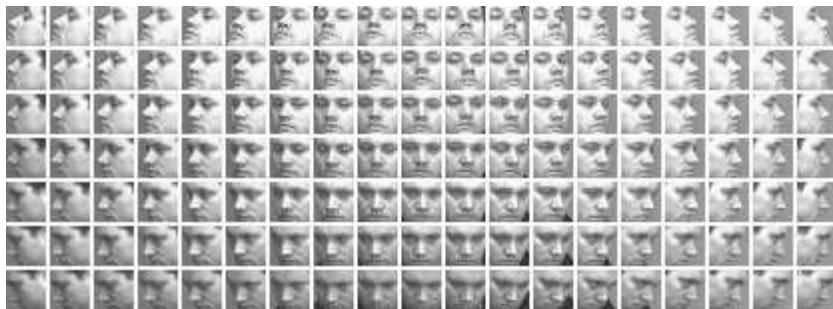


Abbildung 5.13: Trainingsbeispiele aus dem QMUL-Multiview-Face-Datensatz (*Gong et al.*, 2000). Die Testpersonen wurden gebeten, sich so zu bewegen, dass die zweidimensionale Mannigfaltigkeit für zwei Drehwinkel abgedeckt wird. Wir möchten, dass Lernalgorithmen solche Mannigfaltigkeitskoordinaten erkennen und separieren können. Abbildung 20.6 stellt diesen Prozess dar.

Damit haben Sie das Ende von Teil I erreicht. Sie haben die grundlegenden Konzepte aus der Mathematik und dem Machine Learning kennengelernt, auf denen die weiteren Teile des Buchs aufbauen. Sie sind nun bereit für das weitere Studium des Deep Learnings.

II

Tiefe Netze: Zeitgemäße Verfahren

Dieser Teil fasst den aktuellen Stand der praxisrelevanten Anwendungen auf dem Gebiet des Deep Learnings zusammen.

Die Geschichte des Deep Learnings ist lang und steckt voller Ambitionen. Es gibt noch einige Vorschläge, deren Erfolg sich noch nicht erwiesen hat. Und es gibt noch ambitionierte Ziele, die wir noch nicht erreicht haben. Diese weniger entwickelten Zweige des Deep Learnings behandeln wir im letzten Teil des Buchs.

In diesem Teil soll es dagegen um Ansätze gehen, die grundsätzlich funktionieren und in der Branche weithin genutzt werden.

Das moderne Deep Learning stellt ein leistungsstarkes Framework für überwachtes Lernen zur Verfügung. Durch Hinzufügen weiterer Schichten und zusätzlicher Einheiten in einer Schicht können tiefe Netze immer komplexere Funktionen abbilden. Die meisten Aufgaben, bei denen ein Eingabevektor einem Ausgabevektor zugeordnet wird und die sich von einem Menschen rasch und leicht erledigen lassen, können auch via Deep Learning erledigt werden. Die Voraussetzung dafür ist, dass ausreichend große Modelle und ausreichend umfassende Datensätze mit Trainingsbeispielen, die mit Labels gekennzeichnet sind, zur Verfügung stehen. Andere Aufgaben, die sich nicht als einfache Eins-zu-eins-Zuordnungen von Vektoren beschreiben lassen, sowie Aufgaben, über denen Menschen eine Weile brüten müssen, entziehen sich derzeit noch dem Deep Learning.

Dieser Teil des Buchs beschreibt die wesentliche Technologie zur parametrischen Funktionsapproximation, die hinter nahezu allen modernen praktischen Anwendungen im Deep Learning steckt. Wir beginnen mit der Beschreibung des tiefen Feedforward-Netz-Modells, das zur Repräsentation dieser Funktionen eingesetzt wird. Anschließend stellen wir fortgeschrittene Verfahren zur Regularisierung und Optimierung derartiger Modelle vor. Das Skalieren dieser Modelle für große umfangreiche Eingangsdaten wie hochauflösende Bilder oder lange zeitliche Abfolgen erfordert Spezialisierung. Wir stellen das CNN zum Skalieren für große Bilder sowie das RNN für die Verarbeitung zeitlicher Abfolgen vor. Schließlich stellen wir allgemeine Richtlinien für die praxisorientierte Methodologie auf, die beim Konzipieren, Erstellen und Konfigurieren einer Anwendung mit Deep-Learning-Elementen zum Einsatz kommt, und betrachten einige Einsatzbeispiele.

Diese Kapitel sind für die Praxis von großer Bedeutung. Wenn Sie also Deep-Learning-Algorithmen implementieren und verwenden möchten, um Probleme in der Praxis zu lösen, sollten Sie besonders aufmerksam weiterlesen.

6

Tiefe Feedforward-Netze

Tiefe Feedforward-Netze werden auch **neuronale Feedforward-Netze** oder **mehrschichtige Perzeptren** (MLPs) genannt. Sie sind die Deep-Learning-Modelle schlechthin. Das Ziel eines Feedforward-Netzes besteht darin, eine Funktion f^* zu approximieren. Zum Beispiel ordnet bei einem Klassifikator $y = f^*(\mathbf{x})$ eine Eingabe \mathbf{x} einer Kategorie y zu. Ein Feedforward-Netz definiert eine Zuordnung $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ und lernt den Wert der Parameter $\boldsymbol{\theta}$, die aus der besten Funktionsapproximation resultieren.

Das **Feedforward** in der Bezeichnung stammt daher, dass die Informationen durch die auszuwertende Funktion von \mathbf{x} ausgehend über die Zwischenrechnungen zur Bestimmung von f und schließlich zur Ausgabe \mathbf{y} fließen, also in Verarbeitungsrichtung (nach vorn) durchgereicht werden. Es gibt keine **Feedback**-Verbindungen (Rückkoppelungen), über die Ausgaben des Modells wieder in das Modell Eingang finden. Wenn neuronale Feedforward-Netze um Feedback-Verbindungen ergänzt werden, handelt es sich um in Kapitel 10 vorgestellte **RNNs**.

Feedforward-Netze sind in der praktischen Deep-Learning-Anwendung äußerst wichtig. Sie bilden die Grundlage vieler bedeutender kommerzieller Anwendungen. Zum Beispiel handelt es sich bei den CNNs zur Objekterkennung in Fotografien um eine spezielle Form von Feedforward-Netzen. Feedforward-Netze stellen eine wichtige Stufe auf dem Weg zu RNNs dar, die hinter vielen Anwendungen zur Verarbeitung natürlicher Sprache stecken.

Neuronale Feedforward-Netze werden als **Netze** bezeichnet, da sie typischerweise durch Zusammensetzen vieler verschiedener Funktionen dargestellt werden. Das Modell wird mit einem gerichteten azyklischen Graphen in Verbindung gebracht, der beschreibt, wie die Funktionen zusammengestellt werden. Wir können zum Beispiel die drei Funktionen $f^{(1)}$, $f^{(2)}$ und $f^{(3)}$

zur Kette $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$ verknüpfen. Diese Kettenstrukturen finden Sie in fast allen neuronalen Netzen. In diesem Fall ist $f^{(1)}$ die **erste Schicht** des Netzes, $f^{(2)}$ die **zweite Schicht** usw. Die Gesamtlänge der Kette gibt die **Tiefe** des Modells an. Die Bezeichnung »Deep Learning« entstammt direkt diesem Konzept der Tiefe. Die letzte Schicht eines Feed-forward-Netzes wird **Ausgabeschicht** genannt. Während des Trainings eines neuronalen Netzes steuern wir $f(\mathbf{x})$ auf $f^*(\mathbf{x})$. Die Trainingsdaten enthalten verrauchte approximative Beispiele von $f^*(\mathbf{x})$ für verschiedene Trainingspunkte. Jedes Beispiel \mathbf{x} ist mit einem Label $y \approx f^*(\mathbf{x})$ versehen. Die Trainingsbeispiele geben direkt an, was die Ausgabeschicht in jedem Punkt \mathbf{x} tun muss – sie muss einen Wert erzeugen, der nah an y liegt. Das Verhalten der anderen Schichten wird von den Trainingsdaten nicht vorgegeben. Der Lernalgorithmus muss selbstständig entscheiden, wie mithilfe dieser Schichten das gewünschte Ergebnis erzielt wird; die Trainingsdaten enthalten keine Angaben darüber, was in den einzelnen Schichten geschieht. Stattdessen muss der Lernalgorithmus entscheiden, wie mit diesen Schichten eine Approximation von f^* bestmöglich implementiert werden kann. Da die Trainingsdaten die gewünschte Ausgabe dieser Schichten nicht zeigen, werden sie auch als **verdeckte Schichten** bezeichnet.

Diese Art von Netzen heißt *neural*, weil sie entfernt von der Neurowissenschaft inspiriert sind. Jede verdeckte Schicht ist üblicherweise vektorwertig. Die Dimensionalität dieser verdeckten Schichten bestimmt die **Breite** des Modells. Man könnte sagen, dass jedes Element des Vektors die Rolle eines Neurons übernimmt. Anstatt die Schicht als einzelne Vektor-zu-Vektor-Funktion zu betrachten, können wir uns die Schicht als Ansammlung vieler **Einheiten** vorstellen, die parallel agieren; jede dieser Einheiten stellt eine Vektor-zu-Skalar-Funktion dar. Jede Einheit ähnelt insofern einem Neuron, als sie Eingaben von vielen anderen Einheiten erhält und ihren eigenen Aktivierungswert berechnet. Die Idee, viele Schichten vektorwertiger Repräsentationen einzusetzen, stammt aus der Neurowissenschaft. Die Auswahl der Funktionen $f^{(i)}(\mathbf{x})$ zum Berechnen dieser Repräsentationen beruht ebenfalls entfernt auf neurowissenschaftlichen Beobachtungen der Funktionen, die von biologischen Neuronen berechnet werden. Die moderne Forschung an neuronalen Netzen stützt sich allerdings auf viele mathematische und ingenieurwissenschaftliche Disziplinen. Auch haben neuronale Netze nicht das Ziel, das Gehirn perfekt zu modellieren. Sie können sich Feedforward-Netze am besten als Maschinen zur Funktionsapproximation vorstellen, die eine statistische Generalisierung erzielen sollen, wobei durchaus Ideen und Erkenntnisse aus dem Wissen über das Gehirn einfließen. Aber es handelt sich nicht um Modelle von Gehirnfunktionen.

Einen Startpunkt für ein Verständnis von Feedforward-Netzen stellt die Betrachtung von linearen Modellen und den Möglichkeiten zur Überwindung ihrer Einschränkungen dar. Lineare Modelle wie die logistische Regression und die lineare Regression sind reizvoll, weil sie effizient und zuverlässig sowohl in geschlossener Form als auch mit konvexer Optimierung angepasst werden können. Ein offenkundiger Nachteil von linearen Modellen ist die auf lineare Funktionen eingeschränkte Modellkapazität, die verhindert, dass das Modell die Interaktion zwischen zwei beliebigen Eingangsvariablen verstehen kann.

Um lineare Modelle auf die Repräsentation für nichtlineare Funktionen von \mathbf{x} auszudehnen, müssen wir das lineare Modell nicht auf \mathbf{x} selbst, sondern auf transformierte Eingangsdaten $\phi(\mathbf{x})$ anwenden, wobei ϕ eine nichtlineare Transformation ist. Ebenso können wir den in Abschnitt 5.7.2 vorgestellten Kernel-Trick nutzen, um einen nichtlinearen Lernalgorithmus anhand der impliziten ϕ -Zuordnung zu erhalten. Sie können sich vorstellen, dass ϕ eine Menge von Merkmalen zur Beschreibung von \mathbf{x} bereitstellt – oder eine neue Repräsentation von \mathbf{x} .

Damit stellt sich die Frage, wie die Zuordnung ϕ ausgewählt werden sollte.

1. Eine Möglichkeit besteht in der Verwendung eines sehr allgemeinen ϕ , zum Beispiel dem unendlich-dimensionalen ϕ , das implizit von Kernel-Maschinen auf Basis des RBF-Kernels eingesetzt wird. Wenn die Dimension von $\phi(\mathbf{x})$ hoch genug ist, können wir stets auf eine ausreichend große Kapazität zur Anpassung der Trainingsdatenmenge zugreifen – allerdings bleibt die Fähigkeit zur Generalisierung der Testdatenmenge oft schwach. Sehr generische Merkmalszuordnungen arbeiten meist nur mit dem Prinzip der lokalen Glattheit und codieren nicht genug vorherige Informationen zum Lösen fortgeschrittener Probleme.
2. Eine weitere Möglichkeit besteht darin, ϕ manuell zu konstruieren. Das war tatsächlich bis zum Aufkommen des Deep Learnings der vorherrschende Ansatz. Er erfordert Jahrzehnte menschlicher Bemühungen für jede einzelne Aufgabe. Die Fachleute müssen sich auf die jeweiligen Gebiete spezialisieren, zum Beispiel die Spracherkennung oder die Computer Vision, ohne dass es bereichsübergreifende Vorteile oder Gemeinsamkeiten gäbe.
3. Beim Deep Learning wird die Strategie verfolgt, ϕ zu lernen. Dazu nutzen wir ein Modell $y = f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{w}) = \phi(\mathbf{x}; \boldsymbol{\theta})^\top \mathbf{w}$. Wir verfügen

nun über die Parameter θ zum Erlernen von ϕ anhand einer breiten Klasse von Funktionen und die Parameter w zum Zuordnen von $\phi(x)$ zur gewünschten Ausgabe. Das ist ein Beispiel für ein tiefes Feedforward-Netz, in dem ϕ eine verdeckte Schicht definiert. Dieser Ansatz ist die einzige der drei Möglichkeiten, die die Konvexität des Trainingsproblems ignoriert – aber seine Vorteile überwiegen die Nachteile. Dabei parametrisieren wir die Repräsentation als $\phi(x; \theta)$ und verwenden den Optimierungsalgorithmus, um das θ zu finden, das einer guten Repräsentation entspricht. Wenn wir möchten, kann dieser Ansatz auch den Vorteil der hohen generischen Nutzbarkeit der ersten Möglichkeit einbeziehen. Dazu verwenden wir einfach eine sehr breit aufgestellte Familie $\phi(x; \theta)$. Deep Learning kann sich auch den Vorteil der zweiten Möglichkeit zunutze machen. Menschliche Fachleute können ihr Wissen durch den Entwurf von Familien $\phi(x; \theta)$, denen sie eine gute Funktion zutrauen, codieren, um die Generalisierung zu unterstützen. Der Vorteil dabei ist, dass der Mensch nur die passende allgemeine Funktionsfamilie finden muss und keine Zeit mit der Suche nach der exakten Funktion selbst verschwendet.

Dieses allgemeine Prinzip zur Modellverbesserung durch Erlernen von Merkmalen geht über die in diesem Kapitel beschriebenen Feedforward-Netze hinaus. Wir werden es bei allen Arten von Modellen, die das Buch behandelt, wieder aufgreifen. Feedforward-Netze nutzen das Prinzip zum Erlernen deterministischer Zuordnung zwischen x und y , bei denen keine Feedback-Verbindungen vorliegen. Andere Modelle, die Sie noch kennenlernen, wenden diese Prinzipien zum Erlernen von stochastischen Zuordnungen, Funktionen mit Feedback und Wahrscheinlichkeitsverteilungen über einen Einzelvektor an.

Als Einstieg in dieses Kapitel dient ein einfaches Beispiel für ein Feedforward-Netz. Danach behandeln wir jede der Designentscheidungen bei der Umsetzung eines solchen Netzes. Zunächst erfordert das Trainieren eines Feedforward-Netzes viele Designentscheidungen, die mit einem linearen Modell identisch sind: das Auswählen des Optimierers, die Kostenfunktion und die Art der Ausgabeeinheiten. Wir fassen diese Grundlagen des Lernens auf Gradientenbasis kurz zusammen und fahren dann mit einigen der Designentscheidungen fort, die nur bei Feedforward-Netzen notwendig sind. Mit den Feedforward-Netzen haben Sie das Konzept einer verdeckten Schicht kennengelernt. Diese erfordert die Wahl der **Aktivierungsfunktionen**, mit denen die Werte der verdeckten Schicht berechnet werden. Wir müssen uns auch um die Netzarchitektur kümmern, darunter die Anzahl der Schichten, die Verbindung zwischen den einzelnen Schichten und der Anzahl der Einheiten

in jeder Schicht. Für das Lernen in tiefen neuronalen Netzen müssen die Gradienten komplizierter Funktionen berechnet werden. Wir führen den **Backpropagation**-Algorithmus und seine modernen Generalisierungen ein, die eine effiziente Berechnung dieser Gradienten ermöglichen. Abschließend befassen wir uns kurz mit einigen historischen Aspekten.

6.1 Beispiel: Erlernen von XOR

Zum besseren Verständnis entwickeln wir im Folgenden ein komplettes Feedforward-Netz für eine sehr einfache Aufgabe, nämlich das Erlernen der XOR-Funktion.

Die XOR-Funktion (»exklusiv ODER«) ist eine Operation mit zwei Binärwerten x_1 und x_2 . Ist genau einer dieser Binärwerte gleich 1, gibt die XOR-Funktion 1 zurück. Andernfalls gibt sie 0 zurück. Die XOR-Funktion stellt die zu erlernende Zielfunktion $y = f^*(\mathbf{x})$ bereit. Unser Modell stellt eine Funktion $y = f(\mathbf{x}; \boldsymbol{\theta})$ bereit, unser Lernalgorithmus passt die Parameter $\boldsymbol{\theta}$ an, damit f und f^* einander möglichst ähnlich werden.

In diesem einfachen Beispiel kümmern wir uns nicht um die statistische Generalisierung. Wir möchten, dass unser Netz für die vier Punkte $\mathbb{X} = \{[0, 0]^\top, [0, 1]^\top, [1, 0]^\top \text{ und } [1, 1]^\top\}$ korrekt arbeitet. Wir trainieren das Netz für die Gesamtheit dieser vier Punkte. Die einzige Herausforderung ist das Anpassen der Trainingsdatenmenge.

Wir können sie als Regressionsproblem betrachten und eine Verlustfunktion mit mittlerem quadratischen Fehler einsetzen. Wir haben diese Verlustfunktion gewählt, um die Berechnungen für dieses Beispiel so übersichtlich wie möglich zu halten. Im praktischen Alltag ist der MQF normalerweise keine angemessene Kostenfunktion für das Modellieren binärer Daten. Passendere Ansätze werden in Abschnitt 6.2.2.2 vorgestellt.

Für die gesamte Trainingsdatenmenge berechnet, ergibt sich folgende MQF-Verlustfunktion:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\mathbf{x} \in \mathbb{X}} (f^*(\mathbf{x}) - f(\mathbf{x}; \boldsymbol{\theta}))^2. \quad (6.1)$$

Nun müssen wir die Form unseres Modells, $f(\mathbf{x}; \boldsymbol{\theta})$, auswählen. Angenommen, wir entscheiden uns für ein lineares Modell, bei dem $\boldsymbol{\theta}$ aus \mathbf{w} und b besteht. Unser Modell ist definiert als

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^\top \mathbf{w} + b. \quad (6.2)$$

Wir können $J(\theta)$ in geschlossener Form anhand der Normalgleichungen relativ zu w und b minimieren.

Nach dem Lösen der Normalgleichungen erhalten wir $w = \mathbf{0}$ und $b = \frac{1}{2}$. Das lineare Modell gibt einfach überall 0,5 aus. Warum geschieht das? Abbildung 6.1 zeigt, wieso ein lineares Modell die XOR-Funktion nicht repräsentieren kann. Eine Möglichkeit zur Lösung dieses Problems besteht darin, ein Modell zu verwenden, das einen unterschiedlichen Merkmalsraum erlernt, in dem ein lineares Modell die Lösung darstellen kann.

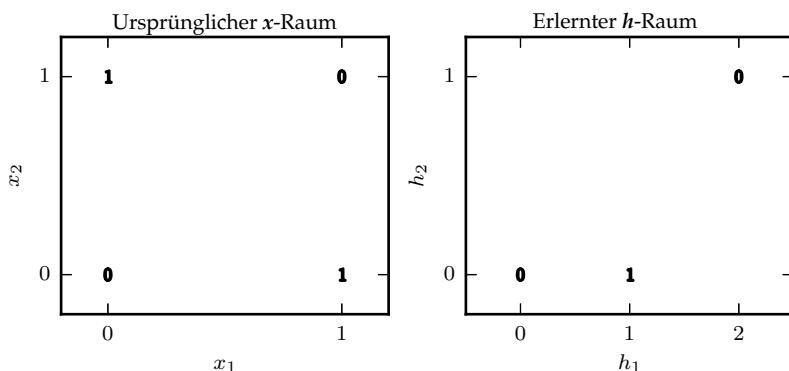


Abbildung 6.1: Lösen des XOR-Problems durch Erlernen einer Repräsentation. Die **fett** gedruckten Zahlen geben den Wert an, den die erlernte Funktion für jeden der Punkte ausgeben muss. (*Links*) Ein direkt auf die Originaldaten angesetztes lineares Modell kann die XOR-Funktion nicht implementieren. Für $x_1 = 0$ muss die Ausgabe des Modells zunehmen, da x_2 zunimmt. Für $x_1 = 1$ muss die Ausgabe des Modells abnehmen, da x_2 abnimmt. Ein lineares Modell muss einen unveränderlichen Koeffizienten w_2 auf x_2 anwenden. Es kann daher den Wert x_1 nicht verwenden, um den Koeffizienten für x_2 zu ändern und scheitert somit an der Lösung. (*Rechts*) Im transformierten Raum, der durch die mittels eines neuronalen Netzes extrahierten Merkmale dargestellt wird, kann ein lineares Modell das Problem lösen. In unserem Beispiel wurden die beiden Punkte, deren Ausgabewert 1 betragen muss, zu einem Punkt im Merkmalsraum zusammengefasst. Anders ausgedrückt: Die nichtlinearen Merkmale haben sowohl $\mathbf{x} = [1, 0]^\top$ als auch $\mathbf{x} = [0, 1]^\top$ im Merkmalsraum zu einem einzelnen Punkt zusammengeführt, $\mathbf{h} = [1, 0]^\top$. Das lineare Modell kann nun für die Funktion angeben, dass diese in h_1 zunimmt und in h_2 abnimmt. In diesem Beispiel führt das Erlernen des Merkmalsraums lediglich zu einer höheren Modellkapazität, die eine Anpassung der Trainingsdatenmenge ermöglicht. In realistischeren Anwendungen können erlernte Repräsentationen auch zur Fähigkeit zur Generalisierung des Modells beitragen.

Insbesondere zeigen wir ein einfaches Feedforward-Netz mit einer verdeckten Schicht, die zwei verdeckte Einheiten enthält. Abbildung 6.2 enthält eine Darstellung des Modells. Dieses Feedforward-Netz weist einen Vektor

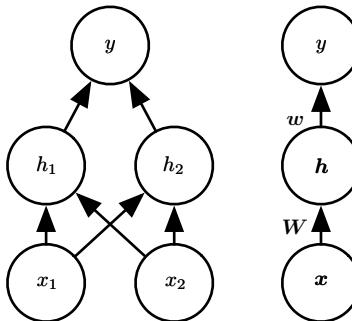


Abbildung 6.2: Zwei Darstellungen eines Feedforward-Netzes. Es handelt sich um das Netz, mit dem wir das XOR-Beispiel gelöst haben. Es verfügt über eine einzelne verdeckte Schicht mit zwei Einheiten. (*Links*) In dieser Darstellung ist jede Einheit als Knoten des Graphen zu sehen. Sie ist explizit und unmissverständlich, nimmt aber bei größeren Netzen schnell zu viel Platz ein. (*Rechts*) In dieser Darstellung steht ein Knoten im Graphen für den gesamten Vektor aller Aktivierungen einer Schicht. Sie ist deutlich kompakter. Manchmal beschriften wir die Kanten im Graphen mit den Parameternamen, die die Beziehung zwischen zwei Schichten beschreiben. Hier geben wir an, dass eine Matrix \mathbf{W} die Zuordnung von \mathbf{x} zu \mathbf{h} beschreibt und ein Vektor \mathbf{w} die Zuordnung von \mathbf{h} zu y . Meist lassen wir die Parameter für die einzelnen Schichten bei dieser Art der Darstellung weg.

mit verdeckten Einheiten \mathbf{h} auf, die durch eine Funktion $f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$ berechnet werden. Die Werte dieser verdeckten Einheiten werden dann als Eingabe für eine zweite Schicht genutzt. Die zweite Schicht ist die Ausgabeschicht des Netzes. Bei der Ausgabeschicht handelt es sich noch immer um ein lineares Regressionsmodell, das nun aber auf \mathbf{h} anstelle von \mathbf{x} angewandt wird. Das Netz enthält daher die beiden verketteten Funktionen $\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$ und $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$ und das vollständige Modell lautet $f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$.

Welche Funktion soll $f^{(1)}$ berechnen? Lineare Modelle waren uns bisher von großem Nutzen. Darum liegt es nahe, auch $f^{(1)}$ linear zu gestalten. Doch leider wäre bei einer linearen $f^{(1)}$ das gesamte Feedforward-Netz eine lineare Funktion seiner Eingangsdaten. Lassen wir den Achsenschnittpunktsterm für einen Augenblick beiseite und nehmen an, es sei $f^{(1)}(\mathbf{x}) = \mathbf{W}^\top \mathbf{x}$ und $f^{(2)}(\mathbf{h}) = \mathbf{h}^\top \mathbf{w}$. Dann gilt $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{W}^\top \mathbf{x}$. Wir könnten diese Funktion als $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}'$ schreiben, mit $\mathbf{w}' = \mathbf{W}\mathbf{w}$.

Offensichtlich müssen wir eine nichtlineare Funktion zum Beschreiben der Merkmale verwenden. Die meisten neuronalen Netze nutzen dazu eine affine Transformation, die mithilfe erlernter Parameter gesteuert wird, gefolgt von einer unveränderlichen nichtlinearen Funktion, die als Aktivierungsfunktion

bezeichnet wird. Wir greifen hier auf dieses Verfahren zurück und definieren $\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{c})$, wobei \mathbf{W} die Gewichte einer linearen Transformation angibt und \mathbf{c} die Verzerrung. Zuvor haben wir bei der Beschreibung eines linearen Regressionsmodells einen Vektor der Gewichte und einen skalaren Verzerrungsparameter zum Beschreiben einer affinen Transformation zwischen einem Eingabevektor und einem Ausgabeskalar genutzt. Jetzt beschreiben wir eine affine Transformation von einem Vektor \mathbf{x} zu einem Vektor \mathbf{h} , sodass ein vollständiger Vektor mit Verzerrungsparametern benötigt wird. Die Aktivierungsfunktion g wird normalerweise als Funktion gewählt, die elementweise angewandt wird, mit $h_i = g(\mathbf{x}^\top \mathbf{W}_{:,i} + c_i)$. In modernen neuronalen Netzen wird empfohlen, die **ReLU** (engl. *rectified linear unit*, dt. *rektifizierte lineare Einheit*) (Jarrett et al., 2009; Nair und Hinton, 2010; Glorot et al., 2011a) zu nutzen, die durch die in Abbildung 6.3 dargestellte Aktivierungsfunktion $g(z) = \max\{0, z\}$ definiert wird.

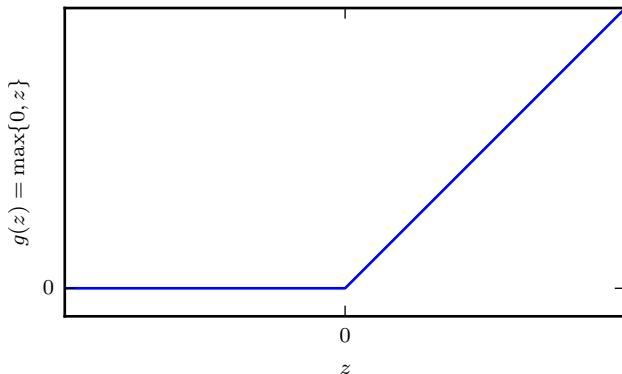


Abbildung 6.3: Die ReLU-Funktion. Diese Aktivierungsfunktion ist die üblicherweise empfohlene Aktivierungsfunktion für die meisten neuronalen Feedforward-Netze. Durch das Anwenden dieser Funktion auf die Ausgabe einer linearen Transformation ergibt sich eine nichtlineare Transformation. Die Funktion ist beinahe linear, allerdings handelt es sich um eine stückweise lineare Funktion mit zwei linearen Abschnitten. Da ReLUs nahezu linear sind, erhalten sie viele der Eigenschaften, die das einfache Optimieren von linearen Modellen mithilfe von Methoden auf Gradientenbasis erlauben. Außerdem erhalten sie viele der Eigenschaften, die zu einer guten Fähigkeit zur Generalisierung von linearen Modellen beitragen. Ein allgemeiner Grundsatz in der Informatik besagt, dass sich komplexe Systeme aus einfachen Komponenten erstellen lassen. So wie der Speicher einer Turing-Maschine nur die Zustände 0 oder 1 speichern können muss, lässt sich aus ReLU-Funktionen ein universeller Funktionsapproximator bauen.

Wir können nun unser komplettes Netz wie folgt angeben:

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^\top \max\{0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}\} + b. \quad (6.3)$$

Damit kommen wir zu einer Lösung des XOR-Problems. Es sei

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad (6.4)$$

$$\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad (6.5)$$

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad (6.6)$$

und $b = 0$.

Nun können wir verfolgen, wie das Modell eine Reihe von Eingaben verarbeitet. Es sei \mathbf{X} die Entwurfsmatrix, die alle vier Punkte im binären Eingaberaum enthält, jeweils ein Beispiel pro Zeile:

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \quad (6.7)$$

Der erste Schritt im neuronalen Netz ist die Multiplikation der Eingabematrix mit der Gewichtungsmatrix der ersten Schicht:

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \quad (6.8)$$

Nun addieren wir den Verzerrungsvektor \mathbf{c} und erhalten

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.9)$$

In diesem Raum liegen alle Beispiele auf einer Linie mit der Steigung 1. Wenn wir der Linie folgen, muss die Ausgabe bei 0 beginnen, bis 1 steigen

und dann wieder auf 0 fallen. Ein lineares Modell kann eine solche Funktion nicht implementieren. Um den Wert für h für jedes Beispiel zu berechnen, wenden wir die rektifizierte lineare Transformation an:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.10)$$

Diese Transformation hat die Beziehung zwischen den Beispielen geändert. Sie liegen nicht mehr auf einer Linie. Wie Abbildung 6.1 zeigt, liegen sie nun in einem Raum, in dem ein lineares Modell zur Lösung eingesetzt werden kann.

Abschließend führen wir eine Multiplikation mit dem Gewichtungsvektor w durch:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \quad (6.11)$$

Das neuronale Netz hat die korrekte Antwort für jedes Beispiel aus dem Batch ermittelt.

In unserem Beispiel haben wir einfach die Lösungen vorgegeben und dann gezeigt, dass kein Fehler auftritt. In der Realität gibt es möglicherweise Milliarden von Modellparametern und Milliarden von Trainingsbeispielen, die das hier erfolgte Erraten der Lösung nicht erlauben. Stattdessen kann ein Optimierungsalgorithmus auf Gradientenbasis Parameter suchen, die nur zu einem geringen Fehler führen. Die für das XOR-Problem beschriebene Lösung liegt im globalen Minimum der Verlustfunktion – auch das Gradientenabstiegsverfahren könnte gegen diesen Punkt konvergieren. Es gibt andere, gleichwertige Lösungen für das XOR-Problem, die sich ebenfalls mittels Gradientenabstiegsverfahrens finden lassen. Der Konvergenzpunkt des Gradientenabstiegsverfahrens ist von den Anfangswerten der Parameter abhängig. In der Praxis findet das Gradientenabstiegsverfahren keine eindeutigen, leicht verständlichen, ganzzahligen Lösungen wie in diesem Beispiel.

6.2 Lernen auf Gradientenbasis

Design und Training eines neuronalen Netzes unterscheidet sich kaum vom Training anderer Machine-Learning-Modelle mit Gradientenabstiegsverfahren. In Abschnitt 5.10 haben wir gezeigt, wie ein Machine-Learning-Algorithmus durch Angeben von Optimierungsverfahren, Kostenfunktion und Modellfamilie entsteht.

Der größte Unterschied zwischen den bisher betrachteten linearen Modellen und neuronalen Netzen ist, dass die Nichtlinearität eines neuronalen Netzes dazu führt, dass die interessantesten Verlustfunktionen nichtkonvex werden. Daher werden neuronale Netze meist mithilfe iterativer Optimierungen auf Gradientenbasis trainiert, die die Kostenfunktion auf einen sehr niedrigen Wert steuern, und nicht mithilfe von Lösungswegen für lineare Gleichungen (die beim Training linearer Regressionsmodelle zum Einsatz kommen) oder mithilfe von konvexen Optimierungsalgorithmen mit globalen Konvergenzgarantien (die für logistische Regressionen oder SVMs genutzt werden). Die Konvergenz konvexer Optimierungen beginnt mit beliebigen Ausgangsparametern (in der Theorie – in der Praxis kann es trotz ihrer Robustheit zu numerischen Problemen kommen). Das stochastische Gradientenabstiegsverfahren für nichtkonvexe Verlustfunktionen hat häufig keine solche Konvergenzgarantie und hängt stark von den Werten der Ausgangsparameter ab. In neuronalen Feedforward-Netzen kommt es darauf an, sämtliche Gewichte mit kleinen Zufallswerten zu initialisieren. Die Verzerrungen können auf Null oder sehr kleine positive Werte initialisiert werden. Die iterativen Optimierungsalgorithmen auf Gradientenbasis zum Trainieren von Feedforward-Netzen und nahezu allen anderen tiefen Modellen werden detailliert in Kapitel 8 behandelt, die Parameterinitialisierung speziell in Abschnitt 8.4. Bis auf Weiteres reicht es aus, wenn Sie verstehen, dass der Trainingsalgorithmus fast immer darauf basiert, dass der Gradient auf die eine oder andere Art in Richtung Kostenfunktion absteigt. Die spezifischen Algorithmen sind Verbesserungen und Verfeinerungen des Gradientenabstiegsverfahrens, das wir in Abschnitt 4.3 vorgestellt haben. Insbesondere handelt es sich oft um Verbesserungen des Algorithmus für das stochastische Gradientenabstiegsverfahren aus Abschnitt 5.9.

Wir können Modelle wie lineare Regression und Support Vector Machines natürlich auch anhand des Gradientenabstiegsverfahrens trainieren – und in der Tat ist das ein gängiger Weg, wenn die Trainingsdatenmenge extrem groß ist. Von diesem Standpunkt aus betrachtet unterscheidet sich das Training eines neuronalen Netzes nicht sonderlich von dem anderer Modelle. Die Berechnung des Gradienten ist beim neuronalen Netz etwas komplizierter,

lässt sich aber dennoch effizient und exakt durchführen. In Abschnitt 6.5 zeigen wir, wie der Gradient mithilfe des Backpropagation-Algorithmus und moderner Generalisierung dieses Algorithmus ermittelt werden kann.

Wie bei anderen Machine-Learning-Modellen müssen wir zur Anwendung des Lernens auf Gradientenbasis eine Kostenfunktion auswählen und angeben, wie die Ausgabe des Modells repräsentiert werden soll. Wenden wir uns nun erneut den Designentscheidungen unter besonderer Berücksichtigung neuronaler Netze zu.

6.2.1 Kostenfunktionen

Ein wichtiger Aspekt bei der Konstruktion eines tiefen neuronalen Netzes besteht in der Auswahl der Kostenfunktion. Glücklicherweise werden für neuronale Netze mehr oder weniger dieselben Kostenfunktionen wie für andere parametrische Modelle, beispielsweise lineare Modelle, verwendet.

In den meisten Fällen definiert unser parametrisches Modell eine Verteilung $p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$ und wir setzen einfach das Prinzip der Maximum Likelihood ein, nutzen also die Kreuzentropie zwischen den Trainingsdaten und den Modellvorhersagen als Kostenfunktion.

Manchmal geht es noch einfacher und wir treffen keine Vorhersage über eine vollständige Wahrscheinlichkeitsverteilung für \mathbf{y} , sondern sagen lediglich eine statistische Größe von \mathbf{y} – bedingt durch \mathbf{x} – vorher. Spezialisierte Verlustfunktionen ermöglichen das Trainieren einer Näherung dieser Schätzwerte.

Die zum Trainieren eines neuronalen Netzes verwendete vollständige Kostenfunktion kombiniert häufig eine der hier beschriebenen primären Kostenfunktionen mit einem Regularisierungsterm. In Abschnitt 5.2.2 haben Sie bereits einige einfache Beispiele für die Regularisierung linearer Modelle kennengelernt. Weight Decay für lineare Modelle lässt sich auch unverändert auf tiefe neuronale Netze anwenden und gehört zu den beliebtesten Regularisierungsverfahren. Weitere fortgeschrittene Regularisierungsverfahren für neuronale Netze finden Sie in Kapitel 7.

6.2.1.1 Erlernen bedingter Verteilungen mit Maximum Likelihood

Die meisten modernen neuronalen Netze werden anhand der Maximum Likelihood trainiert. Das bedeutet, dass die Kostenfunktion einfach die negative Log-Likelihood ist, die auch als Kreuzentropie zwischen den Trainingsdaten

und der Modellverteilung beschrieben wird. Diese Kostenfunktion ergibt sich aus

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}). \quad (6.12)$$

Die spezifische Form der Kostenfunktion ändert sich von Modell zu Modell, und zwar abhängig von der spezifischen Form von $\log p_{\text{model}}$. Die Entwicklung der genannten Gleichung führt normalerweise zu einigen Ter- men, die nicht von den Modellparametern abhängig sind und somit verworfen werden können. Wie Sie in Abschnitt 5.5.1 gesehen haben, erhalten wir für $p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{I})$ zum Beispiel die Kosten des mittleren quadratischen Fehlers

$$J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})\|^2 + \text{const} \quad (6.13)$$

bis zu einem Skalierungsfaktor von $\frac{1}{2}$ und einen nicht von $\boldsymbol{\theta}$ abhängigen Term. Die verworfene Konstante basiert auf der Varianz der Normalverteilung, die wir in diesem Fall nicht parametrisiert haben. Sie haben bereits gesehen, dass die Äquivalenz zwischen der Maximum-Likelihood-Schätzung mit einer Ausgabeverteilung und Minimierung des mittleren quadratischen Fehlers für ein lineares Modell gilt; tatsächlich gilt sie ungeachtet der verwendeten Funktion $f(\mathbf{x}; \boldsymbol{\theta})$ zur Vorhersage des Mittelwerts der Normalverteilung.

Ein Vorteil bei der Ermittlung der Kostenfunktion aus der Maximum Likelihood ist, dass der Aufwand für die Entwicklung der Kostenfunktionen für jedes Modell entfällt. Durch Spezifizieren eines Modells $p(\mathbf{y} \mid \mathbf{x})$ wird automatisch eine Kostenfunktion $\log p(\mathbf{y} \mid \mathbf{x})$ festgelegt.

Es ist ein wiederkehrendes Thema bei der Entwicklung neuronaler Netze, dass der Gradient der Kostenfunktion groß und vorhersagbar genug sein muss, um als gute Richtschnur für den Lernalgorithmus zu dienen. Funktionen, die ihre Sättigung erreichen (sehr flach werden), untergraben dieses Ziel, da der Gradient dadurch sehr klein wird. Häufig geschieht dies, weil die Aktivierungsfunktionen für die Ausgabe der verdeckten Einheiten oder der Ausgabeeinheiten ihre Sättigung erreichen. Die negative Log-Likelihood hilft, das Problem bei vielen Modellen zu vermeiden. Mehrere Ausgabeeinheiten nutzen eine exp-Funktion, die bei stark negativem Argument eine Sättigung erreicht. Die log-Funktion der negativen Log-Likelihood-Kostenfunktion kehrt den exp einiger Ausgabeeinheiten um. Wir behandeln diese Interaktion zwischen der Kostenfunktion und der Auswahl der Ausgabeeinheit in Abschnitt 6.2.2.

Eine ungewöhnliche Eigenschaft der Kosten der Kreuzentropie für die Maximum-Likelihood-Schätzung besteht darin, dass sie üblicherweise kein

Minimum aufweist, wenn sie auf die in der Praxis häufig verwendeten Modelle angewandt wird. Für diskrete Ausgabevervariablen werden die meisten Modelle so parametrisiert, dass keine Wahrscheinlichkeit von Null oder Eins dargestellt werden kann, aber eine weitestgehende Annäherung daran möglich ist. Die logistische Regression ist ein Beispiel für ein solches Modell. Wenn das Modell bei reellwertigen Ausgabevervariablen die Dichte der Ausgabeverteilung steuern kann (zum Beispiel durch Erlernen des Varianzparameters einer Ausgabevernormalverteilung), dann wird es möglich, den korrekten Ausgaben der Trainingsdatenmenge eine extrem hohe Dichte zuzuweisen, was dazu führt, dass die Kreuzentropie sich der negativen Unendlichkeit annähert. Die in Kapitel 7 beschriebenen Regularisierungsverfahren bieten mehrere Möglichkeiten zum Verändern des Lernproblems, damit das Modell diesen Weg nicht zum Einstreichen uneingeschränkter Belohnungen nutzen kann.

6.2.1.2 Erlernen bedingter statistischer Größen

Anstatt eine komplette Wahrscheinlichkeitsverteilung $p(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ zu erlernen, interessieren wir uns häufig nur für eine bedingte statistische Größe für \mathbf{y} auf Basis von \mathbf{x} .

Wir können zum Beispiel über eine Näherung $f(\mathbf{x}; \boldsymbol{\theta})$ verfügen, die wir direkt zur Vorhersage des Mittelwerts von \mathbf{y} nutzen möchten.

Wenn wir ein ausreichend leistungsfähiges neuronales Netz verwenden, können wir uns vorstellen, dass es eine beliebige Funktion f aus einer großen Funktionsklasse repräsentieren kann, wobei die Klasse lediglich durch Merkmale wie Stetigkeit und Eingeschränktheit begrenzt wird – nicht etwa durch eine spezifische parametrische Form. Von diesem Standpunkt aus können wir die Kostenfunktion statt als Funktion als **Funktional** betrachten. Ein Funktional ist eine Zuordnung von Funktionen in die reellen Zahlen. Wir können uns das Lernen als Auswählen einer Funktion anstelle einer Menge von Parametern vorstellen. Wir können unser Kostenfunktional so auslegen, dass sein Minimum auf einer von uns ausgesuchten Funktion liegt. So können wir das Kostenfunktional mit dem Minimum auf der Funktion konstruieren, die \mathbf{x} dem erwarteten Wert von \mathbf{y} anhand von \mathbf{x} zuordnet. Das Lösen eines Optimierungsproblems für eine Funktion setzt ein mathematisches Werkzeug namens **Variationsrechnung** voraus; wir beschreiben es in Abschnitt 19.4.2. Für das Verständnis dieses Kapitels müssen Sie die Variationsrechnung nicht verstehen. Für den Moment reicht es aus, zu wissen, dass die Variationsrechnung zur Herleitung der folgenden beiden Ergebnisse genutzt werden kann.

Unser erstes Ergebnis auf Grundlage der Variationsrechnung zeigt, dass die Lösung des Optimierungsproblems

$$f^* = \arg \min_f \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|^2 \quad (6.14)$$

zu

$$f^*(\mathbf{x}) = \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}(\mathbf{y} | \mathbf{x})} [\mathbf{y}] \quad (6.15)$$

führt, sofern diese Funktion in derselben Klasse liegt, für die wir die Optimierung durchführen. Anders ausgedrückt: Wenn wir anhand von unendlich vielen Stichproben (engl. *samples*) aus der wahren datengenerierenden Verteilung trainieren können, würde das Minimieren des MQF-Kostenfunktion eine Funktion ergeben, die den Mittelwert von \mathbf{y} für jeden Wert von \mathbf{x} vorhersagt.

Unterschiedliche Kostenfunktionen führen zu unterschiedlichen statistischen Größen. Ein zweites Ergebnis aus der Variationsrechnung besagt, dass

$$f^* = \arg \min_f \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|_1 \quad (6.16)$$

zu einer Funktion führt, die den *Medianwert* von \mathbf{y} für jedes \mathbf{x} vorhersagt, sofern eine Funktion durch die Funktionsfamilie beschrieben werden kann, für die wir die Optimierung durchführen. Diese Kostenfunktion wird häufig als **mittlerer absoluter Fehler** bezeichnet.

Leider führen mittlerer quadratischer Fehler und mittlerer absoluter Fehler in Kombination mit der Optimierung auf Gradientenbasis häufig zu schlechten Ergebnissen. Einige Ausgabeeinheiten mit Sättigung erzeugen in Verbindung mit diesen Kostenfunktionen sehr kleine Gradienten. Das ist einer der Gründe dafür, dass die Kreuzentropie als Kostenfunktion beliebter ist als der mittlere quadratische Fehler oder der mittlere absolute Fehler, auch wenn das Schätzen einer vollständigen Verteilung $p(\mathbf{y} | \mathbf{x})$ nicht erforderlich ist.

6.2.2 Ausgabeeinheiten

Die Wahl der Kostenfunktion ist eng an die Wahl der Ausgabeeinheit gekoppelt. Meist nutzen wir einfach die Kreuzentropie zwischen der Datenverteilung und der Modellverteilung. Die Art der Darstellung der Ausgabe bestimmt dann die Form der Kreuzentropie-Funktion.

Jede Art von Einheit neuronaler Netze, die als Ausgabe verwendet werden kann, kann auch als verdeckte Einheit genutzt werden. Hier konzentrieren wir uns auf die Nutzung dieser Einheiten als Ausgaben des Modells, aber

prinzipiell lassen sie sich auch intern nutzen. Wir beschäftigen uns noch genauer mit diesen Einheiten und der Verwendung als verdeckte Einheiten in Abschnitt 6.3.

In diesem Abschnitt gehen wir davon aus, dass das Feedforward-Netz eine Menge verdeckter Merkmale bereitstellt, die definiert sind als $\mathbf{h} = f(\mathbf{x}; \boldsymbol{\theta})$. Die Rolle der Ausgabeschicht besteht darin, eine zusätzliche Transformation von den Merkmalen vorzunehmen, um die Aufgabe des Netzes abzuschließen.

6.2.2.1 Lineare Einheiten für Ausgabe-Normalverteilungen

Eine einfache Art von Ausgabeeinheiten beruht auf einer affinen Transformation ohne Nichtlinearität. Man spricht hier oft lediglich von linearen Einheiten.

Aus den Merkmalen \mathbf{h} erzeugt eine Schicht linearer Ausgabeeinheiten einen Vektor $\hat{\mathbf{y}} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$.

Lineare Ausgabeschichten dienen häufig zum Erzeugen des Mittelwerts einer bedingten Normalverteilung:

$$p(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I}). \quad (6.17)$$

Das Maximieren der Log-Likelihood entspricht dann dem Minimieren des mittleren quadratischen Fehlers.

Die Maximum Likelihood erleichtert auch das Erlernen der Kovarianz der Normalverteilung; außerdem lässt sich die Kovarianz der Normalverteilung leichter als Funktion der Eingabe bilden. Allerdings muss die Kovarianz auf eine positiv definite Matrix für sämtliche Eingaben eingeschränkt werden. Derartige Bedingungen lassen sich mit einer linearen Ausgabeschicht nur schwer einhalten, sodass meist andere Ausgabeeinheiten zum Parametrisieren der Kovarianz verwendet werden. In Abschnitt 6.2.2.4 behandeln wir gleich Ansätze zum Modellieren der Kovarianz.

Da lineare Einheiten nicht sättigen, stellen sie keine große Schwierigkeit für Optimierungsalgorithmen auf Gradientenbasis dar und können mit einer Vielzahl von Optimierungsalgorithmen genutzt werden.

6.2.2.2 Sigmoide Einheiten für Bernoulli-Ausgabeverteilungen

Bei vielen Aufgaben muss der Wert einer binären Variable y vorhergesagt werden. Klassifizierungsprobleme mit zwei Klassen lassen sich so darstellen.

Der Maximum-Likelihood-Ansatz definiert eine Bernoulli-Verteilung über y mit der Bedingung \mathbf{x} .

Zur Definition einer Bernoulli-Verteilung reicht eine einzelne Zahl. Das neuronale Netz muss lediglich $P(y = 1 | \mathbf{x})$ vorhersagen. Damit diese Zahl eine gültige Wahrscheinlichkeit aufweist, muss sie im Intervall $[0, 1]$ liegen.

Um diese Bedingung zu erfüllen, müssen wir sorgfältig vorgehen. Angenommen, wir würden eine lineare Einheit nutzen und einen Schwellenwert festlegen, um eine gültige Wahrscheinlichkeit zu erhalten:

$$P(y = 1 | \mathbf{x}) = \max \left\{ 0, \min \left\{ 1, \mathbf{w}^\top \mathbf{h} + b \right\} \right\}. \quad (6.18)$$

Damit erhalten wir eine gültige bedingte Verteilung, aber wir können diese nicht effektiv mit dem Gradientenabstiegsverfahren trainieren. Immer wenn $\mathbf{w}^\top \mathbf{h} + b$ außerhalb des Intervalls $[0, 1]$ läge, würde der Gradient der Ausgabe des Modells bezüglich seiner Parameter $\mathbf{0}$ werden. Ein Gradient von $\mathbf{0}$ stellt normalerweise ein Problem dar, da der Lernalgorithmus nicht länger eine Richtschnur zur Verbesserung der entsprechenden Parameter hat.

Stattdessen ist es besser, einem anderen Ansatz zu folgen, der sicherstellt, dass bei einer falschen Antwort des Modells ein starker Gradient vorliegt. Dieser Ansatz beruht auf sigmoiden Ausgabeeinheiten in Kombination mit der Maximum Likelihood.

Eine sigmoide Ausgabeeinheit ist definiert durch

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{h} + b), \quad (6.19)$$

wobei σ die logistische Sigmoidfunktion aus Abschnitt 3.10 ist.

Sie können sich die sigmoide Ausgabeeinheit als zweiteilig vorstellen. Zunächst nutzt sie eine lineare Schicht zum Berechnen von $z = \mathbf{w}^\top \mathbf{h} + b$. Anschließend nutzt sie die sigmoide Aktivierungsfunktion zum Umwandeln von z in eine Wahrscheinlichkeit.

Wir lassen die Abhängigkeit von \mathbf{x} für den Moment beiseite, um zu behandeln, wie eine Wahrscheinlichkeitsverteilung mit dem Wert z über y gebildet wird. Das Sigmoid kann durch Konstruieren einer nicht normalisierten Wahrscheinlichkeitsverteilung $\tilde{P}(y)$, deren Summe von 1 verschieden ist, motiviert werden. Wir können dann eine passende Konstante als Divisor verwenden, um eine gültige Wahrscheinlichkeitsverteilung zu erhalten. Wenn wir von der Annahme ausgehen, dass die nicht normalisierte Log-Wahrscheinlichkeiten (engl. *log-probabilities*) in y und z linear sind, können wir potenzieren, um die nicht normalisierten Wahrscheinlichkeiten zu bestimmen. Anschließend normalisieren wir und stellen fest, dass eine Bernoulli-Verteilung vorliegt, die einer sigmoidalen Transformation von z folgt:

$$\log \tilde{P}(y) = yz, \quad (6.20)$$

$$\tilde{P}(y) = \exp(yz), \quad (6.21)$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^1 \exp(y'z)}, \quad (6.22)$$

$$P(y) = \sigma((2y - 1)z). \quad (6.23)$$

Wahrscheinlichkeitsverteilungen auf Basis von Potenzierung und Normalisierung sind in der Fachliteratur zur statistischen Modellierung gängig. Die z -Variable zur Definition einer solchen Verteilung über binäre Variablen wird **Logit** genannt.

Dieser Weg zum Vorhersagen der Wahrscheinlichkeiten im Log-Raum ist für das Maximum Likelihood Learning prädestiniert. Da die verwendete Kostenfunktion für die Maximum Likelihood $-\log P(y | \boldsymbol{x})$ ist, hebt der log in der Kostenfunktion den exp des Sigmoids auf. Ohne diesen Effekt könnte die Sättigung des Sigmoids das Lernen auf Gradientenbasis an guten Fortschritten hindern. Die Verlustfunktion für das Maximum Likelihood Learning einer mittels Sigmoid parametrisierten Bernoulli-Verteilung ist

$$J(\boldsymbol{\theta}) = -\log P(y | \boldsymbol{x}) \quad (6.24)$$

$$= -\log \sigma((2y - 1)z) \quad (6.25)$$

$$= \zeta((1 - 2y)z). \quad (6.26)$$

Diese Herleitung nutzt einige Eigenschaften aus Abschnitt 3.10. Wenn wir den Verlust als Softplus-Funktion schreiben, stellen wir fest, dass nur für ein sehr stark negatives $(1 - 2y)z$ eine Sättigung eintritt. Es kommt nur zur Sättigung, wenn das Modell bereits die richtige Antwort gefunden hat – also $y = 1$ und z sehr positiv ist oder $y = 0$ und z sehr negativ ist. Bei falschem Vorzeichen für z kann das Argument der Softplus-Funktion, $(1 - 2y)z$, zu $|z|$ vereinfacht werden. Da $|z|$ bei falschem Vorzeichen von z groß wird, nähert sich die Softplus-Funktion der Rückgabe des Arguments $|z|$ an. Die Ableitung nach z nähert sich $\text{sign}(z)$ an, sodass die Softplus-Funktion im Grenzbereich eines drastisch fehlerhaften z den Gradienten überhaupt nicht verkleinert. Diese Eigenschaft ist nützlich, da das Lernen auf Gradientenbasis so ein fehlerhaftes z schnell korrigieren kann.

Wenn wir andere Verlustfunktionen wie den MQF verwenden, kann jedes Mal eine Sättigung des Verlusts auftreten, wenn $\sigma(z)$ gesättigt ist. Die sigmoide Aktivierungsfunktion sättigt gegen 0, wenn z stark negativ wird, und gegen 1, wenn z stark positiv wird. Dabei kann der Gradient zu klein werden, um ihn noch zum Lernen zu verwenden – unabhängig davon, ob das Modell die korrekte oder die falsche Antwort gefunden hat. Daher ist die Maximum Likelihood der bevorzugte Ansatz für das Trainieren von sigmoiden Ausgabeeinheiten.

Analytisch betrachtet ist der Logarithmus der Sigmoidfunktion stets definiert und endlich, da das Sigmoid nur Werte aus dem offenen Intervall $(0, 1)$ zurückgibt und nicht den gesamten geschlossenen Intervall der gültigen Wahrscheinlichkeiten $[0, 1]$ betrachtet. Bei der Softwareimplementierung lassen sich rechnerische Probleme vermeiden, indem die negative Log-Likelihood als Funktion von z geschrieben wird, nicht als eine Funktion von $\hat{y} = \sigma(z)$. Weist die Sigmoidfunktion einen Unterlauf in Richtung 0 auf, dann führt der Logarithmus von \hat{y} zur negativen Unendlichkeit.

6.2.2.3 softmax-Einheiten für Multinoulli-Ausgabeverteilungen

Wenn wir eine Wahrscheinlichkeitsverteilung über eine diskrete Variable mit n möglichen Werten darstellen möchten, können wir die softmax-Funktion verwenden. Sie ist eine Art Generalisierung der Sigmoidfunktion, die zur Repräsentation einer Wahrscheinlichkeitsverteilung über eine binäre Variable diente.

softmax-Funktionen werden meist als Ausgabe eines Klassifikators benutzt, um die Wahrscheinlichkeitsverteilung über n verschiedene Klassen darzustellen. Seltener werden softmax-Funktionen auch im Modell selbst eingesetzt, um eine von n unterschiedlichen Möglichkeiten für eine interne Variable auszuwählen.

Im Fall der binären Variablen möchten wir eine einzelne Zahl bestimmen:

$$\hat{y} = P(y = 1 | \mathbf{x}). \quad (6.27)$$

Da diese Zahl zwischen 0 und 1 liegen muss und einen für die Optimierung der Log-Likelihood auf Gradientenbasis geeigneten Logarithmus aufweisen soll, entscheiden wir uns stattdessen dafür, eine Zahl $z = \log \tilde{P}(y = 1 | \mathbf{x})$ vorherzusagen. Durch Potenzierung und Normalisierung haben wir eine Bernoulli-Verteilung erhalten, die einer Sigmoidfunktion folgt.

Um eine Generalisierung für eine diskrete Variable mit n Werten vorzunehmen, müssen wir einen Vektor $\hat{\mathbf{y}}$ erzeugen, mit $\hat{y}_i = P(y = i | \mathbf{x})$. Nicht nur muss jedes Element \hat{y}_i zwischen 0 und 1 liegen, sondern der gesamte Vektor muss sich zu 1 aufsummieren, damit er eine gültige Wahrscheinlichkeitsverteilung darstellt. Derselbe Ansatz, der für die Bernoulli-Verteilung funktioniert hat, lässt sich auch auf die Multinoulli-Verteilung anwenden. Zunächst sagt eine lineare Schicht die nicht normalisierten Log-Wahrscheinlichkeiten vorher:

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}, \quad (6.28)$$

wobei $z_i = \log \tilde{P}(y = i | \mathbf{x})$ ist. Die softmax-Funktion kann dann \mathbf{z} potenzieren und normalisieren, um das gewünschte $\hat{\mathbf{y}}$ zu erhalten. Formal ergibt sich die softmax-Funktion aus

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \quad (6.29)$$

Wie bei der logistischen Sigmoidfunktion passt die exp-Funktion gut für das Trainieren der softmax zur Ausgabe eines Zielwerts y mithilfe der Maximum Log-Likelihood. In diesem Fall möchten wir $\log P(y = i; \mathbf{z}) = \log \text{softmax}(\mathbf{z})_i$ maximieren. Das Definieren der softmax als exp gelingt ganz leicht, da \log in der Log-Likelihood den exp von softmax aufheben kann:

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j). \quad (6.30)$$

Der erste Term in Gleichung 6.30 zeigt, dass die Eingabe z_i stets direkt zur Kostenfunktion beiträgt. Da dieser Term keine Sättigung erreichen kann, wissen wir, dass der Lernprozess auch dann fortgesetzt werden kann, wenn der Beitrag von z_i zum zweiten Term in Gleichung 6.30 sehr klein wird. Beim Maximieren der Log-Likelihood wird dem ersten Term nach z_i erhöht, während dem zweiten nach \mathbf{z} insgesamt nach unten steuert. Um ein Gespür für den zweiten Term, $\log \sum_j \exp(z_j)$, zu erhalten, sollten Sie sich bewusst machen, dass dieser Term grob durch $\max_j z_j$ approximiert werden kann. Diese Approximation beruht auf der Vorstellung, dass $\exp(z_k)$ für jedes z_k , das deutlich kleiner als $\max_j z_j$ ist, unwesentlich ist. Diese Approximation zeigt uns, dass die Kostenfunktion der negativen Log-Likelihood die aktivste falsche Vorhersage stets am stärksten bestraft. Wenn die korrekte Antwort bereits die größte Eingabe für die softmax aufweist, heben sich der Term $-z_i$ und die Terme $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$ in etwa auf. Dieses Beispiel trägt dann nur wenig zu den Gesamtkosten des Trainings bei, die vielmehr von den anderen, noch nicht korrekt klassifizierten Beispielen dominiert werden.

Bisher haben wir uns nur mit einem Beispiel befasst. Insgesamt führt die nicht regularisierte Maximum Likelihood dazu, dass das Modell Parameter erlernt, die dafür sorgen, dass die softmax die Bruchteile jedes beobachteten Ergebnisses der Trainingsdatenmenge vorhersagt:

$$\text{softmax}(\mathbf{z}(\mathbf{x}; \boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^m \mathbf{1}_{y^{(j)}=i, \mathbf{x}^{(j)}=\mathbf{x}}}{\sum_{j=1}^m \mathbf{1}_{\mathbf{x}^{(j)}=\mathbf{x}}}. \quad (6.31)$$

Da die Maximum Likelihood ein konsistenter Schätzer ist, kommt es garantiert dazu, sofern die Modellfamilie die Verteilung der Trainingsdaten

überhaupt vorhersagen kann. In der Praxis kann das Modell aufgrund einer eingeschränkten Modellkapazität und einer unvollkommenen Optimierung diese Bruchteile lediglich approximieren.

Viele Zielfunktionen mit Ausnahme der Log-Likelihood funktionieren nicht so gut mit der softmax-Funktion. Insbesondere scheitern Zielfunktionen, die keinen log zum Aufheben des exp der softmax nutzen, am Lernen, wenn das Argument für den exp stark negativ wird und der Gradient so verschwindet. Gerade der quadratische Fehler ist eine schlechte Verlustfunktion für softmax-Einheiten und kann dazu führen, dass das Modell nicht zum Ändern der Ausgabe trainiert werden kann, obwohl es sehr souveräne, aber fehlerhafte Vorhersagen macht (Bridle, 1990). Warum diese anderen Verlustfunktionen scheitern können, zeigt eine Untersuchung der softmax-Funktion selbst.

Wie das Sigmoid kann auch die softmax-Aktivierung eine Sättigung erreichen. Die Sigmoidfunktion weist eine einzelne Ausgabe auf, die gesättigt wird, wenn ihre Eingabe stark negativ bzw. stark positiv ist. Die softmax weist mehrere Ausgabewerte auf. Diese Ausgabewerte erreichen eine Sättigung, wenn die Unterschiede zwischen den Eingabewerten extrem groß sind. Wenn die softmax gesättigt ist, sind auch viele Kostenfunktionen, die auf der softmax basieren, gesättigt – es sei denn, sie können die Aktivierungsfunktion der Sättigung umkehren.

Um zu erkennen, dass die softmax-Funktion auf den Unterschied zwischen den Eingaben reagiert, hilft es, sich vor Augen zu führen, dass die softmax-Ausgabe invariant gegenüber der Ergänzung desselben Skalars zu all ihren Eingaben ist:

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} + c). \quad (6.32)$$

Anhand dieser Eigenschaft können wir eine numerisch robuste Variante der softmax herleiten:

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} - \max_i z_i). \quad (6.33)$$

Die neu formulierte Version ermöglicht es, die softmax mit nur kleinen numerischen Fehlern zu berechnen, selbst wenn z besonders große oder stark negative Zahlen enthält. Beim Untersuchen der numerisch robusten Variante erkennen wir, dass die softmax-Funktion durch die Abweichung ihrer Argumente gegenüber $\max_i z_i$ gesteuert wird.

Eine Ausgabe $\text{softmax}(\mathbf{z})_i$ sättigt gegen 1, wenn die zugehörige Eingabe maximal ($z_i = \max_i z_i$) und z_i deutlich größer als alle anderen Eingaben ist. Die Ausgabe $\text{softmax}(\mathbf{z})_i$ kann auch gegen 0 sättigen, wenn z_i nicht maximal ist, sondern das Maximum viel größer ist. Dies ist eine Generalisierung

der Sättigung von sigmoiden Einheiten, die ähnliche Schwierigkeiten beim Lernen verursachen kann, wenn keine Kompensation in die Verlustfunktion eingebaut wurde.

Das Argument z der softmax-Funktion lässt sich auf zwei Arten erzeugen. Die übliche Art baut auf eine vorherige Schicht im neuronalen Netz, die jedes Element von z ausgibt, wie oben anhand der linearen Schicht $z = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$ beschrieben. Das ist zwar einfach, führt aber irgendwann zu einer Überparametrisierung der Verteilung. Die Bedingung, dass die n Ausgaben insgesamt 1 ergeben müssen, bedeutet, dass nur $n - 1$ Parameter erforderlich sind; die Wahrscheinlichkeit des n -ten Werts ergibt sich aus der Differenz zwischen den ersten $n - 1$ Wahrscheinlichkeiten und 1. Wir können daher vorgeben, dass ein Element von z unveränderlich ist. Zum Beispiel können wir fordern, dass $z_n = 0$ ist. Tatsächlich macht die sigmoide Einheit genau dies. Die Definition $P(y = 1 | \mathbf{x}) = \sigma(z)$ entspricht der Definition $P(y = 1 | \mathbf{x}) = \text{softmax}(\mathbf{z})_1$ mit zweidimensionalem \mathbf{z} und $z_1 = 0$. Sowohl das Argument $n - 1$ als auch das Argument n nähern sich der softmax an und können dieselbe Menge von Wahrscheinlichkeitsverteilungen beschreiben, aber doch unterschiedliche Lerndynamiken aufweisen. In der Praxis gibt es nur selten große Unterschiede zwischen der überparametrisierten Version und der bedingten Version, sodass es einfacher ist, die überparametrisierte Version zu implementieren.

Aus neurowissenschaftlicher Sicht ist es interessant, sich die softmax als Möglichkeit vorzustellen, eine Art Wettbewerb zwischen den Einheiten auszurufen, die daran teilnehmen: Die softmax-Ausgaben ergeben in Summe stets 1, sodass eine Erhöhung einer Einheit notwendigerweise zu einer Verringerung in den Werten der anderen führt. Das ist ein Analogon zur lateralen Hemmung, die vermutlich zwischen einander naheliegenden Neuronen im Kortex besteht. Im ungünstigsten Fall (wenn der Unterschied zwischen dem größten a_i und den anderen Werten groß ist) gilt das **Winner-Take-All-Prinzip** (eine der Ausgaben ist fast 1, die anderen fast 0).

Der Name »softmax« kann irreführend sein. Die Funktion ist enger mit der Funktion $\arg \max$ als der \max -Funktion verwandt. Der Begriff »soft« gibt an, dass die softmax-Funktion stetig und differenzierbar ist. Die Funktion $\arg \max$, deren Ergebnis als One-hot-Vektor dargestellt wird, ist weder stetig noch differenzierbar. Die softmax-Funktion stellt damit eine glattere Version von $\arg \max$ dar. Die entsprechende glatte Ausführung der maximum-Funktion ist $\text{softmax}(\mathbf{z})^\top \mathbf{z}$. Vermutlich wäre die Bezeichnung »softargmax« korrekter, aber softmax hat sich nun einmal eingebürgert.

6.2.2.4 Andere Ausgabetypen

Meist werden die zuvor beschriebenen linearen, sigmoiden und softmax-Ausgabeeinheiten verwendet. Neuronale Netze können auf nahezu jede gewünschte Ausgabeschicht generalisieren. Das Prinzip der Maximum Likelihood dient als Richtschnur für den Aufbau einer guten Kostenfunktion für nahezu beliebige Arten von Ausgabeschichten.

Grundsätzlich empfiehlt das Prinzip der Maximum Likelihood für eine bedingte Verteilung $p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$ die Verwendung von $-\log p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$ als Kostenfunktion.

Sie können sich vorstellen, dass das neuronale Netz eine Funktion $f(\mathbf{x}; \boldsymbol{\theta})$ repräsentiert. Die Ausgaben dieser Funktion sind keine direkten Vorhersagen des Werts \mathbf{y} . Stattdessen liefert $f(\mathbf{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$ die Parameter für eine Verteilung über y . Unsere Verlustfunktion lässt sich dann als $-\log p(\mathbf{y}; \boldsymbol{\omega}(\mathbf{x}))$ interpretieren.

Ein Beispiel: Wir möchten die Varianz einer bedingten Normalverteilung für \mathbf{y} mit \mathbf{x} erlernen. Im einfachen Fall – wenn die Varianz σ^2 eine Konstante ist – gibt es einen geschlossenen Ausdruck, da der Maximum-Likelihood-Schätzer der Varianz einfach das empirische Mittel des Quadrats der Differenz zwischen den Beobachtungen \mathbf{y} und ihrem erwarteten Wert sind. Ein berechnungstechnisch aufwendigerer Ansatz, der ohne für Sonderfälle geschriebenen Code auskommt, bindet die Varianz als eine Eigenschaft der durch $\boldsymbol{\omega} = f(\mathbf{x}; \boldsymbol{\theta})$ gesteuerten Verteilung $p(\mathbf{y} | \mathbf{x})$ ein. Die negative Log-Likelihood $-\log p(\mathbf{y}; \boldsymbol{\omega}(\mathbf{x}))$ gibt dann eine Kostenfunktion mit den passenden Ausdrücken an, die zum inkrementellen Erlernen der Varianz durch unser Optimierungsverfahren erforderlich sind. Im einfachen Fall – wenn die Standardabweichung nicht von der Eingabe abhängig ist – können wir einen neuen Parameter im Netz erstellen, der direkt in $\boldsymbol{\omega}$ kopiert wird. Dieser neue Parameter kann abhängig von der Art der Parametrisierung der Verteilung entweder σ selbst sein oder ein Parameter v , der für σ^2 steht, oder ein Parameter β , der für $\frac{1}{\sigma^2}$ steht. Wir möchten vielleicht, dass unser Modell einen anderen Varianzbetrag in \mathbf{y} für unterschiedliche Werte von \mathbf{x} vorhersagt. Dies wird als **heteroskedastisches** Modell bezeichnet. Im heteroskedastischen Fall geben wir lediglich vor, dass die Varianz einem der von $f(\mathbf{x}; \boldsymbol{\theta})$ ausgegebenen Werte entsprechen muss. Eine typische Methode hierfür besteht darin, die Normalverteilung mithilfe der Präzision anstelle der Varianz anzugeben (vgl. Gleichung 3.22). Im mehrdimensionalen Fall wird meist eine Diagonal-Präzisionsmatrix eingesetzt:

$$\text{diag}(\boldsymbol{\beta}) \quad (6.34)$$

Dieser Ansatz funktioniert gut mit dem Gradientenabstiegsverfahren, da die Formel für die Log-Likelihood der durch β parametrisierten Normalverteilung lediglich eine Multiplikation mit β_i und eine Addition von $\log \beta_i$ einschließt. Der Gradient von Multiplikations-, Additions- und Logarithmusberechnungen verhält sich angemessen. Wenn wir dagegen die Ausgabe anhand der Varianz parametrisieren würden, wäre eine Division erforderlich. Die Division wird in der Nähe von Null beliebig steil. Zwar sind große Gradienten für das Lernen hilfreich, aber beliebig große Gradienten führen meist zu einer Instabilität. Wenn wir die Ausgabe anhand der Standardabweichung parametrisieren würden, wäre zum Berechnen der Log-Likelihood neben der Division auch noch eine Quadrierung erforderlich. Der Gradient kann durch das Quadrieren in der Nähe von Null verschwinden, sodass das Erlernen quadrierter Parameter schwierig wird. Ungeachtet davon, ob wir die Standardabweichung, Varianz oder Präzision verwenden, müssen wir sicherstellen, dass die Kovarianzmatrix der Normalverteilung positiv definit ist. Da die Eigenwerte der Präzisionsmatrix der Kehrwert der Eigenwerte der Kovarianzmatrix sind, ist damit auch die Präzisionsmatrix positiv definit. Wenn wir eine Diagonalmatrix oder das skalare Vielfache der Diagonalmatrix verwenden, müssen wir nur dafür sorgen, dass die Ausgabe des Modells Positivität aufweist. Wenn wir a als reine Aktivierung des Modells zum Bestimmen der Diagonal-Präzisionsmatrix betrachten, können wir die Soft-plus-Funktion zum Bestimmen eines positiven Präzisionsvektors verwenden: $\beta = \zeta(a)$. Diese Vorgehensweise gilt ebenso bei Verwendung der Varianz oder Standardabweichung anstelle der Präzision oder bei der Verwendung eines skalaren Vielfaches der Identität anstelle einer Diagonalmatrix.

Es ist eher selten, eine Kovarianz- oder Präzisionsmatrix zu erlernen, deren Struktur umfangreicher als die einer Diagonalmatrix ist. Ist die Kovarianz voll und bedingt, muss die Parametrisierung garantieren, dass die vorhergesagte Kovarianzmatrix positiv definit wird. Dazu schreiben wir $\Sigma(\mathbf{x}) = \mathbf{B}(\mathbf{x})\mathbf{B}^\top(\mathbf{x})$, wobei \mathbf{B} eine Quadratmatrix ohne Nebenbedingungen ist. Ein praktisches Problem für eine Matrix mit vollem Rang besteht darin, dass die Berechnung der Likelihood aufwendig ist: Eine $d \times d$ -Matrix erfordert die Berechnung $O(d^3)$ für die Determinante und Inverse von $\Sigma(\mathbf{x})$ (bzw. gleichwertig und üblicher die Eigenwertzerlegung ihrer selbst oder von $\mathbf{B}(\mathbf{x})$).

Häufig möchten wir eine multimodale (oder mehrgipflige) Regression vornehmen, also die tatsächlichen Werte einer bedingten Verteilung $p(\mathbf{y} | \mathbf{x})$ mit mehreren unterschiedlichen Gipfeln im \mathbf{y} -Raum für denselben Wert von \mathbf{x} vorhersagen. In diesem Fall stellt eine gaußsche Mischverteilung eine natürliche Repräsentation für die Ausgabe dar (*Jacobs et al., 1991; Bishop,*

1994). Neuronale Netze mit gaußschen Mischverteilungen als Ausgabe werden häufig als **Mixture-Density-Netze** bezeichnet. Eine Ausgabe einer gaußschen Mischverteilung mit n Komponenten wird durch die bedingte Wahrscheinlichkeitsverteilung definiert:

$$p(\mathbf{y} \mid \mathbf{x}) = \sum_{i=1}^n p(c = i \mid \mathbf{x}) \mathcal{N}(\mathbf{y}; \boldsymbol{\mu}^{(i)}(\mathbf{x}), \boldsymbol{\Sigma}^{(i)}(\mathbf{x})). \quad (6.35)$$

Das neuronale Netz muss drei Ausgaben aufweisen: Einen Vektor zur Definition von $p(c = i \mid \mathbf{x})$, eine Matrix, die $\boldsymbol{\mu}^{(i)}(\mathbf{x})$ für alle i angibt, und einen Tensor, der $\boldsymbol{\Sigma}^{(i)}(\mathbf{x})$ für alle i angibt. Diese Ausgaben müssen mehrere Bedingungen erfüllen:

1. Komponenten der Mischung: $p(c = i \mid \mathbf{x})$: Diese bilden eine Multinoulli-Verteilung über die n verschiedenen Komponenten, die zu einer latenten Variable¹ c gehören. Sie lassen sich normalerweise mit softmax über einen n -dimensionalen Vektor ermitteln, um sicherzustellen, dass diese Ausgaben positiv sind und in der Summe 1 ergeben.
2. Mittelwerte $\boldsymbol{\mu}^{(i)}(\mathbf{x})$: Diese geben das Zentrum oder den Mittelwert für die i -te normalverteilte Komponente an und unterliegen keinen Bedingungen (typischerweise ohne Nichtlinearität für diese Ausgabeeinheiten). Ist \mathbf{y} ein d -Vektor, muss das Netz eine $n \times d$ -Matrix mit allen n dieser d -dimensionalen Vektoren ausgeben. Das Erlernen dieser Mittelwerte anhand der Maximum Likelihood ist etwas komplexer als das Erlernen der Mittelwerte einer Verteilung mit nur einem Ausgabemodus. Wir möchten nur das Mittel der Komponente anpassen, die tatsächlich die Beobachtung erzeugt hat. In der Praxis wissen wir allerdings nicht, welche Komponente für einzelne Beobachtungen verantwortlich ist. Der Ausdruck der negativen Log-Likelihood gewichtet den Beitrag jedes Beispiels zum Verlust jeder Komponente anhand der Wahrscheinlichkeit, dass diese Komponente das Beispiel erzeugt hat.
3. Kovarianzen $\boldsymbol{\Sigma}^{(i)}(\mathbf{x})$: Diese geben die Kovarianzmatrix für jede Komponente i an. Wie beim Erlernen einer einzelnen normalverteilten Komponente verwenden wir meist eine Diagonalmatrix, um uns das

¹ Wir betrachten c als latent, da wir es in den Daten nicht beobachten: Für die Eingabe \mathbf{x} und den Zielwert \mathbf{y} können wir nicht mit Bestimmtheit wissen, welche normalverteilte Komponente für \mathbf{y} verantwortlich war, aber wir können uns vorstellen, dass \mathbf{y} durch die Wahl einer dieser Komponenten erzeugt wurde; und wir können diese nicht beobachtete Wahl zu einer Zufallsvariable machen.

Berechnen der Determinanten zu ersparen. Wie beim Erlernen der Mittelwerte der Mischung, wird die Maximum Likelihood dadurch verkompliziert, dass jedem Punkt jeder Komponente der Mischung eine anteilige Verantwortlichkeit zugewiesen werden muss. Das Gradientenabstiegsverfahren folgt automatisch dem korrekten Prozess, wenn im Rahmen des Mischmodells die korrekte Spezifikation der negativen Log-Likelihood vorgegeben wird.

Es gibt Berichte, nach denen die Optimierung auf Gradientenbasis für bedingte gaußsche Mischverteilungen (bei der Ausgabe neuronaler Netze) unzuverlässig sein kann, zum Teil als Folge der Divisionen (durch die Varianz), die rechnungsmäßig instabil sein können (wenn eine Varianz zu klein für ein bestimmtes Beispiel wird und so sehr große Gradienten erzeugt). Eine Lösung besteht im **Gradienten-Clipping** (siehe Abschnitt 10.11.1), eine andere im heuristischen Skalieren der Gradienten (*Murray und Larochelle, 2014*).

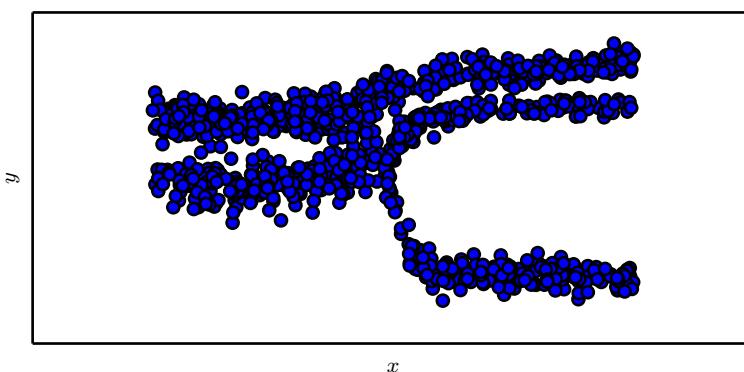


Abbildung 6.4: Aus einem neuronalen Netz gezogene Stichproben mit einer Mixture-Density-Ausgabeschicht. Die Eingabe x stammt aus einer Gleichverteilung, die Ausgabe y aus $p_{\text{model}}(y | x)$. Das neuronale Netz kann die nichtlinearen Zuordnungen der Eingabe zu den Parametern der Ausgabeverteilung erlernen. Diese Parameter enthalten die Wahrscheinlichkeiten dafür, welche der drei Komponenten der Mischung die Ausgabe erzeugen wird, sowie die Parameter der einzelnen Komponenten der Mischung. Jede der Komponenten ist normalverteilt und weist einen vorhergesagten Mittelwert und eine Varianz auf. All diese Aspekte der Ausgabeverteilung sind bezüglich der Eingabe x auf nichtlineare Weise variabel.

Die Ausgaben einer gaußschen Mischverteilung sind besonders effektiv für generative Sprachmodelle (*Schuster, 1999*) und Bewegungen physischer Objekte (*Graves, 2013*). Das Mixture-Density-Verfahren ermöglicht dem Netz die Darstellung mehrerer Ausgabemodus sowie die Steuerung der Varianz seiner Ausgabe, was für ein hohes Maß an Qualität in diesen

reellwertigen Definitionsbereichen unabdingbar ist. Abbildung 6.4 zeigt ein solches Mixture-Density-Netz.

Grundsätzlich möchten wir weiterhin größere Vektoren \mathbf{y} mit mehr Variablen modellieren und immer umfassendere Strukturen für diese Ausgabevariablen vorgeben. Wenn unser neuronales Netz zum Beispiel eine Sequenz von Zeichen ausgeben soll, die einen Satz bildet, dann können wir weiterhin das Maximum-Likelihood-Prinzip auf unser Modell $p(\mathbf{y}; \boldsymbol{\omega}(\mathbf{x}))$ anwenden. In diesem Fall wird das Modell, mit dem wir \mathbf{y} beschreiben, komplex genug, um den Rahmen dieses Kapitels zu sprengen. In Kapitel 10 beschreiben wir, wie RNNs zum Definieren solcher Modelle für Sequenzen verwendet werden. Teil III befasst sich mit fortschrittlichen Verfahren zur Modellierung beliebiger Wahrscheinlichkeitsverteilungen.

6.3 Verdeckte Einheiten

Bisher haben wir uns in erster Linie um die Designentscheidungen für neuronale Netze gekümmert, die am häufigsten für parametrische Machine-Learning-Modelle, die mit einer Optimierung auf Gradientenbasis trainiert werden, zum Einsatz kommen. Jetzt wenden wir uns einem Thema zu, das ausschließlich für neuronale Feedforward-Netze von Bedeutung ist. Die Frage hierzu lautet: Wie wird die Art der verdeckten Einheiten ausgewählt, die in den verdeckten Schichten des Modells genutzt werden?

Das Design verdeckter Einheiten ist ein extrem geschäftiges Forschungsfeld; es gibt derzeit kaum definitive theoretische Grundsätze, die als Richtlinie dienen können.

ReLUs sind eine hervorragende Standardwahl für die verdeckte Einheit. Aber es sind noch viele weitere Arten von verdeckten Einheiten möglich. Welche davon wann zum Einsatz kommen sollte, ist nicht einfach zu beantworten. (Allerdings sind, wie erwähnt, ReLUs im Allgemeinen eine angemessene Wahl.) Hier möchten wir Ihnen zeigen, was die Nutzung der einzelnen Arten von verdeckten Einheiten antreibt. Das vermittelt Ihnen ein Gespür dafür, mit welcher Einheit Sie es versuchen sollten. Eine definitive Antwort auf die Frage nach der besten Einheit im Voraus zu geben, ist nahezu unmöglich. Der Designprozess beruht auf Versuch und Irrtum. Man vermutet, dass eine bestimmte Art von verdeckter Einheit gut funktionieren könnte. Anschließend wird ein Netz mit dieser Art trainiert und seine Leistung auf einer Validierungsdatenmenge bewertet.

Für einige verdeckte Einheiten in dieser Aufstellung ist tatsächlich keine Differentiation an allen Eingabepunkten möglich. Zum Beispiel ist die rekti-

fizierte lineare Funktion (engl. *rectified linear function*) $g(z) = \max\{0, z\}$ in $z = 0$ nicht differenzierbar. Das scheint ein K.-o.-Kriterium für die Nutzung von g mit einem Lernalgorithmus auf Gradientenbasis zu sein. In der Praxis funktioniert das Gradientenabstiegsverfahren jedoch gut genug, damit diese Modelle auf Machine-Learning-Aufgaben angesetzt werden können. Das liegt zum Teil daran, dass Trainingsalgorithmen für neuronale Netze selten ein lokales Minimum der Kostenfunktion erreichen, sondern deren Wert vielmehr erheblich reduzieren (vgl. Abbildung 4.3). (Diese Konzepte werden in Kapitel 8 genauer behandelt.) Da wir vom Training nicht erwarten, tatsächlich einen Punkt zu erreichen, an dem der Gradient **0** ist, reicht es aus, wenn die Minima der Kostenfunktion den Punkten mit nicht definiertem Gradienten entsprechen. Nicht differenzierbare verdeckte Einheiten sind normalerweise nur für eine kleine Anzahl von Punkten nicht differenzierbar. Grundsätzlich weist eine Funktion $g(z)$ eine linksseitige Ableitung auf, die durch die Steigung der Funktion unmittelbar links von z definiert wird, sowie eine rechtsseitige Ableitung, die durch die Steigung der Funktion unmittelbar rechts von z definiert wird. Eine Funktion ist nur dann in z differenzierbar, wenn sowohl die linksseitige als auch die rechtsseitige Ableitung definiert ist und sie gleich sind. Die für neuronale Netze verwendeten Funktionen verfügen üblicherweise über definierte linksseitige und rechtsseitige Ableitungen. Im Fall $g(z) = \max\{0, z\}$ ist die linksseitige Ableitung in $z = 0$ gleich 0 und die rechtsseitige Ableitung ist 1. Softwareimplementierungen für das Training neuronaler Netze geben normalerweise eine der einseitigen Ableitungen aus und melden nicht, dass die Ableitung nicht definiert ist oder zu einem Fehler führt. Das lässt sich heuristisch begründen, da die Optimierung auf Gradientenbasis in einem digitalen Computer sowieso einem numerischen Fehler unterliegt. Wenn eine Funktion $g(0)$ berechnen soll, ist es extrem unwahrscheinlich, dass der zugrunde liegende Wert wirklich 0 war. Stattdessen war es höchstwahrscheinlich ein sehr niedriger Wert ϵ , der auf 0 abgerundet wurde. Je nach Umfeld sind auch theoretisch ansprechendere Begründungen möglich, aber diese gelten meist nicht für das Trainieren neuronaler Netze. Der Knackpunkt ist, dass wir in der Praxis die Nichtdifferenzierbarkeit der im Folgenden beschriebenen Aktivierungsfunktionen von verdeckten Einheiten ungestraft ignorieren können.

Sofern nicht anders angegeben, lassen sich die meisten verdeckten Einheiten so beschreiben: Sie nehmen einen Vektor mit Eingaben \mathbf{x} entgegen, berechnen eine affine Transformation $\mathbf{z} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$ und wenden dann eine elementweise nichtlineare Funktion $g(\mathbf{z})$ an. Die meisten verdeckten Einheiten sind voneinander nur über die Wahl der Form der Aktivierungsfunktion $g(\mathbf{z})$ unterscheidbar.

6.3.1 Rektifizierte lineare Einheiten (ReLUs) und ihre Generalisierung

ReLUs nutzen die Aktivierungsfunktion $g(z) = \max\{0, z\}$.

Diese Einheiten lassen sich leicht optimieren, da sie linearen Einheiten stark ähneln. Der einzige Unterschied zwischen einer linearen Einheit und einer ReLU besteht darin, dass Letztere für die Hälfte ihres Definitionsbereichs Null ausgibt. So bleiben die Ableitungen durch eine ReLU groß, sofern die Einheit aktiv ist. Die Gradienten sind nicht nur groß, sondern auch konsistent. Die zweite Ableitung der Rektifizierung ist fast überall 0. Die Ableitung der Rektifizierung ist überall dort, wo die Einheit aktiv ist, 1. Somit ist die Gradientenrichtung sehr viel nützlicher für das Lernen, als dies bei Aktivierungsfunktionen mit Auswirkungen zweiter Ordnung der Fall wäre.

ReLUs werden meist zusätzlich zu einer affinen Transformation eingesetzt:

$$\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b}). \quad (6.36)$$

Beim Initialisieren der Parameter der affinen Transformation ist es häufig nützlich, alle Elemente von \mathbf{b} auf einen kleinen positiven Wert einzustellen, zum Beispiel 0,1. Dadurch wird es höchst wahrscheinlich, dass die ReLUs zu Beginn für die meisten Eingaben der Trainingsdatenmenge aktiv sind und die Ableitungen hindurchgehen.

Es gibt mehrere Generalisierungen von ReLUs. Die meisten davon funktionieren ähnlich gut wie ReLUs und manchmal besser.

Ein Nachteil der ReLUs ist, dass sie nicht mittels Verfahren auf Gradientenbasis anhand von Beispielen lernen können, für die ihre Aktivierung Null ist. Diverse Generalisierungen von ReLUs garantieren, dass überall der Gradient erreicht wird.

Drei Generalisierungen von ReLUs basieren auf der Nutzung einer von Null verschiedenen Steigung α_i für $z_i < 0$: $h_i = g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$. **Rektifizierung des Absolutbetrags** (engl. *Absolute Value Rectification*) hält $\alpha_i = -1$ fest, um $g(z) = |z|$ zu erhalten. Sie wird bei der Objekterkennung in Bildern eingesetzt (Jarrett et al., 2009), denn dabei ist es sinnvoll, nach Merkmalen zu suchen, die invariant gegenüber einer Polaritätsumkehr der Eingabe-Beleuchtung (engl. *input illumination*) sind. Andere Generalisierungen von ReLUs lassen sich vielfältiger einsetzen. Eine **Leaky ReLU** (Maas et al., 2013) hält α_i auf einem kleinen Wert wie 0,01 fest, während eine **parametrische ReLU** (kurz PReLU) α_i als erlernbaren Parameter behandelt (He et al., 2015).

Maxout-Einheiten (*Goodfellow et al.*, 2013a) generalisierten ReLUs noch weiter. Statt eine elementweise Funktion $g(z)$ zu verwenden, teilen Maxout-Einheiten \mathbf{z} in Gruppen mit k Werten auf. Jede Maxout-Einheit gibt dann das größte Element einer dieser Gruppen aus:

$$g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j, \quad (6.37)$$

wobei $\mathbb{G}^{(i)}$ die Menge der Indizes der Eingaben der Gruppe i , $\{(i-1)k+1, \dots, ik\}$ ist. Damit kann eine stückweise lineare Funktion erlernt werden, die auf mehrere Richtungen im Eingaberaum \mathbf{x} reagiert.

Eine Maxout-Einheit kann eine stückweise lineare, konvexe Funktion mit bis zu k Teilen erlernen. Man kann also sagen, dass Maxout-Einheiten die *Aktivierungsfunktion selbst erlernen*, nicht nur die Beziehung zwischen Einheiten. Ist k groß genug, kann eine Maxout-Einheit das Approximieren einer beliebigen konvexen Funktion mit der gewünschten Genauigkeit erlernen. Insbesondere kann eine Maxout-Schicht mit zwei Teilen lernen, dieselbe Funktion der Eingabe \mathbf{x} zu implementieren, wie eine klassische Schicht dies tut, und zwar mit der ReLU-Funktion, der Absolutbetragrektifizierungsfunktion, der Leaky oder der parametrischen ReLU. Sie kann auch lernen, eine ganz andere Funktion zu implementieren. Die Maxout-Schicht wird natürlich anders als diese anderen Schichttypen parametrisiert, sodass die Lerndynamiken sich sogar in Fällen unterscheiden, in denen Maxout dieselbe Funktion \mathbf{x} wie eine der anderen Schichttypen zu implementieren erlernt.

Jede Maxout-Einheit ist nun nicht nur über einen, sondern über k Gewichtungsvektoren parametrisiert. Daher benötigen Maxout-Einheiten normalerweise eine stärkere Regularisierung als ReLUs. Sie können auch ohne Regularisierung gut funktionieren, wenn die Trainingsdatenmenge groß und die Anzahl der Teile pro Einheit klein ist (*Cai et al.*, 2013).

Maxout-Einheiten bieten noch einige andere Vorteile. In manchen Fällen lassen sich statistische und Berechnungsvorteile erzielen, wenn weniger Parameter benötigt werden. Wenn die von n unterschiedlichen linearen Filtern erfassten Merkmale ohne Informationsverlust zusammengefasst werden können, indem der Höchstwert für jede Gruppe aus k Merkmalen gezogen wird, benötigt die nächste Schicht zum Beispiel k -mal weniger Gewichte.

Da hinter jeder Einheit mehrere Filter stecken, weisen Maxout-Einheiten eine gewisse Redundanz auf, die sie bis zu einem gewissen Grad vor einem als **katastrophales Vergessen** bezeichneten Phänomen schützt. Dabei vergessen neuronale Netze, wie sie Aufgaben bewältigen müssen, für die sie bereits trainiert wurden (*Goodfellow et al.*, 2014a).

ReLUs und all diese Generalisierungen beruhen auf dem Prinzip, dass Modelle einfacher zu optimieren sind, wenn ihr Verhalten sich eher linear verhält. Dieser Grundsatz gilt auch außerhalb der tiefen linearen Netze. RNNs können aus Sequenzen lernen und erzeugen eine Sequenz von Zuständen und Ausgaben. Werden sie trainiert, müssen Informationen durch mehrere zeitliche Schritte propagierte werden – das ist viel einfacher, wenn einige lineare Berechnungen daran beteiligt sind (und einige Richtungsableitungen einen Betrag nahe 1 aufweisen). Zu den rekurrenten Netzarchitekturen mit der besten Leistung gehört das LSTM (engl. *long short-term memory*), das Informationen durch die Zeit mittels Aufsummierung propagierte – eine sehr zielgerichtete Art der linearen Aktivierung. Wir gehen in Abschnitt 10.10 näher darauf ein.

6.3.2 Logistische Sigmoidfunktion und Tangens hyperbolicus

Vor dem Aufkommen der ReLUs setzten die meisten neuronalen Netze als Aktivierungsfunktion auf die logistische Sigmoidfunktion

$$g(z) = \sigma(z) \quad (6.38)$$

oder den Tangens hyperbolicus

$$g(z) = \tanh(z). \quad (6.39)$$

Diese Aktivierungsfunktionen sind eng miteinander verwandt, denn $\tanh(z) = 2\sigma(2z) - 1$.

Sie haben bereits gesehen, wie sigmoide Einheiten als Ausgabeeinheiten zur Vorhersage der Wahrscheinlichkeit, dass eine binäre Variable 1 ist, verwendet werden. Anders als stückweise lineare Einheiten sättigen sigmoide Einheiten im Großteil ihres Definitionsbereichs – die Sättigung erfolgt bei einem hohen Wert, wenn z stark positiv ist, und bei einem niedrigen Wert, wenn z stark negativ ist; außerdem sind die Einheiten nur dann stark von ihrer Eingabe abhängig, wenn z nahe 0 ist. Die weitverbreitete Sättigung der sigmoiden Einheiten kann sich als großes Problem für das Lernen auf Gradientenbasis erweisen. Daher wird mittlerweile davon abgeraten, sie als verdeckte Einheiten in Feedforward-Netzen einzusetzen. Die Verwendung als Ausgabeeinheit ist mit dem Einsatz des Lernens auf Gradientenbasis kompatibel, wenn eine passende Kostenfunktion die Sättigung der Sigmoidfunktion in der Ausgabeschicht ignoriert kann.

Wenn eine Sigmoidfunktion als Aktivierungsfunktion genutzt werden muss, leistet der Tangens hyperbolicus normalerweise bessere Arbeit als

die logistische Sigmoidfunktion. Er ähnelt eher der identischen Abbildung (auch identische Funktion), da $\tanh(0) = 0$ für $\sigma(0) = \frac{1}{2}$ ist. Da \tanh der identischen Abbildung in der Nähe von 0 ähnelt, erinnert das Trainieren eines tiefen neuronalen Netzes $\hat{y} = \mathbf{w}^\top \tanh(\mathbf{U}^\top \tanh(\mathbf{V}^\top \mathbf{x}))$ an das Trainieren eines linearen Modells $\hat{y} = \mathbf{w}^\top \mathbf{U}^\top \mathbf{V}^\top \mathbf{x}$, sofern die Aktivierungen des Netzes klein gehalten werden können. Das erleichtert das Trainieren des \tanh -Netzes.

Sigmoide Aktivierungsfunktionen sind abseits von Feedforward-Netzen häufiger anzutreffen. RNNs, viele probabilistische Modelle und einige Autoencoder schließen die Nutzung stückweiser linearer Aktivierungsfunktionen aus und machen sigmoide Einheiten trotz der Nachteile durch die Sättigung reizvoll.

6.3.3 Andere verdeckte Einheiten

Es gibt noch viele weitere Arten von verdeckten Einheiten, die allerdings weniger häufig eingesetzt werden.

Grundsätzlich funktionieren viele differenzierbare Funktionen ganz wunderbar. Viele unveröffentlichte Aktivierungsfunktionen funktionieren ebenso gut wie ihre verbreiteten Schwestern. Ein konkretes Beispiel: Wir haben ein Feedforward-Netz mithilfe von $\mathbf{h} = \cos(\mathbf{W}\mathbf{x} + \mathbf{b})$ auf den MNIST-Datensatz angesetzt und eine Fehlerquote von weniger als einem Prozent erhalten – die Funktion ist den klassischen Aktivierungsfunktionen also durchaus gewachsen. Während der Forschung an und der Entwicklung von neuen Verfahren werden meist viele unterschiedliche Aktivierungsfunktionen ausprobiert. Dabei stellt sich heraus, dass diverse Variationen in der normalen Praxis ähnliche Leistungen an den Tag legen. Andererseits werden neue Arten von verdeckten Einheiten daher auch nur veröffentlicht, wenn sie eine deutliche Verbesserung gegenüber bisherigen Einheiten darstellen. Verdeckte Einheiten mit ähnlicher Leistung gibt es wie Sand am Meer – eine Veröffentlichung wäre einfach nicht interessant genug.

Eine Aufzählung aller in der Literatur erwähnten Arten von verdeckten Einheiten wäre unpraktisch, weswegen wir nur einige besonders nützliche und markante vorstellen.

Eine Möglichkeit besteht darin, gänzlich auf eine Aktivierung $g(z)$ zu verzichten. Dann spielt also die identische Abbildung die Rolle der Aktivierungsfunktion. Sie haben bereits festgestellt, dass eine lineare Einheit als Ausgabe eines neuronalen Netzes nützlich sein kann. Aber sie kann auch als verdeckte Einheit dienen. Besteht jede Schicht des neuronalen

Netzes nur aus linearen Transformationen, ist das Netz insgesamt linear. Allerdings dürfen durchaus einige Schichten des neuronalen Netzes rein linear sein. Betrachten Sie eine Schicht im neuronalen Netz mit n Eingaben und p Ausgaben: $\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$. Wir können diese durch zwei Schichten ersetzen, von denen eine die Gewichtungsmatrix \mathbf{U} und die andere die Gewichtungsmatrix \mathbf{V} verwendet. Wenn die erste Schicht keine Aktivierungsfunktion enthält, faktorisieren wir praktisch die Gewichtungsmatrix der ursprünglichen Schicht auf Basis von \mathbf{W} . Der faktorierte Ansatz besteht im Berechnen von $\mathbf{h} = g(\mathbf{V}^\top \mathbf{U}^\top \mathbf{x} + \mathbf{b})$. Führt \mathbf{U} zu q Ausgaben, enthalten \mathbf{U} und \mathbf{V} gemeinsam nur $(n+p)q$ Parameter, wohingegen \mathbf{W} np Parameter enthält. Ist q klein, lässt sich die Anzahl der Parameter so deutlich reduzieren. Das führt dazu, dass die lineare Transformation auf einen niedrigen Rang beschränkt werden muss, aber derartig niederrangige Beziehungen sind oft ausreichend. Lineare verdeckte Einheiten bieten somit eine wirkungsvolle Möglichkeit zur Reduzierung der Anzahl der Parameter in einem Netz.

softmax-Einheiten sind eine andere Art von Einheit, die normalerweise als Ausgabe (vgl. Abschnitt 6.2.2.3), manchmal aber auch als verdeckte Einheit genutzt wird. softmax-Einheiten stellen natürlicherweise eine Wahrscheinlichkeitsverteilung über eine diskrete Variable mit k möglichen Werten dar, sodass sie als eine Art Schalter dienen können. Solche verdeckten Einheiten werden meist nur in recht fortschrittlichen Architekturen eingesetzt, in denen der Speicherinhalt geändert wird (vgl. Abschnitt 10.12).

Zu den gängigeren Arten verdeckter Einheiten gehören auch diese:

- **Radiale Basisfunktion** (RBF), Einheit: $h_i = \exp\left(-\frac{1}{\sigma_i^2} \|\mathbf{W}_{:,i} - \mathbf{x}\|^2\right)$. Diese Funktion wird aktiver, wenn sich \mathbf{x} einem Template $\mathbf{W}_{:,i}$ annähert. Da für die meisten \mathbf{x} eine Sättigung bei 0 eintritt, ist die Optimierung ggf. komplex.
- **Softplus**: $g(a) = \zeta(a) = \log(1 + e^a)$. Dies ist eine glatte Version des Rectifiers, die von *Dugas et al.* (2001) zur Funktionsapproximation und von *Nair und Hinton* (2010) für die bedingten Verteilungen von ungerichteten probabilistischen Modellen eingeführt wurde. *Glorot et al.* (2011a) haben Softplus und den Rectifier miteinander verglichen und festgestellt, dass Letzterer die besseren Ergebnisse liefert. Allgemein wird von Softplus abgeraten. Softplus zeigt, dass die Leistung der Arten von verdeckten Einheiten stark kontraintuitiv sein kann – man erwartet aufgrund der Differenzierbarkeit an jeder Stelle oder

weniger starken Sättigung einen Vorteil gegenüber dem Rectifier, der empirisch nicht gegeben ist.

- **Hard tanh.** Die Form erinnert an tanh und den Rectifier, aber anders als der zweite ist diese Funktion eingeschränkt: $g(a) = \max(-1, \min(1, a))$. Sie wurde von *Collobert* (2004) eingeführt.

Das Design verdeckter Einheiten ist weiterhin ein aktives Forschungsfeld. Gewiss gibt es noch viele nützliche verdeckte Einheiten, die nur darauf warten, entdeckt zu werden.

6.4 Architekturdesign

Eine wesentliche Designüberlegung bei neuronalen Netzen liegt in der Entscheidung für eine Architektur. Das Wort **Architektur** bezieht sich auf den Gesamtaufbau des Netzes, also die Anzahl der vorgesehenen Einheiten und deren Verbindung untereinander.

Die meisten neuronalen Netze sind in mehrere Gruppen von Einheiten unterteilt, die Schichten genannt werden. In den meisten neuronalen Netzarchitekturen werden diese Schichten in Form einer Kette angeordnet, sodass jede Schicht eine Funktion ihrer Vorgängerschicht darstellt. In dieser Struktur ergibt sich die erste Schicht aus

$$\mathbf{h}^{(1)} = g^{(1)} \left(\mathbf{W}^{(1)\top} \mathbf{x} + \mathbf{b}^{(1)} \right); \quad (6.40)$$

die zweite Schicht ergibt sich aus

$$\mathbf{h}^{(2)} = g^{(2)} \left(\mathbf{W}^{(2)\top} \mathbf{h}^{(1)} + \mathbf{b}^{(2)} \right); \quad (6.41)$$

usw. In solchen Kettenstrukturen besteht die primäre architektonische Überlegung darin, wie tief das Netz sein soll und wie breit jede der Schichten. Wie wir noch zeigen, reicht bereits ein Netz mit nur einer verdeckten Schicht aus, um die Trainingsdatenmenge anzupassen. Tiefere Netze kommen häufig mit sehr viel weniger Einheiten pro Schicht und sehr viel weniger Parametern aus. Gleichzeitig sind sie zu häufigen Generalisierungen über die Testdatenmenge imstande, aber auch schwieriger zu optimieren. Die ideale Netzarchitektur für eine Aufgabe muss in Experimenten bestimmt werden, in denen der Validierungsdatenfehler beobachtet wird.

6.4.1 Eigenschaften und Tiefe der universellen Approximation

Ein lineares Modell, das mithilfe der Matrizenmultiplikation eine Zuordnung von Merkmalen zu Ausgaben vornimmt, kann per definitionem nur lineare Funktionen abbilden. Es ist leicht zu trainieren, da viele Verlustfunktionen in Verbindung mit linearen Modellen zu konvexen Optimierungsproblemen führen. Leider sollen unsere Systeme jedoch häufig nichtlineare Funktion erlernen.

Auf den ersten Blick könnten wir vermuten, dass zum Erlernen einer nichtlinearen Funktion eine spezielle Modelfamilie für die entsprechende Nichtlinearität entwickelt werden muss. Zum Glück stellen Feedforward-Netze mit verdeckten Schichten ein Framework für universelle Approximation (engl. *universal approximation*) bereit. Vor allem das **Theorem der universellen Approximation** (Hornik et al., 1989; Cybenko, 1989) besagt, dass ein Feedforward-Netz mit einer linearen Ausgabeschicht und mindestens einer verdeckten Schicht mit einer beliebigen »zwingenden« Aktivierungsfunktion (z. B. der logistischen Sigmoidfunktion) jede Borel-messbare Funktion aus einem endlich-dimensionalen Raum einer anderen mit einem beliebigen gewünschten von Null verschiedenen Fehlerbetrag approximieren kann, sofern dem Netz genügend verdeckte Einheiten zur Verfügung gestellt werden. Die Ableitungen des Feedforward-Netzes können sogar die Ableitungen der Funktion angemessen gut approximieren (Hornik et al., 1990). Das Konzept der Borel-Messbarkeit würde den Rahmen dieses Buchs sprengen. Für unsere Zwecke reicht es aus zu betonen, dass jede stetige Funktion einer geschlossenen und eingeschränkten Teilmenge des \mathbb{R}^n Borel-messbar ist und somit durch ein neuronales Netz approximiert werden kann. Ein neuronales Netz kann auch jede beliebige Funktionszuordnung aus einem endlich-dimensionalen diskreten Raum in einen anderen approximieren. Während die ursprünglichen Theoreme zunächst in Begrifflichkeiten von Einheiten mit Aktivierungsfunktionen, die für stark negative und stark positive Argumente zur Sättigung führen, ausgedrückt worden sind, wurden die Theoreme der universellen Approximation auch für eine breitere Klasse von Aktivierungsfunktionen bewiesen, darunter die aktuell häufig genutzte ReLU (Leshno et al., 1993).

Das Theorem der universellen Approximation bedeutet, dass wir ungetacht der zu erlernenden Funktion wissen, dass ein großes mehrschichtiges Perzeptron diese Funktion *repräsentieren* kann. Es ist allerdings nicht garantiert, dass der Trainingsalgorithmus die Funktion *erlernen* kann. Auch wenn das mehrschichtige Perzeptron die Funktion repräsentieren kann, könnte

das Erlernen doch aus zwei unterschiedlichen Gründen scheitern: Erstens ist der für das Training verwendete Optimierungsalgorithmus möglicherweise nicht in der Lage, die Parameterwerte zu ermitteln, die der gewünschten Funktion entsprechen. Zweitens wählt der Trainingsalgorithmus womöglich als Folge einer Überanpassung die falsche Funktion aus. Sie erinnern sich (siehe Abschnitt 5.2.1), dass das No-Free-Lunch-Theorem zeigt, dass es keinen universell überlegenen Machine-Learning-Algorithmus gibt. Feedforward-Netze stellen insofern ein universelles System zur Repräsentation von Funktionen dar, als es für jede Funktion ein Feedforward-Netz gibt, das diese Funktion approximiert. Es gibt kein Universalverfahren zum Untersuchen einer Trainingsdatenmenge mit spezifischen Beispielen und zum Auswählen einer Funktion, die auf nicht in der Trainingsdatenmenge enthaltene Punkte generalisiert.

Gemäß dem Theorem der universellen Approximation gibt es ein Netz, das groß genug ist, um jeden gewünschten Genauigkeitsgrad zu erreichen, aber das Theorem macht keine Angabe dazu, wie groß dieses Netz ist. Barron (1993) stellt einige Regeln zur Größe von Netzen mit einer Schicht auf, die zur Approximation einer breiten Funktionsklasse benötigt werden. Leider wird im schlimmsten Fall eine exponentielle Anzahl verdeckter Einheiten benötigt (möglicherweise sogar mit einer verdeckten Einheit für jede Eingabekonfiguration, die unterschieden werden muss). Das lässt sich am binären Fall am leichtesten zeigen: Die Anzahl der möglichen binären Funktionen für die Vektoren $v \in \{0, 1\}^n$ beträgt 2^{2^n} , wobei für das Auswählen einer solchen Funktion 2^n Bits erforderlich sind, die wiederum im Allgemeinen $O(2^n)$ Freiheitsgrade benötigen.

Zusammenfassend reicht ein Feedforward-Netz mit einer Schicht aus, um jede beliebige Funktion darzustellen, aber möglicherweise ist diese Schicht für den praktischen Einsatz zu groß und kann am Erlernen und korrekten Generalisieren scheitern. In vielen Fällen führt der Einsatz tieferer Modelle zu einer Reduzierung der Einheiten, die zur Repräsentation der gewünschten Funktion erforderlich sind, sodass die Höhe des Generalisierungsfehlers verringert werden kann.

Diverse Funktionsfamilien lassen sich wirkungsvoll durch eine Architektur mit Tiefen oberhalb eines Wertes d approximieren, aber für Tiefen kleiner oder gleich d wird ein sehr viel größeres Modell benötigt. In vielen Fällen ist die Anzahl der verdeckten Einheiten, die das flache Modell benötigt, in n exponentiell. Solche Ergebnisse wurden erstmals für Modelle belegt, die den stetigen, differenzierbaren neuronalen Netzen für das Machine Learning nicht ähneln, doch mittlerweile wurde auch für diese der Nachweis erbracht. Die ersten Ergebnisse galten für Schaltkreise mit Logikgattern (*Håstad*,

1986). Spätere Arbeiten dehnten die Ergebnisse auf lineare Schwellwert-einheiten (engl. *threshold units*) mit nicht-negativen Gewichten (*Håstad und Goldmann*, 1991; *Hajnal et al.*, 1993) und anschließend auf Netze mit stetigwertigen Aktivierungen aus (*Maass*, 1992; *Maass et al.*, 1994). Viele moderne neuronale Netze setzen ReLUs ein. *Leshno et al.* (1993) haben gezeigt, dass die flachen Netze mit einer breiten Familie nichtpolynomialer Aktivierungsfunktionen – darunter ReLUs – Eigenschaften der universellen Approximation aufweisen, ohne dass diese Ergebnisse die Fragen der Tiefe oder Effizienz angehen; sie besagen lediglich, dass ein ausreichend breites Rectifier-Netz jede beliebige Funktion darstellen könnte. *Montufar et al.* (2014) haben gezeigt, dass mit einem tiefen Rectifier-Netz darstellbare Funktionen eine exponentielle Anzahl verdeckter Einheiten in einem flachen Netz (eine verdeckte Schicht) voraussetzen können. Genauer ausgedrückt haben sie gezeigt, dass stückweise lineare Netze (die sich mittels Nichtlinearitäten des Rectifiers oder mittels Maxout-Einheiten erzielen lassen) Funktionen mit einer Anzahl von Bereichen repräsentieren können, die in der Netztiefe exponentiell sind. Abbildung 6.5 zeigt, wie ein Netz mit Rektifizierung des Absolutbetrags Spiegelbilder der berechneten Funktion auf einer verdeckten Einheit bezüglich des Eingabewerts der verdeckten Einheit erzeugt. Jede verdeckte Einheit gibt an, wo der Eingaberaum gefaltet werden muss, um Spiegelreaktionen (auf beiden Seiten der Absolutbetrag-Nichtlinearität) zu erzeugen. Durch Zusammensetzen dieser Faltvorgänge erhalten wir eine exponentiell große Anzahl von stückweise linearen Bereichen, die sämtliche Arten von regelmäßigen (also sich wiederholenden) Mustern erfassen können.

Das Haupttheorem in *Montufar et al.* (2014) besagt, dass die Anzahl der durch ein tiefes Rectifier-Netz mit d Eingaben, einer Tiefe l und n Einheiten in jeder verdeckten Schicht erzeugten linearen Bereiche

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right) \quad (6.42)$$

beträgt, also in der Tiefe l exponentiell ist. Im Falle von Maxout-Netzen mit k Filtern pro Einheit beträgt die Anzahl der linearen Bereiche

$$O\left(k^{(l-1)+d}\right). \quad (6.43)$$

Natürlich gibt es keine Garantie dafür, dass die Art der Funktionen, die wir in Machine-Learning-Anwendungen (und insbesondere in der KI) erlernen möchten, eine solche Eigenschaft teilen.

Wir können auch aus statistischen Gründen ein tiefes Modell wählen. Jedes Mal wenn wir uns für einen bestimmten Machine-Learning-Algorithmus entscheiden, drücken wir implizit einige Annahmen bezüglich der Art

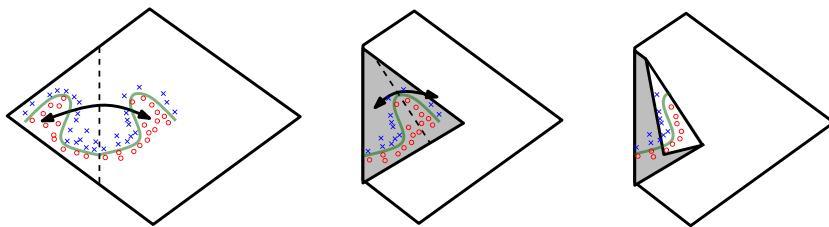


Abbildung 6.5: Eine intuitive geometrische Erklärung des exponentiellen Vorteils tieferer Rectifier-Netze (formal beschrieben in *Montufar et al.* (2014)). (Links) Eine Absolutbetragrektrifizierungseinheit weist für jedes Paar Spiegelpunkte in den Eingabewerten dieselbe Ausgabe auf. Die Spiegelsymmetriearchse ergibt sich aus der Hyperebene der Gewichte und der Verzerrung der Einheit. Eine auf dieser Einheit berechnete Funktion (die gebogene, graue Entscheidungsfläche) stellt ein Spiegelbild eines einfacheren Musters entlang der Symmetriearchse dar. (Mitte) Die Funktion lässt sich durch Falten des Raums um die Symmetriearchse ermitteln. (Rechts) Eine weitere Wiederholung des Musters lässt sich auf die erste falten (in einer nachgelagerten Einheit), um erneut eine Symmetrie zu erzielen (die nunmehr vier Mal mit zwei verdeckten Schichten wiederholt ist). (Abbildung mit freundlicher Genehmigung von *Montufar et al.* (2014) wiedergegeben)

der Funktion aus, die der Algorithmus erlernen soll. Die Wahl eines tieferen Modells codiert eine sehr allgemeine Überzeugung, dass die zu erlernende Funktion aus mehreren einfacheren Funktionen zusammengesetzt sein sollte. Das lässt sich vom Standpunkt des Representation Learnings so ausdrücken, dass wir glauben, dass das Lernproblem durch Ermitteln einer Reihe zugrunde liegender Faktoren der Variation gelöst werden kann, die wiederum durch noch simplere, zugrunde liegende Faktoren der Variation beschrieben werden können. Wir können den Einsatz einer tiefen Architektur auch so interpretieren, dass wir unsere Überzeugung ausdrücken, dass es sich bei der zu erlernenden Funktion um ein Computerprogramm handelt, das aus mehreren Schritten besteht, die jeweils die Ausgaben des vorherigen Schrittes nutzen. Diese Zwischenausgaben müssen keine Faktoren der Variation sein, sondern es kann sich um eine Art Zähler oder Pointer handeln, mit denen das Netz die interne Verarbeitung organisiert. Empirisch scheint eine größere Tiefe zu einer besseren Generalisierungsfähigkeit bei vielen Aufgaben zu führen (*Bengio et al.*, 2007; *Erhan et al.*, 2009; *Bengio*, 2009; *Mesnil et al.*, 2011; *Ciresan et al.*, 2012; *Krizhevsky et al.*, 2012; *Sermanet et al.*, 2013; *Farabet et al.*, 2013; *Couprie et al.*, 2013; *Kahou et al.*, 2013; *Goodfellow et al.*, 2014d; *Szegedy et al.*, 2014a). In Abbildung 6.6 und Abbildung 6.7 werden Beispiele für einige dieser empirischen Ergebnisse gezeigt. Diese Ergebnisse deuten an, dass der Einsatz tiefer Architekturen in der Tat eine

nützliche Annahme über den Funktionsraum ausdrückt, den ein Modell erlernt.

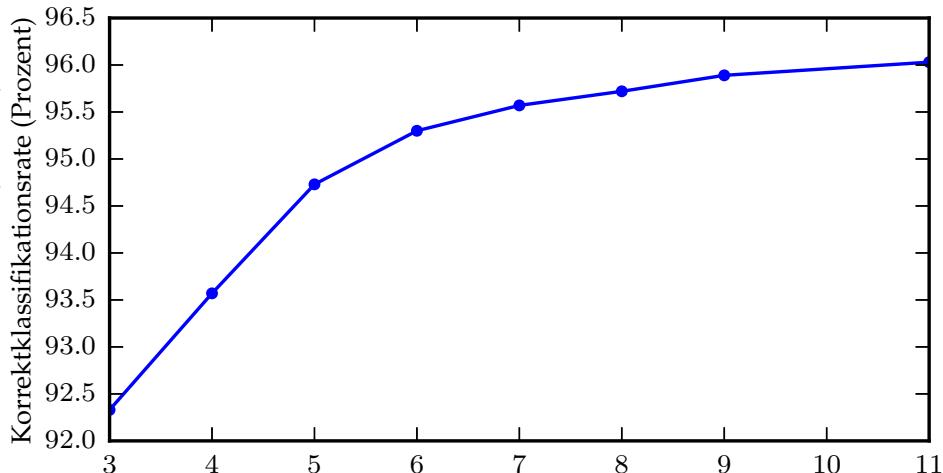


Abbildung 6.6: Auswirkung der Tiefe. Empirische Ergebnisse zeigen, dass tiefere Netze besser generalisieren, wenn sie zum Transkribieren mehrstelliger Zahlen aus Hausnummernfotos eingesetzt werden. Daten aus *Goodfellow et al.* (2014d). Die Korrektklassifikationsrate der Testdatenmenge nimmt mit steigender Tiefe kontinuierlich zu. Abbildung 6.7 zeigt ein Kontrollexperiment, das nachweist, dass andere Steigerungen der Modellgröße nicht dieselbe Wirkung haben.

6.4.2 Weitere Überlegungen zur Architektur

Bisher haben wir neuronale Netze als einfache Ketten bzw. Abfolgen von Schichten beschrieben. Dabei standen besonders die Tiefe des Netzes und die Breite jeder Schicht im Fokus. In der Praxis stellen sich neuronale Netze jedoch sehr viel vielfältiger dar.

Viele neuronale Netzarchitekturen wurden für ganz bestimmte Aufgaben entwickelt. Zum Beispiel werden spezialisierte Architekturen für Computer Vision als CNNs bezeichnet – wir gehen in Kapitel 9 noch darauf ein. Feedforward-Netze können für die Sequenzverarbeitung auch als RNNs generalisiert werden, für die eigene Überlegungen zur Architektur gelten – mehr dazu in Kapitel 10.

Obwohl es allgemein üblich ist, müssen die Schichten nicht unbedingt in einer Kette verbunden sein. Viele Architekturen nutzen eine Hauptkette und ergänzen diese um zusätzliche Merkmale, die als eine Art Umgehungsstraße zwischen der Schicht i und einer Schicht $i + 2$ oder höher dienen, zum Beispiel sogenannte Skip Connections. Diese Skip Connections machen den

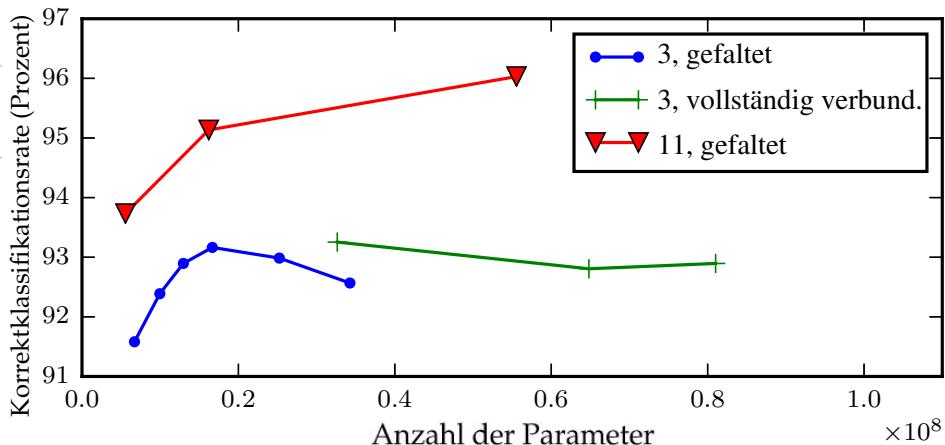


Abbildung 6.7: Auswirkung der Parameteranzahl. Tiefere Modelle neigen zu einer höheren Leistungsfähigkeit. Das liegt nicht nur an der Modellgröße. Dieses Experiment aus *Goodfellow et al.* (2014d) zeigt, dass eine höhere Anzahl der Parameter in den Schichten eines CNNs ohne gleichzeitige Erhöhung der Tiefe nicht annähernd so wirkungsvoll beim Steigern der Leistungsfähigkeit der Testdatenmenge ist, wie diese Abbildung zeigt. Die Legende gibt die Tiefe des Netzes der einzelnen Kurven an und zeigt außerdem, ob die Kurve eine Variation der Größe der gefalteten oder der vollständig verbundenen Schicht (engl. *fully connected layer*) darstellt. Es ist zu sehen, dass es bei flachen Modellen in diesem Kontext ab etwa 20 Millionen Parametern zur Überanpassung kommt, während tiefe Modelle auch bei mehr als 60 Millionen Parametern noch profitieren. Somit scheint die Verwendung eines tiefen Modells eine nützliche Bevorzugung über den Funktionsraum auszudrücken, den das Modell zu lernen vermag. Insbesondere deutet sich eine Überzeugung an, dass die Funktion aus vielen einfacheren Funktionen zusammengesetzt ist. Das könnte entweder zum Erlernen einer Repräsentation führen, die wiederum aus einfacheren Repräsentationen zusammengesetzt ist (z. B. Ecken, die anhand von Kanten definiert werden), oder zum Erlernen eines Programms mit voneinander sequenziell abhängigen Schritten (z. B. zunächst Aufspüren einer Objektmenge, in der dann eine Trennung in die einzelnen Objekte erfolgt, die in einem weiteren Schritt erkannt werden).

Verlauf des Gradienten von den Ausgabeschichten zu Schichten in der Nähe des Eingangs einfacher.

Eine weitere Grundsatzüberlegung beim Architekturdesign ist die Art der Verbindung zwischen zwei Schichten. In einer Standardschicht im neuronalen Netz, die durch eine lineare Transformation mit einer Matrix \mathbf{W} beschrieben wird, ist jede Eingabeeinheit mit jeder Ausgabeeinheit verbunden. Viele spezialisierte Netze in den kommenden Kapiteln weisen weniger Verbindungen auf, sodass jede Einheit der Eingabeschicht nur mit einer

kleinen Teilmenge der Einheiten in der Ausgabeschicht verbunden ist. Diese Verfahrensweisen zum Reduzieren der Anzahl der Verbindungen verringern die Anzahl der Parameter sowie die Menge der erforderlichen Berechnungen zur Auswertung des Netzes; allerdings sind sie häufig stark aufgabenabhängig. Zum Beispiel nutzen die in Kapitel 9 behandelten CNNs spezialisierte Muster von spärlichen Verbindungen (engl. *sparse connections*), die für Computer-Vision-Probleme sehr wirkungsvoll sind. Es ist schwierig, in diesem Kapitel ausführliche Tipps zur Architektur eines generischen neuronalen Netzes zu geben. In den folgenden Kapiteln entwickeln wir bestimmte Architekturstrategien, die sich für unterschiedliche Anwendungsbereiche als praktikabel erwiesen haben.

6.5 Backpropagation und andere Algorithmen zur Differentiation

Wenn wir mit einem Feedforward-Netz eine Eingabe \mathbf{x} zu einer Ausgabe $\hat{\mathbf{y}}$ verarbeiten, fließen die Daten vorwärts durch das Netz. Die Eingabe \mathbf{x} stellt die anfänglichen Daten dar, die dann an die verdeckten Einheiten der einzelnen Schichten übertragen (propagiert) werden, bis letztlich die Ausgabe $\hat{\mathbf{y}}$ vorliegt. Dieser Prozess wird als **Forward-Propagation** bezeichnet. Während des Trainierens kann die Forward-Propagation immer weiter durchgeführt werden, bis Skalarkosten in Höhe von $J(\boldsymbol{\theta})$ entstehen. Beim – häufig mit **Backprop** abgekürzten – **Backpropagation**-Algorithmus (Rumelhart et al., 1986a) dürfen Daten zum Berechnen des Gradienten auch von der Kostenseite rückwärts durch das Netz fließen.

Die Berechnung eines analytischen Ausdrucks für den Gradienten ist relativ einfach, aber die numerische Berechnung eines solchen Ausdrucks kann sehr aufwendig sein. Der Backpropagation-Algorithmus verwendet dafür eine einfache und günstige Möglichkeit.

Der Begriff »Backpropagation« wird häufig fälschlicherweise auf den gesamten Lernalgorithmus für mehrschichtige neuronale Netze angewandt. Tatsächlich bezieht er sich aber nur auf das Verfahren zur Gradientenberechnung; ein anderer Algorithmus, z. B. das stochastische Gradientenabstiegsverfahren, übernimmt dann das Lernen anhand dieses Gradienten. Auch wird die Backpropagation häufig lediglich auf mehrschichtige neuronale Netze bezogen, obwohl sie prinzipiell Ableitungen beliebiger Funktionen berechnen kann (für einige Funktionen besteht die korrekte Reaktion darin, anzugeben, dass die Ableitung der Funktion nicht definiert ist). Wir zeigen nun, wie der Gradient $\nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{y})$ für eine beliebige Funktion f bestimmt

wird, wobei \mathbf{x} eine Menge von Variablen ist, deren Ableitungen gesucht werden, und \mathbf{y} eine weitere Menge von Variablen, die als Eingangsdaten für die Funktion dienen, deren Ableitungen aber nicht benötigt werden. In Lernalgorithmen wird meist der Gradient der Kostenfunktion bezüglich der Parameter benötigt: $\nabla_{\theta} J(\theta)$. Viele Machine-Learning-Aufgaben berechnen auch andere Ableitungen, entweder im Rahmen des Lernprozesses oder zur Analyse des erlernten Modells. Für diese Aufgaben kann der Backpropagation-Algorithmus ebenfalls eingesetzt werden, denn er ist nicht auf die Gradientenberechnung der Kostenfunktion dieser Parameter eingeschränkt. Die Idee zur Berechnung von Ableitungen durch das Propagieren von Daten in einem Netz ist sehr allgemein und lässt sich auch für die Berechnung von Werten wie der Jacobi-Matrix einer Funktion f mit mehreren Ausgaben nutzen. Wir beschränken die vorliegende Beschreibung auf die gängigen Fälle, in denen f über eine Ausgabe verfügt.

6.5.1 Berechnungsgraphen

Bisher haben wir neuronale Netze mit einer recht informellen graphischen Darstellung behandelt. Für eine genauere Beschreibung des Backpropagation-Algorithmus ist eine exaktere Darstellung durch Berechnungsgraphen hilfreich.

Es gibt viele Möglichkeiten zur Formalisierung von Berechnungen in Form von Graphen. Wir verwenden einen Knoten im Graphen zur Darstellung einer Variable. Bei der Variable kann es sich um einen Skalar, einen Vektor, eine Matrix, einen Tensor oder eine beliebige andere Art von Variable handeln.

Zur Formalisierung unserer Graphen benötigen wir außerdem **Operationen**. Eine Operation ist eine einfache Funktion mit einer oder mehreren Variablen. Unsere graphische Darstellung wird durch eine Reihe zulässiger Operationen ergänzt. Funktionen, die komplexer als die verfügbaren Operationen sind, können durch Kombinieren mehrerer Operationen beschrieben werden.

Ohne Einschränkung der Allgemeinheit definieren wir eine Operation so, dass nur eine einzige Ausgabevariable zurückgegeben werden kann. Da die Ausgabevariable über mehrere Eingaben verfügen kann (z.B. einen Vektor), wird die Allgemeinheit nicht eingeschränkt. Softwareimplementierungen der Backpropagation unterstützen normalerweise Operationen mit mehreren Ausgaben, aber in unserer Beschreibung verzichten wir darauf, da dies zu vielen weiteren Details führen würde, die für das grundlegende Verständnis nicht von Bedeutung sind.

Wenn eine Variable y durch eine Operation mit einer Variable x berechnet wird, zeichnen wir eine gerichtete Kante von x zu y . Manchmal ergänzen wir den Ausgabeknoten um den Namen der Operation, manchmal lassen wir diesen Hinweis weg, weil die Operation sich aus dem Kontext ergibt.

Beispiele für Berechnungsgraphen finden Sie in Abbildung 6.8.

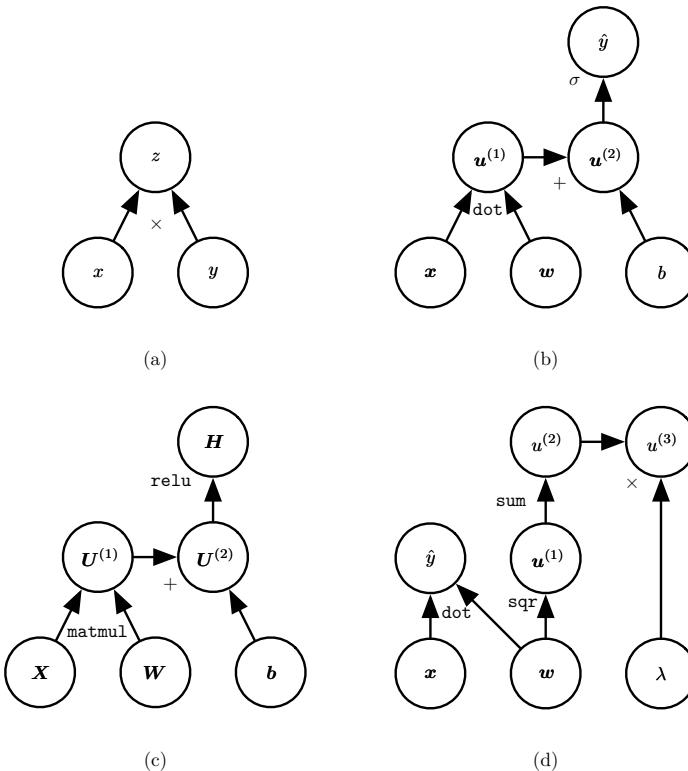


Abbildung 6.8: Beispiele für Berechnungsgraphen. (a) Der Graph, in dem mithilfe der Operation \times $z = xy$ berechnet wird. (b) Der Graph für die Vorhersage der logistische Regression $\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w} + b)$. Einige Zwischengrößen in der algebraischen Notation kommen ohne Namen aus, benötigen aber Namen im Graphen. Wir bezeichnen die i -te dieser Variablen einfach als $\mathbf{u}^{(i)}$. (c) Der Berechnungsgraph für den Ausdruck $\mathbf{H} = \max\{0, \mathbf{X}\mathbf{W} + \mathbf{b}\}$ zum Berechnen einer Entwurfsmatrix für die Aktivierungen der ReLU \mathbf{H} anhand der Entwurfsmatrix mit einem Mini-Batch der Eingaben \mathbf{X} . (d) Die Beispiele a–c haben maximal eine Operation pro Variable genutzt, aber es sind auch mehrere Operationen denkbar. Dies ist ein Berechnungsgraph, in dem mehrere Operationen auf die Gewichte \mathbf{w} eines linearen Regressionsmodells angewandt werden. Die Gewichte werden sowohl für die Vorhersage \hat{y} als auch der Weight-Decay-Strafterm $\lambda \sum_i w_i^2$ genutzt.

6.5.2 Kettenregel in der Analysis

Die Kettenregel in der Analysis (nicht zu verwechseln mit der Produktregel für Wahrscheinlichkeiten) dient zur Berechnung der Funktionsableitungen durch Kombinieren von Funktionen, deren Ableitungen bekannt sind. Der Backpropagation-Algorithmus berechnet die Kettenregel unter Berücksichtigung einer besonders effizienten Reihenfolge der Operationen.

Sei x eine reelle Zahl und seien f und g jeweils Funktionen, die eine reelle Zahl einer anderen reellen Zahl zuordnen. Angenommen, es gilt $y = g(x)$ und $z = f(g(x)) = f(y)$. Dann besagt die Kettenregel: dass

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}. \quad (6.44)$$

Wir können über den skalaren Fall hinaus generalisieren. Angenommen, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, g ordnet von \mathbb{R}^m nach \mathbb{R}^n zu und f von \mathbb{R}^n nach \mathbb{R} . Wenn $\mathbf{y} = g(\mathbf{x})$ und $z = f(\mathbf{y})$, dann ist

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \quad (6.45)$$

In Vektornotation entspricht dies

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^{\top} \nabla_{\mathbf{y}} z, \quad (6.46)$$

mit $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ als $n \times m$ -Jacobi-Matrix von g .

Wir sehen, dass der Gradient der Variable \mathbf{x} durch Multiplikation einer Jacobi-Matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ mit einem Gradienten $\nabla_{\mathbf{y}} z$ bestimmt werden kann. Der Backpropagation-Algorithmus führt diese Multiplikation für jede Operation im Graphen aus.

Normalerweise nutzen wir die Backpropagation für Tensoren mit beliebiger Dimensionalität, nicht einfach nur für Vektoren. Vom Konzept entspricht dies exakt der Backpropagation mit Vektoren. Der einzige Unterschied besteht darin, wie die Zahlen in einem Raster angeordnet werden, um einen Tensor zu bilden. Stellen Sie sich vor, dass jeder Tensor vor der Backpropagation zu einem Vektor abgeplattet wird. Anschließend wird ein vektorwertiger Gradient berechnet und schließlich der Gradient wieder zu einem Tensor geformt. In dieser neuen Sicht ist die Backpropagation weiterhin einfach eine Multiplikation von Jacobi-Matrizen mit Gradienten.

Um den Gradienten eines Wertes z bezüglich eines Tensors \mathbf{X} zu kennzeichnen, schreiben wir $\nabla_{\mathbf{X}} z$ – als wäre \mathbf{X} ein Vektor. Die Indizes in \mathbf{X} weisen

nun mehrere Koordinaten auf: So enthält der Index für einen 3-D-Tensor drei Koordinaten. Wir können durch Verwendung einer einzelnen Variable i zur Darstellung des kompletten Index-Tupels abstrahieren. Für alle möglichen Index-Tupels i ergibt $(\nabla_{\mathbf{X}} z)_i$ dann $\frac{\partial z}{\partial \mathbf{X}_i}$. Das funktioniert genau wie für alle möglichen ganzzahligen Indizes i in einem Vektor: $(\nabla_x z)_i$ ergibt $\frac{\partial z}{\partial x_i}$. Anhand dieser Notation können wir die Kettenregel für Tensoren schreiben. Wenn $\mathbf{Y} = g(\mathbf{X})$ und $z = f(\mathbf{Y})$ ist, dann ist

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}. \quad (6.47)$$

6.5.3 Rekursive Anwendung der Kettenregel, um Backpropagation zu erreichen

Anhand der Kettenregel ist es einfach, einen algebraischen Ausdruck für den Gradienten eines Skalars bezüglich eines beliebigen Knotens im Berechnungsgraphen für diesen Skalar zu notieren. Für die tatsächliche Berechnung des Ausdrucks mit einem Computer sind jedoch zusätzliche Überlegungen erforderlich.

Insbesondere werden viele Teilausdrücke möglicherweise mehrmals im Gesamtausdruck für den Gradienten wiederholt. Jedes Verfahren, das den Gradienten berechnet, muss entscheiden, ob diese Teilausdrücke gespeichert oder mehrfach berechnet werden. Abbildung 6.9 zeigt ein Beispiel für solche wiederholten Teilausdrücke. In manchen Fällen wäre die doppelte Berechnung desselben Teilausdrucks eine Verschwendug von Ressourcen. Für komplexe Graphen können exponentiell viele dieser überflüssigen Berechnungen vorkommen, sodass eine naive Implementierung der Kettenregel undurchführbar wird. In anderen Fällen ist die doppelte Berechnung desselben Teilausdrucks vielleicht eine legitime Möglichkeit, den Speicherbedarf auf Kosten der längeren Laufzeit zu reduzieren.

Wir beginnen mit einer Version des Backpropagation-Algorithmus, die eine direkte Berechnung des eigentlichen Gradienten vorschreibt (siehe Algorithmus 6.2 und Algorithmus 6.1 für die entsprechende Vorwärtsberechnung), und zwar in der Reihenfolge, in der sie tatsächlich erfolgt und gemäß der rekursiven Anwendung der Kettenregel. Diese Berechnungen können direkt durchgeführt werden. Alternativ ist die Beschreibung des Algorithmus als symbolische Spezifikation des Berechnungsgraphen zur Berechnung der Backpropagation einen Blick wert. Allerdings sind Manipulation und Konstruktion des symbolischen Graphen für die Gradientenberechnung bei diesem Ansatz nicht explizit. Ein solcher Ansatz wird mit Algorithmus 6.5

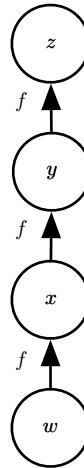


Abbildung 6.9: Ein Berechnungsgraph, der bei der Gradientenberechnung zu wiederholten Teilausdrücken führt. Sei $w \in \mathbb{R}$ die Eingabe für den Graphen. Wir wenden dieselbe Funktion $f : \mathbb{R} \rightarrow \mathbb{R}$ als Operation in jedem Schritt der Kette an: $x = f(w)$, $y = f(x)$, $z = f(y)$. Zum Berechnen von $\frac{\partial z}{\partial w}$ wenden wir Gleichung 6.44 an und erhalten:

$$\frac{\partial z}{\partial} \quad (6.48)$$

$$= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial} \quad (6.49)$$

$$= f'(y) f'(x) f'(w) \quad (6.50)$$

$$= f'(f(f(w))) f'(f(w)) f'(w). \quad (6.51)$$

Gleichung 6.50 empfiehlt eine Implementierung, in der wir die Werte für $f(w)$ nur einmal berechnen und dann in der Variable x ablegen. Diesen Ansatz nutzt auch der Backpropagation-Algorithmus. Eine andere Herangehensweise wird in Gleichung 6.51 empfohlen, wo der Teilausdruck $f(w)$ mehrfach auftritt. In der Alternative wird $f(w)$ jedes Mal neu berechnet. Wenn zum Vorhalten dieser Ausdrücke nur wenig Speicher benötigt wird, bietet der Backpropagation-Ansatz gemäß Gleichung 6.50 einen Vorteil, da die Laufzeit verkürzt wird. Allerdings ist Gleichung 6.51 ebenfalls eine gültige Implementierung der Kettenregel, die bei knappem Speicher ihre Berechtigung hat.

in Abschnitt 6.5.6 vorgestellt; dort generalisieren wir auch für Knoten, die beliebige Tensoren enthalten.

Betrachten Sie zunächst einen Berechnungsgraphen, der beschreibt, wie ein einzelner Skalar $u^{(n)}$ bestimmt wird (zum Beispiel der Verlust bei einem Trainingsbeispiel). Dieser Skalar ist die Größe, deren Gradienten wir ermitteln möchten – bezüglich der n_i Eingangsknoten $u^{(1)}$ bis $u^{(n_i)}$. Anders

Algorithmus 6.1 Ein Verfahren für die Berechnungen zur Zuordnung der n_i Eingangsdaten $u^{(1)}$ bis $u^{(n_i)}$ zu einer Ausgabe $u^{(n)}$. Hiermit wird ein Berechnungsgraph definiert, in dem jeder Knoten den numerischen Wert $u^{(i)}$ berechnet, indem eine Funktion $f^{(i)}$ auf die Menge der Argumente $\mathbb{A}^{(i)}$ angewandt wird, die aus den Werten der vorherigen Knoten $u^{(j)}$, $j < i$ besteht, mit $j \in Pa(u^{(i)})$. Die Eingabe des Berechnungsgraphen ist der Vektor \mathbf{x} ; er wird in den ersten n_i Knoten $u^{(1)}$ bis $u^{(n_i)}$ eingestellt. Die Ausgabe des Berechnungsgraphen wird aus dem letzten (Ausgabe-)Knoten $u^{(n)}$ gelesen.

```
for  $i = 1, \dots, n_i$  do  
   $u^{(i)} \leftarrow x_i$   
end for  
for  $i = n_i + 1, \dots, n$  do  
   $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$   
   $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$   
end for  
return  $u^{(n)}$ 
```

formuliert: Wir möchten $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ für alle $i \in \{1, 2, \dots, n_i\}$ berechnen. Beim Anwenden der Backpropagation zum Berechnen der Gradienten im Gradientenabstiegsverfahren über die Parameter gibt $u^{(n)}$ die Kosten für ein Beispiel oder einen Mini-Batch an, während $u^{(1)}$ bis $u^{(n_i)}$ den Modellparametern entsprechen.

Wir gehen davon aus, dass die Knoten des Graphen so angeordnet sind, dass wir ihre Ausgabe nacheinander berechnen können, und zwar beginnend mit $u^{(n_i+1)}$ hin zu $u^{(n)}$. Wie in Algorithmus 6.1 definiert, ist jeder Knoten $u^{(i)}$ mit einer Operation $f^{(i)}$ verknüpft und wird durch Berechnen der Funktion

$$u^{(i)} = f(\mathbb{A}^{(i)}) \quad (6.52)$$

bestimmt, wobei $\mathbb{A}^{(i)}$ die Menge aller Knoten darstellt, die Eltern von $u^{(i)}$ sind.

Der Algorithmus schreibt die Forward-Propagation vor, die sich in einem Graphen \mathcal{G} vornehmen lässt. Für die Backpropagation können wir einen Berechnungsgraphen erstellen, der von \mathcal{G} abhängig ist und eine weitere Reihe von Knoten ergänzt. Diese bilden einen Teilgraphen \mathcal{B} mit einem Knoten für jeden Knoten von \mathcal{G} . Die Berechnung in \mathcal{B} läuft in exakt umgekehrter Reihenfolge zur Berechnung in \mathcal{G} ab, und jeder Knoten von \mathcal{B} berechnet

die Ableitung $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ des Knotens $u^{(i)}$ im vorwärts gerichteten Graphen. Das geschieht mithilfe der Kettenregel für die Skalarausgabe $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \quad (6.53)$$

gemäß Algorithmus 6.2. Der Teilgraph \mathcal{B} enthält exakt eine Kante für jede Kante zwischen den Knoten $u^{(j)}$ und $u^{(i)}$ von \mathcal{G} . Die Kante von $u^{(j)}$ zu $u^{(i)}$ ist mit der Berechnung von $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ verknüpft. Außerdem wird das Skalarprodukt für jeden Knoten gebildet, und zwar zwischen dem bereits für die Knoten $u^{(i)}$ berechneten Gradienten, die Kinder von $u^{(j)}$ sind, und dem Vektor, der die partiellen Ableitungen $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ für eben diese Kinderknoten $u^{(i)}$ enthält. Zusammenfassend skaliert der Berechnungsaufwand für die Backpropagation linear mit der Anzahl der Kanten in \mathcal{G} , wobei die Berechnung für jede Kante

Algorithmus 6.2 Vereinfachte Version des Backpropagation-Algorithmus zum Berechnen der Ableitungen von $u^{(n)}$ bezüglich der Variablen im Graphen. Dieses Beispiel soll ihr Verständnis erweitern, indem es einen vereinfachten Fall zeigt, in dem sämtliche Variablen Skalare sind; wir möchten hier die Ableitungen bezüglich $u^{(1)}, \dots, u^{(n_i)}$ berechnen. Diese vereinfachte Version berechnet die Ableitungen aller Knoten im Graphen. Der Berechnungsaufwand für den Algorithmus ist proportional zur Anzahl der Kanten im Graphen, sofern die partielle Ableitung für jede Kante gleich lang dauert. Er weist dieselbe Ordnung wie die Anzahl der Berechnungen für die Forward-Propagation auf. Jedes Auftreten von $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ ist eine Funktion der Eltern $u^{(j)}$ von $u^{(i)}$, sodass die Knoten des Vorwärtsgraphen mit den für den Backpropagation-Graphen hinzugefügten verknüpft werden.

Ausführen der Forward-Propagation (in diesem Beispiel Algorithmus 6.1) zum Ermitteln der Aktivierungen für das Netz.

Initialisieren von `grad_table`, einer Datenstruktur zum Speichern der berechneten Ableitungen. Der Eintrag `grad_table[u(i)]` enthält das Ergebnis für $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

```
grad_table[u(n)] ← 1
for j = n - 1 herunter bis 1 do
    Die nächste Zeile berechnet  $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$  anhand gespeicherter Werte:
    grad_table[u(j)] ←  $\sum_{i:j \in Pa(u^{(i)})} \text{grad\_table}[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$ 
end for
return {grad_table[u(i)] | i = 1, ..., ni}
```

der Berechnung einer partiellen Ableitung (eines Knotens bezüglich eines seiner Elternteile) sowie die Durchführung einer Multiplikation und einer Addition entspricht. Unten generalisieren wir diese Analyse für tensorwertige Knoten – eine andere Möglichkeit zum Gruppieren mehrerer Skalarwerte in einem Knoten und zum Aktivieren von effizienteren Implementierungen.

Der Backpropagation-Algorithmus soll die Anzahl der gemeinsamen Teilausdrücke ungeachtet des Speicherplatzes reduzieren. So entspricht seine Ordnung pro Knoten im Graphen einem Jacobi-Produkt. Das lässt sich an der Tatsache zeigen, dass die Backpropagation (Algorithmus 6.2) jede Kante zwischen den Knoten $u^{(j)}$ und $u^{(i)}$ des Graphen exakt einmal aufsucht, um die zugehörige partielle Ableitung $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ zu ermitteln. Somit vermeidet die Backpropagation die exponentielle Explosion in wiederholten Teilausdrücken. Andere Algorithmen können mehrere Teilausdrücke ebenfalls vermeiden, indem sie Vereinfachungen am Berechnungsgraphen durchführen. Oder sie sparen Speicherplatz durch die erneute Berechnung anstelle des Vorhaltens einzelner Teilausdrücke. Wir kommen nach einer Beschreibung des Backpropagation-Algorithmus selbst darauf zurück.

6.5.4 Berechnen der Backpropagation im vollständig verbundenen mehrschichtigen Perzepron

Um die obige Definition der Berechnung der Backpropagation besser zu verstehen, betrachten wir den spezifischen Graphen eines vollständig verbundenen mehrschichtigen MLPs.

Algorithmus 6.3 zeigt zunächst die Forward-Propagation, die Parameter dem überwachten Verlust $L(\hat{\mathbf{y}}, \mathbf{y})$ eines einzelnen (Eingabe, Zielwert) Trainingsbeispiels (\mathbf{x}, \mathbf{y}) zuordnet. Dabei ist $\hat{\mathbf{y}}$ die Ausgabe des neuronalen Netzes, wenn \mathbf{x} in der Eingabe angegeben wird.

Algorithmus 6.4 zeigt als Nächstes die zugehörige Berechnung zum Anwenden des Backpropagation-Algorithmus auf diesen Graphen.

Algorithmen 6.3 und 6.4 sind einfache und leicht verständliche Beispiele. Allerdings sind sie für ein spezifisches Problem spezialisiert.

Moderne Softwareimplementierungen beruhen auf der generalisierten Form der Backpropagation, die nachstehend in Abschnitt 6.5.6 beschrieben wird, da sie für beliebige Berechnungsgraphen angepasst werden kann, indem eine Datenstruktur explizit zur Darstellung symbolischer Berechnungen manipuliert wird.

Algorithmus 6.3 Forward-Propagation in einem typischen tiefen neuronalen Netz und die Berechnung der Kostenfunktion. Der Verlust $L(\hat{\mathbf{y}}, \mathbf{y})$ ist abhängig von der Ausgabe $\hat{\mathbf{y}}$ und dem Zielwert \mathbf{y} (Abschnitt 6.2.1.1 gibt Beispiele für Verlustfunktionen). Um die Gesamtkosten J zu ermitteln, kann der Verlust zu einem Regularisierer $\Omega(\theta)$ addiert werden, wobei θ alle Parameter umfasst (Gewichte und Verzerrungen). Algorithmus 6.4 zeigt, wie die Gradienten von J für die Parameter \mathbf{W} und \mathbf{b} berechnet werden. Aus Gründen der Einfachheit verwendet dieses Beispiel nur ein Eingabebeispiel \mathbf{x} . In praktischen Anwendungen sollte ein Mini-Batch zum Einsatz kommen. Abschnitt 6.5.7 enthält eine realistischere Darstellung.

Require: Netztiefe, l
Require: $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$, die Gewichtungsmatrizen des Modells
Require: $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$, die Verzerrungsparameter des Modells
Require: \mathbf{x} , die Eingangsdaten für den Prozess
Require: \mathbf{y} , der Zielwert der Ausgabe
$$\mathbf{h}^{(0)} = \mathbf{x}$$

for $k = 1, \dots, l$ **do**
 $\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}$
 $\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$
end for
 $\hat{\mathbf{y}} = \mathbf{h}^{(l)}$
 $J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta)$

6.5.5 Symbol-to-Symbol-Ableitungen

Algebraische Ausdrücke und Berechnungsgraphen arbeiten beide mit **Symbolen** oder Variablen, denen keine festen Werte zugeordnet sind. Diese algebraischen und graphenbasierten Darstellungen werden **symbolische Darstellungen** genannt. Beim eigentlichen Verwenden oder Trainieren eines neuronalen Netzes müssen wir den Symbolen bestimmte Werte zuweisen. Wir ersetzen eine symbolische Eingabe im Netz \mathbf{x} durch einen bestimmten **numerischen** Wert, beispielsweise $[1, 2, 3, 765, -1, 8]^\top$.

Einige Ansätze für die Backpropagation nehmen einen Berechnungsgraphen und eine Menge numerischer Werte für die Eingaben des Graphen, um dann eine Menge numerischer Werte zur Beschreibung des Gradienten an diesen Eingabewerten zurückzugeben. Wir nennen diesen Ansatz **Symbol-to-Number-Differentiation**. Er wird in Bibliotheken wie Torch (*Collobert et al.*, 2011b) und Caffe (*Jia*, 2013) eingesetzt.

Algorithmus 6.4 Rückwärtsberechnung für das tiefe neuronale Netz aus Algorithmus 6.3, in der neben der Eingabe \mathbf{x} auch ein Zielwert \mathbf{y} verwendet wird. Diese Berechnung ergibt die Gradienten der Aktivierung $\mathbf{a}^{(k)}$ für jede Schicht k , ausgehend von der Ausgabeschicht rückwärts zur ersten verdeckten Schicht. Von diesen Gradienten, die als Angabe zur Sollveränderung der einzelnen Schichtausgaben mit dem Ziel einer Fehlerreduzierung betrachtet werden können, lässt sich der Gradient der Parameter für jede Schicht bestimmen. Die Gradienten der Gewichte und Verzerrungen können direkt als Teil einer stochastischen Gradientenaktualisierung (Aktualisierung unverzüglich nach dem Berechnen der Gradienten) oder in Verbindung mit anderen Optimierungsverfahren auf Gradientenbasis genutzt werden.

Nach der Vorwärtsberechnung den Gradienten der Ausgabeschicht berechnen:

$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$
for $k = l, l - 1, \dots, 1$ **do**

Den Gradienten in der Ausgabe der Schicht in einen Gradienten in der Aktivierung vor Nichtlinearität umwandeln (elementweise Multiplikation, sofern f elementweise ist):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot f'(\mathbf{a}^{(k)})$$

Gradienten für Gewichte und Verzerrungen berechnen (einschließlich des Regularisierungsterms, sofern erforderlich):

$$\begin{aligned}\nabla_{\mathbf{b}^{(k)}} J &= \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\theta) \\ \nabla_{\mathbf{W}^{(k)}} J &= \mathbf{g} \mathbf{h}^{(k-1)\top} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\theta)\end{aligned}$$

Die Gradienten bezüglich der Aktivierungen der nächstniedrigeren verdeckten Schicht propagieren:

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)\top} \mathbf{g}$$

end for

Eine andere Herangehensweise besteht darin, einem Berechnungsgraphen zusätzliche Knoten hinzuzufügen, die eine symbolische Darstellung der gewünschten Ableitungen enthalten. Dieser Ansatz wird in Theano (*Bergstra et al., 2010; Bastien et al., 2012*) und TensorFlow (*Abadi et al., 2015*) genutzt. Abbildung 6.10 zeigt ein Beispiel für die Funktionsweise. Der wesentliche Vorteil hierbei ist, dass die Ableitungen in derselben Darstellungsform wie der ursprüngliche Ausdruck abgebildet werden. Da die Ableitungen lediglich einen weiteren Berechnungsgraphen darstellen, kann die Backpropagation erneut durchgeführt werden, um durch Differenzieren höhere Ableitungen aus den Ableitungen zu erhalten. (Die Berechnung von Ableitungen höherer Ordnung wird in Abschnitt 6.5.10 behandelt.)

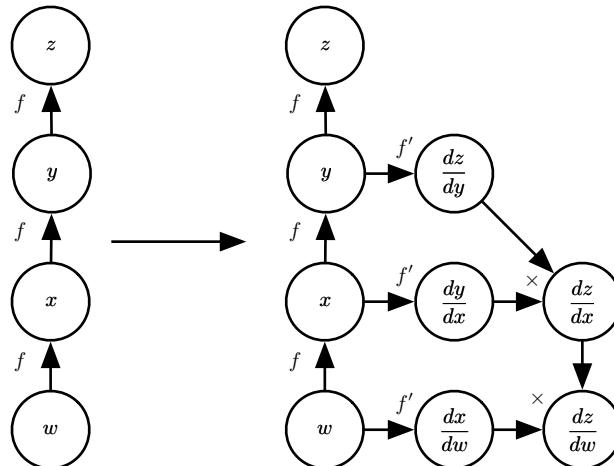


Abbildung 6.10: Ein Beispiel für den Symbol-to-Symbol-Ansatz beim Berechnen von Ableitungen. Dabei muss der Backpropagation-Algorithmus überhaupt nicht auf tatsächliche feste Zahlenwerte zugreifen. Stattdessen werden Knoten zu einem Berechnungsgraphen hinzugefügt, die die Berechnung dieser Ableitungen beschreiben. Ein generischer Algorithmus zur Bewertung des Graphen kann später die Ableitungen für beliebige Zahlenwerte berechnen. (*Links*) In diesem Beispiel beginnen wir mit einem Graphen für $z = f(f(f(w)))$. (*Rechts*) Der Backpropagation-Algorithmus wird mit der Vorgabe durchgeführt, den Graphen für einen Ausdruck zu konstruieren, der $\frac{dz}{dw}$ entspricht. In diesem Beispiel geben wir nicht an, wie der Backpropagation-Algorithmus funktioniert. Es soll lediglich das gewünschte Ergebnis verdeutlichen: einen Berechnungsgraphen mit einer symbolischen Darstellung der Ableitung.

Wir nutzen den zuletzt genannten Ansatz und beschreiben den Backpropagation-Algorithmus durch Konstruieren eines Berechnungsgraphen für die Ableitungen. Jede Teilmenge des Graphen kann später anhand bestimmter Zahlenwerte berechnet werden. So vermeiden wir genaue Angaben zum Zeitpunkt der einzelnen Berechnungen. Stattdessen dient ein generischer Algorithmus zur Graphenbewertung zur Berechnung der einzelnen Knoten, sobald die Werte ihrer Eltern vorliegen.

Die Beschreibung des Symbol-to-Symbol-Ansatzes subsumiert den Symbol-to-Number-Ansatz. Der Symbol-to-Number-Ansatz führt exakt die Berechnungen durch, die in dem Graphen angegeben sind, der mittels Symbol-to-Symbol-Ansatz erzeugt wurde. Der Hauptunterschied besteht darin, dass der Symbol-to-Number-Ansatz den Graphen nicht freilegt.

6.5.6 Allgemeine Backpropagation

Der Backpropagation-Algorithmus ist sehr einfach gehalten. Zum Berechnen des Gradienten eines Skalars z bezüglich einer seiner Vorgänger \mathbf{x} im Graphen beginnen wir mit der Feststellung, dass sich der Gradient bezüglich z aus $\frac{dz}{dz} = 1$ ergibt. Wir können jetzt den Gradienten für alle Eltern von z im Graphen berechnen, indem wir den aktuellen Gradienten mit der Jacobi-Matrix der Operation multiplizieren, durch die z entstanden ist. Durch fortgesetztes Multiplizieren mit den Jacobi-Matrizen rückwärts durch den Graphen erreichen wir irgendwann \mathbf{x} . Für jeden Knoten, der bei dieser Rückwärtsreise von z über zwei oder mehr Pfade erreicht werden kann, addieren wir einfach die Gradienten der unterschiedlichen Pfade an diesem Knoten.

Formal ausgedrückt: Jeder Knoten im Graphen \mathcal{G} entspricht einer Variable. Für die maximale Allgemeinheit beschreiben wir diese Variable als einen Tensor \mathbf{V} . Tensoren können generell beliebig viele Dimensionen aufweisen. Sie subsumieren Skalare, Vektoren und Matrizen.

Wir gehen davon aus, dass jede Variable \mathbf{V} mit den folgenden Subroutinen verknüpft ist:

- **get_operation(\mathbf{V})**: Dies ist die Operation zum Berechnen von \mathbf{V} , dargestellt durch die Kanten, die im Berechnungsgraphen in \mathbf{V} enden. Zum Beispiel könnte es eine Klasse in Python oder C++ geben, die die Matrizenmultiplikation und die Funktion `get_operation` darstellt. Angenommen, wir haben eine Variable, die durch Matrizenmultiplikation entstanden ist: $\mathbf{C} = \mathbf{AB}$. Dann gibt `get_operation(\mathbf{V})` einen Pointer auf eine Instanz der zugehörigen C++-Klasse zurück.
- **get_consumers(\mathbf{V}, \mathcal{G})**: Dies gibt die Liste der Variablen aus, die im Berechnungsgraphen \mathcal{G} Kinder von \mathbf{V} sind.
- **get_inputs(\mathbf{V}, \mathcal{G})**: Dies gibt die Liste der Variablen aus, die im Berechnungsgraphen \mathcal{G} Eltern von \mathbf{V} sind.

Jede Operation op ist auch mit einer Operation bprop verknüpft. Diese Operation bprop kann das Produkt aus Jacobi-Matrix und Vektor berechnen (vgl. Gleichung 6.47). So erzielt der Backpropagation-Algorithmus eine hohe Allgemeinheit. Jede Operation muss wissen, wie die Backpropagation entlang der Kanten des Graphen erfolgt, an der sie beteiligt ist. Wir können zum Beispiel eine Matrizenmultiplikation zum Erzeugen einer Variable $\mathbf{C} = \mathbf{AB}$ verwenden. Angenommen, der Gradient eines Skalars z

bezüglich \mathbf{C} ergibt sich aus \mathbf{G} . Die Matrizenmultiplikation muss zwei Backpropagation-Regeln definieren – eine für jedes Eingabeargument. Wenn wir die `bprop`-Methode aufrufen, um den Gradienten bezüglich \mathbf{A} für einen Gradienten der Ausgabe von \mathbf{G} abzufragen, dann muss die `bprop`-Methode der Matrizenmultiplikation angeben, dass der Gradient bezüglich \mathbf{A} sich aus $\mathbf{G}\mathbf{B}^\top$ ergibt. Ebenso muss, wenn wir die `bprop`-Methode für den Gradienten bezüglich \mathbf{B} aufrufen, die Matrizenoperation die `bprop`-Methode eigenverantwortlich implementieren und angeben, dass sich der gesuchte Gradient aus $\mathbf{A}^\top\mathbf{G}$ ergibt. Der Backpropagation-Algorithmus selbst muss keine Differentiationsregeln kennen. Er muss lediglich die `bprop`-Regeln der einzelnen Operationen mit den korrekten Argumenten aufrufen. Formal muss `op.bprop(inputs, X, G)`

$$\sum_i (\nabla_{\mathbf{x}} \text{op.f(inputs)}_i) G_i \quad (6.54)$$

zurückgeben; dabei handelt es sich lediglich um eine Implementierung der Kettenregel aus Gleichung 6.47.

Hier ist `inputs` eine Liste für die Operation. `op.f` ist die mathematische Funktion, die die Operation implementiert. \mathbf{X} ist die Eingabe, deren Gradienten wir ermitteln möchten, und \mathbf{G} ist der Gradient auf der Ausgabe der Operation.

Die `op.bprop`-Methode sollte stets annehmen, dass alle Eingangsdaten sich voneinander unterscheiden – auch wenn dies nicht der Fall ist. Wenn der Operator `mul` zum Beispiel zwei Instanzen von x durchläuft, um x^2 zu berechnen, muss die `op.bprop`-Methode noch immer x als Ableitung bezüglich beider Eingaben zurückgeben. Der Backpropagation-Algorithmus ergänzt diese beiden Argumente gemeinsam später, um $2x$, also die korrekte totale Ableitung von x , zu ermitteln.

Softwareimplementierungen der Backpropagation bieten normalerweise sowohl die Operationen als auch deren `bprop`-Methoden an, sodass Anwender von Deep-Learning-Softwarebibliotheken die Backpropagation direkt in Graphen nutzen können, die mit gängigen Operationen wie Matrizenmultiplikation, Exponenten, Logarithmen usw. erzeugt wurden. Softwareentwickler, die eine neue Implementierung der Backpropagation erstellen, oder fortgeschrittene Anwender, die eine eigene Operation in einer bestehenden Bibliothek ergänzen möchten, müssen im Normalfall die `op.bprop`-Methode für neue Operationen manuell ableiten.

Der Backpropagation-Algorithmus ist in Algorithmus 6.5 formal beschrieben.

Algorithmus 6.5 Das äußere Gerüst des Backpropagation-Algorithmus. Dieser Teil ist für die Einrichtung und die Aufräumarbeiten zuständig. Die eigentliche Arbeit erfolgt größtenteils in der Subroutine `build_grad` von Algorithmus 6.6

Require: \mathbb{T} , die Zielmenge der Variablen, deren Gradienten berechnet werden müssen

Require: \mathcal{G} , der Berechnungsgraph

Require: z , die zu differenzierende Variable

Sei \mathcal{G}' eine bereinigte Form von \mathcal{G} , die nur Knoten enthält, die Vorgänger von z und Nachfahren der Knoten in \mathbb{T} sind.

Initialisieren von `grad_table`, einer Datenstruktur, die Tensoren mit ihren Gradienten verknüpft

`grad_table`[z] $\leftarrow 1$

for \mathbf{V} in \mathbb{T} **do**

`build_grad`($\mathbf{V}, \mathcal{G}, \mathcal{G}', \text{grad_table}$)

end for

Ausgeben von `grad_table` eingeschränkt auf \mathbb{T}

In Abschnitt 6.5.2 haben wir erklärt, dass die Backpropagation entwickelt wurde, um die mehrfache Berechnung desselben Teilausdrucks in der Kettenregel zu vermeiden. Der naive Algorithmus könnte infolge solch wiederholter Teilausdrücke eine exponentielle Laufzeit beanspruchen. Nachdem wir nun den Backpropagation-Algorithmus spezifiziert haben, können wir uns mit dem Berechnungsaufwand befassen. Wenn wir davon ausgehen, dass zum Berechnen jeder Operation in etwa derselbe Aufwand anfällt, können wir den Berechnungsaufwand anhand der Anzahl der ausgeführten Operationen untersuchen. Bedenken Sie, dass in diesem Zusammenhang eine Operation die grundlegende Einheit unseres Berechnungsgraphen ist. Jede dieser Einheiten kann durchaus aus mehreren arithmetischen Operationen bestehen – zum Beispiel könnte in dem Graphen die Matrizenmultiplikation als eine Operation gelten. Die Gradientenberechnung in einem Graphen mit n Knoten führt niemals mehr als $O(n^2)$ Operationen aus und speichert niemals mehr Ausgaben als für $O(n^2)$ Operationen. Auch hier zählen wir die Operationen im Berechnungsgraphen, nicht die einzelnen Operationen auf der zugrunde liegenden Hardware. Tatsächlich kann die Laufzeit für jede einzelne Operation stark schwanken. So kann die Multiplikation von zwei Matrizen, die jeweils mehrere Millionen Einträge enthalten, einer einzelnen Operation im Graphen entsprechen. Sie stellen fest, dass die Gradientenberechnung maximal $O(n^2)$ Operationen erfordert, da während der Forward-Propagation

Algorithmus 6.6 Die innere Schleife, also die Subroutine `build_grad(V, G, G', grad_table)` des Backpropagation-Algorithmus, die durch den in Algorithmus 6.5 definierten Backpropagation-Algorithmus aufgerufen wird.

Require: V , die Variable, deren Gradient zu G und `grad_table` addiert werden soll

Require: G , der zu ändernde Graph

Require: G' , die Einschränkung von G auf Knoten, die zum Gradienten beitragen

Require: `grad_table`, eine Datenstruktur, die Knoten ihren Gradienten zuordnet

if V enthalten ist in `grad_table` **then**

Ausgeben von `grad_table[V]`

end if

$i \leftarrow 1$

for C in `get_consumers(V, G')` **do**

$op \leftarrow get_operation(C)$

$D \leftarrow build_grad(C, G, G', grad_table)$

$G^{(i)} \leftarrow op.bprop(get_inputs(C, G'), V, D)$

$i \leftarrow i + 1$

end for

$G \leftarrow \sum_i G^{(i)}$

`grad_table[V] = G`

Einfügen von G und der Operationen, aus denen G erzeugt wird, in G

Ausgeben von G

im schlimmsten Fall alle n Knoten des ursprünglichen Graphen bearbeitet werden (je nach gesuchtem Wert muss möglicherweise nicht der gesamte Graph verarbeitet werden). Der Backpropagation-Algorithmus fügt für jede Kante im ursprünglichen Graphen ein Produkt aus einer Jacobi-Matrix und einem Vektor hinzu, das sich durch $O(1)$ Knoten ausdrücken lässt. Da der Berechnungsgraph ein gerichteter, azyklischer Graph ist, weist dieser maximal $O(n^2)$ Kanten auf. Für die in der Praxis üblicherweise eingesetzten Graphen stellt sich die Lage noch besser dar. Die meisten Kostenfunktionen für neuronale Netze ähneln grob einer Kette, sodass die Backpropagation einen (Kosten-)Aufwand von $O(n)$ aufweist. Das ist sehr viel besser als beim naiven Ansatz, bei dem möglicherweise exponentiell mehr Knoten verarbeitet werden müssen. Dieser potenziell exponentielle Aufwand lässt sich durch

Erweitern und Neufassen der rekursiven Kettenregel (Gleichung 6.53) in nicht rekursiver Form zeigen:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{Pfad } (u^{(\pi_1)}, u^{(\pi_2)}, \dots, u^{(\pi_t)}), \\ \text{ von } \pi_1=j \text{ bis } \pi_t=n}} \prod_{k=2}^t \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \quad (6.55)$$

Da die Anzahl der Pfade zwischen den Knoten j und n exponentiell in der Pfadlänge wachsen kann, kann auch die Anzahl der Ausdrücke in obiger Summe (also die Anzahl dieser Pfade) exponentiell mit der Tiefe des Forward-Propagation-Graphen wachsen. Dieser hohe Aufwand entstünde, da dieselbe Berechnung für $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ häufig wiederholt würde. Um eine solche Neuberechnung zu vermeiden, stellen wir uns die Backpropagation als einen Algorithmus zum Ausfüllen einer Tabelle vor, der sich das Speichern von Zwischenergebnissen $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ zunutze macht. Jedem Knoten im Graphen ist eine Tabellenzelle zum Speichern des Gradienten für diesen Knoten zugewiesen. Durch geordnetes Eintragen der Zellwerte vermeidet die Backpropagation das Wiederholen vieler häufiger Teilausdrücke. Diese Vorgehensweise wird manchmal als **dynamische Programmierung** bezeichnet.

6.5.7 Beispiel: Backpropagation für ein MLP-Training

Als Beispiel verfolgen wir den Ablauf des Backpropagation-Algorithmus für das Trainieren eines mehrschichtigen Perzeptrons (MLP).

Wir entwickeln hier ein sehr einfaches mehrschichtiges Perzepron mit nur einer verdeckten Schicht. Für das Training des Modells nutzen wir das stochastische Gradientenabstiegsverfahren anhand eines Mini-Batches. Der Backpropagation-Algorithmus dient zum Berechnen des Gradienten des Aufwands für einen Mini-Batch. Insbesondere setzen wir einen Mini-Batch mit Beispielen aus der Trainingsdatenmenge ein, die als Entwurfsmatrix \mathbf{X} und Vektor der zugehörigen Klassen-Label \mathbf{y} formatiert ist. Das Netz berechnet eine Schicht mit verdeckten Merkmalen $\mathbf{H} = \max\{0, \mathbf{X}\mathbf{W}^{(1)}\}$. Um die Darstellung zu vereinfachen, verzichten wir in diesem Modell auf Verzerrungen. Wir gehen davon aus, dass unsere graphische Darstellung eine `relu`-Operation enthält, die $\max\{0, \mathbf{Z}\}$ elementweise berechnen kann. Die Vorhersagen der nicht normalisierten Log-Wahrscheinlichkeiten über Klassen ergeben sich somit aus $\mathbf{HW}^{(2)}$. Wir gehen davon aus, dass unsere graphische Darstellung eine `cross_entropy`-Operation enthält, die zum Berechnen der Kreuzentropie zwischen den Zielwerten \mathbf{y} und der durch diese nicht normalisierten Log-Wahrscheinlichkeiten definierten Wahrscheinlichkeitsverteilung dient. Die resultierende Kreuzentropie definiert den Aufwand J_{MLE} . Durch

Minimieren der Kreuzentropie erfolgt die Maximum-Likelihood-Schätzung des Klassifikators. Allerdings führen wir einen Regularisierungsterm ein, um das Beispiel realistischer zu gestalten. Der Gesamtaufwand

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} (W_{i,j}^{(1)})^2 + \sum_{i,j} (W_{i,j}^{(2)})^2 \right) \quad (6.56)$$

besteht aus der Kreuzentropie und einem Weight Decay mit dem Koeffizienten λ . Der Berechnungsgraph ist in Abbildung 6.11 dargestellt.

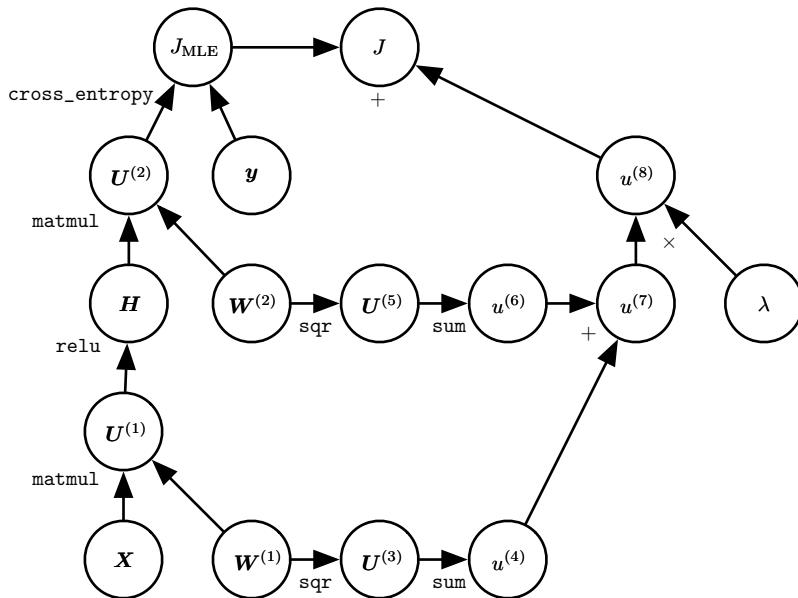


Abbildung 6.11: Der Berechnungsgraph zum Berechnen des Trainingsaufwands für unser Beispiel eines einschichtigen MLPs anhand des Kreuzentropieverlusts und Weight Decay

Der Berechnungsgraph für den Gradienten des Beispiels ist so groß, dass ein Aufzeichnen oder Lesen sehr mühselig wäre. Das zeigt einen der Vorteile des Backpropagation-Algorithmus: Er kann automatisch Gradienten erzeugen, die für einen Softwareentwickler zwar leicht umzusetzen, aber nur sehr mühsam abzuleiten sind.

Wir können das Verhalten des Backpropagation-Algorithmus betrachten, wenn wir uns den Forward-Propagation-Graphen aus Abbildung 6.11 ansehen. Für das Training möchten wir sowohl $\nabla_{W^{(1)}} J$ als auch $\nabla_{W^{(2)}} J$ ermitteln. Es gibt zwei unterschiedliche Pfade, die von J zurück zu den Gewichten führen: Der eine verläuft durch den Aufwand für die Kreuzentropie,

der andere durch den Aufwand des Weight Decays. Der Aufwand für den Weight Decay ist relativ einfach; er trägt stets $2\lambda \mathbf{W}^{(i)}$ zum Gradienten auf $\mathbf{W}^{(i)}$ bei.

Der andere Pfad, über den Aufwand für die Kreuzentropie, ist ein wenig komplexer. Sei \mathbf{G} der Gradient der nicht normalisierten Log-Wahrscheinlichkeiten $\mathbf{U}^{(2)}$ aus der `cross_entropy`-Operation. Der Backpropagation-Algorithmus muss nun zwei unterschiedliche Zweige erkunden. Auf dem kürzeren Zweig wird $\mathbf{H}^\top \mathbf{G}$ zum Gradienten auf $\mathbf{W}^{(2)}$ addiert; dabei wird die Backpropagation-Regel als zweites Argument der Matrizenmultiplikation genutzt. Der andere Zweig entspricht der längeren Kette, die sich tiefer durch das Netz erstreckt. Zunächst berechnet der Backpropagation-Algorithmus $\nabla_{\mathbf{H}} J = \mathbf{G} \mathbf{W}^{(2)\top}$ mit der Backpropagation-Regel als erstem Argument der Matrizenmultiplikation. Dann verwendet die `relu`-Operation die Backpropagation-Regel, um die Komponenten des Gradienten zu streichen, die Elementen von $\mathbf{U}^{(1)}$ kleiner als 0 entsprechen. Das Ergebnis sei \mathbf{G}' . Im letzten Schritt des Backpropagation-Algorithmus wird die Backpropagation-Regel als zweites Argument der `matmul`-Operation eingesetzt, um $\mathbf{X}^\top \mathbf{G}'$ zum Gradienten auf $\mathbf{W}^{(1)}$ zu addieren.

Nachdem diese Gradienten berechnet sind, erfolgt auf ihrer Grundlage eine Anpassung der Parameter durch den Algorithmus für das Gradientenabstiegsverfahren oder einen anderen Optimierungsalgorithmus.

Der Berechnungsaufwand für das mehrschichtige Perzepron wird also vom Aufwand für die Matrizenmultiplikation dominiert. Während der Forward-Propagation multiplizieren wir mit jeder Gewichtungsmatrix. Das führt zu $O(w)$ Multiplikations-Additionen, wobei w die Anzahl der Gewichte ist. Während der rückwärtigen Propagation multiplizieren wir mit der Transponierten jeder Gewichtungsmatrix – der Berechnungsaufwand ist dabei identisch. In puncto Speicher besteht der wesentliche Aufwand für den Algorithmus darin, dass wir die Eingaben für die Nichtlinearität der verdeckten Schicht speichern müssen. Dieser Wert wird ab dem Zeitpunkt seiner Berechnung bis zur Rückkehr des rückwärtigen Laufs an denselben Punkt gespeichert. Der Speicheraufwand entspricht daher $O(mn_h)$, wobei m die Anzahl der Beispiele im Mini-Batch ist und n_h die Anzahl der verdeckten Einheiten.

6.5.8 Komplikationen

Unsere Beschreibung des Backpropagation-Algorithmus hier ist einfacher als die tatsächlichen praktischen Umsetzungen.

Wie bereits erwähnt, haben wir Operationen als Funktion definiert, die nur einen einzelnen Tensor ausgibt, was eine unübliche Einschränkung darstellt. Die meisten Softwareimplementierungen müssen Operationen unterstützen, die mehrere Tensoren zurückgeben können. Um zum Beispiel sowohl den Höchstwert eines Tensors als auch dessen Index zu berechnen, werden optimalerweise beide in einem Speicherdurchlauf bestimmt. Für die effizienteste Berechnung wird also ein Verfahren benötigt, das eine Operation mit zwei Ausgaben nutzt.

Wir haben noch nicht beschrieben, wie der Speicherbedarf für die Backpropagation gesteuert werden kann. Häufig spielt die Aufsummierung vieler Tensoren eine Rolle bei der Backpropagation. Im naiven Ansatz wird jeder einzelne Tensor separat berechnet, um dann in einem zweiten Schritt die Summe zu bilden. Der naive Ansatz führt zu einem extrem hohen Speicherbedarf, der sich durch Einsatz eines einzelnen Puffers, zu dem die einzelnen Werte jeweils nach der Berechnung addiert werden, vermeiden lässt.

Echte Implementierungen der Backpropagation müssen auch mit unterschiedlichen Datentypen umgehen, zum Beispiel 32-Bit-Gleitkommawerten, 64-Bit-Gleitkommawerten und ganzzahligen Werten. Die Regeln für den Umgang mit diesen Typen erfordern besondere Aufmerksamkeit.

Einige Operationen weisen nicht definierte Gradienten auf – diese Fälle müssen unbedingt verfolgt werden, um anzugeben, ob der angeforderte Gradient nicht definiert ist.

Die Differentiation in der Realität wird durch weitere technische Einzelheiten noch komplizierter. Diese Hindernisse sind keineswegs unüberwindbar. Sie haben in diesem Kapitel die wichtigsten Hilfsmittel zum Berechnen von Ableitungen kennengelernt, aber Sie müssen sich auch bewusst machen, dass es viele weitere Kleinigkeiten gibt, die zu berücksichtigen sind.

6.5.9 Differentiation außerhalb der Deep-Learning-Forschungsgemeinde

Die Deep-Learning-Forschungsgemeinde hat sich ein Stück weit von der breiteren Wissenschaft der Informatik isoliert und hat größtenteils ihre eigene Kultur und Herangehensweise in Bezug auf den Umgang mit Differentiation entwickelt. Allgemeiner gesagt befasst sich der Bereich der **automatischen Differentiation** damit, wie Ableitungen algorithmisch berechnet werden können. Der hier beschriebene Backpropagation-Algorithmus ist nur eine von vielen Methoden für das automatische Differenzieren. Es handelt sich um einen Sonderfall einer breiteren Klasse von technischen Verfahren, den

sogenannten **Rückwärtsmodus**. Andere Ansätze werten die Teilausdrücke der Kettenregel in anderer Reihenfolge aus. Grundsätzlich ist das Bestimmen der Berechnungsreihenfolge, die zum geringsten Berechnungsaufwand führt, eine komplizierte Aufgabenstellung. Das Ermitteln der optimalen Reihenfolge der Operationen zum Berechnen des Gradienten ist insofern NP-vollständig (*Naumann*, 2008), als eine Vereinfachung der algebraischen Ausdrücke in ihrer am wenigsten aufwendigen Form erforderlich sein kann.

Ein Beispiel: Gegeben sind die Variablen p_1, p_2, \dots, p_n zur Darstellung von Wahrscheinlichkeiten und die Variablen z_1, z_2, \dots, z_n zur Darstellung nicht normalisierter Log-Wahrscheinlichkeiten. Wir definieren

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)}, \quad (6.57)$$

um die softmax-Funktion aus Potenzierung, Summenbildung und Division zu erstellen und einen Kreuzentropieverlust $J = -\sum_i p_i \log q_i$ zu konstruieren. Ein mathematisch entsprechend gebildeter Mensch kann feststellen, dass die Ableitung von J bezüglich z_i eine recht einfache Form annimmt, nämlich $q_i - p_i$. Der Backpropagation-Algorithmus ist nicht dazu in der Lage, den Gradienten auf diese Weise zu vereinfachen, sondern er propagiert explizit Gradienten durch alle Logarithmus- und Potenzierungsoperationen im ursprünglichen Graphen. Einige Softwarebibliotheken wie Theano (*Bergstra et al.*, 2010; *Bastien et al.*, 2012) können einige algebraische Einstellungen vornehmen, um Verbesserungen gegenüber dem Graphen des reinen Backpropagation-Algorithmus zu erzielen.

Wenn der Vorwärtsgraph \mathcal{G} einen einzelnen Ausgabeknoten besitzt und jede partielle Ableitung $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ mit einem konstanten Rechenaufwand bestimmt werden kann, garantiert die Backpropagation, dass die Anzahl der Berechnungen für die Gradientenberechnung von derselben Größenordnung wie bei der Vorwärtsberechnung ist (vgl. Algorithmus 6.2), da jede lokale partielle Ableitung $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ nur einmal mit einer zugehörigen Multiplikation und Addition für die rekursive Kettenregel-Bildung (Gleichung 6.53) berechnet werden muss. Die Gesamtberechnung beträgt somit $O(\# \text{ Kanten})$. Potenziell ist eine Reduzierung möglich, die allerdings eine Vereinfachung des im Rahmen der Backpropagation konstruierten Berechnungsgraphen erfordert, was eine NP-vollständige Aufgabe ist. Implementierungen wie Theano und TensorFlow nutzen Heuristiken auf Basis der Zuordnung bekannter Vereinfachungsmuster zu iterativen Versuchen, um den Graphen zu vereinfachen. Wir haben die Backpropagation nur für die Berechnung eines skalaren Ausgabegradienten definiert, aber sie kann auch auf die Berechnung einer Jacobi-Matrix ausgedehnt werden (einer von k verschiedenen Skalar-

knoten im Graphen oder eines tensorwertigen Knotens mit k Werten). Eine naive Implementierung benötigt dann eventuell k -mal mehr Berechnungen: Für jeden skalarinternen Knoten im ursprünglichen Vorwärtsgraphen berechnet die naive Implementierung anstelle eines einzelnen gleich k Gradienten. Wenn die Anzahl der Ausgaben des Graphen größer als die Anzahl seiner Eingaben ist, kann es von Vorteil sein, eine andere Form der automatischen Differentiation, den sogenannten **Vorwärtsmodus**, zu nutzen. Die Vorwärtsakkumulation wurde zur Echtzeitberechnung von Gradienten in RNNs vorgeschlagen, zum Beispiel in (*Williams und Zipser*, 1989). Dieser Ansatz umgeht das Erfordernis, Werte und Gradienten für den gesamten Graphen zu speichern, indem er die effiziente Berechnung zugunsten des Speicherplatzbedarfs einschränkt. Die Beziehung zwischen Vorwärts- und Rückwärtsmodus ist analog zur Beziehung zwischen der linksseitigen und rechtsseitigen Multiplikation einer Reihe von Matrizen, beispielsweise

$$\mathbf{ABCD}, \quad (6.58)$$

wobei die Matrizen als Jacobi-Matrizen betrachtet werden können. Wenn \mathbf{D} zum Beispiel ein Spaltenvektor ist und \mathbf{A} viele Zeilen enthält, verfügt der Graph über eine Ausgabe und viele Eingaben. Für die Multiplikationen vom Ende in Gegenrichtung werden nur Produkte aus Matrizen und Vektoren benötigt. Diese Reihenfolge entspricht dem Rückwärtsmodus. Dagegen wird bei Multiplikation von links eine Reihe von Matrizenprodukten benötigt, was die Berechnung insgesamt deutlich aufwendiger macht. Wenn \mathbf{A} weniger Zeilen aufweist als \mathbf{D} Spalten, dann ist es allerdings weniger aufwendig, die Multiplikationen von links nach rechts durchzuführen, also den Vorwärtsmodus zu nutzen.

In vielen Forschungsgemeinden außerhalb des Machine Learnings ist es üblicher, Software zur Differentiation zu implementieren, die direkt mit dem klassischen Programmcode arbeitet, zum Beispiel Python oder C, und automatisch Programme erzeugt, die in diesen Sprachen geschriebene Funktionen differenzieren. In der Deep-Learning-Forschungsgemeinde werden Berechnungsgraphen meist als explizite Datenstrukturen dargestellt, die mithilfe spezialisierter Bibliotheken erzeugt wurden. Der spezialisierte Ansatz hat den Nachteil, dass die Entwickler der Bibliotheken `bprop`-Methoden für jede Operation definieren müssen und so die Anwender auf diese Operationen einschränken. Allerdings bietet der spezialisierte Ansatz auch den Vorteil, dass für jede Operation individuelle Backpropagation-Regeln entwickelt werden können, wodurch sich Geschwindigkeit oder Stabilität auf eine nicht offensichtliche Weise verbessern lässt, die mit einem automatisierten Verfahren vermutlich nicht abgebildet werden kann.

Die Backpropagation ist somit nicht der einzige oder beste Weg zur Gradientenberechnung. Es handelt sich allerdings um eine praktische Methode, die in der Deep-Learning-Forschungsgemeinde gute Dienste leistet. Möglicherweise wird es künftig andere Methoden der Differentiation für tiefe Netze geben, wenn die Anwender des Deep Learnings Fortschritte aus dem breiteren Anwendungsgebiet der automatischen Differentiation erkennen und nutzen.

6.5.10 Ableitungen höherer Ordnung

Einige Software-Frameworks unterstützen auch Ableitungen höherer Ordnung. Im Deep-Learning-Bereich gehören auf jeden Fall Theano und TensorFlow dazu. Diese Bibliotheken nutzen dieselbe Art von Datenstruktur zum Beschreiben der Ausdrücke für Ableitungen wie zum Beschreiben der ursprünglichen Funktion, die differenziert wird. Das bedeutet, dass der symbolische Mechanismus für die Differentiation auf Ableitungen angewandt werden kann.

Im Deep-Learning-Kontext wird nur selten die zweite Ableitung einer Skalarfunktion berechnet. Stattdessen interessieren wir uns meist für die Eigenschaften der Hesse-Matrix. Für eine Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$ hat die Hesse-Matrix die Größe $n \times n$. In typischen Deep-Learning-Anwendungen ist n die Anzahl der Parameter im Modell – und das kann schnell in die Milliarden gehen. Die gesamte Hesse-Matrix lässt sich daher unmöglich darstellen.

Statt die Hesse-Matrix explizit zu berechnen, verwendet der typische Deep-Learning-Ansatz die **Krylow-Verfahren**. Die Krylow-Verfahren sind eine Sammlung iterativer Techniken zum Ausführen diverser Operationen, zum Beispiel zum näherungsweisen Invertieren einer Matrix oder zum Suchen der Approximationen an deren Eigenvektoren oder Eigenwerten; dabei kommen ausschließlich Matrix-Vektor-Produkte zum Einsatz.

Um die Krylow-Verfahren für die Hesse-Matrix zu nutzen, müssen wir einfach nur das Produkt der Hesse-Matrix \mathbf{H} und eines beliebigen Vektors \mathbf{v} bilden. Ein direkter Ansatz (*Christianson, 1992*) hierfür ist die Berechnung

$$\mathbf{H}\mathbf{v} = \nabla_x \left[(\nabla_x f(x))^\top \mathbf{v} \right]. \quad (6.59)$$

Beide Gradientenberechnungen in diesem Ausdruck können automatisch durch die passende Softwarebibliothek bestimmt werden. Beachten Sie, dass der äußere Gradientenausdruck den Gradienten einer Funktion des inneren Gradientenausdrucks nutzt.

Wenn v selbst ein Vektor ist, der von einem Berechnungsgraphen erzeugt wird, muss unbedingt angegeben werden, dass die automatische Differenziation nicht durch den Graphen differenzieren darf, aus dem v entstanden ist.

Obwohl das Berechnen der Hesse-Matrix im Allgemeinen nicht empfohlen wird, ist es mithilfe von Produkten aus Hesse-Matrizen und Vektoren doch möglich. Dazu wird einfach $\mathbf{H}e^{(i)}$ für alle $i = 1, \dots, n$ berechnet, wobei $e^{(i)}$ der One-hot-Vektor mit $e_i^{(i)} = 1$ ist und alle anderen Elemente gleich 0 sind.

6.6 Historische Anmerkungen

Feedforward-Netze stellen sich als effiziente nichtlineare Funktionsapproximatoren auf Basis des Gradientenabstiegsverfahrens zur Minimierung des Fehlers in einer Funktionsapproximation dar. Von diesem Standpunkt aus betrachtet, ist das moderne Feedforward-Netz der Gipfel eines jahrhunder telangen Fortschritts im Bereich der allgemeinen Funktionsapproximation.

Die Kettenregel, die dem Backpropagation-Algorithmus zugrunde liegt, wurde im 17. Jahrhundert erfunden (*Leibniz*, 1676; *L'Hôpital*, 1696). Analysis und Algebra wurden schon lange Zeit zum Lösen von Optimierungsproblemen in geschlossener Form eingesetzt, aber das Gradientenabstiegsverfahren wurde als Verfahren zur iterativen Annäherung der Lösung von Optimierungsproblemen erst im 19. Jahrhundert eingeführt (*Cauchy*, 1847).

Anfang der 1940er-Jahre wurden diese Verfahren zur Funktionsapproximation für Machine-Learning-Modelle wie das Perzepron eingesetzt. Allerdings beruhten die frühen Modelle auf linearen Modellen. Kritiker wie Marvin Minsky zeigten mehrere Mängel der Familie linearer Modelle auf, zum Beispiel ihr Unvermögen, die XOR-Funktion zu erlernen. Das brachte den gesamten Ansatz der neuronalen Netze in Verruf.

Das Erlernen nichtlinearer Funktionen erforderte die Entwicklung eines mehrschichtigen Perzeprons und ein Mittel zur Gradientenberechnung mit einem solchen Modell. Effiziente Anwendungen der Kettenregel auf Basis der dynamischen Programmierung kamen in den 1960er- und 1970er-Jahren auf; sie wurden primär in Steuerungsanwendungen genutzt (*Kelley*, 1960; *Bryson und Denham*, 1961; *Dreyfus*, 1962; *Bryson und Ho*, 1969; *Dreyfus*, 1973), aber auch für die Empfindlichkeitsanalyse (*Linnainmaa*, 1976). *Werbos* (1981) schlug vor, diese Verfahren für das Trainieren künstlicher neuronaler Netze einzusetzen. Die Idee wurde schließlich in die Praxis umgesetzt, nachdem sie auf unterschiedliche Art von mehreren Parteien wiederentdeckt wurde (*LeCun*, 1985; *Parker*, 1985; *Rumelhart et al.*, 1986a). Das Buch

Parallel Distributed Processing stellt die Ergebnisse einiger der ersten erfolgreichen Experimente mit der Backpropagation in einem Kapitel (*Rumelhart et al.*, 1986b) vor, das einen bedeutenden Beitrag zur Beliebtheit der Backpropagation geleistet und eine sehr geschäftige Phase der Forschung über mehrschichtige neuronale Netze eingeläutet hat. Die von den Autoren, insbesondere Rumelhart und Hinton, vertretenen Ansichten erstrecken sich weit über die Backpropagation hinaus. Dazu gehören maßgebliche Ideen zur möglichen rechnerischen Implementierung diverser zentraler Aspekte der Wahrnehmung und des Lernens, die als »Konnektionismus« bezeichnet wurden, da diese Denkrichtung die Bedeutung der Verbindungen zwischen Neuronen als den Ort, an dem Lernen und Gedächtnis stattfinden, betont. Hierzu gehört insbesondere der Begriff der verteilten Repräsentation (*Hinton et al.*, 1986).

Auf der Welle des Erfolgs der Backpropagation gewann die Forschung an neuronalen Netzen an Beliebtheit, um schließlich Anfang der 1990er-Jahre einen Höhepunkt zu erreichen. Anschließend wurden andere Machine-Learning-Verfahren beliebter, bis 2006 ein erneuter Aufschwung des modernen Deep Learnings begann.

Die Grundideen hinter den modernen Feedforward-Netzen haben sich seit den 1980er-Jahren kaum geändert. Noch immer werden derselbe Backpropagation-Algorithmus und dieselben Ansätze im Gradientenabstiegsverfahren genutzt. Die wesentlichen Verbesserungen bei der Leistung neuronaler Netze zwischen 1986 und 2015 lassen sich zwei Faktoren zuschreiben, nämlich erstens den größeren Datensätzen, die die Herausforderung der statistischen Generalisierung für neuronale Netze stark reduziert haben, und zweitens dem Umstand, dass neuronale Netze infolge leistungsfähigerer Computer und einer besseren Softwareinfrastruktur immer stärker gewachsen sind. Eine kleine Anzahl algorithmischer Veränderungen hat die Leistung neuronaler Netze ebenfalls spürbar verbessert.

Eine dieser Änderungen ist der Ersatz des mittleren quadratischen Fehlers durch die Kreuzentropiefamilie der Verlustfunktionen. Der mittlere quadratische Fehler war in den 1980ern und 1990ern beliebt, wurde aber nach und nach durch Kreuzentropieverluste und das Maximum-Likelihood-Prinzip ersetzt, als sich diese Ideen in der Statistik- und Machine-Learning-Gemeinde verbreiteten. Die Nutzung der Kreuzentropieverluste hat die Modellleistung mit sigmoiden und softmax-Ausgaben deutlich gesteigert. Zuvor wirkten Sättigung und langsames Lernen bei Verwendung der MQF-Verlustfunktion als Einschränkung.

Eine weitere wesentliche Algorithmusänderung, die zu einer deutlich gesteigerten Leistung in Feedforward-Netzen beigetragen hat, war der Ersatz der sigmoiden verdeckten Einheiten durch stückweise linear verdeckte Einheiten, beispielsweise die ReLUs. Die Rektifizierung anhand der Funktion $\max\{0, z\}$ wurde in frühen neuronalen Netzmodellen eingeführt und lässt sich bis zum Cognitron und Neocognitron zurückverfolgen (Fukushima, 1975, 1980). Diese frühen Modelle nutzten keine ReLUs, sondern wandten die Rektifizierung auf nichtlineare Funktionen an. Trotz der frühzeitigen Beliebtheit der Rektifizierung wurde sie in den 1980er-Jahren größtenteils durch Sigmoide ersetzt, vielleicht weil Sigmoide bei sehr kleinen neuronalen Netzen besser funktionieren. Zu Beginn der 2000er waren ReLUs verschrien, da man der Meinung war, Aktivierungsfunktionen mit nicht differenzierbaren Punkten meiden zu müssen. Das änderte sich um 2009. Jarrett *et al.* (2009) stellten fest, dass »der Einsatz einer rektifizierenden Nichtlinearität der wichtigste Einzelfaktor für die Verbesserung eines Erkennungssystems« ist, neben diversen anderen Faktoren im Aufbau der neuronalen Netzarchitektur.

Für kleine Datensätze zeigten Jarrett *et al.* (2009), dass der Einsatz rektifizierender Nichtlinearitäten noch viel wichtiger als das Ermitteln der Gewichte der verdeckten Schichten ist. Zufällige Gewichte reichen aus, um nützliche Informationen in einem rektifizierten linearen Netz zu propagieren und der obersten Klassifizierungsschicht zu ermöglichen, unterschiedliche Merkmalsvektoren den Klassenidentitäten zuzuweisen.

Wenn mehr Daten zur Verfügung stehen, zieht der Lernprozess genügend nützliche Kenntnisse, um die Leistung zufällig gewählter Parameter zu überflügeln. Glorot *et al.* (2011a) haben gezeigt, dass das Lernen in tiefen rektifizierten linearen Netzen viel einfacher als in tiefen Netzen mit gekrümmter oder zweiseitiger Sättigung in den Aktivierungsfunktionen ist.

ReLUs sind auch von historischem Interesse, da sie zeigen, dass die Neurowissenschaft weiterhin auf die Entwicklung von Deep-Learning-Algorithmen wirkte. Glorot *et al.* (2011a) ermunterten aus biologischer Sicht zum Einsatz von ReLUs. Die halb-rektifizierende Nichtlinearität sollte die folgenden Eigenschaften biologischer Neuronen erfassen: (1) Für einige Eingaben sind biologische Neuronen vollständig inaktiv. (2) Für einige Eingaben ist die Ausgabe eines biologischen Neurons proportional zu seiner Eingabe. (3) Meist arbeiten biologische Neuronen unter Bedingungen, in denen sie inaktiv sind, d. h. sie sollten **schwache Aktivierungen** (engl. *sparse activations*) aufweisen.

Als 2006 die moderne Phase des Deep Learnings begann, hatten Feedforward-Netze nach wie vor einen schlechten Ruf. In der Zeit von etwa 2006 bis

2012 herrschte allgemein die Überzeugung, dass Feedforward-Netze nur dann gut funktionieren, wenn sie von anderen Modellen, beispielsweise probabilistischen Modellen, unterstützt werden. Heute wissen wir, dass Feedforward-Netze mit den richtigen Ressourcen und technischen Ansätzen sehr gut arbeiten. Das Lernen auf Gradientenbasis in Feedforward-Netzen ist heute ein Instrument zur Entwicklung probabilistischer Modelle, zum Beispiel für den Variational Autoencoder (VAE) und Generative-Adversarial-Netze (GANs), die in Kapitel 20 beschrieben werden. Statt als unzuverlässige Technologie zu gelten, die von anderen Techniken gestützt werden muss, hat sich das Lernen auf Gradientenbasis in Feedforward-Netzen seit 2012 zur leistungsfähigen Technologie gemausert, die Anwendung in vielen Machine-Learning-Aufgaben findet. Im Jahr 2006 hat die Forschungsgemeinde das unüberwachte Lernen als Unterstützung für überwachtes Lernen eingesetzt und mittlerweile ist es ironischerweise gängiger, das überwachte Lernen zur Unterstützung des unüberwachten Lernens zu nutzen.

Feedforward-Netze bieten noch immer ungenutztes Potenzial. Zukünftig erwarten wir ihre Anwendung für viele weitere Aufgaben sowie Fortschritte bei Optimierungsalgorithmen und beim Modelldesign, die ihre Leistung weiter steigern. In diesem Kapitel haben wir uns in erster Linie mit der Modellfamilie der neuronalen Netze befasst. In den nächsten Kapiteln widmen wir uns der Nutzung dieser Modelle, insbesondere der Regularisierung und dem Training.

7

Regularisierung

Eine zentrale Herausforderung im Machine Learning besteht darin, Algorithmen zu entwickeln, die nicht nur mit den Trainingsdaten gut zurechtkommen, sondern auch mit neuen Eingangsdaten. Viele Verfahren im Deep Learning sind explizit darauf ausgelegt, den Testfehler zu reduzieren, was aber zu einem höheren Trainingsfehler führen kann. In ihrer Gesamtheit werden diese Verfahren als Regularisierungsverfahren bezeichnet. In der Praxis des Deep Learnings stehen Ihnen viele Arten der Regularisierung zur Verfügung. Tatsächlich ist die Entwicklung effektiverer Regularisierungsverfahren bis heute eines der wesentlichen Forschungsgebiete in dieser Disziplin.

In Kapitel 5 haben Sie die grundlegenden Konzepte für Generalisierung, Unteranpassung, Überanpassung, Verzerrung, Varianz und Regularisierung kennengelernt. Wenn Sie diese Begriffe noch nicht kennen, lesen Sie zunächst Kapitel 5, bevor Sie mit diesem hier fortfahren.

Im Folgenden befassen wir uns detaillierter mit der Regularisierung, insbesondere mit Regularisierungsverfahren für tiefe Modelle oder Modelle, die als Bausteine für tiefe Modelle dienen können.

In einigen Abschnitten geht es um Standardkonzepte im Machine Learning. Wenn Sie damit bereits vertraut sind, können Sie die jeweiligen Abschnitte einfach überspringen. Allerdings behandeln wir vorrangig die Bedeutung dieser grundlegenden Konzepte für den speziellen Fall der neuronalen Netze.

In Abschnitt 5.2.2 haben wir die Regularisierung definiert als »jede Änderung, die wir an einem Lernalgorithmus vornehmen, die zum Ziel hat, dessen Generalisierungsfehler – nicht aber dessen Trainingsfehler – zu reduzieren«. Es gibt viele Regularisierungsverfahren. Einige davon legen zusätzliche Bedingungen für ein Machine-Learning-Modell fest, andere schränken die Parameterwerte ein. Wieder andere ergänzen die Zielfunktion

um Terme, die man sich als weiche Bedingung (engl. *soft constraint*) für die Parameterwerte vorstellen kann. Werden diese zusätzlichen Bedingungen und Strafterme sorgfältig ausgewählt, können sie die Leistung auf der Testdatenmenge steigern. Manchmal sollen diese Bedingungen und Strafterme ein gewisses Vorwissen repräsentieren. In anderen Fällen dienen sie dazu, eine allgemeine Präferenz für eine einfachere Modellklasse auszudrücken, um so die Generalisierung zu fördern. Einige Strafterme und Bedingungen werden benötigt, um ein unterbestimmtes Problem zu einem bestimmten zu machen. Andere Arten der Regularisierung, die sogenannten Ensemblemethoden, fassen mehrere Hypothesen zur Erklärung der Trainingsdaten zusammen.

Im Deep-Learning-Kontext beruhen die meisten Regularisierungsverfahren auf der Regularisierung von Schätzern. Bei der Regularisierung eines Schätzers wird eine höhere Verzerrung zugunsten einer reduzierten Varianz in Kauf genommen. Ein wirkungsvoller Regularisierer findet dabei eine vorteilhafte Balance, bei der die Varianz erheblich reduziert wird, ohne die Verzerrung zu sehr zu erhöhen. Als wir in Kapitel 5 Generalisierung und Überanpassung behandelt haben, standen drei Fälle im Fokus, bei denen die trainierte Modellfamilie entweder (1) den tatsächlichen datengenerierenden Prozess ausgeschlossen hat – entsprechend einer Unteranpassung und der Einführung einer Verzerrung –, oder (2) dem tatsächlichen datengenerierenden Prozess entsprach oder (3) den generierenden Prozess und dabei möglicherweise andere eventuelle generierende Prozesse eingeschlossen hat – ein Überanpassungsbereich, in dem die Varianz anstelle der Verzerrung den Schätzfehler dominiert. Das Ziel der Regularisierung ist es, ein Modell aus dem dritten Bereich für den zweiten Bereich zu verwenden.

In der Praxis muss eine überkomplexe Modellfamilie nicht unbedingt die Zielfunktion oder den tatsächlichen datengenerierenden Prozess oder überhaupt eine gute Approximation dieser beiden enthalten. Wir haben fast nie Zugang zu dem tatsächlichen datengenerierenden Prozess, können also nie sicher sagen, ob die geschätzte Modellfamilie den generierenden Prozess enthält oder nicht. Die meisten Anwendungen von Deep-Learning-Algorithmen finden jedoch in Bereichen statt, in denen der tatsächliche datengenerierende Prozess mit fast hundertprozentiger Sicherheit außerhalb der Modellfamilie liegt. Deep-Learning-Algorithmen werden typischerweise in extrem komplexen Bereichen eingesetzt, zum Beispiel für Bilder, Audiosequenzen und Texte, bei denen der wahre generierende Prozess letztlich die Simulation der Grundgesamtheit einschließt. Bis zu einem gewissen Maß versuchen wir uns an der Quadratur (datengenerierender Prozess) des Kreises (Modellfamilie).

Um die Komplexität des Modells zu kontrollieren, reicht es also nicht aus, einfach ein Modell der richtigen Größe mit der korrekten Anzahl von Parametern zu finden. Vielmehr stellen wir vielleicht – und in praktischen Deep-Learning-Szenarios nahezu immer – fest, dass das am besten passende Modell (im Sinne eines möglichst kleinen Generalisierungsfehlers) ein großes Modell ist, das entsprechend regularisiert wurde.

Wir betrachten nun diverse Verfahren zum Erstellen solch großer tiefer regularisierter Modelle.

7.1 Parameter-Norm-Strafterme

Die Regularisierung ist ein schon Jahrzehnte vor dem Aufkommen des Deep Learnings verwendetes Verfahren. Lineare Modelle wie die lineare Regression und die logistische Regression ermöglichen einfache, zielgerichtete und effektive Regularisierungsverfahren.

Viele Regularisierungsansätze beruhen darauf, die Kapazität des Modells – ob neuronales Netz, lineare Regression oder logistische Regression – durch Verwendung eines Parameter-Norm-Strafterms $\Omega(\boldsymbol{\theta})$ in der Zielfunktion J einzuschränken. Wir kennzeichnen die regularisierte Zielfunktion mit \tilde{J} :

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}), \quad (7.1)$$

wobei $\alpha \in [0, \infty)$ ein Hyperparameter ist, der den relativen Beitrag des Norm-Strafterms Ω relativ zur Standard-Zielfunktion J gewichtet. Ist α gleich 0, erfolgt keine Regularisierung. Größere Werte für α führen zu einer stärkeren Regularisierung.

Wenn unser Trainingsalgorithmus die regularisierte Zielfunktion \tilde{J} minimiert, wird dies sowohl die ursprüngliche Zielfunktion J für die Trainingsdaten als auch ein Maß der Größe der Parameter $\boldsymbol{\theta}$ (oder einer Teilmenge der Parameter) verringern. Abhängig von der Wahl der Parameter-Norm Ω wird eventuell eine andere Lösung bevorzugt. In diesem Abschnitt behandeln wir die Auswirkungen der einzelnen Normen, wenn diese als Strafterme für die Modellparameter eingesetzt werden.

Bevor wir uns dem Regularisierungsverhalten der verschiedenen Normen zuwenden, möchten wir anmerken, dass für neuronale Netze üblicherweise ein Parameter-Norm-Strafterm Ω gewählt wird, der *nur die Gewichte* der affinen Transformation der einzelnen Schichten bestraft, die Verzerrungen jedoch nicht regularisiert. Für eine exakte Anpassung der Verzerrungen sind meist weniger Daten als für die Gewichte erforderlich. Jedes Gewicht gibt an, wie zwei Variablen interagieren. Um ein Gewicht gut anzupassen, müssen

beide Variablen unter wechselnden Umständen beobachtet werden. Jede Verzerrung steuert nur eine einzelne Variable. Wenn wir die Verzerrungen nicht regularisieren, führen wir also nicht zu viel Varianz ein. Außerdem kann die Regularisierung der Verzerrungsparameter zu einem hohen Maß an Unteranpassung führen. Daher nutzen wir den Vektor \mathbf{w} zum Angeben aller Gewichte, die einem Norm-Strafterm unterliegen sollen, und den Vektor $\boldsymbol{\theta}$ für alle Parameter, also sowohl \mathbf{w} als auch die nicht regularisierten Parameter.

Im Kontext der neuronalen Netze ist es manchmal wünschenswert, einen anderen Strafterm mit einem eigenen α -Koeffizienten für jede Schicht des Netzes zu verwenden. Da die Suche nach dem korrekten Wert vieler Hyperparameter sehr aufwendig sein kann, ist es dennoch vernünftig, denselben Weight Decay für alle Schichten zu nutzen und so den Suchraum einzuschränken.

7.1.1 L^2 -Parameter-Regularisierung

In Abschnitt 5.2.2 haben Sie bereits eine der einfachsten und gängigsten Arten des Parameter-Norm-Strafterms kennengelernt, nämlich den L^2 -Parameter-Norm-Strafterm, der meist als **Weight Decay** bezeichnet wird. Dieses Regularisierungsverfahren nähert die Gewichte dem Ursprung¹ an, indem ein Regularisierungsterm $\Omega(\boldsymbol{\theta}) = \frac{1}{2} \|\mathbf{w}\|_2^2$ zur Zielfunktion hinzugefügt wird. In anderen Wissenschaftsbereichen wird die L^2 -Regularisierung auch **Ridge-Regression** oder **Tikhonov-Regularisierung** genannt.

Wenn wir den Gradienten der regularisierten Zielfunktion betrachten, können wir gewisse Erkenntnisse über das Verhalten der Weight-Decay-Regularisierung erlangen. Für eine einfachere Darstellung lassen wir den Verzerrungsparameter weg, sodass $\boldsymbol{\theta}$ einfach \mathbf{w} ist. Ein solches Modell weist die folgende Gesamtzielfunktion auf:

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2} \mathbf{w}^\top \mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y}), \quad (7.2)$$

dazu der entsprechende Parametergradient

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \quad (7.3)$$

¹ Allgemeiner betrachtet könnten wir die Parameter so regularisieren, dass sie in der Nähe eines bestimmten Punktes im Raum liegen; dabei kommt es überraschenderweise noch immer zu einem Regularisierungseffekt. Allerdings erzielen wir bessere Ergebnisse mit einem Wert, der näher am wahren Wert liegt. Wenn wir nicht wissen, ob der korrekte Wert positiv oder negativ ist, bietet sich Null als Standardwert an. Da die Regularisierung der Modellparameter gegen Null sehr viel üblicher ist, konzentrieren wir uns in unserer Abhandlung auf diesen Sonderfall.

Um die Gewichte in einem einzelnen Gradientenschritt anzupassen, führen wir diese Anpassung durch:

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon (\alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})). \quad (7.4)$$

Anders notiert stellt sich die Anpassung so dar:

$$\mathbf{w} \leftarrow (1 - \epsilon \alpha) \mathbf{w} - \epsilon \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \quad (7.5)$$

Wie Sie sehen, hat das Hinzufügen des Weight-Decay-Terms die Lernregel so verändert, dass der Gewichtsvektor multiplikativ bei jedem Schritt um einen konstanten Faktor abnimmt, bevor die eigentliche Gradientenaktualisierung erfolgt. Dies geschieht in einem Einzelschritt. Aber was geschieht im gesamten Trainingsverlauf?

Wir vereinfachen die Analyse nochmals, indem wir eine quadratische Approximation an die Zielfunktion in der Nähe des Werts der Gewichte vornehmen, der zu einem minimalen nicht regularisierten Trainingsaufwand führt, $\mathbf{w}^* = \arg \min_{\mathbf{w}} J(\mathbf{w})$. Ist die Zielfunktion wirklich quadratisch (wie beim Anpassen eines linearen Regressionsmodells mit mittlerem quadratischen Fehler), dann ist die Approximation perfekt. Die Approximation \hat{J} ergibt sich aus

$$\hat{J}(\boldsymbol{\theta}) = J(\mathbf{w}^*) + \frac{1}{2} (\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H} (\mathbf{w} - \mathbf{w}^*), \quad (7.6)$$

wobei \mathbf{H} die Hesse-Matrix von J bezüglich \mathbf{w} an der Stelle \mathbf{w}^* ist. In dieser quadratischen Approximation gibt es keinen Term erster Ordnung, da \mathbf{w}^* als Minimum definiert ist, in dem der Gradient verschwindet. Ebenso können wir, da \mathbf{w}^* die Stelle eines Minimums von J ist, schließen, dass \mathbf{H} positiv semidefinit ist.

Das Minimum von \hat{J} tritt auf, wenn der Gradient

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H} (\mathbf{w} - \mathbf{w}^*) \quad (7.7)$$

gleich $\mathbf{0}$ ist.

Um die Auswirkung von Weight Decay zu untersuchen, ändern wir Gleichung 7.7 und ergänzen den Weight-Decay-Gradienten. Wir können nun das Minimum der regularisierten Version von \hat{J} bestimmen. Wir verwenden die Variable $\tilde{\mathbf{w}}$ zur Darstellung der Stelle des Minimums.

$$\alpha \tilde{\mathbf{w}} + \mathbf{H} (\tilde{\mathbf{w}} - \mathbf{w}^*) = 0 \quad (7.8)$$

$$(\mathbf{H} + \alpha \mathbf{I}) \tilde{\mathbf{w}} = \mathbf{H} \mathbf{w}^* \quad (7.9)$$

$$\tilde{\mathbf{w}} = (\mathbf{H} + \alpha \mathbf{I})^{-1} \mathbf{H} \mathbf{w}^* \quad (7.10)$$

Wenn sich α dem Wert 0 annähert, nähert sich die regularisierte Lösung $\tilde{\mathbf{w}}$ dem Wert \mathbf{w}^* an. Aber was geschieht, wenn α zunimmt? Da \mathbf{H} \mathbf{H} und symmetrisch ist, ist eine Zerlegung in eine Diagonalmatrix Λ und eine orthonormale Basis aus Eigenvektoren \mathbf{Q} möglich, sodass $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$ ist. Durch Zerlegung in Gleichung 7.10 erhalten wir

$$\tilde{\mathbf{w}} = (\mathbf{Q}\Lambda\mathbf{Q}^\top + \alpha \mathbf{I})^{-1} \mathbf{Q}\Lambda\mathbf{Q}^\top \mathbf{w}^* \quad (7.11)$$

$$= [\mathbf{Q}(\Lambda + \alpha \mathbf{I})\mathbf{Q}^\top]^{-1} \mathbf{Q}\Lambda\mathbf{Q}^\top \mathbf{w}^* \quad (7.12)$$

$$= \mathbf{Q}(\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{Q}^\top \mathbf{w}^*. \quad (7.13)$$

Weight Decay skaliert also \mathbf{w}^* neu entlang der durch die Eigenvektoren von \mathbf{H} definierten Achsen. Insbesondere wird die am i -ten Eigenvektor von \mathbf{H} ausgerichtete Komponente von \mathbf{w}^* um einen Faktor $\frac{\lambda_i}{\lambda_i + \alpha}$ skaliert. (Die Funktionsweise dieser Art Skalierung wurde in Abbildung 2.3 erläutert. Sehen Sie ggf. dort nach.)

In den Richtungen, in denen die Eigenwerte von \mathbf{H} relativ groß sind, zum Beispiel $\lambda_i \gg \alpha$, ist die Auswirkung der Regularisierung relativ klein. Doch Komponenten mit $\lambda_i \ll \alpha$ werden beinahe auf eine Größe von 0 reduziert. Dieser Effekt wird in Abbildung 7.1 dargestellt.

Nur Richtungen, in denen die Parameter stark zur Reduzierung der Zielfunktion beitragen, bleiben relativ intakt. In Richtungen, die nicht zur Reduzierung der Zielfunktion beitragen, zeigt uns ein kleiner Eigenwert der Hesse-Matrix, dass die Bewegung in dieser Richtung den Gradienten nicht merklich erhöht. Komponenten des Gewichtsvektors, die derart unwichtigen Richtungen entsprechen, werden mithilfe der Regularisierung im gesamten Training verringert.

Bisher haben wir Weight Decay bezüglich seiner Auswirkung auf die Optimierung einer abstrakten allgemein quadratischen Kostenfunktion betrachtet. Welche Beziehung besteht zwischen diesen Auswirkungen und dem Machine Learning im Speziellen? Das können wir durch Untersuchung der linearen Regression herausfinden. Bei diesem Modell ist die wahre Kostenfunktion quadratisch und daher für die bisher genutzte Art der Analyse zugänglich. Wenn wir das tun, können wir einen Sonderfall derselben Ergebnisse erhalten, allerdings steht die Lösung nun im Kontext der Trainingsdaten. Für die lineare Regression ist die Kostenfunktion die Summe der quadratischen Fehler:

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}). \quad (7.14)$$

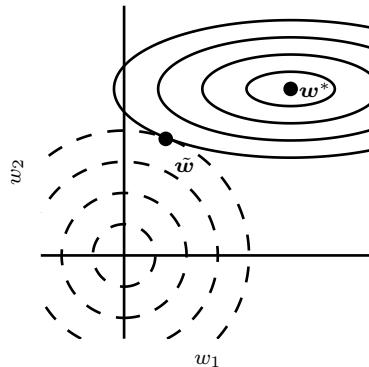


Abbildung 7.1: Darstellung des Effekts der L^2 -Regularisierung (Weight Decay) auf den Wert eines optimalen \mathbf{w} . Die durchgezogenen Ellipsen stehen für gleichwertige Konturen des nicht regularisierten Ziels. Die gepunkteten Kreise stehen für gleichwertige Konturen des L^2 -Regularisierers. Im Punkt $\tilde{\mathbf{w}}$ erreichen diese konkurrierenden Ziele ein Gleichgewicht. In der ersten Dimension ist der Eigenwert der Hesse-Matrix von J klein. Die Zielfunktion steigt mit zunehmender horizontaler Entfernung von \mathbf{w}^* nur geringfügig. Da die Zielfunktion keine starke Neigung in dieser Richtung ausdrückt, wirkt der Regularisierer in dieser Achse stark. Der Regularisierer zieht w_1 in die Nähe von Null. In der zweiten Dimension reagiert die Zielfunktion sehr empfindlich auf Bewegungen weg von \mathbf{w}^* . Der entsprechende Eigenwert ist groß, was eine starke Krümmung andeutet. Daher beeinflusst der Weight Decay die Lage von w_2 kaum.

Wenn wir die L^2 -Regularisierung hinzunehmen, wird die Zielfunktion zu

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\alpha\mathbf{w}^\top \mathbf{w}. \quad (7.15)$$

Damit werden die Normalgleichungen für die Lösung geändert, und zwar von

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (7.16)$$

in

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}. \quad (7.17)$$

Die Matrix $\mathbf{X}^\top \mathbf{X}$ in Gleichung 7.16 ist proportional zur Kovarianzmatrix $\frac{1}{m} \mathbf{X}^\top \mathbf{X}$. Mittels L^2 -Regularisierung wird diese Matrix in Gleichung 7.17 durch $(\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1}$ ersetzt. Die neue Matrix entspricht der ursprünglichen, ergänzt um α in der Diagonalen. Die Diagonaleinträge der Matrix entsprechen der Varianz der einzelnen Eingabemerkmale. Wir stellen fest, dass die L^2 -Regularisierung dazu führt, dass der Lernalgorithmus für die Eingabe \mathbf{X} eine höhere Varianz »wahrnimmt«, wodurch die Gewichte für

Merkmale abnehmen, deren Kovarianz mit dem Zielwert der Ausgabe im Vergleich zur hinzugefügten Varianz gering ist.

7.1.2 L^1 -Regularisierung

Obwohl der L^2 -Weight-Decay die gängigste Form des Weight Decays ist, gibt es doch auch andere Möglichkeiten, die Anzahl der Modellparameter zu bestrafen. Eine davon ist die L^1 -Regularisierung.

Formal ist die L^1 -Regularisierung für den Modellparameter \mathbf{w} wie folgt definiert:

$$\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |w_i|. \quad (7.18)$$

Es handelt sich also um die Summe der Absolutbeträge der einzelnen Parameter.² Wir widmen uns nun der Auswirkung der L^1 -Regularisierung auf das einfache lineare Regressionsmodell ohne Verzerrungsparameter, das wir bereits für die Analyse der L^2 -Regularisierung herangezogen haben. Vor allem möchten wir die Unterschiede zwischen der L^1 - und der L^2 -Regularisierung herausstellen. Wie bei L^2 -Weight-Decay steuert auch der L^1 -Weight-Decay die Stärke der Regularisierung durch Skalieren des Strafterms Ω anhand eines positiven Hyperparameters α . Somit ergibt sich die regularisierte Zielfunktion $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$ aus

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \|\mathbf{w}\|_1 + J(\mathbf{w}; \mathbf{X}, \mathbf{y}), \quad (7.19)$$

mit dem entsprechenden Gradienten (eigentlich Subgradienten)

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \text{sign}(\mathbf{w}) + \nabla_{\mathbf{w}} J(\mathbf{X}, \mathbf{y}; \mathbf{w}), \quad (7.20)$$

wobei $\text{sign}(\mathbf{w})$ einfach das elementweise angewandte Vorzeichen von \mathbf{w} ist.

In Gleichung 7.20 erkennen wir sofort, dass die Auswirkung der L^1 -Regularisierung sich deutlich von der der L^2 -Regularisierung unterscheidet. Insbesondere ist klar, dass der Beitrag der Regularisierung zum Gradienten nicht länger linear mit jedem w_i skaliert, sondern dass es sich um einen konstanten Faktor mit einem Vorzeichen gleich $\text{sign}(w_i)$ handelt. Eine Folge dieser Form des Gradienten ist, dass wir nicht unbedingt saubere algebraische Lösungen für die quadratischen Approximationen von $J(\mathbf{X}, \mathbf{y}; \mathbf{w})$ sehen, wie dies in der L^2 -Regularisierung der Fall war.

² Wie bei der L^2 -Regularisierung könnten wir die Parameter zu einem Wert regularisieren, der nicht gleich Null ist, nämlich einem Parameterwert $\mathbf{w}^{(o)}$. In diesem Fall würde die L^1 -Regularisierung den Term $\Omega(\boldsymbol{\theta}) = \|\mathbf{w} - \mathbf{w}^{(o)}\|_1 = \sum_i |w_i - w_i^{(o)}|$ einführen.

Unser einfaches lineares Modell weist eine quadratische Kostenfunktion auf, die wir durch ihre Taylorreihe repräsentieren können. Alternativ könnten wir uns vorstellen, dass es sich um eine gekürzte Taylorreihe handelt, die die Kostenfunktion eines ausgeklügelteren Modells approximiert. Der Gradient ergibt sich in diesem Fall aus

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \quad (7.21)$$

wobei \mathbf{H} wiederum die Hesse-Matrix von J bezüglich \mathbf{w} an der Stelle \mathbf{w}^* ist.

Da der L^1 -Strafterm für eine ganz allgemeine Hesse-Matrix keine sauberen algebraischen Ausdrücke zulässt, treffen wir die weitere vereinfachende Annahme, dass die Hesse-Matrix diagonal ist, $\mathbf{H} = \text{diag}([H_{1,1}, \dots, H_{n,n}])$, wobei alle $H_{i,i} > 0$ sind. Diese Annahme gilt, sofern die Daten des linearen Regressionsproblems vorverarbeitet wurden und dabei jegliche Korrelation zwischen den Eingabemerkmale entfernt wurde, beispielsweise mittels Hauptkomponentenanalyse (engl. *principal components analysis*, PCA).

Unsere quadratische Approximation der L^1 -regularisierten Zielfunktion wird in eine Summe über die Parameter zerlegt:

$$\hat{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = J(\mathbf{w}^*; \mathbf{X}, \mathbf{y}) + \sum_i \left[\frac{1}{2} H_{i,i} (\mathbf{w}_i - \mathbf{w}_i^*)^2 + \alpha |\mathbf{w}_i| \right]. \quad (7.22)$$

Für das Problem der Minimierung dieser approximativen Kostenfunktion gibt es eine analytische Lösung (für jede Dimension i) der folgenden Form:

$$w_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}. \quad (7.23)$$

Betrachten wir den Fall $w_i^* > 0$ für alle i . Es gibt zwei mögliche Ergebnisse:

1. Den Fall, in dem $w_i^* \leq \frac{\alpha}{H_{i,i}}$ ist. Hier ist der optimale Wert von w_i unter der regularisierten Zielfunktion einfach $w_i = 0$. Dies tritt ein, weil der Beitrag von $J(\mathbf{w}; \mathbf{X}, \mathbf{y})$ zur regularisierten Zielfunktion $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$ in Richtung i von der L^1 -Regularisierung quasi erdrückt wird, wodurch der Wert von w_i auf Null gedrückt wird.
2. Den Fall, in dem $w_i^* > \frac{\alpha}{H_{i,i}}$ ist. In diesem Fall verschiebt die Regularisierung den optimalen Wert von w_i nicht auf Null, sondern verschiebt ihn um einen Abstand, der gleich $\frac{\alpha}{H_{i,i}}$ ist, in diese Richtung.

Etwas Ähnliches geschieht für $w_i^* < 0$; allerdings macht der L^1 -Strafterm w_i um $\frac{\alpha}{H_{i,i}}$ oder 0 weniger negativ.

Im Vergleich zu der L^2 -Regularisierung führt die L^1 -Regularisierung zu einer Lösung, die **dünnbesetzter** (engl. *more sparse*) ist. Dünnbesetzt bedeutet in diesem Zusammenhang, dass einige Parameter einen optimalen Wert von Null haben. Die dünne Besetzung (engl. *sparsity*) der L^1 -Regularisierung ist gegenüber der L^2 -Regularisierung ein qualitativ unterschiedliches Verhalten. Gleichung 7.13 ergab die Lösung \tilde{w} für die L^2 -Regularisierung. Wenn wir diese Gleichung unter der Annahme einer diagonalen und positiv definiten Hesse-Matrix \mathbf{H} (die wir in unserer Analyse der L^1 -Regularisierung eingeführt haben) erneut betrachten, stellen wir fest, dass $\tilde{w}_i = \frac{H_{i,i}}{H_{i,i} + \alpha} w_i^*$ ist. Ist w_i^* von Null verschieden, dann bleibt \tilde{w}_i von Null verschieden. Das zeigt, dass die L^2 -Regularisierung nicht zu dünnbesetzten Parametern führt, dies aber bei der L^1 -Regularisierung durchaus möglich ist, wenn α nur groß genug ist.

Die durch die L^1 -Regularisierung eingeführte dünne Besetzung wurde umfassend als Mechanismus zur **Merkmalsauswahl** verwendet. Die Merkmalsauswahl vereinfacht ein Machine-Learning-Problem durch Auswahl einer Teilmenge aus den verfügbaren Merkmalen. Insbesondere integriert das bekannte LASSO-Modell (engl. *least absolute shrinkage and selection operator model*) (Tibshirani, 1995) einen L^1 -Strafterm mit einem linearen Modell und einer Kostenfunktion nach der Methode der kleinsten Quadrate. Der L^1 -Strafterm führt dazu, dass eine Teilmenge der Gewichte Null wird, die entsprechenden Merkmale also vermutlich ungestraft verworfen werden können.

In Abschnitt 5.6.1 haben Sie erfahren, dass viele Regularisierungsverfahren als bayessche MAP-Inferenz (maximum a posteriori inference) betrachtet werden können und vor allem die L^2 -Regularisierung einer bayesschen MAP-Inferenz mit einer normalverteilten A-priori-Wahrscheinlichkeit der Gewichte entspricht. Bei der L^1 -Regularisierung wird der Strafterm $\alpha\Omega(\mathbf{w}) = \alpha \sum_i |w_i|$ verwendet, um eine Kostenfunktion äquivalent zum Log-a-priori-Term zu regularisieren, die durch bayessche MAP-Inferenz maximiert wird, wenn die A-priori-Annahme eine isotrope Laplace-Verteilung (Gleichung 3.26) über $\mathbf{w} \in \mathbb{R}^n$ ist:

$$\log p(\mathbf{w}) = \sum_i \log \text{Laplace}(w_i; 0, \frac{1}{\alpha}) = -\alpha \|\mathbf{w}\|_1 + n \log \alpha - n \log 2. \quad (7.24)$$

Für das Lernen mittels Maximierung bezüglich \mathbf{w} können wir die Terme $\log \alpha - n \log 2$ ignorieren, da sie nicht von \mathbf{w} abhängig sind.

7.2 Norm-Strafterme als Optimierung unter Nebenbedingungen

Gegeben sei eine mittels Parameter-Norm-Strafterm regularisierte Kostenfunktion:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha \Omega(\boldsymbol{\theta}). \quad (7.25)$$

Laut Abschnitt 4.4 können wir eine Funktion, die Bedingungen unterworfen ist, durch Konstruieren einer allgemeinen Lagrange-Funktion minimieren, die aus der ursprünglichen Zielfunktion und einer Reihe von Straftermen besteht. Jeder Strafterm ist das Produkt aus einem Koeffizienten (dem sogenannten Karush-Kuhn-Tucker- oder KKT-Multiplikator) und einer Funktion, die angibt, ob die Bedingung erfüllt ist. Um $\Omega(\boldsymbol{\theta})$ auf einen kleineren Wert als eine Konstante k einzuschränken, könnten wir eine allgemeine Lagrange-Funktion

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k) \quad (7.26)$$

konstruieren. Die Lösung des Problems unter Nebenbedingungen ergibt sich aus

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha). \quad (7.27)$$

Wie in Abschnitt 4.4 beschrieben, erfordert die Lösung des Problems $\boldsymbol{\theta}$ und auch α zu modifizieren. Abschnitt 4.5 enthält ein ausgearbeitetes Beispiel der linearen Regression mit einer L^2 -Bedingung. Viele unterschiedliche Verfahren sind denkbar – einige verwenden das Gradientenabstiegsverfahren, andere analytische Lösungen bei einem Gradienten von Null –, aber in jedem Fall muss α zunehmen, wenn $\Omega(\boldsymbol{\theta}) > k$ ist, und abnehmen, wenn $\Omega(\boldsymbol{\theta}) < k$ ist. Jedes positive α regt $\Omega(\boldsymbol{\theta})$ zum Abnehmen an. Der optimale Wert α^* führt dazu, dass $\Omega(\boldsymbol{\theta})$ abnimmt, aber nicht so sehr, dass $\Omega(\boldsymbol{\theta})$ kleiner als k wird.

Um die Auswirkung der Bedingung zu verdeutlichen, können wir α^* festhalten und das Problem als Funktion von $\boldsymbol{\theta}$ betrachten:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \arg \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha^* \Omega(\boldsymbol{\theta}). \quad (7.28)$$

Das entspricht exakt dem regularisierten Trainingsproblem beim Minimieren von \tilde{J} . Wir können uns einen Parameter-Norm-Strafterm somit als Aufrelegung einer Bedingung an die Gewichte vorstellen. Ist Ω die L^2 -Norm, dann ist die Bedingung an die Gewichte, dass sie in einer L^2 -Kugel liegen. Ist Ω die L^1 -Norm, dann ist die Bedingung an die Gewichte, dass sie in

einem Bereich mit eingeschränkter L^1 -Norm liegen. Normalerweise kennen wir die Größe des eingeschränkten Bereichs für den Weight Decay mit dem Koeffizienten α^* nicht, da der Wert von α^* nicht direkt etwas über den Wert von k aussagt. Prinzipiell können wir nach k umstellen, aber die Beziehung zwischen k und α^* hängt von der Form von J ab. Obwohl wir die exakte Größe des eingeschränkten Bereichs nicht kennen, können wir sie grob durch Erhöhen oder Verringern von α steuern – dabei nimmt dieser Bereich zu oder ab. Je größer α , desto kleiner der eingeschränkte Bereich. Je kleiner α , desto größer der Bereich.

Manchmal möchten wir explizite Bedingungen anstelle von Straftermen einsetzen. Wie in Abschnitt 4.4 gezeigt, können wir Algorithmen wie das stochastische Gradientenabstiegsverfahren so ändern, dass ein Abstiegsschritt auf $J(\theta)$ erfolgt, um dann θ wieder auf den nächstgelegenen Punkt zu projizieren, der $\Omega(\theta) < k$ erfüllt. Das ist nützlich, wenn wir eine ungefähre Vorstellung vom passenden Wert k haben und keine Zeit mit der Suche nach dem Wert von α verbringen möchten, der diesem k entspricht.

Ein anderer Grund für die Verwendung expliziter Bedingungen und der Rückprojektion anstelle von Bedingungen mit Straftermen ist, dass Strafterme dazu führen können, dass nichtkonvexe Optimierungsverfahren in einem lokalen Minimum für ein kleines θ stecken bleiben. Beim Trainieren neuronaler Netze zeigt sich das meist durch mehrere »tote Einheiten«. Dabei handelt es sich um Einheiten, die nur einen geringen Beitrag zum Verhalten der vom Netz erlernten Funktion leisten, da die Gewichte, die hinein- oder hinausgelangen, allesamt sehr klein sind. Beim Trainieren mit einem Strafterm für die Norm der Gewichte können sich diese Konfigurationen als lokal optimal erweisen, obwohl es möglich ist, J durch größere Gewichte stark zu reduzieren. Durch Rückprojektion implementierte explizite Bedingungen können in diesen Fällen sehr viel besser funktionieren, da sie die Gewichte nicht dazu anregen, sich dem Ursprung anzunähern. Durch Rückprojektion implementierte explizite Bedingungen zeigen nur einen Effekt, wenn die Gewichte groß werden und versuchen, den eingeschränkten Bereich zu verlassen.

Letztlich können explizite Bedingungen mit Rückprojektion nützlich sein, da sie dem Optimierungsverfahren eine gewisse Stabilität verleihen. Bei hohen Lernraten sind positive Feedbackschleifen denkbar, in denen große Gewichte zu großen Gradienten führen, die wiederum eine starke Anpassung der Gewichte nach sich ziehen. Wenn derartige Anpassungen die Größe der Gewichte stetig erhöhen, entfernt sich θ rasant vom Ursprung, bis es zu einem numerischen Überlauf kommt. Explizite Bedingungen mit Rückprojektion verhindern, dass diese Feedbackschleife die Größe der Ge-

wichte unbegrenzt anwachsen lässt. *Hinton et al.* (2012c) empfehlen die Verwendung von Bedingungen in Kombination mit einer hohen Lernrate, um eine rapide Exploration des Parameterraums bei gleichzeitiger Beibehaltung einer gewissen Stabilität zu ermöglichen.

Vor allem empfehlen *Hinton et al.* (2012c) das von *Srebro und Shraibman* (2005) vorgestellte Verfahren: Bedingungen für die Norm jeder Spalte der Gewichtsmatrix einer Schicht im neuronalen Netz festlegen, statt Bedingungen für die Frobenius-Norm der gesamten Gewichtsmatrix aufzustellen. Die Bedingungen für die Norm in jeder einzelnen Spalte verhindern sehr hohe Gewichte für die einzelnen verdeckten Einheiten. Wenn wir diese Bedingung im Rahmen einer Lagrange-Funktion in einen Strafterm umwandeln würden, wäre diese dem L^2 -Weight-Decay ähnlich, allerdings mit einem separaten KKT-Multiplikator für die Gewichte der einzelnen verdeckten Einheiten. Jeder dieser KKT-Multiplikatoren würde getrennt voneinander dynamisch angepasst, damit jede verdeckte Einheit die Bedingung erfüllt. In der Praxis werden Einschränkungen der Spaltennorm stets als explizite Bedingung mit Rückprojektion umgesetzt.

7.3 Regularisierung und unterbestimmte Probleme

In einigen Fällen ist eine ordentliche Definition von Machine-Learning-Problemen nur mithilfe der Regularisierung möglich. Viele lineare Modelle im Machine Learning, darunter auch die lineare Regression und die PCA (engl. *principal components analysis*, dt. *Hauptkomponentenanalyse*), sind von der Umkehrung der Matrix $\mathbf{X}^\top \mathbf{X}$ abhängig. Das ist nicht möglich, wenn $\mathbf{X}^\top \mathbf{X}$ singulär ist. Diese Matrix kann immer dann singulär sein, wenn die datengenerierende Verteilung in einer Richtung tatsächlich keine Varianz aufweist oder wenn in einer Richtung keine Varianz *beobachtet* wird, da es weniger Beispiele (Zeilen in \mathbf{X}) als Eingabemerkmale (Spalten in \mathbf{X}) gibt. In diesem Fall entsprechen viele Arten der Regularisierung der alternativen Umkehrung von $\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I}$. Diese regularisierte Matrix ist garantiert invertierbar.

Solche linearen Probleme weisen geschlossene Lösungen (engl. *closed form solutions*) auf, wenn die relevante Matrix invertierbar ist. Ein Problem ohne geschlossene Lösung kann auch unterbestimmt sein. Ein Beispiel ist die logistische Regression bei einem Problem, dessen Klassen linear separierbar sind. Wenn ein Gewichtsvektor \mathbf{w} eine optimale Klassifizierung erzielen kann, führt auch $2\mathbf{w}$ zu einer optimalen Klassifizierung und einer höheren Regularisierung.

ren Likelihood. Ein iteratives Optimierungsverfahren wie das stochastische Gradientenabstiegsverfahren führt zu einem fortwährenden Wachstum des Betrags von w , das theoretisch nie endet. In der Praxis erreicht eine numerische Implementierung des Gradientenabstiegsverfahrens irgendwann ausreichend große Gewichte, um einen numerischen Überlauf zu verursachen; an diesem Punkt hängt das weitere Verhalten davon ab, welchen Umgang die Programmierer mit nicht reellen Zahlen vorgesehen haben.

Die meisten Arten der Regularisierung können die Konvergenz iterativer Verfahren für unterbestimmte Probleme garantieren. Zum Beispiel führt Weight Decay dazu, dass das Gradientenabstiegsverfahren die Gewichte nicht mehr erhöht, wenn die Steigung der Likelihood gleich dem Koeffizienten für den Weight Decay ist.

Das Konzept der Regularisierung zum Lösen unterbestimmter Probleme gibt es auch außerhalb des Machine Learnings. Es lässt sich zum Beispiel für einige grundlegende Problemstellungen der linearen Algebra nutzen.

In Abschnitt 2.9 haben wir gezeigt, dass unterbestimmte lineare Gleichungen mithilfe der Moore-Penrose-Pseudoinversen gelöst werden können. Sie erinnern sich gewiss an eine Definition der Pseudoinversen \mathbf{X}^+ einer Matrix \mathbf{X} , nämlich

$$\mathbf{X}^+ = \lim_{\alpha \searrow 0} (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top. \quad (7.29)$$

Jetzt erkennen wir, dass in Gleichung 7.29 eine lineare Regression mit Weight Decay erfolgt. Vor allem ist Gleichung 7.29 der Grenzwert für Gleichung 7.17, da der Regularisierungskoeffizient auf Null sinkt. Wir können also davon ausgehen, dass die Pseudoinverse unterbestimmte Probleme mittels Regularisierung stabilisiert.

7.4 Erweitern des Datensatzes

Damit ein Machine-Learning-Modell besser generalisieren kann, ist eine größere Trainingsdatenmenge von großem Nutzen. Leider ist die Datenmenge in der Praxis eingeschränkt. Um dieses Problem zu umgehen, können wir künstliche Daten erzeugen und zur Trainingsdatenmenge hinzufügen. Für einige Machine-Learning-Aufgaben ist das recht einfach.

Das gilt vor allem bei der Klassifizierung. Ein Klassifikator muss eine komplizierte hochdimensionale Eingabe x zu einer einzelnen Kategorie y zusammenfassen. Er muss also in erster Linie invariant gegenüber einer großen Anzahl von Transformationen sein. Wir können neue (x, y) -Paare

ganz einfach durch Transformieren der x -Eingangsdaten unserer Trainingsdatenmenge erzeugen.

Dieser Ansatz funktioniert bei vielen anderen Aufgaben nicht so leicht. Zum Beispiel lassen sich neue künstliche Daten für eine Dichteschätzung kaum erzeugen, wenn wir das Problem nicht bereits gelöst haben.

Das Erweitern des Datensatzes ist bei einer bestimmten Art von Klassifizierungsproblem besonders wirkungsvoll, nämlich bei der Objekterkennung. Bilder sind hochdimensional und enthalten enorm viele Faktoren der Variation, von denen sich eine Menge problemlos simulieren lassen. Zum Beispiel kann das Verschieben von Bildern um einige Pixel in jeder Richtung die Fähigkeit zur Generalisierung deutlich steigern, sogar wenn das Modell bereits mithilfe der Faltungs- und Pooling-Verfahren aus Kapitel 9 so konzipiert wurde, dass es teilweise verschiebungsinvariant ist. Viele andere Operationen wie das Rotieren oder Skalieren von Bildern haben sich ebenfalls als recht effektiv erwiesen.

Allerdings dürfen die Transformationen nicht dazu führen, dass die korrekte Klasse geändert wird. In der optischen Zeichenerkennung muss zum Beispiel die Unterscheidung zwischen den Buchstaben »b« und »d« sowie den Ziffern »6« und »9« gewährleistet bleiben. Spiegelungen an der Horizontalen oder Drehungen um 180° sind demnach nicht geeignet, um den Datensatz zu erweitern.

Es gibt auch Transformationen, gegenüber denen unsere Klassifikatoren invariant sein sollen, die sich aber nicht leicht ausführen lassen. So lässt sich eine Rotation außerhalb der Ebene nicht als einfache geometrische Manipulation der Eingabepixel vornehmen.

Das Erweitern des Datensatzes ist auch in der Spracherkennung nützlich (*Jaitly und Hinton, 2013*).

Auch das Einbringen von Rauschen in ein neuronales Netz (*Sietsma und Dow, 1991*) ist eine Art der Datenanreicherung. Für viele Klassifizierungsaufgaben und sogar einige Regressionsaufgaben muss die Aufgabe auch dann noch lösbar sein, wenn ein geringfügiges Zufallsrauschen für die Eingangsdaten hinzukommt. Neuronale Netze erweisen sich jedoch nicht als nicht sehr robust gegenüber Rauschen (*Tang und Eliasmith, 2010*). Um die Robustheit neuronaler Netze zu steigern, können sie mit zufallsverrauschten Eingangsdaten trainiert werden. Das Hinzufügen von Rauschen ist Teil einiger Algorithmen für unüberwachtes Lernen, zum Beispiel des Denoising Autoencoders (*Vincent et al., 2008*). Das funktioniert auch, wenn das Rauschen auf die verdeckten Einheiten angewandt wird; das entspricht dem Erweitern des Datensatzes auf mehreren Abstraktionsebenen. *Poole et al.*

(2014) haben kürzlich gezeigt, dass dieser Ansatz sich als extrem wirkungsvoll erweisen kann, solange die Stärke des Rauschens vorsichtig eingestellt wird. *Dropout*, ein leistungsfähiges Regularisierungsverfahren, dem wir uns in Abschnitt 7.12 widmen, kann als Verfahren zum Konstruieren neuer Eingangsdaten durch die *Multiplikation* mit Rauschen verstanden werden.

Beim Vergleich von Benchmark-Ergebnissen im Machine Learning müssen wir unbedingt die Auswirkungen erweiterter Datensätze beachten. Häufig können in mühevoller Kleinarbeit entwickelte Erweiterungsverfahren den Generalisierungsfehler einer Machine-Learning-Technik erheblich verringern. Um die Leistung von zwei Machine-Learning-Algorithmen miteinander zu vergleichen, müssen kontrollierte Experimente durchgeführt werden. Achten Sie beim Vergleich der Machine-Learning-Algorithmen A und B darauf, dass für beide dasselbe fein abgestimmte Erweiterungsverfahren für den Datensatz verwendet wurde. Wenn Algorithmus A ohne Datenanreicherung schlecht abschneidet, Algorithmus B dagegen in Verbindung mit diversen synthetischen Transformationen der Eingangsdaten gut, sind die synthetischen Transformationen vermutlich für die Leistungssteigerung verantwortlich, und nicht Algorithmus B. Manchmal lässt sich die Tatsache, ob ein Experiment ordnungsgemäß kontrolliert wurde, nur subjektiv beurteilen. So ist das Einfügen von Rauschen durch den Machine-Learning-Algorithmus in die Eingangsdaten eine Art Erweiterung des Datensatzes. Normalerweise werden allgemein anwendbare Operationen (wie das Hinzufügen von normalverteiltem Rauschen zu den Eingangsdaten) als Teil des Machine-Learning-Algorithmus betrachtet, wohingegen Operationen, die spezifisch für einen Anwendungsbereich sind (zum Beispiel das zufällige Beschneiden eines Bildes) als separate vorbereitende Schritte betrachtet werden.

7.5 Robustheit gegen Rauschen

Abschnitt 7.4 hat das Verrauschen der Eingangsdaten als Methode für das Erweitern des Datensatzes vorgestellt. Bei einigen Modellen entspricht das Hinzufügen von Rauschen mit infinitesimaler Varianz auf Eingabeseite des Modells dem Auferlegen eines Strafterms für die Norm der Gewichte (*Bishop*, 1995a,b). Im allgemeinen Fall müssen Sie sich bewusst machen, dass ein Hinzufügen von Rauschen viel mehr als nur eine Abnahme der Parameter nach sich ziehen kann, insbesondere wenn das Rauschen den verdeckten Einheiten zugefügt wird. Das Verrauschen der verdeckten Einheiten ist von so großer Bedeutung, dass es eine eigenständige Betrachtung rechtfertigt;

der in Abschnitt 7.12 beschriebene Dropout-Algorithmus ist die wichtigste Entwicklung dieses Ansatzes.

Im Rahmen der Modellregularisierung wurde das Rauschen auch zu den Gewichte hinzugefügt. Dieses Verfahren wurde vor allem bei RNNs genutzt (Jim et al., 1996; Graves, 2011). Sie lässt sich als stochastische Implementierung der bayesschen Inferenz über die Gewichte betrachten. Beim bayesschen Lernen werden die Modellgewichte als unsicher betrachtet und dies wird als eine Wahrscheinlichkeitsverteilung dargestellt, die diese Unsicherheit widerspiegelt. Das Hinzufügen von Rauschen zu den Gewichten ist ein praxistauglicher stochastischer Weg zur Darstellung dieser Unsicherheit.

Das Verrauschen der Gewichte lässt sich auch als (unter gewissen Voraussetzungen) gleichwertig zu einer klassischeren Art der Regularisierung betrachten, wodurch die Stabilität der zu erlernenden Funktion gefördert wird. Betrachten Sie eine Regression, bei der eine Funktion $\hat{y}(\mathbf{x})$ trainiert werden soll, die eine Menge von Merkmalen \mathbf{x} einem Skalar zuordnet, und zwar mithilfe der Kostenfunktion nach Methode der kleinsten Quadrate zwischen den Modellvorhersagen $\hat{y}(\mathbf{x})$ und den wahren Werten y :

$$J = \mathbb{E}_{p(x,y)} \left[(\hat{y}(\mathbf{x}) - y)^2 \right]. \quad (7.30)$$

Die Trainingsdatenmenge besteht aus m mit Labels gekennzeichneten Beispielen $\{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})\}$.

Wir gehen jetzt davon aus, dass wir jeder Repräsentation der Eingangsdaten auch eine zufällige Störung $\epsilon_W \sim \mathcal{N}(\epsilon; \mathbf{0}, \eta \mathbf{I})$ der Gewichte der Netze beifügen. Angenommen, es gibt ein Standard-MLP (mehrschichtiges Perzeptron) mit l Schichten. Wir schreiben für das gestörte Modell $\hat{y}_{\epsilon_W}(\mathbf{x})$. Trotz des Hinzufügens von Rauschen möchten wir noch immer den quadratischen Fehler der Ausgabe des Netzes minimieren. Die Zielfunktion wird daher zu

$$\tilde{J}_W = \mathbb{E}_{p(x,y,\epsilon_W)} \left[(\hat{y}_{\epsilon_W}(\mathbf{x}) - y)^2 \right] \quad (7.31)$$

$$= \mathbb{E}_{p(x,y,\epsilon_W)} \left[\hat{y}_{\epsilon_W}^2(\mathbf{x}) - 2y\hat{y}_{\epsilon_W}(\mathbf{x}) + y^2 \right]. \quad (7.32)$$

Für kleine η entspricht die Minimierung von J mit zu Gewichten hinzugefügtem Rauschen (mit Kovarianz $\eta \mathbf{I}$) der Minimierung von J mit einem zusätzlichen Regularisierungsterm: $\eta \mathbb{E}_{p(x,y)} [\|\nabla_W \hat{y}(\mathbf{x})\|^2]$. Diese Art der Regularisierung regt die Parameter dazu an, Bereiche des Parameterraums aufzusuchen, in denen kleine Störungen der Gewichte nur einen recht kleinen Einfluss auf die Ausgabe haben. Anders ausgedrückt: Das Modell wird in Bereiche gedrückt, in denen es relativ unempfindlich gegenüber kleinen Variationen der Gewichte ist, sodass nicht einfach Minima-Punkte gefunden

werden, sondern Minima, die von ebenen Bereichen umgeben sind (*Hochreiter und Schmidhuber, 1995*). Im vereinfachten Fall der linearen Regression (wo zum Beispiel gilt $\hat{y}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$) vereinfacht sich zu dieser Regularisierungsterm zu $\eta \mathbb{E}_{p(\mathbf{x})} [\|\mathbf{x}\|^2]$, was keine Funktion von Parametern darstellt und somit nicht zum Gradienten von \tilde{J}_W bezüglich des Modellparameters beiträgt.

7.5.1 Hinzufügen von Rauschen

Die meisten Datensätze weisen eine gewisse Anzahl von Fehlern in den y -Labeln auf. Ist y ein Fehler, kann sich das Maximieren von $\log p(y | \mathbf{x})$ als nachteilig erweisen. Eine Möglichkeit, dies zu verhindern, besteht in einer expliziten Modellierung des Rauschens für die Label. Wir können zum Beispiel davon ausgehen, dass für eine kleine Konstante ϵ das Label y der Trainingsdatenmenge mit einer Wahrscheinlichkeit von $1 - \epsilon$ korrekt ist und ansonsten alle anderen möglichen Label korrekt sein können. Diese Annahme lässt sich problemlos analytisch in die Kostenfunktion einbinden, sodass kein explizites Ziehen von verrauschten Beispielen erforderlich ist. Zum Beispiel regularisiert die **Label-Glättung** (engl. *label smoothing*) ein Modell anhand einer softmax-Funktion mit k Ausgabewerten, indem die harten Klassifizierungsziele 0 und 1 durch die Zielwerte $\frac{\epsilon}{k-1}$ bzw. $1 - \epsilon$ ersetzt werden. Anschließend kann der Standard-Kreuzentropieverlust mit diesen weichen Zielvorgaben verwendet werden. Das Maximum Likelihood Learning mit einem softmax-Klassifikator und harten Zielvorgaben konvergiert möglicherweise niemals – die softmax ist nicht in der Lage, eine Wahrscheinlichkeit von genau 0 oder genau 1 vorherzusagen, sodass die Funktion immer größere Gewichte lernt und so bis in alle Ewigkeit immer extremere Vorhersagen macht. Dieses Szenario lässt sich mithilfe von anderen Regularisierungsstrategien wie Weight Decay verhindern. Die Label-Glättung hat den Vorteil, dass das Streben nach festen Wahrscheinlichkeiten vermieden wird, ohne auf die korrekten Klassifizierung zu verzichten. Dieses Verfahren wird seit den 1980ern verwendet und ist in modernen neuronalen Netzen nach wie vor verbreitet (*Szegedy et al., 2015*).

7.6 Halb-überwachtes Lernen

Im Paradigma des halb-überwachten Lernens werden sowohl Beispiele ohne Label aus $P(\mathbf{x})$ als auch mit Labeln gekennzeichnete Beispiele aus $P(\mathbf{x}, \mathbf{y})$ zum Schätzen von $P(\mathbf{y} | \mathbf{x})$ oder Vorhersagen von \mathbf{y} anhand von \mathbf{x} verwendet.

Im Deep-Learning-Kontext bezeichnet das halb-überwachte Lernen üblicherweise das Lernen einer Repräsentation $\mathbf{h} = f(\mathbf{x})$. Das Ziel ist es, eine Repräsentation zu lernen, die alle Beispiele aus derselben Klasse auf eine ähnliche Art repräsentiert. Unüberwachtes Lernen kann nützliche Hinweise zum Gruppieren von Beispielen im Darstellungsraum geben. Beispiele, die im Eingaberaum eng beieinander liegen (Clustering), müssen ähnlichen Repräsentationen zugeordnet werden. Ein linearer Klassifikator im neuen Raum kann in vielen Fällen eine bessere Fähigkeit zur Generalisierung erzielen (*Belkin und Niyogi*, 2002; *Chapelle et al.*, 2003). Eine seit langer Zeit bestehende Variante dieses Ansatzes ist die Anwendung der Hauptkomponentenanalyse als vorbereitender Schritt vor der Anwendung eines Klassifikators (auf die projizierten Daten).

Statt unterschiedliche unüberwachte und überwachte Komponenten im Modell einzusetzen, lassen sich Modelle konstruieren, in denen ein generatives Modell von entweder $P(\mathbf{x})$ oder $P(\mathbf{x}, \mathbf{y})$ Parameter mit einem diskriminativen Modell von $P(\mathbf{y} | \mathbf{x})$ teilt. Anschließend kann das überwachte Kriterium $-\log P(\mathbf{y} | \mathbf{x})$ gegen das unüberwachte oder generative Kriterium (z. B. $-\log P(\mathbf{x})$ oder $-\log P(\mathbf{x}, \mathbf{y})$) abgewogen werden. Das generative Kriterium drückt dann eine bestimmte Form von vorheriger Annahme über die Lösung der Problemstellung des überwachten Lernens aus (*Lasserre et al.*, 2006), nämlich dass die Struktur $P(\mathbf{x})$ mit der Struktur $P(\mathbf{y} | \mathbf{x})$ auf eine Art verbunden ist, die in der gemeinsamen Parametrisierung zum Ausdruck kommt. Durch Kontrollieren des Anteils des generativen Kriteriums im Gesamtkriterium lässt sich ein besserer Kompromiss als mit einem rein generativen oder rein diskriminativen Trainingskriterium finden (*Lasserre et al.*, 2006; *Larochelle und Bengio*, 2008).

Salakhutdinov und Hinton (2008) beschreiben ein Verfahren zum Erlernen der Kernel-Funktion einer für die Regression genutzten Kernel-Maschine, in dem der Einsatz von Beispielen ohne Label für die Modellierung von $P(\mathbf{x})$ zu einer deutlichen Verbesserung von $P(\mathbf{y} | \mathbf{x})$ führt.

Chapelle et al. (2006) gehen näher auf das halb-überwachte Lernen ein.

7.7 Multitask Learning

Multitask Learning (*Caruana*, 1993) verbessert die Fähigkeit zur Generalisierung durch Zusammenfassen der Beispiele aus verschiedenen Aufgaben – das entspricht der Auferlegung weicher Bedingungen für die Parameter. So wie das Hinzufügen von Trainingsbeispielen die Modellparameter in Richtung von Werten drängt, die gut generalisieren, wenn ein Teil eines

Modells über mehrere Aufgaben hinweg gemeinsam genutzt wird, so ist dieser Teil des Modells eingeschränkt in Bezug auf gute Werte (vorausgesetzt, diese gemeinsame Nutzung ist gerechtfertigt), was wiederum häufig zu einer besseren Generalisierung führt.

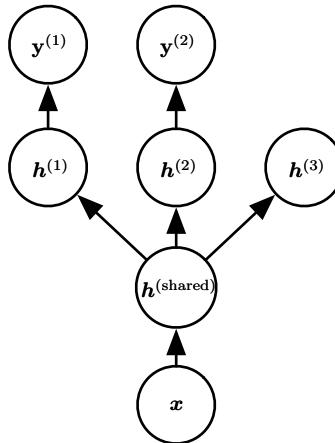


Abbildung 7.2: Multitask Learning lässt sich auf unterschiedliche Weise im Deep Learning einsetzen. Diese Abbildung zeigt den häufigen Fall, in dem die Aufgaben über eine gemeinsame Eingabe verfügen, aber verschiedene Zufallsvariablen für das Ziel beteiligt sind. Die unteren Schichten eines tiefen Netzes (es kann sich um ein überwachtes Feedforward-Netz oder ein Netz mit generativer Komponente und Abwärtspfeilen handeln) können für derartige Aufgaben gemeinsam genutzt werden. Die aufgabenspezifischen Parameter (die mit den Gewichten in bzw. aus $h^{(1)}$ und $h^{(2)}$ verknüpft sind) können zusätzlich zu denen, die zu einer gemeinsamen Repräsentation $h^{(\text{shared})}$ führen, erlernt werden. Es liegt die Annahme zugrunde, dass es einen gemeinsamen Pool von Faktoren gibt, mit denen sich die Variationen in der Eingabe x erklären lassen, während jede Aufgabe mit einer Teilmenge dieser Faktoren verknüpft ist. In diesem Beispiel wird außerdem angenommen, dass die verdeckten Einheiten $h^{(1)}$ und $h^{(2)}$ der oberen Schicht auf jede Aufgabe spezialisiert sind (also $y^{(1)}$ und $y^{(2)}$ vorhersagen), während eine Zwischenrepräsentation $h^{(\text{shared})}$ von allen Aufgaben gemeinsam genutzt wird. Im Kontext des unüberwachten Lernens ist es sinnvoll, dass einige der Faktoren der obersten Ebene mit keiner der Ausgabeaufgaben ($h^{(3)}$) verknüpft sind: Es handelt sich um die Faktoren, die einige der Eingabevariationen erklären, die aber nicht für die Vorhersage von $y^{(1)}$ oder $y^{(2)}$ relevant sind.

Abbildung 7.2 zeigt eine recht häufige Form des Multitask Learnings, bei der verschiedene überwachte Aufgaben (Vorhersage von $y^{(i)}$ anhand von x) dieselbe Eingabe x sowie eine Zwischenrepräsentation $h^{(\text{shared})}$ nutzen und einen gemeinsamen Pool von Faktoren erfassen. Das Modell kann

grundsätzlich in zwei Arten von Teilen und die zugehörigen Parameter aufgeteilt werden:

1. Aufgabenspezifische Parameter (die im Hinblick auf eine gute Fähigkeit zur Generalisierung nur von den Beispielen ihrer Aufgabe profitieren). Dies sind die oberen Schichten des neuronalen Netzes in Abbildung 7.2.
2. Generische Parameter, die allen Aufgaben gemein sind (und die von den zusammengefassten Daten aller Aufgaben profitieren). Dies sind die unteren Schichten des neuronalen Netzes in Abbildung 7.2.

Eine Verbesserung der Fähigkeit zur Generalisierung und des Generalisierungsfehlers (Baxter, 1995) lässt sich durch die gemeinsam genutzten bzw. geteilten Parameter (engl. *shared parameters*) erreichen, deren statistische Aussagekraft deutlich erhöht werden kann (im Verhältnis zur zunehmenden Anzahl der Beispiele für die geteilten Parameter, verglichen mit Modellen für nur eine Aufgabe). Dies geschieht natürlich nur, wenn einige Annahmen über die statistische Beziehung zwischen den diversen Aufgaben zutreffen, also wirklich eine aufgabenübergreifende Eigenschaft vorliegt.

Aus Sicht des Deep Learnings ist die vorrangige zugrunde liegende Überzeugung diese: *Unter den Faktoren, die eine Erklärung für die in den Daten der einzelnen Aufgaben beobachteten Variationen liefern, gibt es einige, die auf zwei oder mehr Aufgaben wirken.*

7.8 Früher Abbruch

Beim Trainieren großer Modelle mit einer ausreichenden Repräsentationskapazität für eine Überanpassung der Aufgabe stellen wir häufig fest, dass der Trainingsfehler im Laufe der Zeit stetig abnimmt, während der Validierungsdatenfehler irgendwann wieder ansteigt. Abbildung 7.3 zeigt dieses Verhalten, das regelmäßig zu beobachten ist.

Wir können also ein Modell mit einem besseren Validierungsdatenfehler (und somit hoffentlich einem besseren Fehler auf den Testdaten) erhalten, indem wir die Parametereinstellung zum Zeitpunkt des niedrigsten Validierungsdatenfehlers auswählen. Wann immer der Fehler der Validierungsdatenmenge kleiner wird, speichern wir eine Kopie der Modellparameter. Wenn der Trainingsalgorithmus terminiert, verwenden wir diese anstelle der letzten Parameter. Der Algorithmus terminiert, wenn sich gegenüber dem besten aufgezeichneten Validierungsfehler über eine zuvor festgelegte Anzahl der Iterationen keine Verbesserung der Parameter feststellen lässt. Dieses Verfahren wird formal in Algorithmus 7.1 beschrieben.

Algorithmus 7.1 Der Meta-Algorithmus für den frühen Abbruch zur Bestimmung der besten Trainingsdauer. Dieser Meta-Algorithmus ist eine generelle Vorgehensweise, die mit einer Vielzahl von Trainingsalgorithmen und Methoden zur Quantifizierung des Fehlers der Validierungsdatenmenge gut zurechtkommt.

n sei die Anzahl der Schritte zwischen den Evaluationen.

p sei die »Geduld«, also die Angabe, wie viele Male ein sich verschlechternder Validierungsdatenfehler beobachtet wird, bevor aufgegeben wird.

θ_o seien die Ausgangsparameter.

$\theta \leftarrow \theta_o$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

while $j < p$ **do**

Anpassen von θ durch Ausführen des Trainingsalgorithmus für n Schritte.

$i \leftarrow i + n$

$v' \leftarrow \text{Validierungsdatenfehler}(\theta)$

if $v' < v$ **then**

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

else

$j \leftarrow j + 1$

end if

end while

Die besten Parameter sind θ^* , die beste Anzahl der Trainingsschritte ist i^* .

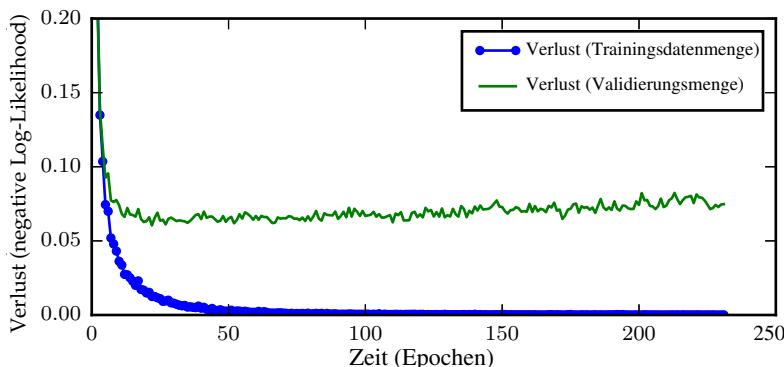


Abbildung 7.3: Die Lernkurven stellen die Veränderung des Verlusts der negativen Log-Likelihood im Laufe der Zeit dar (angegeben als Anzahl der Trainingsiterationen³ über den Datensatz, also als **Epochen**). In diesem Beispiel wird ein Maxout-Netz anhand der MNIST-Sammlung trainiert. Beachten Sie, wie das Trainingsziel im Zeitverlauf stetig abnimmt, aber der durchschnittliche Verlust der Validierungsdatenmenge schließlich wieder ansteigt, sodass eine asymmetrische U-Kurve entsteht.

Dieses Verfahren wird als **früher Abbruch** (engl. *early stopping*) bezeichnet. Es dürfte sich um die gängigste Art der Regularisierung im Deep Learning handeln, weil hier Effektivität und Simplizität zusammenkommen.

Man kann sich den frühen Abbruch als sehr effizienten Algorithmus zur Hyperparameter-Auswahl vorstellen, bei der die Anzahl der Trainingsschritte nur einen weiteren Hyperparameter darstellt. In Abbildung 7.3 wird deutlich, dass dieser Hyperparameter eine U-förmige Leistungskurve für die Validierungsdatenmenge aufweist. Die meisten Hyperparameter zur Steuerung der Modellkapazität weisen diese Form auf (vgl. Abbildung 5.3). Beim frühen Abbruch steuern wir die tatsächliche Kapazität des Modells, indem wir die Anzahl der Schritte für die Anpassung der Trainingsdatenmenge vorgeben. Die meisten Hyperparameter müssen durch eine aufwendige Abfolge von Versuch und Irrtum ermittelt werden; dazu legen wir zu Beginn des Trainings einen Hyperparameter fest, lassen mehrere Trainingsschritte laufen und prüfen dann seine Auswirkung. Der Hyperparameter für die »Trainingsdauer« ist insofern einzigartig, als per definitionem in nur einem Trainingslauf mehrere Werte des Hyperparameters getestet werden. Der einzige signifikante Aufwand bei der automatischen Wahl dieses Hyperparameters anhand des frühen Abbruchs besteht darin, die Evaluation der Validierungsdatenmenge während des Trainings regelmäßig auszuführen. Idealerweise erfolgt dies

³ Eine Trainingsiteration ist ein vollständiger Durchlauf durch alle Beispiele im Datensatz.

parallel zum Training auf einem separaten Rechner, einer separaten CPU oder einer separaten GPU, also nicht auf dem Prozessor, der für das primäre Training verwendet wird. Stehen solche Ressourcen nicht zur Verfügung, kann der Aufwand für diese regelmäßigen Evaluierungen reduziert werden, indem die Validierungsdatenmenge im Vergleich zur Trainingsdatenmenge klein gewählt wird oder indem der Validierungsdatenfehler seltener evaluiert wird und man eine weniger genaue Schätzung der optimalen Trainingsdauer erhält.

Außerdem besteht beim frühen Abbruch die Notwendigkeit, eine Kopie der besten Parameter zu speichern, was zusätzlichen Aufwand bedeutet. Dieser Aufwand ist jedoch meist vernachlässigbar, da diese Parameter in einem langsameren und größeren Speichermedium abgelegt werden können. Denkbar wäre es z. B., für das Training den GPU-Speicher zu nutzen, die optimalen Parameter aber im RAM oder auf einem Datenträger zu speichern. Da die besten Parameter in unregelmäßigen Abständen gespeichert und während des Trainings nicht gelesen werden, wirken sich die gelegentlichen langsamten Schreibvorgänge kaum auf die Gesamttrainingsdauer aus.

Der frühe Abbruch ist eine geringfügige Art der Regularisierung, denn das zugrunde liegende Trainingsverfahren, die Zielfunktion oder die Menge der zulässigen Parameterwerte müssen fast nicht geändert werden. Dieser kann also eingesetzt werden, ohne sich negativ auf die Lerndynamik auszuwirken. Im Gegensatz zum Weight Decay – dieser darf nicht zu stark ausfallen, da das Netz ansonsten in einem schlechten lokalen Minimum festhängt und eine Lösung mit unerwünschten kleinen Gewichten vorliegt.

Der frühe Abbruch kann einzeln oder in Kombination mit anderen Regularisierungsverfahren genutzt werden. Selbst bei Regularisierungsverfahren, bei denen die Zielfunktion zugunsten einer besseren Fähigkeit zur Generalisierung geändert wird, kommt es selten dazu, dass die beste Generalisierung in einem lokalen Minimum des Trainingsziels stattfindet.

Für den frühen Abbruch wird eine Validierungsdatenmenge benötigt, d. h., es werden nicht alle Trainingsdaten in das Modell eingespeist. Um diese Daten bestmöglich zu nutzen, kann nach dem anfänglichen Training, bei dem der frühe Abbruch erfolgt ist, eine weitere Trainingseinheit erfolgen. Im zweiten zusätzlichen Trainingsschritt werden alle Trainingsdaten verwendet. Es gibt zwei grundlegende Vorgehensweisen für dieses zweite Training:

Bei der einen (Algorithmus 7.2) wird das Modell erneut initialisiert und mit allen Daten neu trainiert. Bei diesem zweiten Trainingslauf entspricht die Anzahl der Schritte dem im ersten Lauf mittels Früher-Abbruch-Verfahren ermittelten optimalen Wert. Dabei sind einige Feinheiten zu beachten. Zum

Algorithmus 7.2 Ein Meta-Algorithmus für den frühen Abbruch. Er bestimmt, wie lange das Training dauert, und führt es dann für alle Daten erneut durch.

Sei $\mathbf{X}^{(\text{train})}$ und $\mathbf{y}^{(\text{train})}$ die Trainingsdatenmenge.

Aufteilen von $\mathbf{X}^{(\text{train})}$ und $\mathbf{y}^{(\text{train})}$ in $(\mathbf{X}^{(\text{subtrain})}, \mathbf{X}^{(\text{valid})})$ beziehungsweise $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$.

Ausführen des Früher-Abbruch-Verfahrens (Algorithmus 7.1) ab einem zufälligen $\boldsymbol{\theta}$ mit $\mathbf{X}^{(\text{subtrain})}$ und $\mathbf{y}^{(\text{subtrain})}$ als Trainingsdaten sowie $\mathbf{X}^{(\text{valid})}$ und $\mathbf{y}^{(\text{valid})}$ als Validierungsdaten. Das Ergebnis ist i^* , die optimale Anzahl der Schritte.

Erneutes Setzen von $\boldsymbol{\theta}$ auf Zufallswerte.

Trainieren anhand von $\mathbf{X}^{(\text{train})}$ und $\mathbf{y}^{(\text{train})}$ für i^* Schritte.

Beispiel gibt es keine einfache Möglichkeit zu ermitteln, ob besser dieselbe Anzahl von Parameteranpassungen oder dieselbe Anzahl Durchläufe für den Datensatz erfolgen sollte. In der zweiten Trainingsrunde werden für jeden Durchlauf des Datensatzes mehr Parameteranpassungen benötigt, da die Trainingsdatenmenge größer ist.

Die andere Vorgehensweise für die Nutzung aller Daten besteht darin, die ermittelten Parameter aus der ersten Trainingsrunde beizubehalten und das Training *fortzusetzen*, allerdings anhand aller Daten. In dieser Phase wissen wir nicht länger, nach welcher Anzahl von Schritten wir aufhören sollten. Stattdessen können wir die durchschnittliche Verlustfunktion der Validierungsdatenmenge beobachten und das Training so lange fortsetzen, bis der Wert unter den des Ziels für die Trainingsdatenmenge fällt, an dem das Früher-Abbruch-Verfahren zuvor endete. Diese Vorgehensweise erspart den hohen Aufwand für ein erneutes Trainieren des Modells von Anfang an, verhält sich aber nicht genauso angemessen. Zum Beispiel könnte der Zielwert der Validierungsdatenmenge von der Zielfunktion niemals erreicht werden – es ist also nicht garantiert, dass dieses Verfahren terminiert. Das Verfahren ist formal in Algorithmus 7.3 dargestellt.

Der frühe Abbruch ist auch deshalb nützlich, weil er den Berechnungsaufwand für das Trainingsverfahren senkt. Neben dem offensichtlich geringeren Aufwand als Folge einer eingeschränkten Anzahl von Trainingsiterationen bietet dieser auch den Vorteil einer Regularisierung ohne zusätzliche Strafterme in der Kostenfunktion oder ohne die Berechnung der Gradienten solcher zusätzlichen Terme.

Algorithmus 7.3 Ein Meta-Algorithmus für den frühen Abbruch. Er bestimmt, ab welchem Zielwert wir mit der Überanpassung beginnen, und setzt dann das Training fort, bis dieser Wert erreicht ist.

Sei $\mathbf{X}^{(\text{train})}$ und $\mathbf{y}^{(\text{train})}$ die Trainingsdatenmenge.

Aufteilen von $\mathbf{X}^{(\text{train})}$ und $\mathbf{y}^{(\text{train})}$ in $(\mathbf{X}^{(\text{subtrain})}, \mathbf{X}^{(\text{valid})})$ beziehungsweise $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$.

Ausführen des Früher-Abbruch-Verfahrens (Algorithmus 7.1) ab einem zufälligen $\boldsymbol{\theta}$ mit $\mathbf{X}^{(\text{subtrain})}$ und $\mathbf{y}^{(\text{subtrain})}$ als Trainingsdaten sowie $\mathbf{X}^{(\text{valid})}$ und $\mathbf{y}^{(\text{valid})}$ als Validierungsdaten. Damit wird $\boldsymbol{\theta}$ angepasst.

$$\epsilon \leftarrow J(\boldsymbol{\theta}, \mathbf{X}^{(\text{subtrain})}, \mathbf{y}^{(\text{subtrain})})$$

while $J(\boldsymbol{\theta}, \mathbf{X}^{(\text{valid})}, \mathbf{y}^{(\text{valid})}) > \epsilon$ **do**

 Trainieren anhand von $\mathbf{X}^{(\text{train})}$ und $\mathbf{y}^{(\text{train})}$ für n Schritte.

end while

Funktionsweise des frühen Abbruchs als Regularisierer: Bisher haben wir ausgeführt, dass der frühe Abbruch tatsächlich ein Regularisierungsverfahren *ist*, aber wir haben diese Behauptung nur mithilfe von Lernkurven untermauert, in denen der Validierungsdatenfehler U-förmig ist. Doch mit welchem Mechanismus wird das Modell beim frühen Abbruch tatsächlich regularisiert? *Bishop* (1995a) und *Sjöberg und Ljung* (1995) argumentieren, dass der frühe Abbruch den Optimierungsprozess auf einen relativ kleinen Teil des Parameterraums in der Nähe des Ausgangsparameterwerts $\boldsymbol{\theta}_o$ eingeschränkt (vgl. Abbildung 7.4). Stellen Sie sich τ Optimierungsschritte (entsprechend τ Trainingsiterationen) und eine Lernrate von ϵ vor. Sie können das Produkt $\epsilon\tau$ als ein Maß der tatsächlichen Kapazität betrachten. Angenommen, der Gradient ist eingeschränkt, dann führt die Einschränkung der Anzahl der Iterationen und der Lernrate dazu, dass das Volumen des Parameterraums, das von $\boldsymbol{\theta}_o$ erreicht werden kann, eingeschränkt ist. In diesem Sinne verhält sich $\epsilon\tau$ so, als wäre es der Kehrwert des Koeffizienten, der beim Weight Decay eingesetzt wird.

Tatsächlich lässt sich zeigen, wie – im Falle eines einfachen linearen Modells mit quadratischer Fehlerfunktion und einfaches Gradientenabstiegsverfahren – das Früher-Abbruch-Verfahren der L^2 -Regularisierung entspricht.

Zum Vergleich mit der klassischen L^2 -Regularisierung untersuchen wir einen einfachen Fall, in dem nur lineare Gewichte als Parameter zum Einsatz kommen ($\boldsymbol{\theta} = \mathbf{w}$). Wir können die Kostenfunktion J mit einer quadratischen

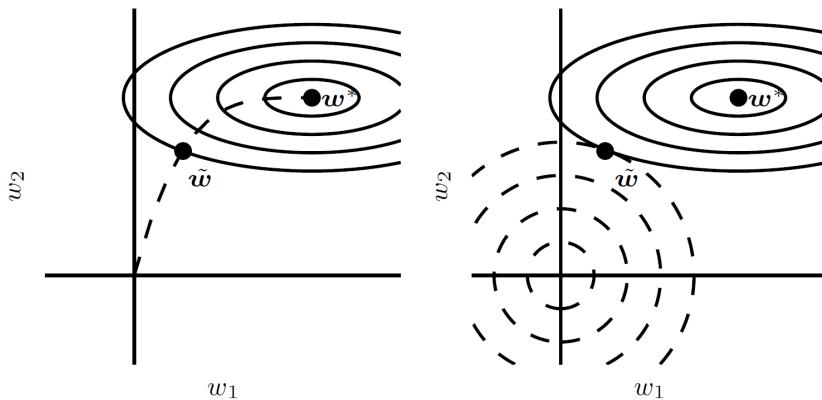


Abbildung 7.4: Darstellung der Auswirkungen des frühen Abbruchs. (*Links*) Die durchgezogenen Linien markieren die Ausdehnung der negativen Log-Likelihood. Die gestrichelte Linie zeigt die Trajektorie des stochastischen Gradientenabstiegsverfahrens, beginnend im Ursprung. Statt nun im Punkt \mathbf{w}^* abzubrechen, der den Aufwand minimiert, führt der frühe Abbruch dazu, dass die Trajektorie bereits im früheren Punkt $\tilde{\mathbf{w}}$ endet. (*Rechts*) Darstellung der Auswirkungen der L^2 -Regularisierung zum Vergleich. Die gestrichelten Kreise markieren die Ausdehnung des L^2 -Strafterms, durch die das Minimum des Gesamtaufwands näher am Ursprung liegt als das Minimum des nicht regularisierten Aufwands.

Approximation in der Umgebung des empirischen optimalen Werts der Gewichte \mathbf{w}^* modellieren:

$$\hat{J}(\boldsymbol{\theta}) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \quad (7.33)$$

wobei \mathbf{H} die Hesse-Matrix von J bezüglich \mathbf{w} an der Stelle \mathbf{w}^* ist. Wenn \mathbf{w}^* ein Minimum von $J(\mathbf{w})$ ist, ist \mathbf{H} positiv semidefinit. Im Rahmen einer lokalen Taylorreihen-Approximation ergibt sich der Gradient aus

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*). \quad (7.34)$$

Wir untersuchen die Trajektorie, die der Parametervektor während des Trainings einnimmt. Aus Gründen der Einfachheit setzen wir den Ausgangsparametervektor auf den Ursprung,⁴ $\mathbf{w}^{(0)} = \mathbf{0}$. Untersuchen wir nun

⁴ In neuronalen Netzen lassen sich Symmetriebrechungen zwischen verdeckten Einheiten nicht durch Initialisieren aller Parameter auf $\mathbf{0}$ erreichen (siehe Abschnitt 6.2). Allerdings ist das Argument für beliebige andere Anfangswerte $\mathbf{w}_{(0)}$ gültig.

das approximative Verhalten des Gradientenabstiegsverfahrens für J durch Analyse des Gradientenabstiegsverfahrens für \hat{J} :

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \epsilon \nabla_{\mathbf{w}} \hat{J}(\mathbf{w}^{(\tau-1)}) \quad (7.35)$$

$$= \mathbf{w}^{(\tau-1)} - \epsilon \mathbf{H}(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*), \quad (7.36)$$

$$\mathbf{w}^{(\tau)} - \mathbf{w}^* = (\mathbf{I} - \epsilon \mathbf{H})(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*). \quad (7.37)$$

Schreiben wir diesen Ausdruck im Raum der Eigenvektoren von \mathbf{H} neu, nutzen wir dabei die Eigenwertzerlegung von \mathbf{H} : $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$, wobei Λ eine Diagonalmatrix und \mathbf{Q} eine orthonormale Eigenvektorenbasis ist.

$$\mathbf{w}^{(\tau)} - \mathbf{w}^* = (\mathbf{I} - \epsilon \mathbf{Q}\Lambda\mathbf{Q}^\top)(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \quad (7.38)$$

$$\mathbf{Q}^\top(\mathbf{w}^{(\tau)} - \mathbf{w}^*) = (\mathbf{I} - \epsilon \Lambda)\mathbf{Q}^\top(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \quad (7.39)$$

Wenn $\mathbf{w}^{(0)} = 0$ und ϵ klein genug für $|1 - \epsilon\lambda_i| < 1$ gewählt ist, dann stellt sich die Trajektorie der Parameter während des Trainings nach τ Parameteranpassungen wie folgt dar:

$$\mathbf{Q}^\top \mathbf{w}^{(\tau)} = [\mathbf{I} - (\mathbf{I} - \epsilon \Lambda)^\tau] \mathbf{Q}^\top \mathbf{w}^*. \quad (7.40)$$

Jetzt kann der Ausdruck für $\mathbf{Q}^\top \tilde{\mathbf{w}}$ in Gleichung 7.13 zur L^2 -Regularisierung folgendermaßen umgeordnet werden:

$$\mathbf{Q}^\top \tilde{\mathbf{w}} = (\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{Q}^\top \mathbf{w}^*, \quad (7.41)$$

$$\mathbf{Q}^\top \tilde{\mathbf{w}} = [\mathbf{I} - (\Lambda + \alpha \mathbf{I})^{-1} \alpha] \mathbf{Q}^\top \mathbf{w}^*. \quad (7.42)$$

Ein Vergleich von Gleichung 7.40 mit Gleichung 7.42 zeigt, dass bei Wahl der Hyperparameter ϵ , α und τ mit

$$(\mathbf{I} - \epsilon \Lambda)^\tau = (\Lambda + \alpha \mathbf{I})^{-1} \alpha \quad (7.43)$$

die L^2 -Regularisierung und der frühe Abbruch als gleichwertig betrachtet werden können (zumindest für die quadratische Approximation der Zielfunktion). Wenn wir noch einen Schritt weitergehen und logarithmieren und die Reihenentwicklung für $\log(1 + x)$ verwenden, können wir schließen, dass – wenn alle λ_i klein sind (also $\epsilon\lambda_i \ll 1$ und $\lambda_i/\alpha \ll 1$) – gilt:

$$\tau \approx \frac{1}{\epsilon\alpha}, \quad (7.44)$$

$$\alpha \approx \frac{1}{\tau\epsilon}. \quad (7.45)$$

Unter diesen Voraussetzungen spielt also die Anzahl der Trainingsiterationen τ eine antiproportionale Rolle zum L^2 -Regularisierungsparameter und die Inverse von $\tau\epsilon$ eine Rolle für den Koeffizienten für den Weight Decay.

Parameterwerte für die Richtungen der signifikanten Krümmung (der Zielfunktion) werden weniger stark regularisiert als für Richtungen mit einer geringeren Krümmung. Das bedeutet für den frühen Abbruch natürlich, dass Parameter, die einer Richtung mit signifikanter Krümmung entsprechen, im Gegensatz zu Parametern, die Richtungen mit weniger starker Krümmung entsprechen, zu einem relativ frühen Lernen tendieren.

Die Herleitungen in diesem Abschnitt haben gezeigt, dass eine Trajektorie der Länge τ in einem Punkt endet, der einem Minimum der L^2 -regularisierten Zielfunktion entspricht. Der frühe Abbruch ist natürlich mehr als eine reine Einschränkung der Trajektorienlänge. Vielmehr beinhaltet er üblicherweise eine Überwachung des Validierungsdatenfehlers, um die Trajektorie an einem bestimmten guten Punkt im Raum enden zu lassen. Daher bietet der frühe Abbruch gegenüber dem Weight Decay den Vorteil, dass er automatisch die korrekte Stärke der Regularisierung ermittelt, während der Weight Decay viele Trainingsläufe mit unterschiedlichen Werten für den Hyperparameter benötigt.

7.9 Parameter Tying und Parameter Sharing

Bisher haben wir in diesem Kapitel den Einsatz von Bedingungen oder Straftermen für Parameter stets bezüglich eines unveränderlichen Bereichs oder Punkts behandelt. So belegt die L^2 -Regularisierung (oder Weight Decay) Modellparameter mit einem Strafterm, wenn diese von einem festen Wert 0 abweichen. Manchmal müssen wir unser Vorwissen über geeignete Werte für die Modellparameter jedoch auf andere Art und Weise ausdrücken. Vielleicht sind die exakten Sollparameterwerte unbekannt, aber wir wissen aus Erfahrung im Bereich der Domänen- und Modellarchitektur, dass bestimmte Abhängigkeiten zwischen den Modellparametern existieren müssen.

Häufig möchten wir für bestimmte Parameter angeben, dass diese nahe beieinander liegen sollen. Ein Beispiel: Wir haben zwei Modelle für dieselbe Klassifizierungsaufgabe (mit derselben Menge von Klassen), aber leicht unterschiedlichen Eingabeverteilungen. Formal handelt es sich um das Modell A mit den Parametern $\mathbf{w}^{(A)}$ und das Modell B mit den Parametern $\mathbf{w}^{(B)}$. Die beiden Modelle ordnen die Eingabe zwei unterschiedlichen, aber verwandten Ausgaben zu: $\hat{y}^{(A)} = f(\mathbf{w}^{(A)}, \mathbf{x})$ und $\hat{y}^{(B)} = g(\mathbf{w}^{(B)}, \mathbf{x})$.

Wenn diese Aufgaben nun ähnlich genug sind (womöglich mit ähnlichen Eingabe- und Ausgabeverteilungen), sind wir vielleicht der Überzeugung, dass die Modellparameter nahe beieinander liegen sollten: $\forall i$, $w_i^{(A)}$ soll nahe bei $w_i^{(B)}$ liegen. Wir können diese Angabe per Regularisierung nutzen. Dazu verwenden wir einen Parameter-Norm-Strafterm der Form $\Omega(\mathbf{w}^{(A)}, \mathbf{w}^{(B)}) = \|\mathbf{w}^{(A)} - \mathbf{w}^{(B)}\|_2^2$. Im Beispiel ist es ein L^2 -Strafterm, aber es gibt auch andere Möglichkeiten.

Dieser Ansatz wurde von *Lasserre et al.* (2006) vorgeschlagen, die die Parameter eines Modells (das als Klassifikator in einem überwachten Paradigma trainiert wurde) so regularisiert haben, dass sie den Parametern eines anderen Modells (das in einem unüberwachten Paradigma trainiert wurde) nahe sind (um die Verteilung der beobachteten Eingangsdaten zu erfassen). Die Architekturen wurden so konstruiert, dass viele der Parameter im Klassifizierungsmodell mit den entsprechenden Parametern im unüberwachten Modell gepaart werden konnten.

Obwohl ein Parameter-Norm-Strafterm eine Möglichkeit darstellt, Parameter so zu regularisieren, dass Sie einander nahe sind, werden stattdessen meist Bedingungen eingesetzt: *Erzwingen der Gleichheit von Parametermengen*. Dieses Regularisierungsverfahren wird häufig als **Parameter Sharing** (dt. *Teilen der Parameter* bzw. *gemeinsame Nutzung der Parameter*) bezeichnet, da wir die einzelnen Modelle oder Modellkomponenten so betrachten, als würden sie dieselbe Parametermenge miteinander teilen. Ein wesentlicher Vorteil von Parameter Sharing gegenüber der Regularisierung für eine Parameter Nähe (mittels Norm-Strafterm) besteht darin, dass nur eine Teilmenge der Parameter (die eindeutige Menge) im Speicher gehalten werden muss. In gewissen Modellen – zum Beispiel im CNN – kann das zu einer deutlichen Reduzierung des Speicherbedarfs für das Modell führen.

7.9.1 CNNs

Parameter Sharing wird besonders gern und häufig in **CNNs** für Computer Vision eingesetzt.

Natürliche Bilder weisen viele statistische Eigenschaften auf, die invariant gegenüber einer Verschiebung sind. So zeigt das Foto einer Katze noch immer eine Katze, wenn der Bildinhalt um ein Pixel nach rechts verschoben wird. In CNNs wird diese Eigenschaft berücksichtigt, indem Parameter für mehrere Bildpositionen geteilt werden. Dasselbe Merkmal (eine verdeckte Einheit mit identischen Gewichten) wird für verschiedene Positionen der Eingabe berechnet. So können mit demselben Suchalgorithmus für Katzen diese

unabhängig davon gefunden werden, ob sie in Spalte i oder in Spalte $i + 1$ des Fotos auftauchen.

Parameter Sharing führt in CNNs zu einer deutlich geringeren Anzahl eindeutiger Modellparameter und somit zu wesentlich größeren Netzen, ohne dass hierfür die Trainingsdaten in gleichem Maß zunehmen müssten. Dies ist eines der besten Beispiele dafür, wie das Wissen über den jeweiligen Anwendungsbereich wirkungsvoll in die Netzarchitektur einfließen kann.

Wir gehen in Kapitel 9 genauer auf CNNs ein.

7.10 Dünnbesetzte Repräsentationen

Beim Weight Decay werden die Modellparameter direkt mit einem Strafterm belegt. Eine andere Vorgehensweise besteht darin, die Aktivierungen der Einheiten in einem neuronalen Netz mit einem Strafterm zu belegen, der dazu führen soll, dass deren Aktivierungen gehemmt werden.

In Abschnitt 7.1.2 haben wir bereits gezeigt, wie die L^1 -Bestrafung zu einer dünnbesetzten Parametrisierung (engl. *sparse parametrization*) führt – also viele der Parameter zu Null (oder nahezu Null) werden. Andererseits beschreibt die dünnbesetzte Repräsentation eine Repräsentation, in der viele Elemente Null (oder nahezu Null) sind. Eine vereinfachte Darstellung dieser Unterscheidung lässt sich für die lineare Regression zeigen:

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix} \quad (7.46)$$

$\mathbf{y} \in \mathbb{R}^m \qquad \mathbf{A} \in \mathbb{R}^{m \times n} \qquad \mathbf{x} \in \mathbb{R}^n$

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix} \quad (7.47)$$

$\mathbf{y} \in \mathbb{R}^m \qquad \mathbf{B} \in \mathbb{R}^{m \times n} \qquad \mathbf{h} \in \mathbb{R}^n$

Der erste Ausdruck ist ein Beispiel für ein dünnbesetztes parametrisiertes lineares Regressionsmodell. Der zweite zeigt eine lineare Regression mit einer dünnbesetzten Repräsentation \mathbf{h} der Daten \mathbf{x} . \mathbf{h} ist also eine Funktion von \mathbf{x} , die in gewissem Sinne die Informationen von \mathbf{x} darstellt, allerdings mit einem dünnbesetzten Vektor.

Die repräsentative Regularisierung (engl. *representational regularization*) wird durch einige der Mechanismen erreicht, die wir bereits für die Parameter-Regularisierung eingesetzt haben.

Die Norm-Strafterm-Regularisierung von *Repräsentationen* erfolgt, indem zur Verlustfunktion J ein Norm-Strafterm für die Repräsentation hinzugefügt wird. Diesen Strafterm schreiben wir $\Omega(\mathbf{h})$. Wie zuvor notieren wir die regularisierte Verlustfunktion als \tilde{J} :

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha \Omega(\mathbf{h}), \quad (7.48)$$

wobei $\alpha \in [0, \infty)$ den relativen Beitrag des Norm-Strafterms gewichtet; größere Werte von α bedeuten also eine stärkere Regularisierung.

So wie ein L^1 -Strafterm für die Parameter zu einer dünnen Besetzung der Parameter führt, führt ein L^1 -Strafterm für die Elemente der Repräsentation zu einer dünnbesetzten Repräsentation: $\Omega(\mathbf{h}) = \|\mathbf{h}\|_1 = \sum_i |h_i|$. Natürlich ist der L^1 -Strafterm nur eine Möglichkeit eines Strafterms, der zu einer dünnbesetzten Repräsentation führen kann. Andere Möglichkeiten verwenden den von einer studentschen t -Verteilung der Repräsentation abgeleiteten Strafterm (*Olshausen und Field*, 1996; *Bergstra*, 2011) und Kullback-Leibler-Divergenz-Strafterm (*Larochelle und Bengio*, 2008), die insbesondere für Repräsentationen mit Elementen nützlich sind, die zwingend in dem Intervall $[0,1]$ liegen müssen. *Lee et al.* (2008) und *Goodfellow et al.* (2009) zeigen Beispiele für Verfahren auf Basis einer Regularisierung der durchschnittlichen Aktivierung über mehrere Beispiele, $\frac{1}{m} \sum_i \mathbf{h}^{(i)}$, die nahe einem Zielwert liegen sollen, wie zum Beispiel ein Vektor, dessen Einträge alle ,01 lauten.

Andere Ansätze erreichen eine dünnbesetzte Repräsentation durch harte Bedingungen (engl. *hard constraint*) für die Aktivierungswerte. So codiert die **Orthogonal Matching Pursuit** (*Pati et al.*, 1993) eine Eingabe \mathbf{x} mit der Repräsentation \mathbf{h} , die das Optimierungsproblem unter Nebenbedingungen

$$\arg \min_{\mathbf{h}, \|\mathbf{h}\|_0 < k} \|\mathbf{x} - \mathbf{W}\mathbf{h}\|^2 \quad (7.49)$$

löst, wobei $\|\mathbf{h}\|_0$ die Anzahl der von Null verschiedenen Einträge von \mathbf{h} angibt. Dieses Problem lässt sich effizient lösen, wenn die Bedingung an \mathbf{W}

ist, orthogonal zu sein. Das Verfahren wird oft OMP- k genannt, wobei der Wert k die Anzahl der zulässigen von Null verschiedenen Merkmale angibt. *Coates und Ng* (2011) zeigte, dass OMP-1 für tiefe Architekturen ein sehr effektiver Merkmalsextraktor sein kann.

Für praktisch jedes Modell, das verdeckte Einheiten aufweist, lässt sich eine dünne Besetzung erreichen. In diesem Buch zeigen wir viele Beispiele für eine Regularisierung mit dünner Besetzung in unterschiedlichen Kontexten.

7.11 Bagging und andere Ensemblemethoden

Bagging (ein Kurzwort für **Bootstrap Aggregating**) ist ein Verfahren zur Reduzierung des Generalisierungsfehlers durch Kombinieren mehrerer Modelle (*Breiman*, 1994). Dazu werden mehrere unterschiedliche Modelle separat voneinander trainiert, bevor alle Modelle gemeinsam über die Ausgabe für Testbeispiele abstimmen. Das ist ein Beispiel für ein allgemeines Verfahren im Machine Learning, das als **Modellmittelung** oder auch *Model Averaging* bezeichnet wird. Man spricht bei diesen Verfahren von **Ensemblemethoden**.

Die Modellmittelung funktioniert, weil verschiedene Modelle bei einer Testdatenmenge normalerweise nicht alle dieselben Fehler machen.

Hier ein Beispiel für k Regressionsmodelle: Angenommen, jedes Modell macht bei jedem Beispiel einen Fehler ϵ_i (die Fehler werden aus einer multivariaten Normalverteilung mit Mittelwert gleich Null und den Varianzen $\mathbb{E}[\epsilon_i^2] = v$ und den Kovarianzen $\mathbb{E}[\epsilon_i \epsilon_j] = c$ als Stichproben gezogen). Dann beträgt der Fehler aus einer gemittelten Vorhersage aller Ensemblemodelle $\frac{1}{k} \sum_i \epsilon_i$. Der erwartete quadratische Fehler des Ensembleprädiktors beträgt

$$\mathbb{E} \left[\left(\frac{1}{k} \sum_i \epsilon_i \right)^2 \right] = \frac{1}{k^2} \mathbb{E} \left[\sum_i \left(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j \right) \right], \quad (7.50)$$

$$= \frac{1}{k} v + \frac{k-1}{k} c. \quad (7.51)$$

Wenn die Fehler perfekt korreliert sind und $c = v$ ist, sinkt der mittlere quadratische Fehler auf v und die Modellmittelung bietet keinerlei Nutzen. Wenn die Fehler perfekt unkorreliert sind und $c = 0$ ist, beträgt der erwartete quadratische Fehler des Ensembles lediglich $\frac{1}{k} v$. Der Erwartungswert für den quadratischen Fehler des Ensembles nimmt also linear mit der Ensemblegröße ab. Anders ausgedrückt: Im Mittel funktioniert das Ensemble mindestens genauso gut wie jedes seiner Teile. Sofern die Teile unabhängige Fehler machen, funktioniert das Ensemble deutlich besser als die Teile.

Unterschiedliche Ensemblemethoden konstruieren das Modellensemble auf unterschiedliche Weise. Zum Beispiel kann jeder Teil des Ensembles aus dem Training eines fundamental anderen Modelltyps mit unterschiedlichen Algorithmen oder Zielfunktionen entstehen. Bagging ist ein Verfahren, bei dem derselbe Modelltyp, derselbe Trainingsalgorithmus und dieselbe Zielfunktion mehrmals wiederverwendet werden können.

Beim Bagging werden auch k verschiedene Datensätze konstruiert. Jeder Datensatz weist dieselbe Anzahl von Beispielen wie der ursprüngliche Datensatz auf und wird gleichzeitig durch Ziehen von Stichproben mit Zurücklegen aus dem ursprünglichen Datensatz konstruiert. Die Wahrscheinlichkeit ist also hoch, dass jeder Datensatz einige Beispiele aus dem ursprünglichen Datensatz **nicht** enthält, dafür aber mehrere Duplikate (im Schnitt finden sich etwa zwei Drittel der Beispiele aus dem ursprünglichen Datensatz in der resultierenden Trainingsdatenmenge, sofern diese dieselbe Größe wie das Original aufweist). Anschließend wird das Modell i anhand des Datensatzes i trainiert. Die Unterschiede zwischen den enthaltenen Beispielen in jedem Datensatz führen zu Unterschieden zwischen den trainierten Modellen. Abbildung 7.5 zeigt ein Beispiel.

Neuronale Netze erreichen eine so breite Vielzahl verschiedener Lösungspunkte, dass die Modellmittelung sogar dann von Nutzen ist, wenn alle Modelle mit demselben Datensatz trainiert werden. Unterschiede bei der Zufallsinitialisierung, bei der zufälligen Auswahl von Mini-Batches, bei den Hyperparametern oder den Ergebnissen nichtdeterministischer Implementierungen der neuronalen Netze führen oft genug dazu, dass die einzelnen Teile des Ensembles partiell unabhängige Fehler machen.

Die Modellmittelung ist eine extrem leistungsfähige und zuverlässige Methode zur Reduzierung des Generalisierungsfehlers. Beim Benchmarking von Algorithmen für wissenschaftliche Abhandlungen wird für gewöhnlich von ihrem Einsatz abgeraten, da jeder Machine-Learning-Algorithmus von der Modellmittelung deutlich profitieren kann – zulasten eines höheren Rechenaufwands und Speicherbedarfs. Daher erfolgen Benchmark-Vergleiche in der Regel mit nur einem Modell.

Gewinner von Machine-Learning-Wettbewerben sind meist Verfahren, die eine Modellmittelung über Dutzende von Modellen durchführen. Ein prominentes Beispiel aus der jüngeren Vergangenheit ist der *Netflix Grand Prize* (Koren, 2009).

Nicht alle Verfahren zum Konstruieren von Ensembles zielen darauf ab, das Ensemble stärker als die einzelnen Modelle zu regularisieren. So konstruiert eine **Boosting** (Freund und Schapire, 1996b,a) genannte Methode

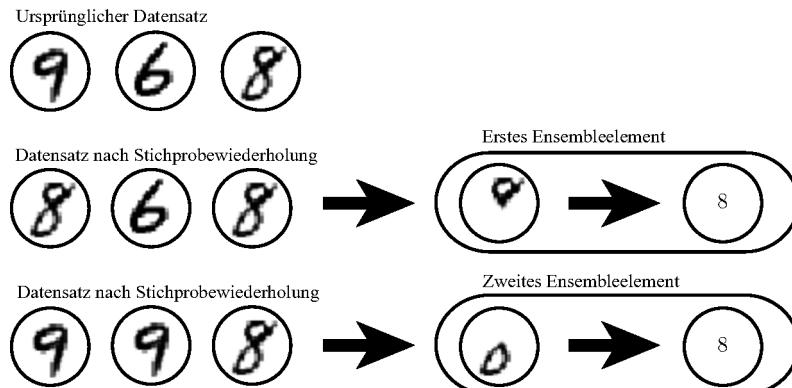


Abbildung 7.5: Eine bildhafte Darstellung der Funktionsweise von Bagging. Hier trainieren wir ein Erkennungsmodul für die Ziffer Acht anhand des oben abgebildeten Datensatzes mit den Elementen 8, 6 und 9. Nun erzeugen wir zwei unterschiedliche Datensätze mittels Stichprobewiederholung. Beim Bagging-Trainingsverfahren wird jeder der Datensätze durch Stichprobenentnahme mit Zurücklegen konstruiert. Der erste Datensatz lässt die 9 weg und wiederholt die 8. In diesem Datensatz lernt das Modul, dass eine Schleife in der oberen Hälfte der Ziffer einer 8 entspricht. Im zweiten Datensatz wird die 9 wiederholt, aber die 6 weggelassen. In diesem Fall lernt das Modul, dass eine Schleife in der unteren Hälfte der Ziffer einer 8 entspricht. Für sich allein genommen sind diese Klassifizierungen eher schwach, aber wenn wir die Ausgabe mitteln, funktioniert das Modul robust und gibt nur die erreichte Konfidenz aus, wenn beide Schleifen für die Ziffer 8 vorhanden sind.

ein Ensemble, das eine höhere Kapazität als seine Einzelmodelle aufweist. Boosting wurde zum Erstellen von Ensembles neuronaler Netze verwendet (*Schwenk und Bengio, 1998*), indem nach und nach weitere neuronale Netze zum Ensemble hinzugefügt wurden. Es wurde auch eingesetzt, um ein einzelnes neuronales Netz als Ensemble zu betrachten (*Bengio et al., 2006a*), wobei nach und nach verdeckte Einheiten zum Netz hinzugefügt wurden.

7.12 Dropout

Dropout (*Srivastava et al., 2014*) ist ein wenig rechenaufwendiges, aber leistungsstarkes Verfahren zur Regularisierung einer großen Modelfamilie. Vereinfacht ausgedrückt ist Dropout ein Verfahren zur praxistauglichen Umsetzung von Bagging für Ensembles mit sehr vielen großen neuronalen Netzen. Beim Bagging werden mehrere Modelle trainiert und für jedes Testbeispiel bewertet. Das wird unpraktisch, wenn jedes Modell ein großes neuronales Netz ist, da Training und Bewertung solcher Netze viel Zeit

und Speicherplatz benötigen. Häufig werden Ensembles mit fünf bis zehn neuronalen Netzen eingesetzt – Szegedy *et al.* (2014a) setzten sechs Netze ein, um die ImageNet Large Scale Visual Recognition Challenge (ILSVRC) zu gewinnen –, aber sobald es mehr werden, ist der Prozess schwer zu handhaben. Dropout bietet eine wenig aufwendige Approximation für das Trainieren und Bewerten eines auf Bagging basierenden (engl. *bagged*) Ensembles mit exponentiell vielen neuronalen Netzen.

Insbesondere trainiert Dropout das Ensemble, das aus allen Teilnetzen besteht, die durch Entfernen von Nicht-Ausgabeeinheiten aus einem zugrunde liegenden Basisnetz gebildet werden können (vgl. Abbildung 7.6). In den meisten modernen neuronalen Netzen, die auf einer Reihe von affinen Transformationen und Nichtlinearitäten basieren, können wir eine Einheit wirksam aus einem Netz entfernen, indem wir ihren Ausgabewert mit Null multiplizieren. Dieses Verfahren erfordert eine geringe Änderung bei bestimmten Modellen wie RBF-Netzen (engl. *radial basis function networks*), bei denen die Differenz zwischen dem Zustand der Einheit und einem Referenzwert gebildet wird. Wir stellen hier den Dropout-Algorithmus aus Gründen der Einfachheit als Multiplikation mit Null dar – aber er kann problemlos so geändert werden, dass er mit anderen Operationen funktioniert, die eine Einheit aus dem Netz entfernen.

Für das Lernen mit Bagging definieren wir k verschiedene Modelle, erzeugen k verschiedene Datensätze durch Ziehen von Stichproben aus der Trainingsdatenmenge mit Zurücklegen und trainieren schließlich Modell i anhand von Datensatz i . Dropout zielt darauf ab, den Prozess mit einer exponentiell großen Anzahl neuronaler Netze zu approximieren. Für das Trainieren mit Dropout setzen wir einen Lernalgorithmus auf Mini-Batch-Basis ein, der kleine Schritte macht, zum Beispiel das stochastische Gradientenabstiegsverfahren. Bei jedem Laden eines Beispiels in einen Mini-Batch wählen wir zufällig eine andere binäre Maske aus, die auf alle Eingabe- und verdeckten Einheiten im Netz angewandt wird. Die Maske für jede Einheit wird unabhängig von allen anderen ausgewählt. Die Wahrscheinlichkeit, einen Maskenwert von 1 zu auszuwählen (mit dem die Einheit einbezogen wird), ist ein Hyperparameter, der vor Beginn des Trainings festgelegt wird. Es handelt sich nicht um eine Funktion des aktuellen Werts der Modellparameter oder des Eingabebeispiels. Üblicherweise wird eine Eingabeeinheit mit der Wahrscheinlichkeit 0,8 und eine verdeckte Einheit mit der Wahrscheinlichkeit 0,5 einbezogen. Anschließend folgt der übliche Ablauf aus Forward-Propagation, Backpropagation und Aktualisierung des Gelernten. Abbildung 7.7 zeigt die Forward-Propagation mit Dropout.

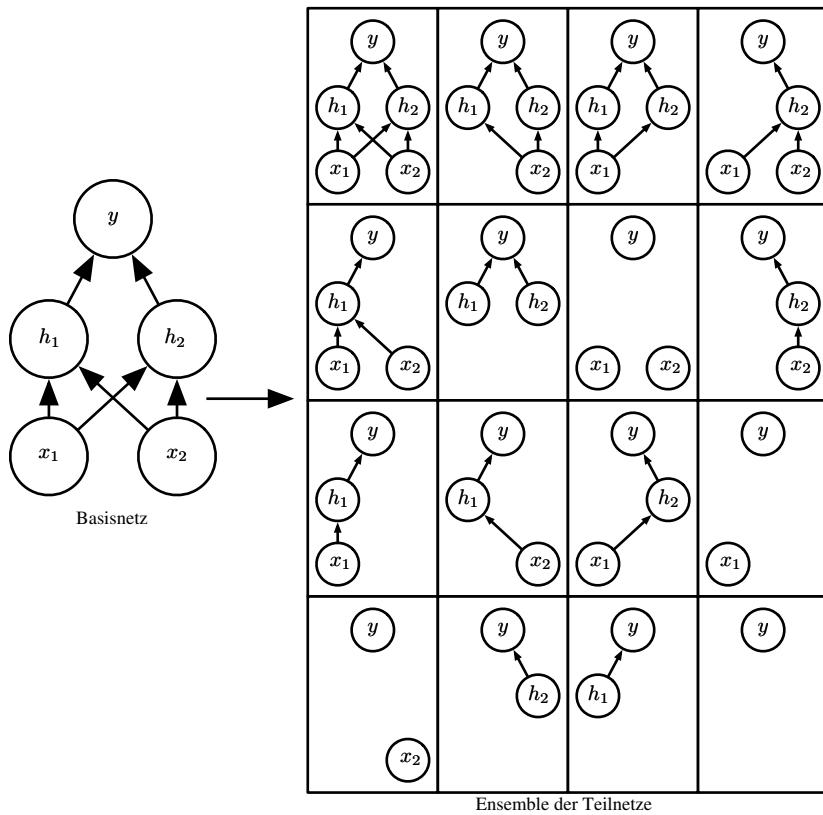


Abbildung 7.6: Dropout trainiert ein Ensemble, das aus allen Teilnetzen besteht, die durch Entfernen von Nicht-Ausgabeeinheiten (engl. *nonoutput units*) aus einem zugrunde liegenden Basisnetz erzeugt werden können. Wir beginnen hier mit einem Basisnetz mit zwei sichtbaren und zwei verdeckten Einheiten. Für diese vier Einheiten gibt es 16 mögliche Teilmengen. Die Abbildung zeigt alle 16 Teilnetze, die durch ein Dropout (Ausschalten, Entfernen) unterschiedlicher Teilmengen von Einheiten des Originalnetzes gebildet werden können. In diesem kleinen Beispiel weist ein Großteil der entstehenden Netze keine Eingabeeinheiten oder keinen Verbindungspfad zwischen der Eingabe und der Ausgabe auf. Bei Netzen mit breiteren Schichten wird dieses Problem bedeutungslos, denn die Wahrscheinlichkeit, beim Dropout alle möglichen Pfade zwischen Eingaben und Ausgaben zu verlieren, wird geringer.

Formal ausgedrückt gibt ein Maskenvektor $\boldsymbol{\mu}$ an, welche Einheiten einbezogen werden, und $J(\boldsymbol{\theta}, \boldsymbol{\mu})$ definiert den Aufwand des Modells, das durch die Parameter $\boldsymbol{\theta}$ und die Maske $\boldsymbol{\mu}$ definiert wird. Für das Dropout-Training muss nun $\mathbb{E}_{\boldsymbol{\mu}} J(\boldsymbol{\theta}, \boldsymbol{\mu})$ minimiert werden. Der Erwartungswert enthält exponentiell viele Terme, aber wir können einen erwartungstreuen Schätzwert seines Gradienten bestimmen, indem wir Werte von $\boldsymbol{\mu}$ als Stichproben ziehen.

Ein Training mit Dropout ist nicht ganz dasselbe wie ein Training mit Bagging. Beim Bagging sind die Modelle alle unabhängig. Beim Dropout nutzen sie gemeinsame Parameter, wobei jedes Modell eine andere Parameter-Teilmenge vom übergeordneten neuronalen Netz erbt. Dieses Parameter Sharing ermöglicht es, eine exponentielle Anzahl von Modellen mit überschaubarem Speicherplatz darzustellen. Beim Bagging wird jedes Modell auf Konvergenz zur jeweiligen Trainingsdatenmenge trainiert. Im Falle von Dropout werden die meisten Modelle gar nicht explizit trainiert – meist ist das Modell so groß, dass es innerhalb der Lebensdauer des Universums unmöglich wäre, alle möglichen Teilnetze als Stichproben zu ziehen. Stattdessen wird jeweils ein winziger Bruchteil der möglichen Teilnetze mit nur einem Schritt trainiert; das Parameter Sharing sorgt dafür, dass die restlichen Teilnetze zu guten Einstellungen für die Parameter gelangen. Dies sind die einzigen Unterschiede. Darüber hinaus folgt Dropout dem Bagging-Algorithmus. Zum Beispiel ist die Trainingsdatenmenge in jedem Teilnetz tatsächlich eine Teilmenge der ursprünglichen Trainingsdatenmenge, die durch Ziehen mit Zurücklegen erzeugt wurde.

Für eine Vorhersage muss ein auf Bagging basierendes Ensemble ein Votum jedes seiner Teile (Elemente) sammeln. Wir bezeichnen diesen Vorgang in diesem Kontext als **Inferenz**. Bisher war für unsere Beschreibung von Bagging und Dropout kein explizit probabilistisches Modell erforderlich. Jetzt gehen wir davon aus, dass die Aufgabe des Modells in der Ausgabe einer Wahrscheinlichkeitsverteilung besteht. Beim Bagging erzeugt jedes Modell i eine Wahrscheinlichkeitsverteilung $p^{(i)}(y | \mathbf{x})$. Die Vorhersage des Ensembles ergibt sich aus dem arithmetischen Mittel all dieser Verteilungen,

$$\frac{1}{k} \sum_{i=1}^k p^{(i)}(y | \mathbf{x}). \quad (7.52)$$

Beim Dropout definiert jedes anhand des Maskenvektors $\boldsymbol{\mu}$ definiertes Teilmodell eine Wahrscheinlichkeitsverteilung $p(y | \mathbf{x}, \boldsymbol{\mu})$. Das arithmetische Mittel aller Masken ergibt sich aus

$$\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y | \mathbf{x}, \boldsymbol{\mu}), \quad (7.53)$$

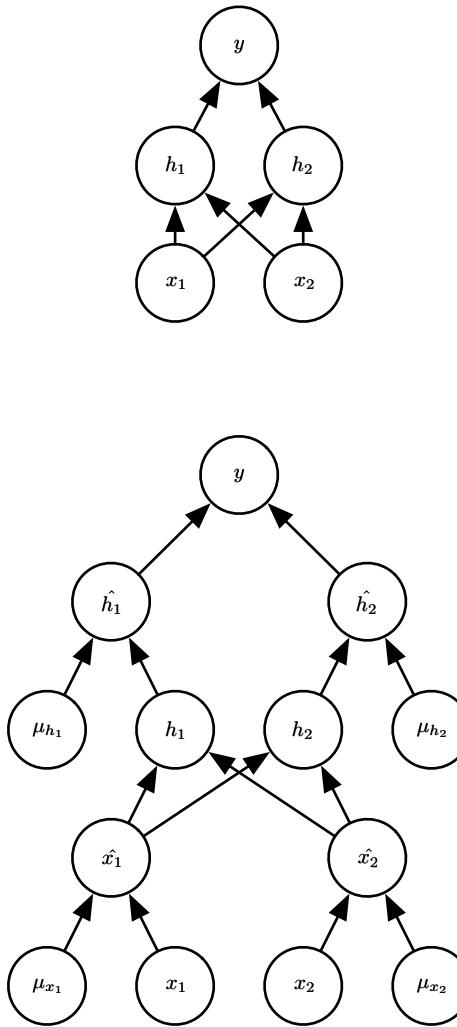


Abbildung 7.7: Ein Beispiel für die Forward-Propagation durch ein Feedforward-Netz mit Dropout. (*Oben*) In diesem Beispiel verwenden wir ein Feedforward-Netz mit zwei Eingabeeinheiten, einer verdeckten Schicht mit zwei verdeckten Einheiten und einer Ausgabeeinheit. (*Unten*) Für die Forward-Propagation mit Dropout wählen wir zufällig einen Vektor μ mit einem Eintrag für jede Eingabe- oder verdeckte Einheit im Netz aus. Die Einträge von μ sind binär und werden unabhängig voneinander als Stichproben gezogen. Die Wahrscheinlichkeit, dass ein Eintrag 1 ist, ist ein Hyperparameter – meist 0,5 für die verdeckten Schichten und 0,8 für die Eingabe. Jede Einheit im Netz wird mit der entsprechenden Maske multipliziert. Anschließend erfolgt die Forward-Propagation im restlichen Netz wie gewohnt. Das entspricht der zufälligen Auswahl eines der Teilnetze aus Abbildung 7.6 und dem Ausführen der Forward-Propagation darin.

wobei $p(\boldsymbol{\mu})$ die Wahrscheinlichkeitsverteilung ist, die zum Trainingszeitpunkt für die Stichprobenentnahme aus $\boldsymbol{\mu}$ verwendet wurde.

Da diese Summe eine exponentielle Anzahl von Termen enthält, ist sie nicht effizient berechenbar, sofern die Modellstruktur keine Vereinfachung ermöglicht. Bisher sind tiefe neuronale Netze nicht dafür bekannt, eine effizient durchführbare Vereinfachung zu ermöglichen. Stattdessen können wir die Inferenz mittels Stichprobenentnahme approximieren, indem wir die Ausgabe vieler Masken mitteln. Häufig sind bereits 10 bis 20 Masken für eine gute Leistung ausreichend.

Ein noch besserer Ansatz ermöglicht uns jedoch, eine gute Approximation an die Vorhersagen des gesamten Ensembles zu erzielen; hierfür wird nur eine Forward-Propagation benötigt. Dazu wechseln wir zur Verwendung des geometrischen Mittels anstelle des arithmetischen Mittels der von den Ensemblelementen vorhergesagten Verteilungen. *Warde-Farley et al.* (2014) geben Argumente und einen empirischen Nachweis dafür, dass das geometrische Mittel in diesem Zusammenhang vergleichbar dem arithmetischen Mittel funktioniert.

Das geometrische Mittel mehrerer Wahrscheinlichkeitsverteilungen ist nicht unbedingt eine Wahrscheinlichkeitsverteilung. Damit das Ergebnis in jedem Fall eine Wahrscheinlichkeitsverteilung ist, stellen wir die Bedingung auf, dass keines der Teilmodelle einem Ereignis eine Wahrscheinlichkeit von 0 zuweisen darf; außerdem normalisieren wir die resultierende Verteilung wieder. Die durch das geometrische Mittel direkt definierte, nicht normalisierte Wahrscheinlichkeitsverteilung ergibt sich aus

$$\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x}) = \sqrt[d]{\prod_{\boldsymbol{\mu}} p(y \mid \mathbf{x}, \boldsymbol{\mu})}, \quad (7.54)$$

wobei d die Anzahl der Einheiten ist, die entfernt werden dürfen. Wir verwenden hier eine Gleichverteilung über $\boldsymbol{\mu}$, um die Darstellung zu vereinfachen – aber auch Ungleichverteilungen sind möglich. Für die Vorhersagen müssen wir das Ensemble wieder normalisieren:

$$p_{\text{ensemble}}(y \mid \mathbf{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \mathbf{x})}. \quad (7.55)$$

Eine wesentliche Erkenntnis (*Hinton et al.*, 2012c) beim Dropout ist, dass wir p_{ensemble} durch Berechnen von $p(y \mid \mathbf{x})$ in einem Modell approximieren können: das Modell mit allen Einheiten, aber den von der Einheit i wegführenden Gewichten, multipliziert mit der Wahrscheinlichkeit, dass die Einheit i einbezogen wird. Diese Änderung soll von der Idee her den richtigen

Erwartungswert der Ausgabe dieser Einheit erfassen. Wir bezeichnen den Ansatz als **Inferenzregel zur Skalierung der Gewichte**. Es gibt bisher kein theoretisches Argument für die Genauigkeit dieser approximativen Inferenzregel in tiefen nichtlinearen Netzen, aber empirisch funktioniert sie sehr gut.

Da wir für eine Inklusionswahrscheinlichkeit (engl. *inclusion probability*) üblicherweise die Wahrscheinlichkeit von $\frac{1}{2}$ verwenden, führt die Regel zur Skalierung der Gewichte meist zu einer Division der Gewichte durch 2 am Ende des Trainings mit anschließender Verwendung des Modells wie gewohnt. Eine andere Möglichkeit, dieses Ergebnis zu erzielen, besteht im Multiplizieren der Zustände der Einheiten während des Trainings mit 2. Ungeachtet des Weges möchten wir, dass die erwartete Gesamteingabe einer Einheit zum Testzeitpunkt in etwa der erwarteten Gesamteingabe der Einheit zum Trainingszeitpunkt entspricht, obwohl zu Letzterem im Schnitt die Hälfte der Einheiten fehlt.

Bei vielen Modellklassen, die keine nichtlinearen verdeckten Einheiten aufweisen, ist die Inferenzregel zur Skalierung der Gewichte exakt. Betrachten wir als einfaches Beispiel einen softmax-Regressionsklassifikator mit n Eingangsvariablen, dargestellt durch den Vektor \mathbf{v} :

$$P(y = y | \mathbf{v}) = \text{softmax}(\mathbf{W}^\top \mathbf{v} + \mathbf{b})_y. \quad (7.56)$$

Wir können durch elementweises Multiplizieren der Eingabe mit einem Binärvektor d die Familie der Teilmodelle indizieren:

$$P(y = y | \mathbf{v}; \mathbf{d}) = \text{softmax}(\mathbf{W}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b})_y. \quad (7.57)$$

Der Ensembleprädiktor wird durch erneutes Normalisieren des geometrischen Mittels über alle Vorhersagen der Ensembleelemente definiert:

$$P_{\text{ensemble}}(y = y | \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(y = y | \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(y = y' | \mathbf{v})}, \quad (7.58)$$

mit

$$\tilde{P}_{\text{ensemble}}(y = y | \mathbf{v}) = \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} P(y = y | \mathbf{v}; \mathbf{d})}. \quad (7.59)$$

Dass die Regel zur Skalierung der Gewichte exakt ist, zeigt sich, wenn wir $\tilde{P}_{\text{ensemble}}$ vereinfachen:

$$\tilde{P}_{\text{ensemble}}(y = y | \mathbf{v}) = \sqrt[2^n]{\prod_{\mathbf{d} \in \{0,1\}^n} P(y = y | \mathbf{v}; \mathbf{d})} \quad (7.60)$$

$$= \sqrt[2^n]{\prod_{d \in \{0,1\}^n} \text{softmax}(\mathbf{W}^\top(\mathbf{d} \odot \mathbf{v}) + \mathbf{b})_y} \quad (7.61)$$

$$= \sqrt[2^n]{\prod_{d \in \{0,1\}^n} \frac{\exp(\mathbf{W}_{y,:}^\top(\mathbf{d} \odot \mathbf{v}) + b_y)}{\sum_{y'} \exp(\mathbf{W}_{y',:}^\top(\mathbf{d} \odot \mathbf{v}) + b_{y'})}} \quad (7.62)$$

$$= \frac{\sqrt[2^n]{\prod_{d \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top(\mathbf{d} \odot \mathbf{v}) + b_y)}}{\sqrt[2^n]{\prod_{d \in \{0,1\}^n} \sum_{y'} \exp(\mathbf{W}_{y',:}^\top(\mathbf{d} \odot \mathbf{v}) + b_{y'})}} \quad (7.63)$$

Da \tilde{P} normalisiert wird, können wir die Multiplikation durch Faktoren, die bezüglich y konstant sind, problemlos vernachlässigen:

$$\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{d \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top(\mathbf{d} \odot \mathbf{v}) + b_y)} \quad (7.64)$$

$$= \exp\left(\frac{1}{2^n} \sum_{d \in \{0,1\}^n} \mathbf{W}_{y,:}^\top(\mathbf{d} \odot \mathbf{v}) + b_y\right) \quad (7.65)$$

$$= \exp\left(\frac{1}{2} \mathbf{W}_{y,:}^\top \mathbf{v} + b_y\right). \quad (7.66)$$

Durch Einsetzen in Gleichung 7.58 erhalten wir einen softmax-Klassifikator mit den Gewichten $\frac{1}{2}\mathbf{W}$.

Die Regel zur Skalierung der Gewichte ist auch in anderen Zusammenhängen exakt, darunter in Regressionsnetzen mit bedingt normalen Ausgaben und in tiefen Netzen mit verdeckten Schichten ohne Nichtlinearitäten. Allerdings stellt die Regel bei tiefen Modellen mit Nichtlinearitäten nur eine Approximation dar. Obschon die Approximation nicht theoretisch charakterisiert wurde, funktioniert sie empirisch häufig gut. *Goodfellow et al.* (2013a) haben experimentell herausgefunden, dass die Approximation durch Skalierung der Gewichte (in Bezug auf die Korrektklassifikationsrate) besser als Monte-Carlo-Approximationen für den Ensembleprädiktor funktionieren können. Das gilt sogar dann noch, wenn die Monte-Carlo-Approximation Stichproben aus bis zu 1.000 Teilnetzen ziehen durfte. *Gal und Ghahramani* (2015) haben festgestellt, dass einige Modelle eine bessere Korrektklassifikationsrate mit zwanzig Stichproben und der Monte-Carlo-Approximation erzielen. Es scheint, dass die optimale Wahl der Inferenzapproximation aufgabenabhängig ist.

Srivastava et al. (2014) haben gezeigt, dass Dropout effektiver als andere rechnerisch wenig aufwendige Standardregularisierer ist, wie zum Beispiel

Weight Decay, Filterbedingungen oder geringfügige Aktivitätregularisierung (engl. *sparse activity regularization*). Dropout kann auch mit anderen Arten der Regularisierung kombiniert werden, um eine weitere Verbesserung zu erzielen.

Ein Vorteil von Dropout ist der geringe Berechnungsaufwand. Für Dropout werden im Training nur $O(n)$ Berechnungen pro Beispiel und Anpassung benötigt, um n zufällige Binärzahlen zu erzeugen und mit dem Zustand zu multiplizieren. Je nach Art der Implementierung wird ein Speicherplatz $O(n)$ zum Ablegen dieser Binärzahlen bis zur Backpropagation benötigt. Der Aufwand für den Einsatz von Inferenz im trainierten Modell ist pro Beispiel gleich hoch – ob nun Dropout verwendet wird oder nicht. Allerdings entsteht Aufwand für die einmalige Division der Gewichte durch 2, bevor die Inferenz auf die Beispiele angewandt wird.

Ein weiterer wesentlicher Vorteil von Dropout besteht darin, dass es die Art des Modells oder Trainingsverfahrens nur unwesentlich einschränkt. Es kommt mit nahezu allen Modellen gut zurecht, die eine verteilte Repräsentation einsetzen, und kann mittels stochastischen Gradientenabstiegsverfahrens trainiert werden. Dazu gehören neuronale Feedforward-Netze, probabilistische Modelle wie Restricted Boltzmann Machines (RBMs, dt. *eingeschränkte Boltzmann-Maschinen*) (Srivastava et al., 2014) sowie RNNs (Bayer und Osendorfer, 2014; Pascanu et al., 2014a). Viele weitere Verfahren zur Regularisierung vergleichbarer Leistung führen zu wesentlich strengeren Einschränkungen der Modellarchitektur.

Zwar ist der Aufwand für den Einsatz von Dropout pro Schritt in einem bestimmten Modell vernachlässigbar, im Gesamtsystem jedoch kann sich ein beträchtlicher Aufwand ergeben. Da Dropout ein Regularisierungsverfahren ist, reduziert es die tatsächliche Kapazität des Modells. Um dem entgegenzuwirken, müssen wir die Modellgröße erhöhen. Typischerweise ist der optimale Validierungsdatenfehler beim Einsatz von Dropout deutlich geringer, aber eben zum Preis eines deutlich größeren Modells und einer viel größeren Anzahl von Iterationen des Trainingsalgorithmus. Bei sehr großen Datensätzen führt die Regularisierung zu einer geringen Reduzierung des Generalisierungsfehlers. In diesen Fällen übersteigen der Berechnungsaufwand für Dropout und die größeren Modelle eventuell die Vorteile der Regularisierung.

Wenn nur sehr wenige mit Labels gekennzeichnete Trainingsbeispiele zur Verfügung stehen, ist Dropout weniger effektiv. Bayessche neuronale Netze (Neal, 1996) sind dem Dropout-Verfahren mit dem *Alternative Splicing*-Datensatz (Xiong et al., 2011) überlegen, der weniger als 5 000 Beispiele enthält

(Srivastava *et al.*, 2014). Sind weitere Datensätze ohne Label verfügbar, ist das unüberwachte Lernen von Merkmalen (engl. *unsupervised feature learning*) dem Dropout überlegen.

Wager *et al.* (2013) haben gezeigt, dass Dropout für die lineare Regression gleichwertig mit dem L^2 -Weight-Decay mit einem jeweils anderen Koeffizienten für den Weight Decay für jedes Eingabemerkmal ist. Die Größe des Koeffizienten für den Weight Decay für jedes Merkmal wird durch seine Varianz bestimmt. Ähnliche Ergebnisse sind auch bei anderen linearen Modellen festzustellen. In tiefen Modellen sind Dropout und Weight Decay nicht gleichwertig.

Die Stochastizität, die beim Training mit Dropout eingesetzt wird, ist keine Voraussetzung für den Erfolg des Ansatzes. Sie ist lediglich ein Mittel zur Approximation der Summe für alle Teilmodelle. Wang und Manning (2013) haben analytische Approximationen dieser Marginalisierung hergeleitet. Ihre Approximation, genannt **Fast Dropout**, führte aufgrund der verringerten Stochastizität bei der Gradientenberechnung zu einer schnelleren Konvergenz. Das Verfahren lässt sich auch zum Testzeitpunkt im Vergleich zu der Approximation mittels Skalierung der Gewichte als erstaunliche (aber auch rechenaufwendigere) Approximation an den Durchschnittswert aller Teilnetze einsetzen. Fast Dropout hat bei Problemen mit kleinen neuronalen Netzen nahezu die Leistungsfähigkeit des normalen Dropout-Verfahrens erreicht, aber noch nicht zu deutlichen Verbesserungen geführt. Auch wurde es noch nicht auf umfangreiche Problemstellungen angewandt.

Für den Regularisierungseffekt von Dropout ist Stochastizität nicht nur nicht erforderlich, sondern auch unzureichend. Um dies zu zeigen, haben Warde-Farley *et al.* (2014) Kontrollexperimente anhand eines Verfahrens namens **Dropout Boosting** entwickelt. Dieses Verfahren wurde von ihnen so konzipiert, dass exakt dasselbe Maskierungsrauschen wie beim klassischen Dropout genutzt wird, jedoch ohne die regularisierende Wirkung. Dropout Boosting trainiert das gesamte Ensemble, sodass insgesamt die Log-Likelihood der Trainingsdatenmenge maximiert wird. In der Art, in der das klassische Dropout analog zum Bagging ist, ist dieser Ansatz analog zum Boosting. Wie gedacht, zeigen Experimente mit Dropout Boosting im Vergleich zu einem Training des gesamten Netzes als einzelnes Modell nahezu keinen Regularisierungseffekt. Das zeigt, dass die Auslegung von Dropout als Bagging zusätzlich zur Auslegung von Dropout als Robustheit gegen Rauschen wertvoll ist. Der Regularisierungseffekt des auf Bagging basierenden Ensembles wird nur erreicht, wenn die stochastisch mittels Stichprobenentnahme gezogenen Ensembleelemente darauf trainiert werden, unabhängig voneinander gut zu funktionieren.

Dropout hat weitere stochastische Ansätze für das Trainieren exponentiell großer Ensembles von Modellen inspiriert, die Gewichte teilen. Drop-Connect ist ein Dropout-Sonderfall, bei dem jedes Produkt eines einzelnen skalaren Gewichts und des Zustands einer einzelnen verdeckten Einheit als Einheit betrachtet wird, die entfernt werden darf (Wan *et al.*, 2013). Das stochastische Pooling ist eine Art Zufallspooling (siehe Abschnitt 9.3) zum Erstellen von Ensembles aus CNNs, bei dem jedes CNN andere räumliche Orte der einzelnen Merkmalskarten überwacht. Damit ist Dropout die bisher am weitesten verbreitete implizite Ensemblemethode.

Eine wesentliche Erkenntnis aus Dropout ist, dass das Trainieren eines Netzes mit stochastischem Verhalten und das Vorhersagen durch Mittelwertbildung für mehrere stochastische Entscheidungen eine Art Bagging mit Parameter Sharing implementiert. Zuvor haben wir Dropout als Bagging für ein Modellensemble beschrieben, das durch Einbeziehen oder Ausschließen von Einheiten gebildet wird. Allerdings muss diese Methode der Modellmitteilung nicht auf Ein- und Ausschluss beruhen. Prinzipiell ist jede Art zufälliger Modifizierung zulässig. In der Praxis müssen wir Modifizierungsfamilien wählen, die es neuronalen Netzen ermöglichen zu lernen, leistungsfähig zu bleiben. Idealerweise sollten wir auch Modellfamilien einsetzen, die eine schnelle approximative Inferenzregel zulassen. Wir können uns für jede Form der Modifizierung, die durch einen Vektor μ parametrisiert wird, vorstellen, dass sie ein Ensemble $p(y | \mathbf{x}, \mu)$ für alle möglichen Werte von μ trainiert. Dabei ist μ nicht auf eine endliche Anzahl von Werten eingeschränkt. μ darf zum Beispiel reellwertig sein. Srivastava *et al.* (2014) haben gezeigt, dass die Multiplikation der Gewichte mit $\mu \sim \mathcal{N}(\mathbf{1}, I)$ dem Dropout-Verfahren auf Basis binärer Masken überlegen sein kann. Da $\mathbb{E}[\mu] = \mathbf{1}$ ist, implementiert das Standardnetz automatisch die approximative Inferenz im Ensemble – ohne dass es einer Skalierung der Gewichte bedarf.

Bisher haben wir Dropout rein als Mittel für ein effizientes approximatives Bagging beschrieben. Eine andere Betrachtungsweise geht darüber hinaus. Dropout trainiert nicht nur ein auf Bagging basierendes Modellensemble, sondern ein Modellensemble, innerhalb dessen verdeckte Einheiten geteilt werden. Jede verdeckte Einheit muss also ungeachtet der anderen verdeckten Einheiten im Modell eine gute Leistung ermöglichen. Verdeckte Einheiten müssen für einen Tausch und Wechsel zwischen den Modellen vorbereitet sein. Hinton *et al.* (2012c) haben sich aus der Biologie inspirieren lassen: Die geschlechtliche Vermehrung, bei der Gene zwischen zwei unterschiedlichen Organismen getauscht werden, führt zu einem evolutionären Druck auf die Gene, nicht nur »gut« zu werden, sondern auch leicht zwischen unterschiedlichen Organismen ausgetauscht werden können. Derartige Gene

und Merkmale sind gegenüber Veränderungen ihrer Umgebung robust, da sie nicht in der Lage sind, sich auf falsche Weise an ungewöhnliche Merkmale eines beliebigen Organismus oder Modells anzupassen. Dropout regularisiert somit jede verdeckte Einheit derart, dass es sich nicht nur um ein gutes Merkmal handelt, sondern um ein Merkmal, das in vielen Kontexten gut ist. *Warde-Farley et al.* (2014) haben das Training mit Dropout mit dem Trainieren großer Ensembles verglichen und gefolgert, dass Dropout zusätzliche Verbesserungen des Generalisierungsfehlers gegenüber den Ergebnissen von Ensembles mit unabhängigen Modellen bietet.

Machen Sie sich bewusst, dass die Vorteile von Dropout größtenteils der Tatsache zuzuschreiben sind, dass das Maskierungsrauschen auf die verdeckten Einheiten angewandt wird. Sie können sich dies als eine Art extrem cleverer und adaptiver Zerstörung der Eingabeinformation vorstellen (nicht als eine Zerstörung der Werte am Eingang). Wenn das Modell zum Beispiel eine verdeckte Einheit h_i erlernt, die Gesichter anhand der Nase identifiziert, entspricht das Entfernen von h_i dem Löschen der Information, dass das Foto eine Nase enthält. Das Modell muss ein anderes h_i erlernen, in dem das Vorhandensein einer Nase entweder redundant codiert ist oder das Gesichter anhand eines anderen Merkmals erkennt, zum Beispiel am Mund. Klassische Verfahren, die nicht strukturiertes Rauschen zur Eingabe hinzufügen, sind nicht in der Lage, die Informationen über eine Nase zufällig aus dem Foto eines Gesichts zu löschen – es sei denn, das Rauschen ist derart stark, dass auch nahezu alle anderen Informationen aus dem Foto entfernt werden. Indem extrahierte Merkmale anstelle der Originalwerte zerstört werden, können bei der Zerstörung sämtliche Kenntnisse über die Eingabeverteilung genutzt werden, die das Modell bisher gewonnen hat.

Ein weiterer wichtiger Aspekt des Dropout-Verfahrens ist der multiplikative Charakter des Rauschens. Bei einem additiven Rauschen mit festem Maßstab könnte eine rektifizierte lineare verdeckte Einheit h_i mit hinzugefügtem Rauschen ϵ einfach lernen, dass h_i sehr groß werden muss, um das hinzugefügte Rauschen ϵ in der Relation unbedeutend werden zu lassen. Multiplikatives Rauschen lässt eine solch pathologische Lösung im Zusammenhang mit der Robustheit gegen Rauschen nicht zu.

Ein weiterer Deep-Learning-Algorithmus, die Batch-Normalisierung, parametrisiert das Modell auf eine Art neu, die den verdeckten Einheiten zum Trainingszeitpunkt sowohl additives als auch multiplikatives Rauschen hinzufügt. Der Hauptzweck der Batch-Normalisierung, ist eine verbesserte Optimierung, aber das Rauschen kann sich regularisierend auswirken und führt manchmal dazu, dass Dropout nicht benötigt wird. Die Batch-Normalisierung, wird in Abschnitt 8.7.1 genauer beschrieben.

7.13 Adversarial Training

In vielen Bereichen sind neuronale Netze dabei, menschliche Leistungen zu erzielen, sofern die Evaluation anhand einer unabhängig und identisch verteilten (u.i.v.) Testdatenmenge erfolgt. Das führt zu der Frage, ob diese Modelle die Aufgabe tatsächlich auf einem menschlichen Level verstehen. Um den Grad des Verstehens eines Netzes hinsichtlich seiner Aufgabe zu überprüfen, können wir nach Beispieldaten suchen, die vom Modell falsch klassifiziert werden. Szegedy et al. (2014b) haben herausgefunden, dass selbst neuronale Netze, die menschliche Genauigkeit erzielen, eine Fehlerquote von nahezu 100 % aufweisen, wenn es um Beispieldaten geht, die absichtlich mithilfe eines Optimierungsverfahrens so konstruiert werden, dass eine Eingabe \mathbf{x}' in der Nähe eines Datenpunkts \mathbf{x} gesucht wird, sodass die Ausgabe des Modells sich in \mathbf{x}' deutlich unterscheidet. In vielen Fällen können \mathbf{x}' und \mathbf{x} einander so ähnlich sein, dass für das menschliche Auge kein Unterschied zwischen dem ursprünglichen Beispiel und dem sogenannten **Adversarial Example** besteht – aber das Netz kann zu stark abweichenden Vorhersagen gelangen. Abbildung 7.8 zeigt ein Beispiel.

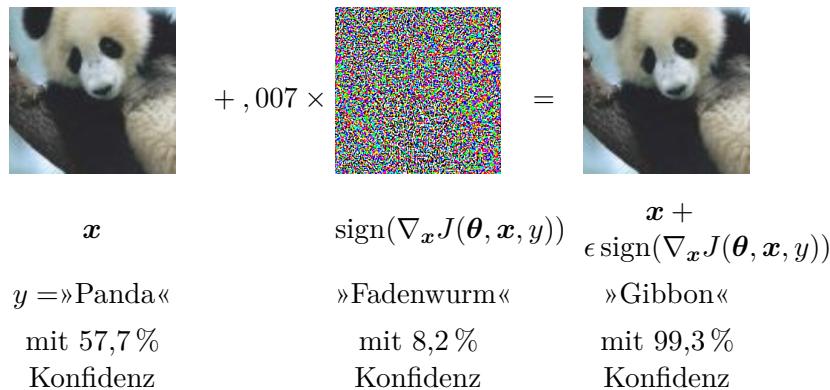


Abbildung 7.8: Erzeugen eines Adversarial Examples für GoogLeNet (Szegedy et al., 2014a) in ImageNet. Durch Hinzufügen eines verschwindend kleinen Vektors, dessen Elemente bezüglich der Eingabe gleich dem Vorzeichen der Elemente des Gradienten der Kostenfunktion sind, können wir die GoogLeNet-Klassifizierung des Bildes verändern. (Abbildung mit freundlicher Genehmigung von Goodfellow et al. (2014b))

Adversarial Examples haben viele Auswirkungen, die nicht in diesem Kapitel behandelt werden können, zum Beispiel in der Computersicherheit. Sie sind für die Regularisierung jedoch von Interesse, da sich damit die Fehlerquote der ursprünglichen u.i.v. Testdatenmenge durch **Adversarial**

Training reduzieren lässt – ein Training mit Adversarial Examples aus der Trainingsdatenmenge (*Szegedy et al., 2014b; Goodfellow et al., 2014b*).

Goodfellow et al. (2014b) haben gezeigt, dass einer der wesentlichen Gründe für solche Adversarial Examples eine ausschließliche Linearität ist. Neuronale Netze setzen sich aus überwiegend linearen Bausteinen zusammen. Dies führt dazu, dass sich die von ihnen implementierte Gesamtfunktion in einigen Experimenten als in hohem Maße linear erweist. Diese linearen Funktionen lassen sich einfach optimieren. Leider kann sich der Wert einer linearen Funktion sehr schnell ändern, wenn sie über viele Eingaben verfügt. Wenn wir jede Eingabe um ϵ ändern, kann sich eine lineare Funktion mit den Gewichten w um bis zu $\epsilon\|w\|_1$ ändern – und das kann wiederum ein sehr hoher Betrag sein, wenn w hochdimensional ist. Das Adversarial Training führt zu einer Abwertung dieses sehr sensiblen lokal linearen Verhaltens, indem das Netz zu lokaler Konstanz in der Umgebung der Trainingsdaten animiert wird. Es wird quasi explizit eine Annahme der lokalen Konstanz in überwachte neuronale Netze eingeführt.

Adversarial Training hilft dabei, die Leistungsfähigkeit einer großen Funktionsfamilie in Verbindung mit einer aggressiven Regularisierung zu verdeutlichen. Rein lineare Modelle wie die logistische Regression sind nicht in der Lage, den Adversarial Examples standzuhalten, da sie zur Linearität gezwungen werden. Neuronale Netze können Funktionen repräsentieren, die nahezu linear bis hin zu nahezu lokal konstant sein können und die somit die Flexibilität aufweisen, lineare Tendenzen in den Trainingsdaten zu erfassen; dabei sind sie gleichzeitig in der Lage, zu lernen, lokalen Störungen standzuhalten.

Adversarial Examples stellen auch Mittel für halb-überwachtes Lernen zur Verfügung. An einem Punkt x , der keinem Label im Datensatz zugeordnet ist, weist das Modell selbst ein Label \hat{y} zu. Das Label \hat{y} des Modells ist vielleicht nicht das korrekte Label, aber wenn das Modell eine hohe Qualität aufweist, ist die Wahrscheinlichkeit hoch, dass \hat{y} das korrekte Label darstellt. Wir können nach einem Adversarial Example x' suchen, das dazu führt, dass der Klassifikator ein Label y' mit $y' \neq \hat{y}$ ausgibt. Adversarial Examples, die nicht mit korrekten Labeln, sondern mit einem vom trainierten Modell bereitgestellten Label erzeugt werden, werden **virtuelle Adversarial Examples** (*Miyato et al., 2015*) genannt. Der Klassifikator kann anschließend so trainiert werden, dass dasselbe Label x und x' zugeordnet wird. So wird der Klassifikator dazu angeregt, eine Funktion zu erlernen, die robust gegenüber kleineren Änderungen überall entlang der Mannigfaltigkeit ist, in der die Daten ohne Label liegen. Der Ansatz beruht auf der Annahme, dass unterschiedliche Klassen normalerweise in nicht miteinander verbundenen

Mannigfaltigkeiten liegen und eine kleine Störung nicht dazu führen dürfte, dass der Sprung von einer in eine andere Klassenmannigfaltigkeit erfolgt.

7.14 Tangentendistanz, Tangenten-Propagation und Mannigfaltigkeit-Tangentenklassifikator

Viele Machine-Learning-Algorithmen versuchen, dem Fluch der Dimensionalität zu entkommen, indem sie davon ausgehen, dass die Daten in der Nähe einer geringdimensionalen Mannigfaltigkeit liegen (siehe Abschnitt 5.11.3).

Zu den frühen Versuchen, diese Mannigfaltigkeit-Hypothese zu nutzen, gehört der **Tangentendistanz**-Algorithmus (*Simard et al.*, 1993, 1998). Es handelt sich dabei um einen nichtparametrischen Nearest-Neighbor-Algorithmus, in dem nicht der allgemeine euklidische Abstand genutzt wird, sondern vielmehr eine Metrik, die aus dem Wissen über die Mannigfaltigkeiten, in deren Nähe sich die Wahrscheinlichkeit konzentriert, abgeleitet wird. Es wird angenommen, dass wir versuchen, Beispiele zu klassifizieren, und dass Beispiele in derselben Mannigfaltigkeit zur selben Kategorie gehören.

Da der Klassifikator invariant gegenüber den lokalen Faktoren der Variation sein sollte, die einer Bewegung auf der Mannigfaltigkeit entsprechen, wäre es sinnvoll, als Nearest-Neighbor-Abstand zwischen den Punkten \mathbf{x}_1 und \mathbf{x}_2 den Abstand zwischen den Mannigfaltigkeiten M_1 und M_2 zu verwenden, zu denen diese Punkte jeweils gehören. Obwohl das rechnerisch kein einfaches Unterfangen ist (hierfür muss ein Optimierungsproblem gelöst werden, mit dem das nächstgelegene Punktpaar in M_1 und M_2 gefunden wird), ist es eine mit wenig Aufwand verbundene Alternative für das lokale Umfeld, M_i anhand der Tangentialebene in \mathbf{x}_i zu approximieren und den Abstand zwischen den beiden Tangenten oder einer Tangentialebene und einem Punkt zu messen. Dies ist durch Lösen eines geringdimensionalen linearen Systems (in der Dimension der Mannigfaltigkeiten) möglich. Natürlich müssen für den Algorithmus die Tangentenvektoren angegeben werden.

Auf ähnliche Weise trainiert der **Tangenten-Propagation**-Algorithmus (*Simard et al.*, 1992) (Abbildung 7.9) einen Klassifikator als neuronales Netz mit einem zusätzlichen Strafterm so, dass jede Ausgabe $f(\mathbf{x})$ des neuronalen Netzes lokal invariant gegenüber bekannten Faktoren der Variation ist. Diese Faktoren der Variation entsprechen der Bewegung entlang der Mannigfaltigkeit, in deren Nähe die Beispiele derselben Klasse konzentriert sind. Die lokale Invarianz wird durch die Bedingung erreicht, dass $\nabla_{\mathbf{x}} f(\mathbf{x})$ orthogonal zu den bekannten Mannigfaltigkeit-Tangentenvektoren (engl. *manifold tangent vectors*) $\mathbf{v}^{(i)}$ in \mathbf{x} ist – oder gleichwertig, dass die Richtungsableitung

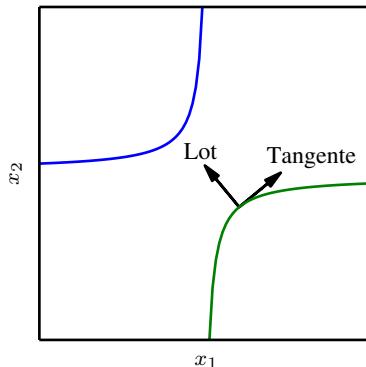


Abbildung 7.9: Darstellung des Konzepts hinter dem Tangenten-Propagation-Algorithmus (Simard et al., 1992) und dem Mannigfaltigkeit-Tangentenklassifikator (Rifai et al., 2011c), die beide die Ausgabefunktion $f(\mathbf{x})$ des Klassifikators regularisieren. Jede Kurve stellt die Mannigfaltigkeit einer anderen Klasse dar (hier eine eindimensionale Mannigfaltigkeit, die in einem zweidimensionalen Raum eingebettet ist). Auf der einen Kurve haben wir einen Einzelpunkt ausgewählt und einen Vektor eingezeichnet, der tangential zur Klassenmannigfaltigkeit verläuft (parallel zur Mannigfaltigkeit und sie berührend), sowie einen Vektor, der lotrecht zur Klassenmannigfaltigkeit verläuft (orthogonal zur Mannigfaltigkeit). In mehreren Dimensionen kann es viele Tangentenrichtungen und viele Lotrichtungen geben. Wir erwarten eine schnelle Änderung der Klassifizierungsfunktion senkrecht zur Mannigfaltigkeit und keine Änderung in Richtung der Klassenmannigfaltigkeit. So-wohl Tangenten-Propagation als auch der Mannigfaltigkeit-Tangentenklassifikator regularisieren $f(\mathbf{x})$ so, dass mit der Bewegung von \mathbf{x} entlang der Mannigfaltigkeit keine große Veränderung eintritt. Für die Tangenten-Propagation müssen die Funktionen zur Berechnung der Tangentenrichtungen manuell angegeben werden (zum Beispiel, dass kleine Verschiebungen von Bildern innerhalb derselben Klassenmannigfaltigkeit stattfinden). Der Mannigfaltigkeit-Tangentenklassifikator schätzt die Richtungen der Mannigfaltigkeit-Tangenten durch Trainieren eines Autoencoders für die Anpassung der Trainingsdaten. Die Verwendung von Autoencodern zum Schätzen von Mannigfaltigkeiten wird in Kapitel 14 beschrieben.

von f in \mathbf{x} in der Richtung $\mathbf{v}^{(i)}$ klein ist, indem ein Regularisierungsstrafterm Ω hinzugefügt wird:

$$\Omega(f) = \sum_i \left((\nabla_{\mathbf{x}} f(\mathbf{x}))^\top \mathbf{v}^{(i)} \right)^2. \quad (7.67)$$

Dieser Regularisierer kann natürlich mittels eines passenden Hyperparameters skaliert werden. Für die meisten neuronalen Netze müssen wir die Summe vieler Ausgaben bilden. Die Verwendung einer einzelnen Ausgabe $f(\mathbf{x})$ dient lediglich zur Vereinfachung des Beispiels. Wie beim Tangentendistanz-Algorithmus werden die Tangentenvektoren a priori abgeleitet, meist

aus dem formalen Wissen über die Auswirkung der Transformationen wie Verschiebung, Drehung und Skalierung von Fotos. Die Tangenten-Propagation wurde nicht nur für das überwachte Lernen (*Simard et al.*, 1992) genutzt, sondern auch im Rahmen des Reinforcement Learnings (*Thrun*, 1995).

Die Tangenten-Propagation ist eng mit dem Erweitern des Datensatzes verwandt. In beiden Fällen codiert der Anwender des Algorithmus ein Vorwissen über die Aufgabe durch Festlegen einer Reihe von Transformationen, die nicht zu einer geänderten Ausgabe des Netzes führen dürfen. Der Unterschied besteht im Falle der Erweiterung des Datensatzes darin, dass das Netz explizit dahingehend trainiert wird, dass unterschiedliche Eingaben, die aus mehr als nur infinitesimalen Transformationen hervorgehen, korrekt klassifiziert werden. Für die Tangenten-Propagation ist kein explizites Aufsuchen eines neuen Eingabepunkts erforderlich. Stattdessen regularisiert sie das Modell analytisch als störungsresistent in den Richtungen der angegebenen Transformation. Das ist zwar ein theoretisch eleganter Ansatz, der aber leider zwei große Nachteile aufweist: Zunächst wird das Modell nur resistent gegen kleinste Störungen regularisiert. Ein explizites Erweitern des Datensatzes führt zu einer Robustheit gegenüber größeren Störungen. Zweitens erweist sich der infinitesimale Ansatz bei Modellen auf Basis von ReLUs als problematisch. Diese Modelle können nur ihre Ableitungen verkleinern, indem sie Einheiten ausschalten oder ihre Gewichte verringern. Sie sind nicht in der Lage, ihre Ableitungen zu verkleinern, indem sie bei einem hohen Wert mit hohen Gewichten sättigen, wie es zum Beispiel für Sigmoidfunktionen oder tanh-Einheiten möglich ist. Das Erweitern des Datensatzes funktioniert gut mit ReLUs, da die unterschiedlichen Teilmengen der rektifizierten Einheiten für unterschiedliche transformierte Versionen der ursprünglichen Eingaben aktiviert werden können.

Die Tangenten-Propagation ist außerdem mit der **Double Backpropagation** (*Drucker und LeCun*, 1992) und dem Adversarial Training (*Szegedy et al.*, 2014b; *Goodfellow et al.*, 2014b) verwandt. Die Double Backpropagation reguliert die Jacobi-Matrix, sodass sie klein wird; das Adversarial Training sucht Eingaben in der Nähe der ursprünglichen Eingaben und trainiert das Modell so, dass für diese dieselbe Ausgabe wie für die ursprünglichen Eingaben erfolgt. Die Tangenten-Propagation und das Erweitern des Datensatzes mithilfe manuell angegebener Transformationen setzen voraus, dass das Modell invariant gegenüber bestimmten angegebenen Änderungsrichtungen in der Eingabe ist. Sowohl Double Backpropagation als auch Adversarial Training benötigen ein Modell, das invariant gegenüber *allen* Änderungsrichtungen der Eingabe ist, sofern die Änderungen klein sind.

So wie das Erweitern des Datensatzes eine nicht infinitesimale Version der Tangenten-Propagation darstellt, ist das Adversarial Training die nicht infinitesimale Version der Double Backpropagation.

Mit dem Mannigfaltigkeit-Tangentenklassifikator (engl. *manifold tangent classifier*) (Rifai et al., 2011c) müssen die Tangentenvektoren nicht mehr im Voraus bekannt sein. Wie wir in Kapitel 14 noch zeigen, können Autoencoder die Mannigfaltigkeit-Tangentenvektoren schätzen. Der Mannigfaltigkeit-Tangentenklassifikator nutzt dieses Verfahren, damit die Tangentenvektoren nicht zuvor vom Anwender bestimmt werden müssen. Wie Abbildung 14.10 zeigt, erstrecken sich diese geschätzten Tangentenvektoren über die klassischen Invarianten hinaus, die sich aus der Bildgeometrie ergeben (z. B. Verschiebung, Drehung und Skalierung), und enthalten Faktoren, die erlernt werden müssen, da sie objektspezifisch sind (z. B. sich bewegende Körperteile). Der für den Mannigfaltigkeit-Tangentenklassifikator vorgeschlagene Algorithmus fällt daher einfach aus: (1) Nutzen eines Autoencoders zum Erlernen der Mannigfaltigkeitsstruktur durch unüberwachtes Lernen und (2) Verwenden dieser Tangenten zum Regularisieren eines neuronalen Netzwerks, das als Klassifikator genutzt wird, wie bei der Tangenten-Propagation (Gleichung 7.67).

In diesem Kapitel haben wir die wichtigsten der allgemeinen Verfahren zur Regularisierung von neuronalen Netzen betrachtet. Die Regularisierung ist ein zentrales Thema im Machine Learning und wird somit in den verbleibenden Kapiteln immer wieder auftauchen. Ebenfalls von zentraler Bedeutung für das Machine Learning ist die Optimierung, der wir uns nun widmen.

8

Optimierung beim Trainieren von tiefen Modellen

Deep-Learning-Algorithmen erfordern in vielen Bereichen Optimierung. Zum Beispiel beinhaltet das Durchführen der Inferenz in Modellen das Lösen eines Optimierungsproblems. Beim Schreiben von Beweisen oder beim Konzipieren von Algorithmen wird häufig analytische Optimierung verwendet. Unter allen Optimierungsproblemen im Deep-Learning-Kontext ist das Trainieren von neuronalen Netzen das schwierigste. Häufig müssen Hunderte von Maschinen Tage oder gar Monate rechnen, um einen einzelnen Vorgang des Trainingsproblems für das neuronale Netz zu lösen. Da dieses Problem eine solch enorme Bedeutung hat und seine Lösung gleichzeitig so aufwendig ist, wurde eine Reihe spezieller Optimierungsmethoden dafür entwickelt. Diese Optimierungsmethoden für das Trainieren neuronaler Netze sind Thema des vorliegenden Kapitels.

Falls Sie mit den Grundlagen der Optimierung auf Gradientenbasis nicht vertraut sind, sollten Sie Kapitel 4 lesen. Dort wird die grundlegende numerische Optimierung kurz erklärt.

Auf den nächsten Seiten geht es vor allem um einen speziellen Fall der Optimierung, nämlich wie Sie die Parameter eines neuronalen Netzes finden, die die Kostenfunktion $J(\theta)$ deutlich minimieren, die üblicherweise sowohl eine Performance-Bewertung beinhaltet, welche auf Basis der gesamten Trainingsdatenmenge evaluiert wird, als auch zusätzliche Regularisierungsterme.

Wir beschreiben zunächst, wie sich die Optimierung als Trainingsalgorithmus für eine Machine-Learning-Aufgabe von der reinen Optimierung unterscheidet. Dann stellen wir einige der konkreten Herausforderungen vor, die eine Optimierung neuronaler Netze so schwierig machen. Anschließend definieren wir einige praxistaugliche Algorithmen, darunter sowohl die Optimierungsalgorithmen selbst als auch Verfahren zur Parameterinitialisierung. Fortschrittlichere Algorithmen passen ihre Lernrate während des Trainings an oder nutzen Informationen aus den zweiten Ableitungen der Kostenfunktion. Zu guter Letzt betrachten wir mehrere Optimierungsverfahren, die durch die Kombination einfacher Optimierungsalgorithmen zu übergeordneten Verfahren gebildet werden.

8.1 Unterschied zwischen Lernen und reiner Optimierung

Optimierungsalgorithmen für das Trainieren von tiefen Modellen unterscheiden sich auf mehrere Arten von klassischen Optimierungsalgorithmen. Machine Learning findet normalerweise indirekt statt. Meist interessieren wir uns für eine Leistungsbewertung P , die bezüglich einer Testdatenmenge definiert wird und auch nicht effizient durchführbar (engl. *intractable*) sein kann. Daher optimieren wir P nur indirekt, nämlich durch Reduzieren einer Kostenfunktion $J(\boldsymbol{\theta})$, um so hoffentlich P zu verbessern. Anders sieht es bei der reinen Optimierung aus, denn hier ist das Minimieren von J das oberste und eigentliche Ziel. Optimierungsalgorithmen für das Trainieren von tiefen Modellen enthalten für gewöhnlich auch eine Spezialisierung hinsichtlich der spezifischen Struktur der Machine-Learning-Zielfunktionen.

Im Normalfall kann die Kostenfunktion als Durchschnittswert über die Trainingsdatenmenge geschrieben werden, zum Beispiel

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, y) \sim \hat{p}_{\text{data}}} L(f(\mathbf{x}; \boldsymbol{\theta}), y), \quad (8.1)$$

wobei L die Verlustfunktion pro Beispiel ist, $f(\mathbf{x}; \boldsymbol{\theta})$ die vorhergesagte Ausgabe für eine Eingabe \mathbf{x} und \hat{p}_{data} die empirische Verteilung. Beim überwachten Lernen ist y der Zielwert der Ausgabe. In diesem Kapitel entwickeln wir den nicht regularisierten überwachten Fall, in dem $f(\mathbf{x}; \boldsymbol{\theta})$ und y die Argumente für L sind. Eine Erweiterung dieser Entwicklung ist dann trivial – wenn zum Beispiel $\boldsymbol{\theta}$ oder \mathbf{x} als Argumente eingebunden oder y als Argumente ausgeschlossen werden sollen. So lassen sich unterschiedliche Arten der Regularisierung oder des unüberwachten Lernens entwickeln.

Gleichung 8.1 definiert eine Zielfunktion für die Trainingsdatenmenge. Meist ziehen wir eine Minimierung der entsprechenden Zielfunktion für einen Erwartungswert der *datengenerierenden Verteilung* p_{data} gegenüber einem Wert für die endliche Trainingsdatenmenge vor:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, y) \sim p_{\text{data}}} L(f(\mathbf{x}; \boldsymbol{\theta}), y). \quad (8.2)$$

8.1.1 Empirische Risikominimierung

Ein Machine-Learning-Algorithmus soll den erwarteten Generalisierungsfehler aus Gleichung 8.2 reduzieren. Diese Größe wird als **Risiko** bezeichnet. Beachten Sie, dass der Erwartungswert für die tatsächlich zugrunde liegende Verteilung p_{data} gilt. Wenn wir die wahre Verteilung $p_{\text{data}}(\mathbf{x}, y)$ kennen würden, wäre die Risikominimierung eine Optimierungsaufgabe, die mittels Optimierungsalgorithmus gelöst werden könnte. Wenn wir $p_{\text{data}}(\mathbf{x}, y)$ nicht kennen, sondern nur eine Menge von Trainingsstichproben zur Verfügung steht, handelt es sich dagegen um ein Machine-Learning-Problem.

Um ein Machine-Learning-Problem wieder in ein Optimierungsproblem umzuwandeln, müssen wir lediglich den erwarteten Trainingsverlust minimieren. Wir müssen also die wahre Verteilung $p(\mathbf{x}, y)$ durch die anhand der Trainingsdatenmenge definierte empirische Verteilung $\hat{p}(\mathbf{x}, y)$ ersetzen. Nun minimieren wir das **empirische Risiko**

$$\mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}(\mathbf{x}, y)} [L(f(\mathbf{x}; \boldsymbol{\theta}), y)] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}), \quad (8.3)$$

wobei m die Anzahl der Trainingsbeispiele ist.

Das Trainingsverfahren, das auf dem Minimieren dieses durchschnittlichen Trainingsfehlers beruht, wird **empirische Risikominimierung** genannt.¹ In diesem Zusammenhang ähnelt Machine Learning nach wie vor einer einfachen Optimierung. Statt das Risiko direkt zu optimieren, optimieren wir das empirische Risiko und hoffen, dass dadurch auch das Risiko erheblich sinkt. Eine Auswahl theoretischer Ergebnisse stellen Bedingungen her, unter denen das tatsächliche Risiko um diverse Beträge sinken dürfte.

Nichtsdestotrotz besteht bei der empirischen Risiko-Minimierung die Gefahr einer Überanpassung. Modelle mit hoher Kapazität können leicht die Trainingsdatenmenge memorieren. In vielen Fällen ist die empirische Risikominimierung nicht wirklich möglich. Die derzeit effektivsten Optimierungsalgorithmen beruhen auf dem Gradientenabstiegsverfahren, aber viele

¹ Anm. zur Übersetzung: Korrekter wäre eigentlich »Minimierung des empirischen Risikos«, denn es ist das Risiko, das empirisch ist, nicht die Minimierung.

nützliche Verlustfunktionen – wie der 0-1-Verlust – weisen keine nützlichen Ableitungen auf (die Ableitung ist entweder Null oder überall undefiniert). Diese beiden Probleme führen dazu, dass die empirische Risikominimierung im Deep-Learning-Kontext nur selten eingesetzt wird. Stattdessen müssen wir einen etwas anderen Ansatz wählen, bei dem die tatsächlich optimierte Größe sich häufig noch stärker von der Größe, die wir eigentlich optimieren möchten, unterscheidet.

8.1.2 Ersatzverlustfunktionen und früher Abbruch

Manchmal lässt sich die wesentliche Verlustfunktion (zum Beispiel der Klassifizierungsfehler) nicht effizient optimieren. So ist die exakte Minimierung des erwarteten 0-1-Verlusts normalerweise nicht effizient optimierbar (exponentiell in der Eingabedimension); das gilt auch für einen linearen Klassifikator (*Marcotte und Savard, 1992*). In diesen Fällen optimieren wir stattdessen eine **Ersatzverlustfunktion** (engl. *surrogate loss function*), die stellvertretend verwendet werden kann und verschiedene Vorteile hat. So wird die negative Log-Likelihood der korrekten Klasse üblicherweise als Ersatz für den 0-1-Verlust verwendet. Die negative Log-Likelihood erlaubt es dem Modell, die bedingte Wahrscheinlichkeit der Klassen anhand der Eingabe zu schätzen. Gelingt dies dem Modell gut, kann es die Klassen auswählen, die zum kleinsten Klassifizierungsfehler im Erwartungswert führen.

Hin und wieder führt eine Ersatzverlustfunktion sogar dazu, dass mehr erlernt werden kann. Zum Beispiel nimmt der 0-1-Verlust der Testdatenmenge häufig sehr lange Zeit ab, nachdem der 0-1-Verlust der Trainingsdatenmenge Null erreicht hat, wenn für das Training ein Log-Likelihood-Ersatz verwendet wird. Das liegt daran, dass selbst bei einem erwarteten 0-1-Verlust von Null eine Verbesserung der Robustheit des Klassifikators durch weiteres Auseinanderziehen der Klassen erreicht werden kann, was zu einem konfidenten und zuverlässigeren Klassifikator führt. Somit werden mehr Informationen aus den Trainingsdaten gewonnen, als es bei einer einfachen Minimierung des durchschnittlichen 0-1-Verlusts der Trainingsdatenmenge möglich gewesen wäre.

Ein sehr wichtiger Unterschied zwischen der Optimierung im Allgemeinen und der Optimierung für Trainingsalgorithmen besteht darin, dass Trainingsalgorithmen normalerweise nicht bei einem lokalen Minimum stoppen. Stattdessen minimiert ein Machine-Learning-Algorithmus eine Ersatzverlustfunktion normalerweise, stoppt aber, wenn ein Konvergenzkriterium auf Grundlage des frühen Abbruchs (Abschnitt 7.8) erfüllt ist. Üblicherweise beruht das Kriterium für den frühen Abbruch auf der tatsächlich

zugrunde liegenden Verlustfunktion, zum Beispiel dem 0-1-Verlust für eine Validierungsdatenmenge, und ist so angelegt, dass der Algorithmus stoppt, wenn sich die ersten Anzeichen für eine Überanpassung zeigen. Wenn das Training endet, weist die Ersatzverlustfunktion häufig noch große Ableitungen auf. Ganz anders bei der reinen Optimierung, wo die Konvergenz des Optimierungsalgorithmus erst als erreicht gilt, wenn der Gradient sehr klein ist.

8.1.3 Batch- und Mini-Batch-Algorithmen

Ein Aspekt, der Machine-Learning-Algorithmen von allgemeinen Optimierungsalgorithmen unterscheidet, ist, dass die Zielfunktion meist als Summe aus Trainingsbeispielen zerlegt wird. Optimierungsalgorithmen für das Machine Learning berechnen für gewöhnlich jede Anpassung der Parameter anhand eines Erwartungswerts der Kostenfunktion, der lediglich mit einer Teilmenge der Terme der gesamten Kostenfunktion geschätzt wurde.

So stellen sich Probleme der Maximum-Likelihood-Schätzung im Log-Raum als eine Zerlegung in die Summe über den einzelnen Beispielen dar:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \quad (8.4)$$

Das Maximieren dieser Summe entspricht dem Maximieren des Erwartungswerts für die empirische Verteilung, die durch die Trainingsdatenmenge definiert ist:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{x}, y; \boldsymbol{\theta}). \quad (8.5)$$

Die meisten der Eigenschaften der Zielfunktion J , die von den meisten unserer Optimierungsalgorithmen verwendet werden, sind gleichzeitig Erwartungswerte für die Trainingsdatenmenge. Zum Beispiel ist der Gradient die meistverwendete Eigenschaft:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\text{model}}(\mathbf{x}, y; \boldsymbol{\theta}). \quad (8.6)$$

Die exakte Berechnung dieses Erwartungswerts ist sehr aufwendig, da dazu das Modell für jedes Beispiel des gesamten Datensatzes evaluiert werden muss. In der Praxis können wir diese Erwartungswerte durch zufälliges Ziehen einer geringen Anzahl von Beispielen aus dem Datensatz und Mittelwertbildung nur dieser Beispiele bestimmen.

Sie haben gelernt, dass der Standardfehler des Mittelwerts (Gleichung 5.46) aus einer Schätzung von n Stichproben gleich σ/\sqrt{n} ist, wobei

σ die tatsächliche Standardabweichung des Werts der Stichproben ist. Der Nenner von \sqrt{n} gibt an, dass der Einsatz weiterer Beispiele zur Schätzung des Gradienten keine linearen Ergebnisse bringt. Vergleichen Sie zwei hypothetische Schätzwerte des Gradienten, einen auf der Grundlage von 100, den anderen auf der Grundlage von 10 000 Beispielen. Der zweite erfordert den 100-fachen Berechnungsaufwand, reduziert aber den Standardfehler des Mittelwerts nur um den Faktor 10. Die meisten Optimierungsalgorithmen konvergieren sehr viel schneller (was die Gesamtberechnung betrifft, nicht die Anzahl der Anpassungen), wenn sie schnell approximative Schätzwerte des Gradienten berechnen dürfen und sich nicht mit der langsamten Berechnung des exakten Gradienten aufhalten müssen.

Eine weitere Überlegung hinter der statistischen Schätzung des Gradienten anhand einer geringen Anzahl von Stichproben ist Redundanz in der Trainingsdatenmenge. Im ungünstigsten Fall sind alle m Stichproben in der Trainingsdatenmenge identische Kopien eines einzigen Elements. Eine Gradientenschätzung auf Basis von Stichproben könnte den korrekten Gradienten mit nur einer Stichprobe ermitteln und so m -mal weniger Rechenaufwand als der naive Ansatz benötigen. In der Praxis ist dieser Fall höchst unwahrscheinlich, aber eine große Anzahl von Beispielen, die jeweils sehr ähnliche Beiträge zum Gradienten leisten, ist durchaus vorstellbar.

Nutzt ein Optimierungsalgorithmus die gesamte Trainingsdatenmenge, spricht man von **Batch-** oder **deterministischen** Gradientenmethoden, da sämtliche Trainingsbeispiele gleichzeitig in einem großen Stapel verarbeitet werden. Aber Obacht! Der englische Begriff »Batch« (dt. *Stapel*) wird auch für Mini-Batches im stochastischen Mini-Batch-Gradientenabstieg verwendet. Der Begriff »Batch-Gradientenabstieg« bezieht sich meist auf die Nutzung der gesamten Trainingsdatenmenge, der Begriff »Batch« zur Beschreibung einer Gruppe von Beispielen dagegen nicht. So wird beispielsweise häufig der Begriff »Batch-Größe« verwendet, um die Größe eines Mini-Batches zu beschreiben.

Optimierungsalgorithmen, die zu einem Zeitpunkt immer nur ein Beispiel gleichzeitig verwenden, werden manchmal auch **stochastische** Verfahren oder **Online**-Verfahren genannt. Der Begriff »Online« gibt meist an, dass die Beispiele aus einem Strom stetig erzeugter Beispiele stammen, nicht aus einer Trainingsdatenmenge mit fester Größe, aus der immer wieder gezogen wird.

Die meisten Algorithmen für das Deep Learning nehmen eine Zwischenstellung ein und nutzen mehr als eines, aber weniger als alle Trainingsbeispiele. Früher sprach man von **Mini-Batch-** oder **stochastischen Mini-**

Batch-Verfahren, aber heute wird meist nur von **stochastischen** Verfahren gesprochen.

Das kanonische Beispiel eines stochastischen Verfahrens ist das stochastische Gradientenabstiegsverfahren (engl. *stochastic gradient descent*, SGD), das in Abschnitt 8.3.1 genauer behandelt wird.

Mini-Batch-Größen richten sich allgemein nach den folgenden Faktoren:

- Größere Batches führen zu einer genaueren Gradientenschätzung, aber der Gewinn ist nichtlinear.
- Multicore-Architekturen sind von besonders kleinen Batches meist extrem unterfordert. Das führt zu einer gewissen absoluten Mindestgröße der Batches, unter der sich keine Verarbeitungszeit einsparen lässt.
- Wenn alle Beispiele im Batch parallel verarbeitet werden sollen (und das ist Usus), muss mit zunehmender Batch-Größe auch der Arbeitsspeicher größer ausfallen. Das ist häufig der beschränkende Hardwarefaktor für die Batch-Größe.
- Je nach Hardware ist eine bestimmte Array-Größe für die Laufzeit von Vorteil. Insbesondere bei Nutzung von GPUs sind Zweierpotenz-Batch-Größen für eine bessere Laufzeit gängig. Übliche Größen liegen hier zwischen 32 und 256, wobei für große Modelle manchmal ein Wert von 16 gewählt wird.
- Kleine Batches können zu einem Regularisierungseffekt führen (*Wilson und Martinez*, 2003), was möglicherweise am Rauschen liegt, das sie dem Lernprozess hinzufügen. Der beste Generalisierungsfehler ergibt sich oft bei einer Batch-Größe von 1. Das Training mit einer so kleinen Batch-Größe erfordert unter Umständen eine geringe Lernrate, da ansonsten aufgrund der großen Varianz in der Gradientenschätzung keine Stabilität gegeben ist. Die Gesamtlaufzeit kann sehr hoch sein, da mehr Schritte erforderlich sind – einerseits aufgrund der reduzierten Lernrate und andererseits, da zur Beobachtung der gesamten Trainingsdatenmenge mehr Schritte notwendig werden.

Diverse Arten von Algorithmen verwenden verschiedene Arten von Informationen aus Mini-Batches auf unterschiedliche Weise. Einige Algorithmen sind anfälliger gegenüber dem Stichprobenfehler als andere – sei es, weil sie Informationen nutzen, die sich mit wenigen Stichproben schlechter schätzen lassen, oder weil sie Informationen auf eine Art nutzen, die eine Verstärkung

des Stichprobenfehlers begünstigt. Verfahren, in denen Anpassungen nur auf Basis des Gradienten \mathbf{g} berechnet werden, sind meist recht robust und kommen auch mit kleineren Batch-Größen, z. B. 100, gut zurecht. Verfahren zweiter Ordnung, die zusätzlich die Hesse-Matrix \mathbf{H} einsetzen und Anpassungen wie $\mathbf{H}^{-1}\mathbf{g}$ berechnen, erfordern im Normalfall größere Batches, z. B. 10 000. Diese großen Batch-Größen sind erforderlich, um Schwankungen in den Schätzwerten für $\mathbf{H}^{-1}\mathbf{g}$ zu minimieren. Ein Beispiel: Sei \mathbf{H} perfekt geschätzt, aber durch eine schlechte Konditionszahl belastet. Durch Multiplikation mit \mathbf{H} oder der Inversen werden bereits vorhandene Fehler verstärkt, also hier die Schätzfehler in \mathbf{g} . Sehr kleine Änderungen des Schätzwerts für \mathbf{g} können daher zu großen Änderungen in der Anpassung $\mathbf{H}^{-1}\mathbf{g}$ führen – sogar bei einer perfekten Schätzung von \mathbf{H} . Natürlich wird \mathbf{H} nur näherungsweise geschätzt, sodass die Anpassung $\mathbf{H}^{-1}\mathbf{g}$ einen höheren Fehler aufweist, als wir für die Anwendung einer schlecht konditionierten Operation auf den Schätzwert von \mathbf{g} vorhersagen würden.

Es ist auch extrem wichtig, die Mini-Batches zufällig auszuwählen. Für die Berechnung eines erwartungstreuen Schätzwerts des erwarteten Gradienten aus einer Menge von Stichproben müssen diese Stichproben unbedingt unabhängig sein. Außerdem möchten wir, dass zwei aufeinanderfolgende Gradientenschätzungen unabhängig voneinander sind, sodass zwei aufeinanderfolgende Mini-Batches mit Beispielen ebenfalls unabhängig voneinander sein sollten. Viele Datensätze sind von sich aus so sortiert, dass aufeinanderfolgende Beispiele eine hohe Korrelation aufweisen. Als Beispiel mag ein medizinischer Datensatz, nämlich eine lange Liste mit Testergebnissen von Blutproben dienen. Diese Liste führt vermutlich zunächst fünf zu unterschiedlichen Zeitpunkten genommene Proben des ersten Patienten auf, dann drei Blutproben des zweiten Patienten, dann Blutproben einer dritten Person usw. Würden wir nun die Beispiele aus der Liste einfach in der vorhandenen Reihenfolge ziehen, wären die Mini-Batches extrem verzerrt, denn jedes davon würde hauptsächlich Daten nur eines Patienten von vielen enthalten. In Fällen, in denen die Reihenfolge des Datensatzes eine gewisse Bedeutung hat, müssen die Beispiele vor dem Auswählen von Mini-Batches durchgemischt werden. Bei sehr großen Datensätzen, zum Beispiel Daten eines Datenzentrums mit Milliarden von Beispielen, ist das wirklich gleichförmige und zufällige Ziehen von Stichproben bei jedem Konstruieren eines Mini-Batches unpraktisch. Zum Glück reicht es in der Praxis aus, die Reihenfolge des Datensatzes einmal zu mischen und ihn dann in gemischter Form zu speichern. Dadurch entsteht eine unveränderliche Menge möglicher Mini-Batches aus aufeinanderfolgenden Beispielen, die alle später trainierten Modelle verwenden. Jedes einzelne Modell ist gezwungen, diese

Reihenfolge bei jedem einzelnen Durchlauf der Trainingsdaten einzuhalten. Diese Abweichung von einem wirklich zufälligen Auswahlprozess scheint sich nicht sonderlich nachteilig auszuwirken. Wird dagegen die Durchmischung der Beispiele nicht in irgendeiner Form durchgeführt, kann die Wirksamkeit des Algorithmus stark eingeschränkt werden.

Viele Optimierungsprobleme im Machine Learning führen die Zerlegung für die Beispiele gut genug durch, dass sich damit parallel vollständig unabhängige Anpassungen über unterschiedliche Beispiele berechnen lassen. Anders ausgedrückt: Wir können die Anpassung, mit der $J(\mathbf{X})$ für einen Mini-Batch mit den Beispielen \mathbf{X} minimiert wird, zeitgleich mit der Anpassung für mehrere andere Mini-Batches berechnen. Derart asynchrone, parallel verteilte Ansätze werden in Abschnitt 12.1.3 genauer behandelt.

Ein interessanter Gedanke beim stochastischen Mini-Batch-Gradientenabstieg ist, dass es dem Gradienten des tatsächlichen *Generalisierungsfehlers* (Gleichung 8.2) folgt, sofern keine Beispiele wiederholt werden. Die meisten Implementierungen des stochastischen Mini-Batch-Gradientenabstiegs durchmischen den Datensatz einmal und nutzen das Ergebnis dann für mehrere Durchläufe. Beim ersten Durchlauf wird mit jedem Mini-Batch ein erwartungstreuer Schätzwert des tatsächlichen Generalisierungsfehlers berechnet. Im zweiten Durchlauf ist der Schätzwert verzerrt, da er erneutes Ziehen von Stichproben (engl. *resampling*) bereits verwendeter Werte gebildet wird und nicht durch Ziehen neuer angemessener Stichproben aus der datengenerierenden Verteilung.

Dass das stochastische Gradientenabstiegsverfahren den Generalisierungsfehler minimiert, lässt sich am leichtesten im Online Learning erkennen, in dem Beispiele oder Mini-Batches aus einem **Datenstrom** gezogen werden. Mit anderen Worten: Statt eine Trainingsdatenmenge mit unveränderlicher Größe zu verwenden, ähnelt der Klassifikator einem lebenden Wesen, das zu jedem Zeitpunkt ein neues Beispiel sieht, wobei jedes Beispiel (\mathbf{x}, y) aus der datengenerierenden Verteilung $p_{\text{data}}(\mathbf{x}, y)$ stammt. In diesem Fall werden keine Beispiele wiederholt – jede Erfahrung ist eine angemessene Stichprobe aus p_{data} .

Die Gleichwertigkeit lässt sich am leichtesten ableiten, wenn sowohl \mathbf{x} als auch y diskret sind. In diesem Fall lässt sich der Generalisierungsfehler (Gleichung 8.2) als Summe notieren:

$$J^*(\boldsymbol{\theta}) = \sum_{\mathbf{x}} \sum_y p_{\text{data}}(\mathbf{x}, y) L(f(\mathbf{x}; \boldsymbol{\theta}), y), \quad (8.7)$$

mit dem exakten Gradienten

$$\mathbf{g} = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \sum_{\mathbf{x}} \sum_y p_{\text{data}}(\mathbf{x}, y) \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}; \boldsymbol{\theta}), y). \quad (8.8)$$

Wir haben dieselbe Tatsache bereits in Gleichung 8.5 und Gleichung 8.6 für die Log-Likelihood bewiesen; nun stellen wir fest, dass sie außer für die Likelihood auch für andere Funktionen L gilt. Ein ähnliches Ergebnis lässt sich ableiten, wenn \mathbf{x} und y stetig sind, unter schwachen Annahmen bezüglich p_{data} und L .

Wir können daher einen erwartungstreuen Schätzer für den exakten Gradienten des Generalisierungsfehlers erhalten, indem wir einen Mini-Batch mit Beispielen $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ und entsprechenden Zielwerten $y^{(i)}$ aus der datengenerierenden Verteilung p_{data} ziehen und anschließend den Gradienten des Verlusts bezüglich der Parameter des Mini-Batches berechnen:

$$\hat{\mathbf{g}} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \quad (8.9)$$

Durch Anpassen von $\boldsymbol{\theta}$ in Richtung $\hat{\mathbf{g}}$ wird das stochastische Gradientenabstiegsverfahren auf den Generalisierungsfehler angewandt.

Das trifft natürlich nur zu, wenn keine Beispiele wiederverwendet werden. Nichtsdestotrotz ist es meist besser, die Trainingsdatenmenge mehrmals zu durchlaufen – es sei denn, sie ist extrem groß. Werden mehrere solche Epochen genutzt, folgt nur die erste davon dem erwartungstreuen Gradienten des Generalisierungsfehlers. Dennoch bieten die zusätzlichen Epochen genügend Vorteile, denn obwohl sie die Lücke zwischen Trainingsfehler und Testfehler vergrößern, verringern sie doch den Trainingsfehler.

Einige Datensätze wachsen enorm an – schneller als die Rechenleistung –, sodass in Machine-Learning-Anwendungen immer häufiger jedes Trainingsbeispiel nur noch einmal verwendet wird oder sogar nur ein unvollständiger Durchlauf der Trainingsdatenmenge erfolgt. Bei sehr großen Trainingsdatenmengen stellt Überanpassung kein Problem dar. Stattdessen rücken Unteranpassung und Berechnungseffizienz in den Fokus. *Bottou und Bousquet (2008)* behandeln die Auswirkung rechnerischer Flaschenhälse als Folge einer wachsenden Anzahl von Trainingsbeispielen auf den Generalisierungsfehler.

8.2 Herausforderungen bei der Optimierung neuronaler Netze

Die Optimierung ist generell eine besonders schwierige Aufgabe. Bisher hat man im Machine Learning die Problematik der generellen Optimierung (engl. *general optimization*) vermieden, indem die Zielfunktion und Bedingungen zielführend entwickelt worden sind, um sicherzustellen, dass das Optimierungsproblem konvex ist. Beim Trainieren neuronaler Netze führt aber kein Weg um den generellen nichtkonvexen Fall herum. Selbst die konvexe Optimierung hat ihre Komplikationen. In diesem Abschnitt stellen wir einige der wesentlichen Herausforderungen bei der Optimierung im Rahmen des Trainings von tiefen Modellen vor.

8.2.1 Schlechte Konditionierung

Einige Herausforderungen betreffen auch die Optimierung konvexer Funktionen. Die bekannteste davon ist die schlechte Konditionierung der Hesse-Matrix \mathbf{H} . Es handelt sich hier um ein sehr allgemeines Problem bei den meisten numerischen Optimierungen – ob konvex oder andersartig –, das in Abschnitt 4.3.1 genauer beschrieben wurde.

Man nimmt an, dass die schlechte Konditionierung in Trainingsproblemen neuronaler Netze vorliegt. Sie kann entstehen, wenn das stochastische Gradientenabstiegsverfahren »hängt«, also sogar sehr kleine Schritte zu einer Zunahme der Kostenfunktion führen.

Wie in Gleichung 4.9 gezeigt, sagt eine Taylor-Entwicklung zweiter Ordnung der Kostenfunktion vorher, dass ein Schritt $-\epsilon\mathbf{g}$ im Gradientenabstiegsverfahren den Aufwand um

$$\frac{1}{2}\epsilon^2\mathbf{g}^\top \mathbf{H}\mathbf{g} - \epsilon\mathbf{g}^\top \mathbf{g} \quad (8.10)$$

erhöht. Eine schlechte Konditionierung des Gradienten wird zu einem Problem, wenn $\frac{1}{2}\epsilon^2\mathbf{g}^\top \mathbf{H}\mathbf{g}$ den Term $\epsilon\mathbf{g}^\top \mathbf{g}$ übersteigt. Um zu bestimmen, ob die schlechte Konditionierung für das Training eines neuronalen Netzes nachteilig ist, können wir das Quadrat der Norm des Gradienten $\mathbf{g}^\top \mathbf{g}$ und den Ausdruck $\mathbf{g}^\top \mathbf{H}\mathbf{g}$ überwachen. Häufig wird die Norm des Gradienten während des Lernens nicht wesentlich kleiner, aber der Term $\mathbf{g}^\top \mathbf{H}\mathbf{g}$ wächst um mehr als eine Größenordnung an. Als Folge davon wird das Lernen sehr langsam, obwohl ein starker Gradient vorliegt, denn die Lernrate muss reduziert werden, um die noch stärkere Krümmung zu kompensieren. Abbil-

dung 8.1 zeigt ein Beispiel eines deutlich zunehmenden Gradienten während des erfolgreichen Trainings eines neuronalen Netzes.

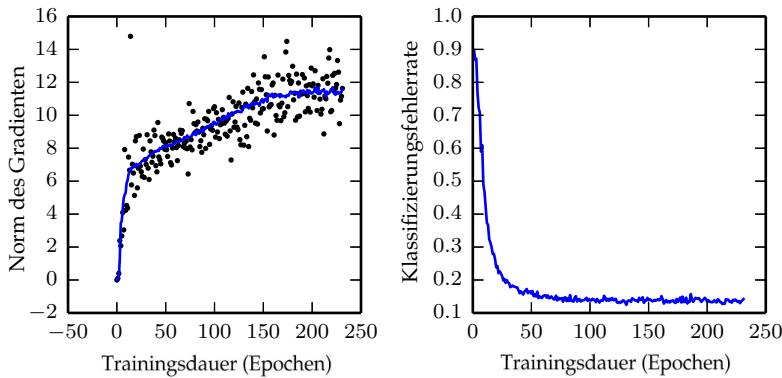


Abbildung 8.1: Das Gradientenabstiegsverfahren erreicht oft keinerlei kritischen Punkt. In diesem Beispiel steigt die Norm des Gradienten während des Trainings eines CNNs zur Objekterkennung. (*Links*) Ein Punktdiagramm zeigt, wie die Normen der einzelnen Gradientenberechnungen sich im Laufe der Zeit entwickeln. Zur besseren Lesbarkeit ist für jede Epoche nur eine Norm des Gradienten dargestellt. Der gleitende Mittelwert aller Norm des Gradienten ist als durchgezogene Linie eingetragen. Die Norm des Gradienten nimmt im Zeitverlauf erkennbar zu, obwohl wir für die Konvergenz des Trainingsverfahrens zu einem kritischen Punkt eine Abnahme erwarten würden. (*Rechts*) Trotz des steigenden Gradienten ist das Training recht erfolgreich. Der Klassifizierungsfehler der Validierungsdatenmenge sinkt auf einen niedrigen Wert.

Obwohl schlechte Konditionierung auch in anderen Bereichen neuronaler Netze vorkommt, sind nicht alle Methoden zu ihrer Vermeidung auch auf neuronale Netze übertragbar. Zum Beispiel ist das Newton-Verfahren ein hervorragendes Hilfsmittel zum Minimieren konvexer Funktionen mit schlecht konditionierten Hesse-Matrizen, aber wie wir in weiteren Abschnitten noch zeigen werden, muss es stark modifiziert werden, bevor es für neuronale Netze verwandt werden kann.

8.2.2 Lokale Minima

Zu den augenfälligsten Merkmalen eines konvexen Optimierungsproblems gehört, dass es auf die Suche eines lokalen Minimums reduziert werden kann. Jedes lokale Minimum ist auch stets ein globales Minimum. Einige konvexe Funktionen weisen einen flachen Bereich anstelle eines punktuellen globalen Minimums auf, aber jeder Punkt in einem solchen Bereich ist eine annehmbare Lösung. Beim Optimieren einer konvexen Funktion ist immer

dann, wenn wir einen beliebigen kritischen Punkt finden, klar, dass wir eine gute Lösung erreicht haben.

Bei nichtkonvexen Funktionen, zu denen neuronale Netze gehören, sind viele lokale Minima möglich. Tatsächlich kann man davon ausgehen, dass jedes tiefe Modell im Wesentlichen immer eine enorm große Anzahl lokaler Minima aufweist. Wir werden allerdings noch sehen, dass das nicht unbedingt ein großes Problem darstellt.

Neuronale Netze und alle Modelle mit mehreren gleichwertig parametrisierten latenten Variablen weisen als Folge des Problems der **Identifizierbarkeit eines Modells** stets mehrere lokale Minima auf. Ein Modell gilt als identifizierbar, wenn eine ausreichend große Trainingsdatenmenge alle Einstellungen der Modellparameter bis auf eine ausschließen kann. Modelle mit latenten Variablen sind häufig nicht identifizierbar, da sich gleichwertige Modelle entwickeln lassen, indem latente Variable untereinander ausgetauscht werden. Wir könnten zum Beispiel in einem neuronalen Netz Schicht 1 durch Austauschen des eingehenden Gewichtungsvektors für die Einheit i mit dem eingehenden Gewichtungsvektor für die Einheit j ändern und dann für die ausgehenden Gewichtungsvektoren ebenso verfahren. Bei m Schichten mit jeweils n Einheiten erhalten wir so $n!^m$ Möglichkeiten zur Anordnung der verdeckten Einheiten. Diese Art der Nichtidentifizierbarkeit wird als **Weight Space Symmetry** (Symmetrie des Gewichtungsraums) bezeichnet.

Viele Arten neuronaler Netze weisen neben der Weight Space Symmetry noch weitere Ursachen für eine Nichtidentifizierbarkeit auf. So können wir in jedem rektifizierten linearen oder Maxout-Netz alle eingehenden Gewichte und Verzerrungen einer Einheit um den Faktor α skalieren, sofern wir alle ihre ausgehenden Gewichte um den Faktor $\frac{1}{\alpha}$ skalieren. Somit liegt – sofern die Kostenfunktion keine Terme wie Weight Decay enthält, die direkt von den Gewichten und nicht von den Ausgaben der Modelle abhängen – jedes lokale Minimum eines rektifizierten linearen oder Maxout-Netzes auf einer $(m \times n)$ -dimensionalen Hyperbel gleichwertiger lokaler Minima.

Die Identifizierbarkeit eines Modells führt dazu, dass die Kostenfunktion eines neuronalen Netzes eine extrem große oder sogar überabzählbar unendliche Menge lokaler Minima aufweisen kann. Allerdings sind all diese lokalen Minima (die Folge der Nichtidentifizierbarkeit sind) für den Wert der Kostenfunktion gleichwertig. Daher stellen diese lokalen Minima keine problematische Form der Nichtkonvexität dar.

Lokale Minima können dann zu einem Problem werden, wenn sie gegenüber dem globalen Minimum einen hohen Aufwand aufweisen. Man könnte

kleine neuronale Netze konstruieren – sogar ohne verdeckte Einheiten –, die lokale Minima aufweisen, die höhere Kosten als das globale Minimum nach sich ziehen (*Sontag und Sussman, 1989; Brady et al., 1989; Gori und Tesi, 1992*). Wenn es häufig lokale Minima mit hohem Aufwand gibt, könnte dies ein ernstzunehmendes Problem für Optimierungsalgorithmen auf Gradientenbasis darstellen.

Ob das Newton-Verfahren im praktischen Einsatz viele lokale Minima mit hohem Aufwand ermitteln kann und ob die Optimierungsalgorithmen diese auffinden, bleiben offene Fragen. Viele Jahre lang war in der Praxis die Ansicht vorherrschend, dass lokale Minima ein allgemein schwieriges Problem bei der Optimierung neuronaler Netze seien. Das scheint heute jedoch nicht mehr der Fall zu sein. Zwar ist das Problem weiterhin Gegenstand aktueller Forschungen, aber Fachleute vermuten mittlerweile, dass die meisten lokalen Minima bei ausreichend großen neuronalen Netzen einen geringen Kostenfunktionswert aufweisen und dass es nicht wichtig ist, ein echtes globales Minimum zu finden, sondern dass es vielmehr darum geht, einen Punkt im Parameterraum zu bestimmen, der einen geringen – aber nicht den niedrigsten – Aufwand aufweist (*Saxe et al., 2013; Dauphin et al., 2014; Goodfellow et al., 2015; Choromanska et al., 2014*).

In der Praxis werden nahezu sämtliche Probleme bei der Optimierung neuronaler Netze den lokalen Minima angelastet. Wir empfehlen, eine sorgfältige Prüfung spezifischer Probleme vorzunehmen. Ein Test zum Ausschließen lokaler Minima als Problem besteht im Plotten der Norm des Gradienten über die Zeit. Wenn die Norm des Gradienten nicht auf eine vernachlässigbare Größe abnimmt, sind weder lokale Minima noch andere Arten kritischer Punkte das Problem. In hochdimensionalen Räumen kann es sich als sehr schwierig erweisen, lokale Minima zuverlässig als Problem zu erkennen. Neben lokalen Minima weisen noch viele weitere Strukturen kleine Gradienten auf.

8.2.3 Plateaus, Sattelpunkte und andere flache Bereiche

In vielen hochdimensionalen nichtkonvexen Funktionen sind lokale Minima (und Maxima) tatsächlich rar gesät – zumindest im Gegensatz zu einer anderen Art von Punkt, an dem der Gradient Null ist: dem Sattelpunkt. Einige Punkte in der Nähe eines Sattels weisen höhere Kosten als dieser auf, andere geringere. Im Sattelpunkt gibt es in der Hesse-Matrix sowohl positive als auch negative Eigenwerte. Bei Punkten, die entlang der den positiven Eigenwerten zugeordneten Eigenvektoren liegen, ist der Aufwand (die Kosten) größer als beim Sattelpunkt, bei Punkten entlang der negativen

Eigenwerte geringer. Sie können sich einen Sattelpunkt so vorstellen, dass er gleichzeitig ein lokales Minimum entlang eines Querschnitts der Kostenfunktion und ein lokales Maximum entlang eines anderen Querschnitts ist. Abbildung 4.5 verdeutlicht dies.

Viele Klassen zufälliger Funktionen legen das folgende Verhalten an den Tag: In geringdimensionalen Räumen sind lokale Minima häufig, wogegen in höherdimensionalen Räumen lokale Minima selten und Sattelpunkte häufiger sind. Für eine solche Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$ wächst das erwartete Verhältnis der Anzahl von Sattelpunkten zu lokalen Minima exponentiell mit n . Um dieses Verhalten besser zu verstehen, beachten Sie, dass die Hesse-Matrix in einem lokalen Minimum nur positive Eigenwerte aufweist. In einem Sattelpunkt verfügt die Hesse-Matrix dagegen über eine Mischung aus positiven und negativen Eigenwerten. Stellen Sie sich vor, dass das Vorzeichen für jeden der Eigenwerte durch Münzwurf bestimmt wird. In einer einzelnen Dimension lässt sich ein lokales Minimum leicht durch einmaliges Werfen der Münze mit dem Ergebnis Kopf ermitteln. Im n -dimensionalen Raum ist es exponentiell unwahrscheinlich, dass alle n Münzwürfe Kopf zeigen. *Dauphin et al. (2014)* behandeln die relevanten theoretischen Grundlagen hierzu.

Eine erstaunliche Eigenschaft vieler zufälliger Funktionen ist, dass die Eigenwerte der Hesse-Matrix in Bereichen mit geringerem Aufwand eher zur Positivität neigen. Für unseren Münzwurf heißt das, dass es wahrscheinlicher ist, dass Kopf n -mal fällt, wenn wir uns an einem kritischen Punkt mit niedrigen Kosten befinden. Außerdem bedeutet es, dass der Aufwand (die Kosten) für lokale Minima sehr viel wahrscheinlicher gering als hoch ist. Kritische Punkte mit hohem Aufwand sind sehr viel wahrscheinlicher Sattelpunkte. Kritische Punkte mit extrem hohem Aufwand sind wahrscheinlicher lokale Maxima.

Das gilt für viele Klassen von zufälligen Funktionen. Aber trifft das auch auf neuronale Netze zu? *Baldi und Hornik (1989)* haben theoretisch gezeigt, dass flache Autoencoder (Feedforward-Netze, die daraufhin trainiert wurden, ihre Eingabe in die Ausgabe zu kopieren, siehe Kapitel 14) ohne Nichtlinearitäten globale Minima und Sattelpunkte aufweisen, aber keine lokalen Minima mit höheren Kosten als beim globalen Minimum. Sie haben – ohne Beweis – beobachtet, dass diese Ergebnisse auch für tiefere Netze ohne Nichtlinearitäten gelten. Die Ausgabe solcher Netze ist eine lineare Funktion der Eingabe, aber sie eignen sich gut als Modelle zur Untersuchung von nichtlinearen neuronalen Netzen, da ihre Verlustfunktion eine nichtkonvexe Funktion ihrer Parameter ist. Solche Netze sind im Grunde genommen lediglich multiple Matrizen, die zusammengesetzt werden. *Saxe*

et al. (2013) lieferten exakte Lösungen für die komplette Lerndynamik in derartigen Netzen und haben gezeigt, dass das Lernen in diesen Modellen viele der qualitativen Merkmale erfasst, die beim Trainieren tiefer Modelle mit nichtlinearen Aktivierungsfunktionen beobachtet werden. *Dauphin et al.* (2014) haben experimentell bestätigt, dass die Konditionierung der Hesse-Matrix verbessert wird, und beobachtet, dass der Trick der Zentrierung äquivalent zu einer weiteren Methode des Boltzmann-Machine-Learnings ist, dem sogenannten *enhanced Gradient* (erweiterter Gradient). *Choromanska et al.* (2014) liefern weitere theoretische Argumente und zeigen, dass dies auch für eine andere Klasse hochdimensionaler zufälliger Funktionen gilt, die mit neuronalen Netzen im Zusammenhang stehen.

Welche Folgen hat die starke Ausbreitung der Sattelpunkte für die Trainingsalgorithmen? Bei Optimierungen erster Ordnung, also Algorithmen, die lediglich Gradienteninformationen nutzen, ist die Lage unklar. Der Gradient kann in der Nähe eines Sattels häufig sehr klein werden. Andererseits scheint das Gradientenabstiegsverfahren erfahrungsgemäß in vielen Fällen in der Lage zu sein, den Sattelpunkten zu entkommen. *Goodfellow et al.* (2015) haben Darstellungen mehrerer Lerntrajektorien modernster neuronaler Netze erstellt – Abbildung 8.2 zeigt ein Beispiel. Diese Visualisierungen weisen eine Abflachung der Kostenfunktion in der Nähe eines markanten Sattels auf, an dem die Gewichte alle Null sind, aber sie zeigen auch, dass die Trajektorie des Gradientenabstiegs diesen Bereich schnell verlässt. *Goodfellow et al.* (2015) behaupten außerdem, dass analytisch gezeigt werden kann, dass ein Gradientenabstiegsverfahren mit stetiger Zeit von nahegelegenen Sattelpunkten abgestoßen und nicht angezogen wird; wobei sich die Situation in realistischeren Verwendungsfällen des Gradientenabstiegsverfahrens unterscheiden könnte.

Für das Newton-Verfahren stellen Sattelpunkte offensichtlich ein Problem dar. Das Gradientenabstiegsverfahren ist für einen Abstieg gemacht, nicht explizit für die Suche nach einem kritischen Punkt. Das Newton-Verfahren dagegen soll einen Punkt finden, in dem der Gradient Null ist. Ohne entsprechende Anpassung kann es auf einen Sattel springen. Die starke Ausbreitung von Sattelpunkten in hochdimensionalen Räumen erklärt vermutlich, warum Verfahren zweiter Ordnung das Gradientenabstiegsverfahren beim Training neuronaler Netze nicht ablösen konnten. *Dauphin et al.* (2014) stellten ein **sattelpunktfreies Newton-Verfahren** (engl. *saddle-free Newton method*) für die Optimierung zweiter Ordnung vor und zeigten, dass es der herkömmlichen Variante deutlich überlegen ist. Es ist schwierig, Verfahren zweiter Ordnung für große neuronale Netze zu skalieren, aber der sattelpunktfreie Ansatz ist diesbezüglich vielversprechend.

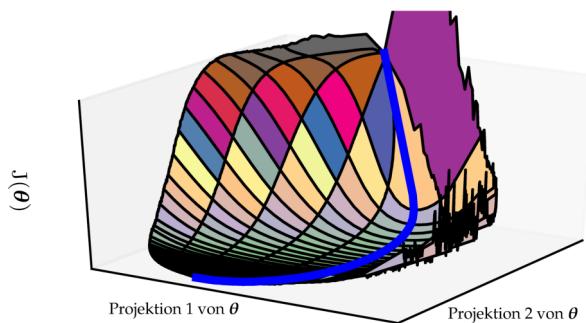


Abbildung 8.2: Eine Darstellung der Kostenfunktion eines neuronalen Netzes. Diese Visualisierungen fallen für neuronale Feedforward-Netze, CNNs und RNNs für die echte Objekterkennung und die Verarbeitung natürlicher Sprache ähnlich aus. Überraschenderweise enthalten die Darstellungen meist kaum verdächtige Hindernisse. Vor dem Aufstieg des stochastischen Gradientenabstiegsverfahrens beim Training sehr großer Modelle etwa im Jahr 2012 dachte man, dass die Oberflächen der Kostenfunktionen in neuronalen Netzen viel mehr nichtkonvexe Strukturen aufweisen würden, als diese Projektionen offenbaren. Das größte Hindernis, das durch diese Projektion aufgezeigt wird, ist ein Sattelpunkt mit hohen Kosten in der Nähe der Parameterinitialisierung. Allerdings zeigt der dicke schwarze Pfad, dass die Trainingstrajektorie des statistischen Gradientenabstiegsverfahrens diesem Sattelpunkt schnell entkommt. Der Großteil der Trainingsdauer findet in dem relativ flachen Tal der Kostenfunktion statt, möglicherweise aufgrund des hohen Rauschens im Gradienten, einer schlechten Konditionierung der Hesse-Matrix in diesem Bereich oder einfach der Notwendigkeit, den hohen »Berg« zu umgehen, der in der Abbildung als indirekter, gewundener Pfad sichtbar ist. (Abbildung mit freundlicher Genehmigung von *Goodfellow et al. (2015)* angepasst)

Es gibt neben Minima und Sattelpunkten noch andere Arten von Punkten, an denen der Gradient Null ist. Maxima ähneln aus Optimierungssicht stark den Sattelpunkten – viele Algorithmen werden davon im Gegensatz zum nicht modifizierten Newton-Verfahren nicht angezogen. Maxima vieler Klassen von zufälligen Funktionen werden im hochdimensionalen Raum exponentiell selten – ebenso wie Minima.

Es mag auch breite flache Bereiche mit einem konstanten Wert geben. Dort sind der Gradient und die Hesse-Matrix immer Null. Derart ausgeartete Orte stellen gewaltige Probleme für alle numerischen Optimierungsalgorithmen dar. In einem konvexen Problem muss ein breiter flacher Bereich vollständig aus globalen Minima bestehen, aber bei einem generellen Optimierungsproblem kann dieser Bereich auch einem hohen Wert der Zielfunktion entsprechen.

8.2.4 Klippen und explodierende Gradienten

Neuronale Netze mit vielen Schichten enthalten häufig extrem steile Bereiche, die an Klippen erinnern (siehe Abbildung 8.3). Sie entstehen, wenn mehrere große Gewichte miteinander multipliziert werden. Angesichts extrem steiler Klippen kann die Gradientenaktualisierung zu einer extremen Parameterverschiebung führen, sodass die Klippe komplett verlassen wird.

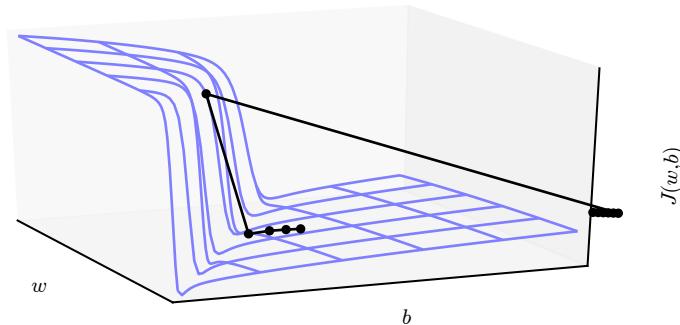


Abbildung 8.3: Die Zielfunktion für stark nichtlineare tiefe neuronale Netze oder für RNNs enthält häufig starke Nichtlinearitäten (engl. *sharp nonlinearities*) im Parameterraum, die aus einer Multiplikation mehrerer Parameter entstehen. Diese Nichtlinearitäten führen an einigen Stellen zu sehr hohen Ableitungen. Wenn die Parameter sich einem solchen Klippenbereich nähern, kann eine Gradientenaktualisierung die Parameter praktisch sehr weit wegkatapultieren, sodass unter Umständen sogar der Großteil der bisher erfolgten Optimierungsarbeit verloren geht. (Abbildung mit freundlicher Genehmigung von Pascanu et al. (2013) angepasst)

Die Klippen sind schwierig, unabhängig davon, ob wir sie von oben oder von unten betrachten, aber zum Glück lassen sich ihre stärksten Auswirkungen mithilfe der als **Gradienten-Clipping** bezeichneten Heuristik (siehe Abschnitt 10.11.1) umgehen. Dazu rufen wir uns ins Gedächtnis zurück, dass der Gradient nicht die optimale Schrittweite vorgibt, sondern nur die optimale Richtung in einem infinitesimal kleinen Bereich. Wenn also der klassische Algorithmus für das Gradientenabstiegsverfahren einen sehr großen Schritt vorschlägt, greift das Gradienten-Clipping ein und verringert die Schrittweite, sodass ein Verlassen des Bereichs, für den der Gradient die Richtung des näherungsweise steilsten Abstiegs angibt, eher unwahrscheinlich wird. Klippen sind in den Kostenfunktionen von RNNs am verbreitetsten, da in diesen Modellen viele Faktoren miteinander multipliziert werden – ein Faktor pro Zeitschritt. Lange zeitliche Abfolgen führen somit zu einer extrem hohen Anzahl von Multiplikationen.

8.2.5 Langfristige Abhangigkeiten

Eine weitere Schwierigkeit fur Optimierungsalgorithmen in neuronalen Netzen entsteht, wenn der Berechnungsgraph sehr tief wird. Feedforward-Netze mit vielen Schichten weisen solche tiefen Berechnungsgraphen auf. Das gilt ebenso fur RNNs (siehe Kapitel 10), in denen sehr tiefe Berechnungsgraphen durch das wiederholte Anwenden derselben Operation in jedem Schritt einer langen zeitlichen Abfolge entstehen. Eine wiederholte Anwendung derselben Parameter fuhrt zu besonders ausgepragten Schwierigkeiten.

Ein Beispiel: Gegeben sei ein Berechnungsgraph, der einen Pfad mit wiederholter Multiplikation mit einer Matrix \mathbf{W} enthalt. Nach t Schritten entspricht dies einer Multiplikation mit \mathbf{W}^t . Angenommen, \mathbf{W} weist eine Eigenwertzerlegung $\mathbf{W} = \mathbf{V} \text{diag}(\boldsymbol{\lambda}) \mathbf{V}^{-1}$ auf. In diesem einfachen Fall sehen wir direkt, dass

$$\mathbf{W}^t = (\mathbf{V} \text{diag}(\boldsymbol{\lambda}) \mathbf{V}^{-1})^t = \mathbf{V} \text{diag}(\boldsymbol{\lambda})^t \mathbf{V}^{-1} \quad (8.11)$$

ist. Alle Eigenwerte λ_i , die nicht in der Nahe eines Absolutbetrags von 1 liegen, explodieren (wenn ihr Betrag groer 1 ist) oder verschwinden (wenn ihr Betrag kleiner 1 ist). Das **Problem der verschwindenden und explodierenden Gradienten** (engl. *vanishing and exploding gradient problem*) bezieht sich auf die Tatsache, dass Gradienten in einem solchen Graphen abhangig von $\text{diag}(\boldsymbol{\lambda})^t$ skaliert werden. Verschwindende Gradienten erschweren das Bestimmen der Parameterrichtung fur eine bessere Kostenfunktion, explodierende Gradienten konnen den Lernprozess instabil werden lassen. Die zuvor beschriebenen Klippen, durch die das Gradienten-Clipping ausgelost wird, sind ein Beispiel fur explodierende Gradienten.

Die hier beschriebene wiederholte Multiplikation mit \mathbf{W} in jedem Zeitschritt ahnelt dem Algorithmus der **Potenzmethode** fur die Suche des grosten Eigenwerts einer Matrix \mathbf{W} und des zugehorigen Eigenvektors. So gesehen ist es nicht uberraschend, dass $\mathbf{x}^\top \mathbf{W}^t$ letztlich alle Komponenten von \mathbf{x} verwirft, die orthogonal zum Haupteigenvektor von \mathbf{W} sind.

RNNs nutzen dieselbe Matrix \mathbf{W} in jedem Zeitschritt. Das gilt allerdings nicht fur Feedforward-Netze, sodass selbst sehr tiefe Feedforward-Netze das Problem der verschwindenden und explodierenden Gradienten grotenteils vermeiden konnen (*Sussillo, 2014*).

Wir wenden uns den Herausforderungen beim Trainieren von RNNs in Abschnitt 10.7 wieder zu; zunachst gehen wir aber naher auf RNNs ein.

8.2.6 Inexakte Gradienten

Die meisten Optimierungsalgorithmen beruhen auf der Annahme, dass wir den exakten Gradienten oder die Hesse-Matrix kennen bzw. ermitteln können. In der Praxis liegen oft nur verrauschte oder sogar verzerrte Schätzwerte dieser Größen vor. Fast jeder Deep-Learning-Algorithmus verlässt sich auf stichprobenbasierende Schätzwerte – zumindest bei der Verwendung eines Mini-Batches mit Trainingsbeispielen zur Gradientenberechnung.

In anderen Fällen ist die Zielfunktion, die wir minimieren möchten, nicht effizient berechenbar. Wenn die Zielfunktion nicht effizient berechenbar ist, so ist normalerweise auch ihr Gradient nicht effizient berechenbar. Dann können wir den Gradienten nur approximieren. Diese Probleme treten meist in Verbindung mit fortschrittlicheren Modellen auf, die wir in Teil III behandeln. Zum Beispiel gibt die kontrastive Divergenz (engl. *contrastive divergence*) uns ein Verfahren zur Approximation des Gradienten der nicht effizient berechenbaren Log-Likelihood einer Boltzmann-Maschine an die Hand.

Diverse Optimierungsalgorithmen für neuronale Netze sind darauf ausgelegt, Unvollkommenheiten bei der Gradientenschätzung zu berücksichtigen. Das Problem lässt sich auch durch Auswahl einer Ersatzverlustfunktion umgehen, die einfacher als der tatsächliche Verlust approximiert werden kann.

8.2.7 Schlechte Korrespondenz zwischen lokaler und globaler Struktur

Vieler der bisher behandelten Probleme beziehen sich auf die Eigenschaften der Verlustfunktion in einem Einzelpunkt – ein Einzelschritt kann sich als schwierig erweisen, wenn $J(\boldsymbol{\theta})$ im aktuellen Punkt $\boldsymbol{\theta}$ schlecht konditioniert ist, wenn $\boldsymbol{\theta}$ auf einer Klippe liegt oder wenn $\boldsymbol{\theta}$ ein Sattelpunkt ist, der den Blick auf den möglichen Gradientenabstieg verschleiert.

All diese Probleme in einem Einzelpunkt lassen sich überwinden und dennoch kann das Ergebnis schlecht sein, wenn die Richtung der stärksten lokalen Verbesserung nicht auf entfernte Bereiche mit sehr viel geringeren Kosten weist.

Goodfellow et al. (2015) argumentieren, dass ein Großteil der Trainingsdauer der Länge der Trajektorie geschuldet ist, die zum Ermitteln der Lösung benötigt wird. Abbildung 8.2 zeigt, dass die Lerntrajektorie eine lange Zeit einen weiten Bogen um eine bergförmige Struktur beschreibt.

Thema vieler Forschungsarbeiten bezüglich der Optimierungsschwierigkeiten war, ob das Training ein globales Minimum, ein lokales Minimum oder einen Sattelpunkt erreicht. In der Praxis erreichen neuronale Netze jedoch keinen derartigen kritischen Punkt. Abbildung 8.1 zeigt, dass neuronale Netze häufig keinen Bereich mit kleinem Gradienten erreichen. Tatsächlich müssen solche kritischen Punkte noch nicht einmal existieren. Zum Beispiel kann der Verlustfunktion $-\log p(y | \mathbf{x}; \boldsymbol{\theta})$ ein globaler Tiefpunkt fehlen; sie könnte sich stattdessen mit zunehmender Konfidenz des Modells asymptotisch einem Wert annähern. Für einen Klassifikator mit diskretem y und $p(y | \mathbf{x})$ aus einer softmax-Funktion kann die negative Log-Likelihood beliebig nahe Null werden, wenn das Modell für jedes Beispiel in der Trainingsdatenmenge eine korrekte Klassifizierung vornimmt, aber es ist unmöglich, tatsächlich den Nullwert zu erreichen. Ebenso kann ein Modell reellwertiger Werte $p(y | \mathbf{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$ eine negative Log-Likelihood aufweisen, die sich der negativen Unendlichkeit annähert – sofern $f(\boldsymbol{\theta})$ den Wert aller y -Ziele der Trainingsdatenmenge korrekt vorhersagen kann, erhöht der Lernalgorithmus β grenzenlos. Abbildung 8.4 zeigt ein Beispiel für eine nicht erfolgreiche lokale Optimierung zum Aufspüren eines guten Werts für die Kostenfunktion, obwohl keine anderen lokalen Minima oder Sattelpunkte existieren.

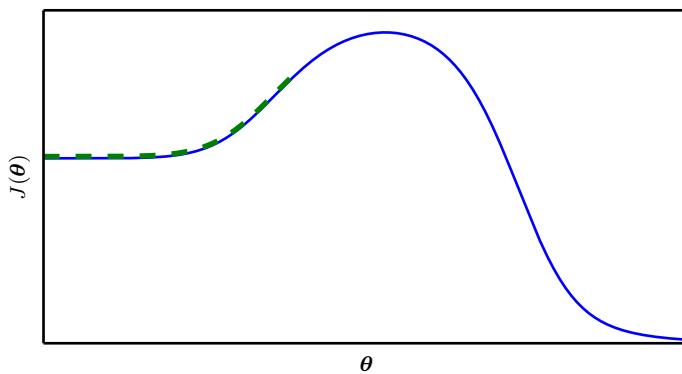


Abbildung 8.4: Die Optimierung auf Basis lokaler Abstiege kann fehlschlagen, wenn die lokale Oberfläche nicht in Richtung der globalen Lösung weist. Diese Abbildung zeigt ein Beispiel dafür – obwohl es keine Sattelpunkte oder lokale Minima gibt. Die Kostenfunktion im Beispiel enthält nur Asymptoten in Richtung niedriger Werte, nicht in Richtung Minima. Das Hauptproblem hierbei liegt in der Initialisierung auf der falschen Seite des »Bergs«, der so nicht umgangen werden kann. Im höherdimensionalen Raum sind Lernalgorithmen häufig in der Lage, solche Berge zu umgehen, aber die Trajektorie dafür kann lang sein und übermäßig viel Zeit kosten (vgl. Abbildung 8.2).

Thema künftiger Forschung muss sein, ein besseres Verständnis der Faktoren zu entwickeln, die die Länge der Lerntrajektorie beeinflussen, und das Ergebnis des Prozesses besser zu charakterisieren.

Viele vorhandene Forschungsrichtungen suchen nach guten Ausgangspunkten für Probleme mit komplexer globaler Struktur, anstatt sich auf die Entwicklung von Algorithmen zu konzentrieren, die nichtlokale Bewegungen nutzen.

Das Gradientenabstiegsverfahren und praktisch alle Lernalgorithmen, die ein erfolgreiches Trainieren neuronaler Netze ermöglichen, bauen auf kleinen lokalen Bewegungen auf. In den bisherigen Abschnitten haben wir uns in erster Linie damit befasst, dass das Bestimmen der korrekten Richtung dieser lokalen Bewegungen keine leichte Angelegenheit ist. Möglicherweise können wir einige Eigenschaften der Zielfunktion, zum Beispiel ihren Gradienten, nur näherungsweise berechnen, sodass unser Schätzwert für die korrekte Richtung eine Verzerrung oder eine Varianz aufweist. In diesen Fällen mag der lokale Abstieg einen angemessen kurzen Pfad zu einer gültigen Lösung aufzeigen oder auch nicht, ohne dass wir dem lokalen Abstiegspfad tatsächlich folgen können. Vielleicht gibt es ein Problem mit der Zielfunktion (schlechte Konditionierung oder unstetige Gradienten), das dazu führt, dass der Bereich, in dem der Gradient ein gutes Modell der Zielfunktion bietet, sehr klein wird. In diesen Fällen mag der lokale Abstieg mit einer Schrittweite ϵ einen angemessen kurzen Pfad zur Lösung aufzeigen, aber wir können die Richtung des lokalen Abstiegs nur mit einer Schrittweite $\delta \ll \epsilon$ berechnen. In diesen Fällen mag der lokale Abstieg einen Pfad zur Lösung aufweisen, aber dieser Pfad enthält viele Schritte, was einen hohen Berechnungsaufwand nach sich zieht. Manchmal geben lokale Informationen keinerlei Hilfestellung, zum Beispiel wenn die Funktion einen breiten und flachen Bereich aufweist oder wenn wir aus irgendeinem Grund exakt einen kritischen Punkt treffen (Letzteres geschieht eigentlich nur bei Verfahren, die explizit nach kritischen Punkten suchen, zum Beispiel dem Newton-Verfahren). In diesen Fällen zeigt der lokale Abstieg keinerlei Pfad zur Lösung auf. In anderen Fällen erweisen sich lokale Bewegungen als zu »gierig« (engl. *greedy*) und führen uns zwar auf einen absteigenden Pfad, der allerdings von der Lösung wegführt (vgl. Abbildung 8.4) oder auf eine unnötig lange Trajektorie zur Lösung (vgl. Abbildung 8.2). Wir wissen derzeit noch nicht, welche dieser Probleme den stärksten Einfluss auf die Schwierigkeiten beim Optimieren neuronaler Netze haben. Das bleibt Gegenstand aktueller Forschungsbestrebungen.

Ungeachtet dessen, welche dieser Probleme die wichtigsten sind, lassen sie sich alle vermeiden, wenn es einen Bereich des Raums gibt, der angemessen direkt über einen Pfad, dem der lokale Abstieg folgen kann, mit einer Lösung

verbunden ist, und wenn wir in der Lage sind, den Lernprozess innerhalb dieses angemessenen Bereichs zu initialisieren. Das zeigt, wie nützlich die Auswahl guter Anfangspunkte für klassische Optimierungsalgorithmen ist.

8.2.8 Theoretische Grenzen der Optimierung

Mehrere theoretische Ergebnisse zeigen, dass es Grenzen bei der Leistung von Optimierungsalgorithmen für neuronale Netze gibt (*Blum und Rivest*, 1992; *Judd*, 1989; *Wolpert und MacReady*, 1997). Allerdings haben diese meist kaum eine praktische Auswirkung.

Einige der theoretischen Ergebnisse gelten nur, wenn die Einheiten eines neuronalen Netzes diskrete Werte ausgeben. Die meisten Einheiten in neuronalen Netzen geben sanft zunehmende Werte aus, die eine Optimierung mittels lokaler Suche möglich machen. Einige theoretische Ergebnisse zeigen, dass es nicht effizient lösbar Problemklassen gibt – aber es ist nicht einfach, zu entscheiden, ob ein bestimmtes Problem in diese Klasse fällt. Weitere Ergebnisse zeigen, dass die Suche einer Lösung für ein Netz mit einer bestimmten Größe nicht effizient durchführbar ist. In der Praxis finden wir die Lösung dennoch, indem wir ein größeres Netz mit sehr viel mehr Parametereinstellungen nutzen. Außerdem interessieren wir uns beim Trainieren neuronaler Netze meist gar nicht für das exakte Minimum einer Funktion, sondern lediglich dafür, ihren Wert so weit zu reduzieren, dass sich ein guter Generalisierungsfehler ergibt. Ob ein Optimierungsalgorithmus dieses Ziel erreichen kann, ist in der theoretischen Analyse extrem schwierig zu sagen. Die Entwicklung realistischer Grenzen für die Leistung von Optimierungsalgorithmen ist daher weiterhin ein wichtiges Ziel der Machine-Learning-Forschung.

8.3 Grundlegende Algorithmen

Wir haben in Abschnitt 4.3 bereits den Algorithmus für das Gradientenabstiegsverfahren vorgestellt, der dem Gradienten einer vollständigen Trainingsdatenmenge abwärts folgt. Der Vorgang lässt sich mithilfe des stochastischen Gradientenabstiegsverfahrens deutlich beschleunigen, bei dem der Gradient zufällig ausgewählter Mini-Batches abwärts verfolgt wird (vgl. Abschnitt 5.9 und Abschnitt 8.1.3).

8.3.1 Stochastisches Gradientenabstiegsverfahren

Das stochastische Gradientenabstiegsverfahren (engl. *stochastic gradient descent*, SGD) und seine Varianten dürfen zu den meistverwendeten Optimierungsalgorithmen im Machine Learning insgesamt und insbesondere im Deep Learning gehören. Wie in Abschnitt 8.1.3 gezeigt, lässt sich eine erwartungstreue Gradientenschätzung bestimmen, indem der durchschnittliche Gradient eines Mini-Batches mit m Beispielen, die u.i.v. aus der datengenerierenden Verteilung gezogen wurden, verwendet wird.

Algorithmus 8.1 zeigt, wie diese Gradientenschätzung mit dem Gradientenabstiegsverfahren durchgeführt wird.

Algorithmus 8.1 Aktualisierung des stochastischen Gradientenabstiegsverfahrens (SGD) in der Trainingsiteration k

Require: Lernrate ϵ_k

Require: Ausgangsparameter $\boldsymbol{\theta}$

while Abbruchbedingung nicht erfüllt **do**

Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen aus der Trainingsdatenmenge $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ mit entsprechenden Zielwerten $\mathbf{y}^{(i)}$.

Berechnen der Gradientenschätzung: $\hat{\mathbf{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$.

Aktualisiere: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\mathbf{g}}$.

end while

Ein unabdingbarer Parameter für den SGD-Algorithmus ist die Lernrate. Bisher haben wir für das SGD stets eine unveränderliche Lernrate ϵ angenommen. In der Praxis muss die Lernrate nach und nach verringert werden, sodass sie zur Iteration k als ϵ_k notiert wird.

Das liegt daran, dass der Gradientenschätzer für das SGD eine Rauschquelle (das zufällige Ziehen von m Trainingsbeispielen) einbringt, die auch bei Erreichen des Minimums nicht verschwindet. Dagegen wird der wahre Gradient einer Gesamt-Kostenfunktion zunächst klein und dann $\mathbf{0}$, wenn wir uns beim Batch-Gradientenabstieg einem Minimum annähern und es erreichen, weswegen der Batch-Gradientenabstieg eine unveränderliche Lernrate nutzen kann. Eine ausreichende Bedingung, die eine Konvergenz des SGDs garantiert, ist

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \quad \text{und} \tag{8.12}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty. \quad (8.13)$$

In der Praxis wird meist die Lernrate bis zur Iteration τ linear verringert:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_{\tau} \quad (8.14)$$

mit $\alpha = \frac{k}{\tau}$. Nach der Iteration τ bleibt ϵ dann üblicherweise konstant.

Die Lernrate kann zwar durch Versuch und Irrtum ermittelt werden, aber meist ist es von Vorteil, sie durch Beobachten von Lernkurven, die die Zielfunktion als Funktion der Zeit darstellen, auszuwählen. Das ist eher Kunst denn Wissenschaft, sodass die meisten Empfehlungen zu diesem Thema mit ein wenig Skepsis betrachtet werden sollten. Für einen linearen Ablauf sind die Parameter ϵ_0 , ϵ_{τ} und τ auszuwählen. Meist kann für τ die Anzahl der Iterationen eingestellt werden, die für einige Hundert Durchläufe durch die Trainingsdatenmenge benötigt werden. ϵ_{τ} sollte in etwa auf ein Prozent des Wertes von ϵ_0 eingestellt werden. Aber wie wird ϵ_0 festgelegt? Ist der Wert zu groß, weist die Lernkurve drastische Schwankungen auf und die Kostenfunktion steigt oft erheblich an. Geringfügige Schwankungen sind in Ordnung, insbesondere, wenn man mit einer stochastischen Kostenfunktion trainiert, wie beispielsweise der Kostenfunktion, die aus der Verwendung von Dropout entsteht. Ist die Lernrate zu gering, macht das Lernen nur langsame Fortschritte. Wenn die anfängliche Lernrate zu gering ist, bleibt das Lernen vielleicht sogar mit einem hohen Kostenwert hängen. Typischerweise liegt die optimale anfängliche Lernrate (im Sinne der Gesamttrainingsdauer und des endgültigen Aufwands) über der Lernrate, die nach den ersten etwa 100 Iterationen die beste Leistung zeigt. Daher sollten die ersten paar Iterationen beobachtet werden, um dann eine Lernrate zu verwenden, die größer als die Lernrate mit der bis dahin besten Leistung ist, ohne so groß zu sein, dass es zu einer höheren Instabilität kommt.

Die wichtigste Eigenschaft des SGDs und der zugehörigen Mini-Batch-Optimierung oder Online-Optimierung auf Gradientenbasis ist, dass die Berechnungsduer pro Update nicht mit der Anzahl der Trainingsbeispiele wächst. So ist auch bei sehr vielen Trainingsbeispielen eine Konvergenz möglich. Ist der Datensatz groß genug, kann das SGD innerhalb eines festgelegten Toleranzbereichs gegen den endgültigen Fehler auf den Testdaten konvergieren, bevor die gesamte Trainingsdatenmenge verarbeitet wurde.

Zur Untersuchung der Konvergenzrate eines Optimierungsalgorithmus wird üblicherweise der **Überschussfehler** $J(\theta) - \min_{\theta} J(\theta)$ gemessen, also der Betrag, um den die gegenwärtige Kostenfunktion den geringstmöglichen Betrag überschreitet. Wird das SGD auf ein konkaves Problem angesetzt,

beträgt der Überschussfehler nach k Iterationen $O(\frac{1}{\sqrt{k}})$, wohingegen er im stark konvexen Fall $O(\frac{1}{k})$ beträgt. Diese Grenzen können nur verbessert werden, wenn zusätzliche Voraussetzungen geschaffen werden.

Der Batch-Gradientenabstieg bietet in der Theorie bessere Konvergenzraten als das SGD. Allerdings besagt die Cramér-Rao-Ungleichung (*Cramér*, 1946; *Rao*, 1945), dass der Generalisierungsfehler nicht schneller abnehmen kann als $O(\frac{1}{k})$. *Bottou und Bousquet* (2008) argumentieren, dass es daher nicht lohnenswert sein könnte, einen Optimierungsalgorithmus zu verfolgen, der bei Machine-Learning-Aufgaben schneller als $O(\frac{1}{k})$ konvergiert – eine schnellere Konvergenz entspricht vermutlich einer Überanpassung. Außerdem verdeckt die asymptotische Analyse viele Vorteile des stochastischen Gradientenabstiegsverfahrens nach einer geringen Anzahl Schritte. Bei großen Datenmengen fällt die Fähigkeit des SGDs, schnelle Anfangsfortschritte bei der Bewertung des Gradienten für sehr wenige Beispiele zu erzielen, mehr ins Gewicht, als seine langsame asymptotische Konvergenz. Die meisten der im weiteren Verlauf des Kapitels beschriebenen Algorithmen erzielen Vorteile, die in der Praxis von Bedeutung sind, aber in den konstanten Faktoren unbedeutend werden, die durch die asymptotische Analyse für $O(\frac{1}{k})$ verdeckt werden. Man kann auch die Vorteile von Batch- und stochastischem Gradientenabstiegsverfahren abwägen, indem die Mini-Batch-Größe während des Lernens nach und nach erhöht wird.

Weitere Informationen zum SGD finden Sie in *Bottou* (1998).

8.3.2 Momentum

Das stochastische Gradientenabstiegsverfahren ist nach wie vor ein beliebtes Optimierungsverfahren, mit der das Lernen allerdings manchmal zäh erfolgt. Mit der Momentum-Methode (engl. *method of momentum*) (*Polyak*, 1964) lässt sich das Lernen beschleunigen, insbesondere bei starker Krümmung, kleinen, aber stetigen Gradienten und rauschbehafteten Gradienten. Der Momentum-Algorithmus akkumuliert einen sich exponentiell verringernden gleitenden Mittelwert der bisherigen Gradienten und setzt die Bewegung in deren Richtung fort. Die Auswirkung des Momentums ist in Abbildung 8.5 dargestellt.

Formal betrachtet führt der Momentum-Algorithmus eine Variable v ein, die die Rolle einer Geschwindigkeit übernimmt. Es handelt sich dabei um die Richtung und Geschwindigkeit, mit der die Parameter den Parameterraum durchqueren. Die Geschwindigkeit ist auf ein sich exponentiell verringerndes Mittel des negativen Gradienten eingestellt. Der aus dem

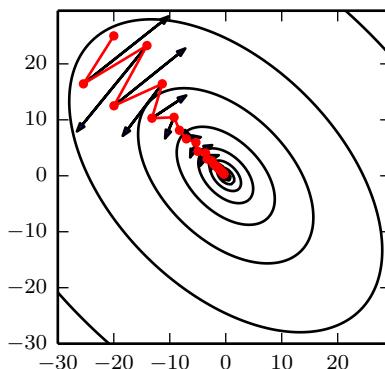


Abbildung 8.5: Momentum soll in erster Linie zwei Probleme lösen, nämlich eine schlechte Konditionierung der Hesse-Matrix und Varianz im stochastischen Gradienten. Hier zeigen wir, wie Momentum das erste dieser Probleme überwindet. Die Höhenlinien stellen eine quadratische Verlustfunktion mit einer schlecht konditionierten Hesse-Matrix dar. Der nach oben links verlaufende Pfad, der die Linien schneidet, zeigt den Pfad der Momentum-Lernregel während der Minimierung dieser Funktion an. In jedem Schritt auf diesem Weg zeichnen wir einen Pfeil ein, der angibt, welchen Schritt der Gradientenabstieg in diesem Punkt nehmen würde. Eine schlecht konditionierte quadratische Zielfunktion ähnelt somit einem langen, schmalen Tal oder einer Schlucht mit sehr steilen Seitenwänden. Das Momentum durchwandert diese Schlucht ganz direkt in Längsrichtung, wohingegen die Gradientenschritte Zeit damit verschwenden, quer zur schmalen Achse der Schlucht hin- und herzuspringen. Sehen Sie sich auch Abbildung 4.6 an, in der das Verhalten des Gradientenabstiegsverfahrens ohne Momentum dargestellt ist.

Englischen übernommene Name **Momentum** (dt. *Impuls*) entstammt einer physikalischen Analogie, in der der negative Gradient eine Kraft ist, die ein Teilchen unter Einhaltung der newtonschen Bewegungsgesetze durch den Parameterraum bewegt. In der Physik ist der Impuls das Produkt aus Masse und Geschwindigkeit. Im Momentum-Lernalgorithmus nehmen wir die Einheitsmasse an, sodass der Geschwindigkeitsvektor \mathbf{v} auch als Impuls (engl. *momentum*) des Teilchens betrachtet werden kann. Ein Hyperparameter $\alpha \in [0, 1)$ bestimmt, wie schnell die Beiträge der bisherigen Gradienten exponentiell abnehmen. Die Update-Regel ergibt sich aus

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\theta} \left(\frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right), \quad (8.15)$$

$$\theta \leftarrow \theta + \mathbf{v}. \quad (8.16)$$

Die Gradientenelemente $\nabla_{\theta} \left(\frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$ werden durch die Geschwindigkeit \mathbf{v} angehäuft. Je größer α relativ zu ϵ ist, desto mehr bishe-

lige Gradienten beeinflussen die aktuelle Richtung. Der SGD-Algorithmus mit Momentum ist in Algorithmus 8.2 zu finden.

Algorithmus 8.2 Stochastisches Gradientenabstiegsverfahren (SGD) mit Momentum

Require: Lernrate ϵ , Momentum-Parameter α

Require: Ausgangsparameter θ , Ausgangsgeschwindigkeit v

while Abbruchbedingung nicht erfüllt **do**

Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen aus der Trainingsdatenmenge $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ mit entsprechenden Zielwerten $\mathbf{y}^{(i)}$.

Berechnen der Gradientenschätzung: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.

Berechnen des Geschwindigkeit-Updates: $v \leftarrow \alpha v - \epsilon g$.

Aktualisiere: $\theta \leftarrow \theta + v$.

end while

Bisher war die Schrittweite einfach das Produkt aus Norm des Gradienten und Lernrate. Jetzt ist die Schrittweite davon abhängig, wie groß und wie gleichgerichtet eine *Sequenz* (zeitliche Abfolge) von Gradienten ist. Die Schrittweite ist dann am größten, wenn viele aufeinanderfolgende Gradienten exakt in dieselbe Richtung weisen. Wenn der Momentum-Algorithmus stets einen Gradienten \mathbf{g} beobachtet, beschleunigt er in Richtung $-\mathbf{g}$, bis eine Endgeschwindigkeit erreicht wird, in der jede Schrittweite diesen Wert beträgt:

$$\frac{\epsilon \|\mathbf{g}\|}{1 - \alpha}. \quad (8.17)$$

Es ist hilfreich, sich den Momentum-Hyperparameter in Begriffen von $\frac{1}{1 - \alpha}$ vorzustellen. Zum Beispiel entspricht $\alpha = 0,9$ einer Multiplikation der Höchstgeschwindigkeit mit 10 relativ zum Algorithmus des Gradientenabstiegsverfahrens.

In der Praxis sind häufige Werte für α 0,5, 0,9 und 0,99. Wie die Lernrate kann auch α im Laufe der Zeit angepasst werden. Normalerweise ist es zunächst klein und wird später erhöht. Das Anpassen von α im Laufe der Zeit ist weniger wichtig als die Abnahme von ϵ im Laufe der Zeit.

Wir können den Momentum-Algorithmus als Simulation eines Teilchens betrachten, das einer newtonischen Dynamik mit stetiger Zeit ausgesetzt ist. Die physikalische Analogie hilft dabei, ein Gespür dafür zu entwickeln, wie sich die Algorithmen für das Momentum und das Gradientenabstiegsverfahren verhalten.

Die Lage des Teilchens wird jederzeit durch $\theta(t)$ angegeben. Das Teilchen ist einer Nettokraft von $f(t)$ ausgesetzt. Diese Kraft beschleunigt das Teilchen:

$$f(t) = \frac{\partial^2}{\partial t^2} \theta(t). \quad (8.18)$$

Statt von einer Differenzialgleichung zweiter Ordnung der Position auszugehen, können wir die Variable $v(t)$ als Geschwindigkeit des Teilchens zum Zeitpunkt t einführen und die newtonsche Dynamik als Differenzialgleichung erster Ordnung neu schreiben:

$$v(t) = \frac{\partial}{\partial t} \theta(t), \quad (8.19)$$

$$f(t) = \frac{\partial}{\partial t} v(t). \quad (8.20)$$

Der Momentum-Algorithmus muss nun die Differenzialgleichungen durch numerische Simulation lösen. Eine einfache numerische Methode zum Lösen von Differenzialgleichungen ist die Euler-Methode, bei der ganz einfach die in der Gleichung definierte Dynamik in Form von kleinen endlichen Schritten in Richtung des einzelnen Gradienten simuliert wird.

Sie kennen nun die Grundform der Momentum-Updates, aber was genau sind die Kräfte? Eine Kraft ist proportional zum negativen Gradienten der Kostenfunktion: $-\nabla_{\theta} J(\theta)$. Diese Kraft stößt das Teilchen entlang der Oberfläche der Kostenfunktion nach unten. Der Algorithmus zum Gradientenabstiegsverfahren würde anhand jedes Gradienten nur einen Schritt machen, aber das newtonsche Szenario im Momentum-Algorithmus nutzt diese Kraft nun, um die Geschwindigkeit des Teilchens zu ändern. Sie können sich das Teilchen als Eishockeypuck vorstellen, der eine Eisfläche hinabrutscht. In steilen Bereichen der Eisfläche legt er an Tempo zu und rutscht weiter in diese Richtung, bis es wieder nach oben geht.

Es wird noch eine weitere Kraft benötigt. Wenn der Gradient der Kostenfunktion die einzige Kraft ist, kommt das Teilchen möglicherweise niemals zur Ruhe. Es wäre quasi ein Eishockeypuck, der auf der einen Seite des Tals hinabrutscht, auf der gegenüberliegenden Seite nach oben gleitet, dann die Richtung umkehrt usw. So gleitet der Puck bis in alle Ewigkeit hin und her – natürlich nur bei einer vollkommenen Eisfläche ohne jede Reibung. Wir bringen daher eine weitere Kraft ein, die proportional zu $-v(t)$ wirkt. Physikalisch betrachtet entspricht diese Kraft dem viskosen oder Strömungswiderstand, als würde das Teilchen sich durch eine viskose Flüssigkeit wie Sirup bewegen. Das führt dazu, dass es im Laufe der Zeit an Energie verliert und schließlich gegen ein lokales Minimum konvergiert.

Warum verwenden wir aber nun $-\mathbf{v}(t)$ und den Strömungswiderstand? Für $-\mathbf{v}(t)$ spricht zum Teil mathematische Bequemlichkeit – eine ganzzahlige Potenz der Geschwindigkeit stellt keine Herausforderung dar. In anderen physikalischen Systemen gibt es andere Arten von Widerstand für andere ganzzahlige Potenzen der Geschwindigkeit. So unterliegt ein Teilchen in der Luft einem Widerstand infolge turbulenter Strömung, dessen Kraft proportional zum Quadrat der Geschwindigkeit ist, während ein Teilchen am Boden der Grenzreibung unterliegt, deren Kraft eine konstante Größe ist. Wir können jede dieser Optionen verwerfen. Widerstand infolge turbulenter Strömung (proportional zum Quadrat der Geschwindigkeit) wird bei geringer Geschwindigkeit sehr schwach. Er ist nicht stark genug, um das Teilchen anzuhalten. Ein Teilchen mit einer von Null verschiedenen Ausgangsgeschwindigkeit, das nur diesem Widerstand ausgesetzt ist, bewegt sich auf ewig von seiner Ausgangsposition weg; der Abstand zum Startpunkt wächst wie $O(\log t)$. Wir müssen daher auf eine geringere Potenz der Geschwindigkeit verwenden. Bei einer Potenz von Null, was der Grenzreibung entspricht, ist die Kraft zu stark. Ist die Kraft infolge des Gradienten der Kostenfunktion klein, aber nicht Null, kann die konstante Kraft infolge der Reibung dazu führen, dass das Teilchen zur Ruhe kommt, bevor ein lokales Minimum erreicht ist. Der Strömungswiderstand umgeht diese beiden Probleme, denn er ist einerseits schwach genug, um dem Gradienten das Induzieren von Bewegung zu erlauben, bis ein Minimum erreicht ist, und andererseits stark genug, um eine Bewegung zu verhindern, wenn der Gradient dazu keinen Anlass bietet.

8.3.3 Nesterow-Momentum

Sutskever et al. (2013) haben eine Variante des Momentum-Algorithmus vorgestellt, die auf Nesterows beschleunigter Gradientenmethode (engl. *accelerated gradient method*) beruht (*Nesterov*, 1983, 2004). Die Update-Regeln ergeben sich hierbei aus

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^m L \left(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta} + \alpha \mathbf{v}), \mathbf{y}^{(i)} \right) \right], \quad (8.21)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}, \quad (8.22)$$

wobei die Parameter α und ϵ eine ähnliche Rolle wie im Standard-Momentum-Verfahren spielen. Der Unterschied zwischen dem Nesterow-Momentum und dem Standard-Momentum ist der Punkt, an dem der Gradient bestimmt wird. Beim Nesterow-Momentum erfolgt dies nach dem Anwenden der aktuellen Geschwindigkeit. Man könnte also sagen, dass das Nesterow-Momentum

versucht, einen *Korrekturfaktor* in das Standardverfahren für das Momentum einzubringen. Den vollständigen Nesterow-Momentum-Algorithmus finden Sie in Algorithmus 8.3.

Im Falle konvexer Batch-Gradienten verändert das Nesterow-Momentum die Konvergenzrate des Überschussfehlers von $O(1/k)$ (nach k Schritten) zu $O(1/k^2)$ (siehe *Nesterov (1983)*). Leider verbessert das Nesterow-Momentum die Konvergenzrate im Falle stochastischer Gradienten nicht.

Algorithmus 8.3 Stochastisches Gradientenabstiegsverfahren (SGD) mit Nesterow-Momentum

Require: Lernrate ϵ , Momentum-Parameter α

Require: Ausgangsparameter θ , Ausgangsgeschwindigkeit v

while Abbruchbedingung nicht erfüllt **do**

Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen aus der Trainingsdatenmenge $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ mit entsprechenden Labeln $\mathbf{y}^{(i)}$.

Anwenden eines Zwischen-Updates: $\tilde{\theta} \leftarrow \theta + \alpha v$.

Berechnen des Gradienten (im Zwischenpunkt):

$\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$.

Berechnen des Geschwindigkeit-Updates: $v \leftarrow \alpha v - \epsilon g$.

Aktualisiere: $\theta \leftarrow \theta + v$.

end while

8.4 Verfahren zur Parameterinitialisierung

Einige Optimierungsalgorithmen sind nicht von Natur aus iterativ und suchen einfach einen Lösungspunkt. Andere sind zwar von Natur aus iterativ, konvergieren aber bei Verwendung für die richtige Klasse von Optimierungsproblemen gegen akzeptable Lösungen, und zwar in einer akzeptablen Zeit und ungeachtet der Initialisierung. Auf Trainingsalgorithmen für das Deep Learning treffen diese vorteilhaften Aussagen leider nicht zu. Vielmehr sind sie normalerweise iterativ und erfordern somit einen Startpunkt für die Iterationen. Außerdem ist das Trainieren tiefer Modelle eine hinreichend komplexe Aufgabe, sodass die meisten Algorithmen von der Auswahl der Initialisierung stark beeinflusst werden. Der Startpunkt kann darüber entscheiden, ob der Algorithmus überhaupt konvergiert. Einige Startpunkte sind so instabil, dass der Algorithmus auf numerische Schwierigkeiten trifft und vollständig versagt. Wenn das Lernen konvergiert, beeinflusst der Startpunkt das Tempo der Konvergenz und ob ihr Ziel ein Punkt mit hohen

oder niedrigen Kosten ist. Punkte mit vergleichbaren Kosten können zusätzlich stark variierende Generalisierungsfehler aufweisen und auch die Generalisierung wird vom Standpunkt beeinflusst.

Moderne Initialisierungsverfahren sind einfach und heuristisch. Das Konstruieren verbesserter Initialisierungsverfahren ist eine schwierige Aufgabe, da wir nicht über ein fundiertes Verständnis der Optimierung in neuronalen Netzen verfügen. Die meisten Verfahren zielen darauf ab, einige gute Eigenschaften bei der Initialisierung des Netzes zu erreichen. Allerdings wissen wir nicht wirklich, welche dieser Eigenschaften unter welchen Umständen erhalten bleiben, nachdem das Lernen einmal begonnen hat. Eine weitere Schwierigkeit besteht darin, dass es einige Startpunkte gibt, die für die Optimierung von Vorteil sind, für die Generalisierung aber von Nachteil. Unser Verständnis des Einflusses des Startpunkts auf die Generalisierung ist besonders unausgereift; es bietet entsprechend geringe oder gar keine Hilfe bei der Wahl des Startpunkts.

Allerdings wissen wir nicht wirklich, welche dieser Eigenschaften unter welchen Umständen erhalten bleiben, nachdem das Lernen einmal begonnen hat. Eine weitere Schwierigkeit besteht darin, dass es einige Startpunkte gibt, die für die Optimierung von Vorteil sind, für die Generalisierung aber von Nachteil. Unser Verständnis des Einflusses des Startpunkts auf die Generalisierung ist besonders primitiv; es bietet entsprechend geringe oder gar keine Hilfe bei der Wahl des Startpunkts.

Möglicherweise die einzige Eigenschaft, die wir mit absoluter Sicherheit kennen, ist diese: Die Ausgangsparameter müssen die Symmetrie zwischen den unterschiedlichen Einheiten »brechen«. Falls zwei verdeckte Einheiten mit derselben Aktivierungsfunktion mit denselben Eingaben verbunden sind, dann müssen diese Einheiten unterschiedliche Ausgangsparameter aufweisen. Hätten sie dieselben Ausgangsparameter, würde ein deterministischer Lernalgorithmus im Rahmen einer deterministischen Kostenfunktion und eines deterministischen Modells diese beiden Einheiten ständig auf dieselbe Weise aktualisieren. Sogar bei Modellen oder Trainingsalgorithmen, die unterschiedliche Updates für unterschiedliche Einheiten mithilfe der Stochastizität berechnen können (z. B. beim Trainieren mit Dropout), ist es meist am besten, jede Einheit so zu initialisieren, dass eine von allen anderen Einheiten verschiedene Funktion berechnet wird. So kann vielleicht sichergestellt werden, dass keine Eingabemuster im Nullraum der Forward-Propagation und keine Gradientenmuster im Nullraum der Backpropagation verloren gehen. Das Ziel, von jeder Einheit eine andere Funktion berechnen zu lassen, fördert die Zufallsinitialisierung der Parameter. Wir könnten explizit nach einer großen Menge von Basisfunktionen suchen, die

alle voneinander verschieden sind, aber das führt häufig zu einem merklichen Berechnungsaufwand. Wenn wir zum Beispiel maximal so viele Ausgaben wie Eingaben haben, könnten wir die Gram-Schmidt-Orthogonalisierung für eine ursprüngliche Gewichtungsmatrix verwenden und so garantieren, dass jede Einheit eine ganz andere Funktion als alle anderen Einheiten berechnet. Die Zufallsinitialisierung aus einer Verteilung mit hoher Entropie über einen hochdimensionalen Raum erfordert weniger Rechenaufwand und weist Einheiten weniger wahrscheinlich die Berechnung derselben Funktion zu.

Normalerweise legen wir als Verzerrungen für jede Einheit heuristisch ausgewählte Konstanten fest und initialisieren lediglich die Gewichte zufällig. Zusätzliche Parameter – zum Beispiel solche zur Codierung der bedingten Varianz einer Vorhersage – werden meist ebenso wie die Verzerrungen auf heuristisch ausgewählte Konstanten festgelegt.

Wir initialisieren nahezu immer sämtliche Gewichte im Modell mit Werten, die zufällig aus einer Normal- oder Gleichverteilung gezogen wurden. Ob wir uns für die Normal- oder die Gleichverteilung entscheiden, scheint keine große Rolle zu spielen, war aber bisher auch nicht Gegenstand umfassender Untersuchungen. Der Maßstab der ursprünglichen Verteilung wirkt sich jedoch stark sowohl auf das Ergebnis des Optimierungsverfahrens als auch auf die Fähigkeit zur Generalisierung des Netzes aus.

Größere ursprüngliche Gewichte führen zu einer stärkeren Symmetriebrechung, wodurch redundante Einheiten besser vermieden werden. Sie helfen auch dabei, einen Signalverlust während der Forward-Propagation oder Backpropagation durch die lineare Komponente jeder Schicht zu vermeiden – größere Werte in der Matrix führen zu größeren Ausgaben in der Matrizenmultiplikation. Wenn die ursprünglichen Gewichte jedoch zu groß sind, kann dies zu einer Explosion der Werte während der Forward-Propagation oder Backpropagation führen. In RNNs können große Gewichte auch für **Chaos** sorgen, beispielsweise eine extreme Anfälligkeit gegenüber kleinen Störungen der Eingabe, sodass das Verhalten der deterministischen Forward-Propagation zufällig erscheint. Bis zu einem gewissen Maß lässt sich das Problem explodierender Gradienten durch Gradienten-Clipping umgehen (Schwellenwertbildung der Gradientenwerte vor einem Schritt im Gradientenabstiegsverfahren). Große Gewichte können ebenfalls extreme Werte verursachen, die zu einer Sättigung der Aktivierungsfunktion und somit dem kompletten Verlust des Gradienten durch gesättigte Einheiten führen. Diese im Wettstreit stehenden Faktoren bestimmen die ideale anfängliche Skalierung der Gewichte.

Eine Betrachtung aus Sicht von Regularisierung und Optimierung kann zu sehr unterschiedlichen Erkenntnissen über die »korrekte« Initialisierung eines Netzes führen. Für die Optimierung müssen die Gewichte groß genug sein, um die Informationen erfolgreich zu propagieren, aber einige Bedenken aufseiten der Regularisierung fordern eher kleinere Gewichte. Der Einsatz eines Optimierungsalgorithmus – beispielsweise des SGDs –, der kleine inkrementelle Änderungen an den Gewichten vornimmt und dazu neigt, in Bereichen zu stoppen, die näher an den Ausgangsparametern liegen (ob durch Hängenbleiben in einem Bereich mit niedrigem Gradienten oder durch Auslösen einer frühen Abbruchbedingung aufgrund von Überanpassung), drückt die Annahme aus, dass die endgültigen Parameter in der Nähe der Ausgangsparameter liegen sollten. Aus Abschnitt 7.8 wissen Sie, dass das Gradientenabstiegsverfahren mit frühem Abbruch für einige Modelle dem Weight Decay gleichwertig ist. Im allgemeinen Fall ist das Gradientenabstiegsverfahren mit frühem Abbruch dem Weight Decay nicht gleich, aber es bietet eine grobe Analogie bezüglich des Effekts der Initialisierung. Wir können uns die Initialisierung der Parameter θ bis θ_0 als ähnlich dem Aufzwingen einer normalverteilten A-priori-Wahrscheinlichkeit $p(\theta)$ mit dem Mittelwert θ_0 vorstellen. Aus dieser Sicht ist es durchaus sinnvoll, θ_0 in der Nähe von 0 auszuwählen. Diese Annahme besagt, dass es wahrscheinlicher ist, dass Einheiten nicht miteinander interagieren, als dass sie interagieren. Einheiten interagieren nur, wenn der Likelihood-Term der Zielfunktion eine starke Präferenz für die Interaktion ausdrückt. Andererseits gibt unsere Annahme sofern wir θ_0 auf große Werte initialisieren an, welche Einheiten miteinander wie interagieren sollten.

Für die Wahl der anfänglichen Skalierung der Gewichte stehen mehrere Heuristiken zur Verfügung. Eine Heuristik initialisiert die Gewichte einer vollständig verbundenen Schicht mit m Eingaben und n Ausgaben durch Ziehen von Stichproben aller Gewichte aus $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$; dagegen raten Glorot und Bengio (2010) zur **normalisierten Initialisierung**

$$W_{i,j} \sim U \left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}} \right). \quad (8.23)$$

Letztere stellt einen Kompromiss zwischen der Initialisierung aller Schichten auf dieselbe Aktivierungsvarianz und der Initialisierung aller Schichten auf dieselbe Gradientenvarianz dar. Die Formel wird anhand der Voraussetzung hergeleitet, dass das Netz nur aus einer Kette von Matrizenmultiplikationen besteht und keine Nichtlinearitäten aufweist. Echte neuronale Netze verstößen offensichtlich gegen diese Voraussetzung, aber viele für das li-

neare Modell entwickelte Verfahrenn kommen auch relativ gut mit den nichtlinearen Gegenstücken zurecht.

Saxe et al. (2013) empfehlen eine Initialisierung auf orthogonale Zufalls-matrizen mit sorgfältig ausgewähltem Skalierungs- oder **Verstärkungsfaktor** g , der die in jeder Schicht verwendete Nichtlinearität berücksichtigt. Sie leiten spezifische Werte für den Skalierungsfaktor für unterschiedliche Arten nichtlinearer Aktivierungsfunktionen her. Dieses Initialisierungsverfahren wird auch durch ein Modell eines tiefen Netzes als Sequenz von Matrixmultiplikationen ohne Nichtlinearitäten gefördert. Unter einem solchen Modell garantiert dieses Initialisierungsverfahren, dass die Gesamtzahl der Trainingsiterationen zum Erreichen der Konvergenz unabhängig von der Tiefe ist.

Erhöhen des Skalierungsfaktors g drückt das Netz in einen Bereich, in dem Aktivierungen bei der Forward-Propagation durch das Netz bei der Norm wachsen, Gradienten dagegen bei der Backpropagation. *Sussillo* (2014) hat gezeigt, dass das korrekte Festlegen des Verstärkungsfaktors ausreicht, um Netze bis zu einer Tiefe von 1 000 Schichten zu trainieren, ohne dass orthogonale Initialisierungen benötigt werden. Eine wesentliche Erkenntnis aus diesem Ansatz ist, dass Aktivierungen und Gradienten in Feedforward-Netzen bei jedem Schritt der Forward-Propagation oder Backpropagation zunehmen oder abnehmen können und so ein Random-Walk-Verhalten (Zufallsbewegung) an den Tag legen. Das liegt daran, dass Feedforward-Netze in jeder Schicht eine andere Gewichtungsmatrix nutzen. Wird der Random Walk auf den Erhalt der Normen abgestimmt, können Feedforward-Netze das Problem der verschwindenden und explodierenden Gradienten größtenteils umgehen, das entsteht, wenn dieselbe Gewichtungsmatrix für jeden Schritt verwendet wird (siehe Abschnitt 8.2.5).

Leider führen diese optimalen Kriterien für das ursprüngliche Gewicht häufig nicht zu einer optimalen Leistung. Das kann drei verschiedene Gründe haben: Erstens besteht die Möglichkeit, dass wir die falschen Kriterien nutzen – vielleicht gibt es damit keinen Vorteil für den Erhalt der Norm eines Signals im gesamten Netz. Zweitens gehen die bei der Initialisierung erzwungenen Eigenschaften möglicherweise nach Beginn des Lernprozesses verloren. Drittens könnte es sein, dass die Kriterien nicht nur die Geschwindigkeit der Optimierung erhöhen, sondern unbeabsichtigt auch den Generalisierungsfehler. In der Praxis müssen wir die Skalierung der Gewichte normalerweise als Hyperparameter behandeln, dessen optimaler Wert grob irgendwo in der Nähe der theoretischen Vorhersagen liegt, aber nicht exakt diesen entspricht.

Ein Nachteil beim Skalieren von Regeln zum Einstellen aller ursprünglichen Gewichtungen auf dieselbe Standardabweichung (z.B. $\frac{1}{\sqrt{m}}$) ist, dass jede einzelne Gewichtung bei großen Schichten extrem klein wird. Martens (2010) hat mit der **gehemmten Initialisierung** (engl. *sparse initialization*) ein alternatives Initialisierungsschema vorgestellt, in dem jede Einheit so initialisiert wird, dass sie exakt k von Null verschiedene Gewichtungen aufweist. Die Idee ist es, die Gesamthöhe der Eingabe der Einheit unabhängig von der Anzahl m der Eingaben zu halten, ohne dass die Größe der einzelnen Gewichtungselemente mit m abnimmt. Die gehemmte Initialisierung verhilft zum Initialisierungszeitpunkt zu mehr Vielfalt unter den Einheiten. Allerdings wird damit auch eine sehr starke Annahme bezüglich der Gewichtungen, für die sehr hohe normalverteilte Werte gewählt wurden, angelegt. Da das Gradientenabstiegsverfahren sehr lange benötigt, um »falsche« hohe Werte zu erreichen, kann dieses Initialisierungsschema Probleme für Einheiten verursachen, zum Beispiel für Maxout-Einheiten, die mehrere Filter aufweisen, die sorgfältig abgestimmt werden müssen.

Wenn die Berechnungsressourcen es erlauben, sollten Sie die anfängliche Skalierung der Gewichtungen für jede Schicht als Hyperparameter behandeln und die Skalierungen mithilfe eines Hyperparameter-Suchalgorithmus (siehe auch Abschnitt 11.4.2) wie der Zufallssuche (engl. *random search*) auswählen. Die Entscheidung für eine gehemmte Initialisierung kann ebenfalls mittels Hyperparameter erfolgen. Natürlich können Sie auch manuell die besten Anfangsgewichtungen suchen. Als Faustregel für diese Suche betrachten Sie den Wertebereich oder die Standardabweichung der Aktivierungen oder Gradienten für einen Mini-Batch mit Daten. Sind die Gewichtungen zu klein, nimmt der Wertebereich der Aktivierungen über den Mini-Batch mit fortschreitender Propagation der Aktivierungen durch das Netz ab. Durch wiederholtes Ermitteln der ersten Schicht mit unzumutbar kleinen Aktivierungen und Erhöhen der entsprechenden Gewichtungen erhält man schließlich ein Netz mit durchgängig hinreichenden Ausgangsaktivierungen. Wenn das Lernen an diesem Punkt noch immer zu langsam vor sich geht, ist der Wertebereich oder die Standardabweichung der Gradienten und der Aktivierungen einen Blick wert. Dieses Verfahren lässt sich prinzipiell automatisieren und ist meist weniger rechenaufwendig als die Hyperparameter-Optimierung auf Basis des Validierungsdatenfehlers, da es auf der Rückkoppelung des Verhaltens des Ursprungsmodells für einen einzelnen Datenbatch beruht, nicht auf der Rückkoppelung eines trainierten Modells der Validierungsdatenmenge. Nachdem der Prozess lange Zeit heuristisch eingesetzt wurde, gibt es seit Kurzem eine formalere Spezifikation und Untersuchung durch Mishkin und Matas (2015).

Bisher haben wir uns auf die Initialisierung der Gewichte konzentriert. Zum Glück ist die Initialisierung anderer Parameter gemeinhin einfacher.

Der Ansatz zum Einstellen der Verzerrungen muss mit dem Ansatz zum Einstellen der Gewichte abgestimmt werden. Eine Einstellung der Verzerrungen auf Null ist mit den meisten Initialisierungsverfahren für Gewichte kompatibel. Es gibt einige wenige Situationen, in denen wir einige Verzerrungen auf von Null verschiedene Werte einstellen:

- Ist eine Verzerrung für eine Ausgabeeinheit vorgesehen, ist es häufig von Vorteil, die Verzerrung so zu initialisieren, dass die passende Randstatistik der Ausgabe erzielt wird. Dazu nehmen wir an, dass die ursprünglichen Gewichte klein genug sind, um die Ausgabe der Einheit nur durch die Verzerrung zu bestimmen. Das begründet eine als Inverse der Aktivierungsfunktion für die Randstatistik der Ausgabe in der Trainingsdatenmenge eingestellte Verzerrung. Wenn die Ausgabe zum Beispiel eine Verteilung über Klassen ist und diese Verteilung eine besonders schiefe Verteilung mit einer Randwahrscheinlichkeit für die Klasse i aus c_i eines Vektors \mathbf{c} ist, können wir den Verzerrungsvektor \mathbf{b} durch Lösen der Gleichung $\text{softmax}(\mathbf{b}) = \mathbf{c}$ bestimmen. Das gilt nicht nur für die Klassifikatoren, sondern auch für die Modelle, die wir in Teil III behandeln, darunter Autoencoder und Boltzmann-Maschinen. Diese Modelle weisen Schichten auf, deren Ausgabe den Eingabedaten \mathbf{x} ähneln sollte; es kann sich als sehr hilfreich erweisen, die Verzerrungen solcher Schichten so zu initialisieren, dass sie zur Randverteilung über \mathbf{x} passen.
- Manchmal möchten wir die Verzerrung so wählen, dass es nicht zu einer zu starken Sättigung bei der Initialisierung kommt. Wir können beispielsweise die Verzerrung einer verdeckten ReLU (engl. *rectified linear unit*) auf 0,1 statt auf 0 setzen, um eine Sättigung der ReLU bei der Initialisierung zu verhindern. Dieser Ansatz ist allerdings nicht mit Initialisierungsverfahren für Gewichte kompatibel, die keine starken Eingaben aus den Verzerrungen erwarten. Sie sollten daher darauf verzichten, diesen Ansatz in Verbindung mit einer Random-Walk-Initialisierung zu verwenden (*Sussillo*, 2014).
- Es kommt vor, dass eine Einheit steuert, welche anderen Einheiten in eine Funktion einbezogen werden. In diesen Fällen gibt es eine Einheit mit der Ausgabe u und eine weitere Einheit $h \in [0, 1]$, die miteinander multipliziert werden, um eine Ausgabe uh zu erzeugen. Wir können h als ein Gate betrachten, das zwischen $uh \approx u$ oder $uh \approx 0$ entscheidet.

Dabei möchten wir die Verzerrung für h so wählen, dass bei der Initialisierung meist $h \approx 1$ gilt. Ansonsten hat u keine Möglichkeit zum Lernen. Ein Beispiel: Jozefowicz et al. (2015) sprechen sich stark dafür aus, die Verzerrung für das Forget-Gate eines LSTM-Modells (vgl. Abschnitt 10.10) auf 1 einzustellen.

Ein anderer häufiger Parametertyp ist der Varianz- oder Präzisionsparameter. Wir können zum Beispiel die lineare Regression mit einer Schätzung der bedingten Varianz anhand des Modells

$$p(y | \mathbf{x}) = \mathcal{N}(y | \mathbf{w}^T \mathbf{x} + b, 1/\beta) \quad (8.24)$$

durchführen, wobei β ein Präzisionsparameter ist. Varianz- oder Präzisionsparameter können normalerweise ungestraft auf 1 gesetzt werden. Ein anderer Ansatz geht davon aus, dass die ursprünglichen Gewichte nahe genug an Null liegen, dass die Auswirkungen der Gewichte beim Setzen der Verzerrungen vernachlässigt werden können; anschließend werden die Verzerrungen so gewählt, dass das korrekte Marginalmittel der Ausgabe entsteht, und die Varianzparameter auf die Randvarianz der Ausgabe in der Trainingsdatenmenge gesetzt.

Neben diesen einfachen konstanten oder Zufallsmethoden zur Initialisierung der Modellparameter ist auch eine Initialisierung mittels Machine Learning denkbar. Eine gängige Vorgehensweise, die in Teil III dieses Buchs noch behandelt werden wird, besteht in der Initialisierung eines überwachten Modells mit den Parametern, die durch ein unüberwachtes Modell erlernt wurden, das anhand derselben Eingaben trainiert wurde. Auch ein überwachtes Training für eine verwandte Aufgabe ist möglich. Sogar ein überwachtes Training einer nicht verwandten Aufgabe kann manchmal zu einer Initialisierung führen, die eine schnellere Konvergenz als eine Zufallsinitialisierung erlaubt. Einige dieser Initialisierungsverfahren führen zu einer schnelleren Konvergenz und besseren Fähigkeit zur Generalisierung, da sie Informationen über die Verteilung in den Ausgangsparametern des Modells codieren. Andere kommen offenkundig in erster Linie gut zurecht, weil die Parameter mit der richtigen Skalierung gewählt oder unterschiedliche Einheiten zur Berechnung unterschiedlicher Funktionen eingestellt wurden.

8.5 Algorithmen mit adaptiven Lernraten

Die Forschung im Bereich neuronaler Netze weiß schon lange, dass die Lernrate einer der am schwierigsten einzustellenden Hyperparameter ist, da sie

die Modellleistung stark beeinflusst. Wie in den Abschnitten 4.3 und 8.2 gezeigt, sind die Kosten häufig extrem empfindlich gegenüber Richtungen im Parameterraum und unempfindlich gegenüber anderen. Der Momentum-Algorithmus kann diese Probleme bis zu einem gewissen Grad abschwächen, allerdings zu den Kosten der Einführung eines weiteren Hyperparameters. Angesichts dessen kommt natürlich die Frage auf, ob es noch andere Möglichkeiten gibt. Wenn wir glauben, dass die Richtungen der Empfindlichkeit in etwa den Achsen entsprechen, ist die Nutzung einer gesonderten Lernrate für jeden Parameter sinnvoll; diese Lernraten können dann während des Lernprozesses automatisch angepasst werden.

Der **Delta-Bar-Delta**-Algorithmus (Jacobs, 1988) ist ein früher heuristischer Ansatz zur Anpassung individueller Lernraten für Modellparameter während des Trainings. Der Ansatz basiert auf einer einfachen Idee: Wenn die partielle Ableitung des Verlusts bezüglich eines bestimmten Modellparameters ihr Vorzeichen beibehält, muss die Lernrate zunehmen. Wechselt das Vorzeichen der partiellen Ableitung, dann muss die Lernrate abnehmen. Diese Art von Regel gilt natürlich nur für die vollständige Batch-Optimierung.

Vor Kurzem wurden eine Reihe inkrementeller Verfahren (oder Verfahren auf Basis von Mini-Batches) zur Anpassung der Lernrate der Modellparameter vorgestellt. In diesem Abschnitt geben wir einen kurzen Abriss einiger dieser Algorithmen.

8.5.1 AdaGrad

Der **AdaGrad**-Algorithmus (Algorithmus 8.4) passt die Lernraten aller Modellparameter individuell an, und zwar durch antiproportionale Skalierung zur Quadratwurzel der Summe aller historischen quadrierten Werte des Gradienten (Duchi *et al.*, 2011). Die Parameter mit der größten partiellen Ableitung des Verlusts weisen eine entsprechend rapide Abnahme ihrer Lernrate auf, wohingegen Parameter mit kleinen partiellen Ableitungen eine relativ geringe Abnahme der Lernrate aufweisen. Unter dem Strich ergibt sich ein größerer Fortschritt in den eher sanft geneigten Richtungen des Parameterraums.

Im Rahmen der konvexen Optimierung bietet der AdaGrad-Algorithmus einige wünschenswerte theoretische Eigenschaften. Empirisch – beim Trainieren tiefer neuronaler Netzmodelle – kann die Anhäufung der Quadrate der Gradienten *von Beginn des Trainings an* jedoch zu einer verfrühten und übermäßigen Abnahme der effektiven Lernrate führen. AdaGrad kommt gut mit einigen – aber nicht allen – Deep-Learning-Modellen zurecht.

Algorithmus 8.4 Der AdaGrad-Algorithmus

Require: Globale Lernrate ϵ

Require: Ausgangsparameter θ

Require: Kleine Konstante δ , vielleicht 10^{-7} , für numerische Stabilität

Initialisieren der Variable für die Gradientenakkumulation $r = \mathbf{0}$

while Abbruchbedingung nicht erfüllt **do**

Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen aus der Trainingsdatenmenge $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ mit entsprechenden Zielwerten $\mathbf{y}^{(i)}$.

Berechnen des Gradienten: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.

Anhäufen des Gradienten: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$.

Berechnen des Updates: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$. (Division und Quadratwurzel elementweise angewandt)

Aktualisiere: $\theta \leftarrow \theta + \Delta\theta$.

end while

8.5.2 RMSProp

Der **RMSProp**-Algorithmus (Root Mean Square Propagation Algorithm) (Hinton, 2012) modifiziert AdaGrad hinsichtlich einer besseren Leistung in nichtkonvexen Fällen, indem die Gradientenakkumulation einem exponentiell gewichteten gleitendem Mittelwert weicht. AdaGrad soll bei konvexen Funktionen schnell konvergieren. Bei nichtkonvexen Funktionen für das Training von neuronalen Netzen kann die Lerntrajektorie viele unterschiedliche Strukturen durchlaufen und irgendwann in einem Bereich landen, der eine lokal konvexe Mulde ist. AdaGrad verringert die Lernrate anhand der gesamten Historie des Quadrats des Gradienten, sodass sie möglicherweise vor Erreichen einer solchen konvexen Struktur zu klein wird. RMSProp verwendet ein exponentiell abnehmendes Mittel, um die Historie der fernen Vergangenheit zu verwerfen; dadurch ist eine schnelle Konvergenz nach dem Aufspüren einer konvexen Mulde möglich – als wäre der AdaGrad-Algorithmus in dieser Mulde initialisiert worden.

RMSProp wird in der Standardform in Algorithmus 8.5 wiedergegeben und in Kombination mit dem Nesterow-Momentum in Algorithmus 8.6. Anders als bei AdaGrad führt die Verwendung des gleitenden Mittelwerts zu einem neuen Hyperparameter ρ , der den Längenmaßstab des gleitenden Mittelwerts steuert.

Algorithmus 8.5 Der RMSProp-Algorithmus

Require: Globale Lernrate ϵ , Abnahmerate ρ
Require: Ausgangsparameter θ
Require: Kleine Konstante δ , meist 10^{-6} , zum Stabilisieren der Division durch kleine Zahlen
Initialisieren der Anhäufungsvariablen $r = 0$
while Abbruchbedingung nicht erfüllt **do**
 Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen aus der Trainingsdatenmenge $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ mit entsprechenden Zielwerten $\mathbf{y}^{(i)}$.
 Berechnen des Gradienten: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.
 Anhäufen des quadratischen Gradienten: $r \leftarrow \rho r + (1 - \rho) \mathbf{g} \odot \mathbf{g}$.
 Berechnen der Parameteranpassung: $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta+r}} \odot \mathbf{g}$.
 ($\frac{1}{\sqrt{\delta+r}}$ elementweise angewandt)
 Aktualisiere: $\theta \leftarrow \theta + \Delta\theta$.
end while

Algorithmus 8.6 RMSProp-Algorithmus mit Nesterow-Momentum

Require: Globale Lernrate ϵ , Abnahmerate ρ , Momentum-Koeffizient α
Require: Ausgangsparameter θ , Ausgangsgeschwindigkeit v
Initialisieren der Anhäufungsvariablen $r = 0$
while Abbruchbedingung nicht erfüllt **do**
 Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen aus der Trainingsdatenmenge $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ mit entsprechenden Zielwerten $\mathbf{y}^{(i)}$.
 Berechnen eines Zwischen-Updates: $\tilde{\theta} \leftarrow \theta + \alpha v$.
 Berechnen des Gradienten: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$.
 Anhäufen des Gradienten: $r \leftarrow \rho r + (1 - \rho) \mathbf{g} \odot \mathbf{g}$.
 Berechnen des Geschwindigkeit-Updates: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot \mathbf{g}$.
 ($\frac{1}{\sqrt{r}}$ elementweise angewandt)
 Aktualisiere: $\theta \leftarrow \theta + v$.
end while

Empirisch hat sich RMSProp als effektiver und praxistauglicher Optimierungsalgorithmus für tiefe neuronale Netze erwiesen. Er gehört aktuell zu den Standard-Optimierungsverfahren im Praxisalltag.

Algorithmus 8.7 Der Adam-Algorithmus

Require: Schrittweite ϵ (Empfohlene Vorgabe: 0,001)
Require: Exponentielle Abnahmeraten (engl. *decay*) für Momenten-Schätz-
werte, ρ_1 und ρ_2 in $[0, 1]$. (Empfohlene Vorgaben: 0,9 bzw. 0,999)
Require: Kleine Konstante δ zur numerischen Stabilisierung (Empfohlene
Vorgabe: 10^{-8})
Require: Ausgangsparameter θ
Initialisieren der Variablen für das statische und das Trägheitsmoment
 $s = \mathbf{0}, r = \mathbf{0}$
Initialisieren des Zeitschritts $t = 0$
while Abbruchbedingung nicht erfüllt **do**
 Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen aus der
 Trainingsdatenmenge $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ mit entsprechenden Zielwerten
 $\mathbf{y}^{(i)}$.
 Berechnen des Gradienten: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
 $t \leftarrow t + 1$
 Aktualisieren des verzerrten Schätzwerts für das statische Moment:
 $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$
 Aktualisieren des verzerrten Schätzwerts für das Trägheitsmoment:
 $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$
 Korrigieren der Verzerrung im statischen Moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$
 Korrigieren der Verzerrung im Trägheitsmoment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$
 Berechnen des Updates: $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ (elementweise angewandte
 Operationen)
 Aktualisiere: $\theta \leftarrow \theta + \Delta\theta$
end while

8.5.3 Adam

Adam (*Kingma und Ba, 2014*) ist ein weiterer Optimierungsalgorithmus mit adaptiver Lernrate (siehe Algorithmus 8.7). Die Bezeichnung »Adam« ist ein Kurzwort für »adaptive Momente«. Im Zusammenhang mit den bisherigen Algorithmen lässt er sich am besten als eine Variante der Kombination aus RMSProp und Momentum mit einigen wichtigen Unterscheidungen beschreiben. Zunächst einmal ist der Impuls (Momentum) in Adam direkt als

Schätzwert des Moments erster Ordnung (mit exponentiellem Gewicht) des Gradienten enthalten. Der direkte Weg zum Hinzufügen von Momentum zu RMSProp besteht darin, es auf die neu skalierten Gradienten anzuwenden. Hinter der Verwendung von Momentum in Kombination mit einer erneuten Skalierung steht keine theoretische Motivation. Zweitens enthält Adam Verzerrungskorrekturen der Schätzwerte für die Momente erster Ordnung (Momentum-Term) und die (nicht zentrierten) Momente zweiter Ordnung, um deren Initialisierung im Ursprung zu berücksichtigen (siehe Algorithmus 8.7). Auch RMSProp bezieht einen Schätzwert des (nicht zentrierten) Moments zweiter Ordnung ein; allerdings fehlt der Korrekturfaktor. Daher kann der Schätzwert für das Moment zweiter Ordnung in RMSprop – anders als in Adam – bereits frühzeitig im Training eine hohe Verzerrung aufweisen. Adam wird insgesamt als recht robust gegenüber der Auswahl der Hyperparameter betrachtet. Allerdings muss die Lernrate manchmal anders als vorgegeben gewählt werden.

8.5.4 Auswählen des passenden Optimierungsalgorithmus

Wir haben eine Reihe verwandter Algorithmen betrachtet, die sich mit der Optimierung von tiefen Modellen durch Anpassen der Lernraten für die einzelnen Modellparameter befassen. Sie fragen sich sicherlich, welchen Algorithmus Sie verwenden sollten.

Leider herrscht hierzu derzeit keine Einigkeit. *Schaul et al.* (2014) haben als wertvollen Beitrag eine Vielzahl von Optimierungsalgorithmen für ein breites Spektrum an Lernaufgaben miteinander verglichen. Die Ergebnisse deuten an, dass Algorithmen mit adaptiven Lernraten (wie RMSProp und AdaDelta) recht robust funktionieren, aber es gab keinen klaren Gewinner.

Derzeit gehören SGD, SGD mit Momentum, RMSProp, RMSProp mit Momentum, AdaDelta und Adam zu den beliebtesten Kandidaten. Die Entscheidung für einen Algorithmus scheint sich momentan in erster Linie danach zu richten, wie gut der Anwender den jeweiligen Algorithmus kennt, da Vertrautheit die Hyperparameter-Abstimmung erleichtert.

8.6 Approximative Verfahren zweiter Ordnung

In diesem Abschnitt behandeln wir die Verfahren zweiter Ordnung, die beim Trainieren tiefer Netze zum Einsatz kommen. Eine ältere Abhandlung hierzu

finden Sie in *LeCun et al.* (1998a). Der Einfachheit halber untersuchen wir von den Zielfunktionen lediglich das empirische Risiko:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}(\mathbf{x}, y)} [L(f(\mathbf{x}; \boldsymbol{\theta}), y)] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \quad (8.25)$$

Die hier vorgestellten Verfahren lassen sich problemlos auf allgemeinere Zielfunktionen erweitern, zum Beispiel jene, die Parameter-Regularisierungsterme einbeziehen (vgl. Kapitel 7).

8.6.1 Newton-Verfahren

In Abschnitt 4.3 haben wir Gradientenmethoden zweiter Ordnung vorgestellt. Anders als Verfahren erster Ordnung nutzen Verfahren zweiter Ordnung die zweiten Ableitungen für eine bessere Optimierung. Am häufigsten wird das Newton-Verfahren eingesetzt. Wir gehen nun näher auf das Newton-Verfahren ein, insbesondere für das Training neuronaler Netze.

Das Newton-Verfahren ist ein Optimierungsverfahren auf Basis einer Taylor-Entwicklung zweiter Ordnung zur Approximation von $J(\boldsymbol{\theta})$ an einen Punkt $\boldsymbol{\theta}_0$, wobei Ableitungen höherer Ordnung ignoriert werden:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0), \quad (8.26)$$

wobei \mathbf{H} die Hesse-Matrix von J bezüglich $\boldsymbol{\theta}$ berechnet für $\boldsymbol{\theta}_0$ ist. Wenn wir dann für den kritischen Punkt dieser Funktion lösen, ergibt sich die Update-Regel der Newton-Parameter:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0). \quad (8.27)$$

Für eine lokal quadratische Funktion (mit positiv definitem \mathbf{H}) springt das Newton-Verfahren durch erneutes Skalieren des Gradienten mit \mathbf{H}^{-1} direkt zum Minimum. Ist die Zielfunktion nicht quadratisch, sondern konvex (es gibt Terme höherer Ordnung), kann diese Aktualisierung iteriert werden, sodass der Trainingsalgorithmus zum Newton-Verfahren aus Algorithmus 8.8 entsteht.

Bei nicht quadratischen Oberflächen kann das Newton-Verfahren iterativ angewandt werden, sofern die Hesse-Matrix positiv definit bleibt. Dazu wird ein iteratives Verfahren mit zwei Schritten benötigt: Zunächst wird die inverse Hesse-Matrix aktualisiert bzw. berechnet (durch Aktualisieren der quadratischen Approximation). Anschließend werden die Parameter gemäß Gleichung 8.27 aktualisiert.

Algorithmus 8.8 Newton-Verfahren mit Ziel

$$J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$$

Require: Ausgangsparameter $\boldsymbol{\theta}_0$ **Require:** Trainingsdatenmenge mit m Beispielen**while** Abbruchbedingung nicht erfüllt **do** Berechnen des Gradienten: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$ Berechnen der Hesse-Matrix: $\mathbf{H} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}}^2 \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$ Berechnen der Inversen zur Hesse-Matrix: \mathbf{H}^{-1} Berechnen des Updates: $\Delta \boldsymbol{\theta} = -\mathbf{H}^{-1} \mathbf{g}$ Aktualisiere: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$ **end while**

In Abschnitt 8.2.3 haben wir gezeigt, dass sich das Newton-Verfahren nur bei positiv definiter Hesse-Matrix eignet. Im Deep Learning ist die Oberfläche der Zielfunktion normalerweise nicht konvex und enthält viele Merkmale – z. B. Sattelpunkte –, die ein Problem für das Newton-Verfahren darstellen.

Wenn nicht alle Eigenwerte der Hesse-Matrix positiv sind, beispielsweise in der Nähe eines Sattels, kann das Newton-Verfahren tatsächlich dazu führen, dass die Aktualisierungen in der falschen Richtung erfolgen. Das lässt sich durch Regularisieren der Hesse-Matrix vermeiden. Gängige Regularisierungsverfahren bestehen im Addieren einer Konstanten α auf der Diagonalen der Hesse-Matrix. Die regularisierte Aktualisierung sieht dann so aus:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [H(f(\boldsymbol{\theta}_0)) + \alpha \mathbf{I}]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0). \quad (8.28)$$

Dieses Regularisierungsverfahren wird in Approximationen an das Newton-Verfahren genutzt, z. B. im Levenberg-Marquardt-Algorithmus (*Levenberg, 1944; Marquardt, 1963*). Sie funktioniert recht gut, sofern die negativen Eigenwerte der Hesse-Matrix noch relativ nah an Null liegen. Bei extremeren Krümmungsrichtungen muss α ausreichend groß sein, um die negativen Eigenwerte zu kompensieren. Mit zunehmendem α wird die Hesse-Matrix jedoch zunehmend von der Diagonalen $\alpha \mathbf{I}$ dominiert und die vom Newton-Verfahren gewählte Richtung konvergiert gegen den durch α geteilten Standardgradienten. Bei stark negativer Krümmung muss α möglicherweise so groß sein, dass das Newton-Verfahren kleinere Schritte als das Gradientenabstiegsverfahren mit einer passend ausgewählten Lernrate macht.

Neben den Herausforderungen, die bestimmte Merkmale der Zielfunktion darstellen (z. B. Sattelpunkte), wird der Einsatz des Newton-Verfahrens beim Trainieren großer neuronaler Netze auch durch den erheblichen Rechenauf-

wand, den es darstellt, begrenzt. Die Anzahl der Elemente in der Hesse-Matrix ist in der Anzahl der Parameter quadriert, sodass für k Parameter (und schon in sehr kleinen neuronalen Netzen kann die Anzahl der Parameter k mehrere Millionen betragen) im Newton-Verfahren eine Inversion einer $k \times k$ -Matrix erforderlich wird – mit einer rechnerischen Komplexität von $O(k^3)$. Da sich die Parameter außerdem bei jeder Aktualisierung ändern, muss die inverse Hesse-Matrix *in jeder Trainingsiteration* berechnet werden. Folglich lassen sich in der Praxis nur Netze mit einer sehr geringen Parameteranzahl mit dem Newton-Verfahren trainieren. Der Rest dieses Abschnitts befasst sich mit Alternativen, die einige Vorteile des Newton-Verfahrens aufgreifen, ohne unter dem damit verbundenen Rechenaufwand zu leiden.

8.6.2 Konjugierte Gradienten

Für konjugierte Gradienten wird der Aufwand zur Berechnung der inversen Hesse-Matrix vermieden, denn der Abstieg erfolgt iterativ in **konjugierten Richtungen**. Der Ansatz ergibt sich aus der sorgfältigen Untersuchung von Schwächen des Verfahrens des steilsten Abstiegs (siehe Abschnitt 4.3), bei dem Liniensuchen iterativ in die dem Gradienten zugeordneten Richtungen erfolgen. Abbildung 8.6 zeigt, wie das Gradientenabstiegsverfahren bei Anwendung in einer quadratischen Mulde auf recht ineffiziente Weise hin und her springt. Das liegt daran, dass jede vom Gradienten vorgegebene Liniensuchrichtung stets orthogonal zur vorherigen Suchrichtung verläuft.

Sei die vorherige Suchrichtung \mathbf{d}_{t-1} . Am Minimum, in dem die Liniensuche endet, beträgt die Ableitung in Richtung \mathbf{d}_{t-1} : $\nabla_{\theta} J(\boldsymbol{\theta}) \cdot \mathbf{d}_{t-1} = 0$. Da der Gradient in diesem Punkt die aktuelle Suchrichtung bestimmt, trägt $\mathbf{d}_t = \nabla_{\theta} J(\boldsymbol{\theta})$ nicht zur Richtung \mathbf{d}_{t-1} bei. Somit ist \mathbf{d}_t orthogonal zu \mathbf{d}_{t-1} . Diese Beziehung zwischen \mathbf{d}_{t-1} und \mathbf{d}_t wird in Abbildung 8.6 für mehrere Iterationen des steilsten Abstiegs dargestellt. Wie die Abbildung zeigt, erhält die Auswahl der orthogonalen Abstiegsrichtungen den Mindestwert entlang der vorherigen Suchrichtungen nicht. Dadurch entsteht das Zickzackmuster – durch Abstieg Richtung Minimum in der aktuellen Gradientenrichtung müssen wir die Zielfunktion in der vorherigen Gradientenrichtung erneut minimieren. Indem wir dem Gradienten am Ende jeder Liniensuche folgen, heben wir den bereits in Richtung der vorherigen Liniensuche gemachten Fortschritt quasi auf. Das Verfahren der konjugierten Gradienten (kurz: CG-Verfahren, von engl. *conjugate gradients*) widmet sich diesem Problem.

Im CG-Verfahren suchen wir eine Suchrichtung, die **konjugiert** zur vorherigen Liniensuchrichtung ist, also den Fortschritt in dieser Richtung

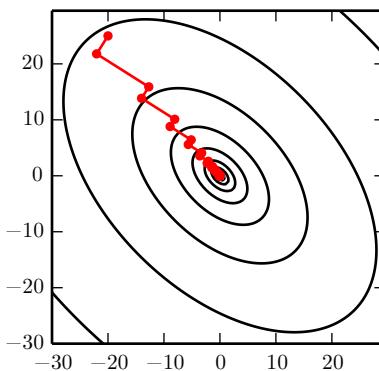


Abbildung 8.6: Das Gradientenabstiegsverfahren am Beispiel der Oberfläche einer quadratischen Kostenfunktion. Bei diesem Verfahren wird der Punkt des geringsten Aufwands entlang der durch den Gradienten im Startpunkt jedes Schritts vorgegebenen Linie angesprungen. Dies löst die Probleme, die in Verbindung mit einer unveränderlichen Lernrate in Abbildung 4.6 zu erkennen sind, doch selbst bei optimaler Schrittweite kommt es zu einem Hin und Her des Algorithmus in Richtung Optimum. Per definitionem ist der Gradient des Endpunkts im Minimum der Zielfunktion einer vorgegebenen Richtung orthogonal zu dieser Richtung.

nicht aufhebt. In der Trainingsiteration t nimmt die nächste Suchrichtung \mathbf{d}_t diese Form an:

$$\mathbf{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \mathbf{d}_{t-1}, \quad (8.29)$$

wobei β_t ein Koeffizient ist, dessen Größe bestimmt, welchen Anteil der Richtung \mathbf{d}_{t-1} wir zur momentanen Suchrichtung hinzufügen sollten.

Zwei Richtungen, \mathbf{d}_t und \mathbf{d}_{t-1} , sind konjugiert, wenn $\mathbf{d}_t^\top \mathbf{H} \mathbf{d}_{t-1} = 0$ ist, wobei \mathbf{H} die Hesse-Matrix ist.

Der direkte Weg zum Auferlegen des Konjugiums nutzt die Berechnung der Eigenvektoren von \mathbf{H} zur Wahl von β_t , verstößt aber gegen unseren Wunsch, ein Verfahren zu entwickeln, das für größere Probleme weniger rechenaufwendig als das Newton-Verfahren ist. Lassen sich konjugierte Richtungen ohne diese Berechnungen bestimmen? Zum Glück ist das möglich.

Zwei gängige Verfahren zum Berechnen von β_t sind diese:

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \quad (8.30)$$

2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \quad (8.31)$$

Für eine quadratische Oberfläche stellen die konjugierten Richtungen sicher, dass der Betrag des Gradienten entlang der bisherigen Richtung nicht zunimmt. Wir bleiben daher bei dem Minimum entlang der vorherigen Richtungen. Als Folge davon werden für das CG-Verfahren in einem k -dimensionalen Parameterraum höchstens k Liniensuchen benötigt, um das Minimum zu erreichen. Der CG-Algorithmus findet sich in Algorithmus 8.9.

Algorithmus 8.9 Das CG-Verfahren

Require: Ausgangsparameter $\boldsymbol{\theta}_0$

Require: Trainingsdatenmenge mit m Beispielen

Initialisieren von $\boldsymbol{\rho}_0 = \mathbf{0}$

Initialisieren von $g_0 = 0$

Initialisieren von $t = 1$

while Abbruchbedingung nicht erfüllt **do**

 Initialisieren des Gradienten $\mathbf{g}_t = \mathbf{0}$

 Berechnen des Gradienten: $\mathbf{g}_t \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$

 Berechnen von $\beta_t = \frac{(\mathbf{g}_t - \mathbf{g}_{t-1})^\top \mathbf{g}_t}{\mathbf{g}_{t-1}^\top \mathbf{g}_{t-1}}$ (Polak-Ribière)

 (Nichtlinearer konjugierter Gradient: Optionales Zurücksetzen von β_t auf Null, z. B. wenn t ein Vielfaches einer Konstanten k ist, beispielsweise $k = 5$)

 Berechnen der Suchrichtung: $\boldsymbol{\rho}_t = -\mathbf{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$

 Durchführen der Liniensuche zum Bestimmen von:

$\epsilon^* = \operatorname{argmin}_{\epsilon} \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}_t + \epsilon \boldsymbol{\rho}_t), \mathbf{y}^{(i)})$

 (Für echte quadratische Kostenfunktionen analytisches Lösen von ϵ^* anstelle einer expliziten Suche danach)

 Aktualisiere: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t$

$t \leftarrow t + 1$

end while

Nichtlineare konjugierte Gradienten Bisher haben wir das Verfahren der konjugierten Gradienten für quadratische Zielfunktionen betrachtet. Natürlich sind das Hauptthema des Kapitels vor allem Optimierungsverfahren für das Trainieren neuronaler Netze und anderer zugehöriger Deep-Learning-Modelle, in denen die entsprechende Zielfunktion alles andere als quadratisch ist. Es mag überraschen, dass das CG-Verfahren auch in diesem Fall noch gilt, wenn auch mit einigen Änderungen. Ohne die Gewissheit, dass die Zielfunktion quadratisch ist, kann nicht sichergestellt werden, dass die konjugierten Richtungen das Minimum der Zielfunktion bisheriger Richtungen beibehalten. Daher sind im **nichtlinearen CG-Algorithmus** gelegentliche

Resets vorgesehen, bei denen das CG-Verfahren mit der Liniensuche entlang des unveränderten Gradienten neu gestartet wird.

Aus der Praxis werden annehmbare Ergebnisse für Anwendungen des nichtlinearen CG-Algorithmus beim Training neuronaler Netze gemeldet, obschon es häufig von Vorteil ist, die Optimierung mit einigen wenigen Iterationen des stochastischen Gradientenabstiegsverfahrens zu initialisieren, bevor das nichtlineare CG-Verfahren übernimmt. Zwar wird der (nichtlineare) CG-Algorithmus traditionell als Batch-Verfahren betrachtet, allerdings wurden auch Mini-Batch-Varianten davon erfolgreich für das Trainieren neuronaler Netze genutzt (*Le et al.*, 2011). Auch wurden bereits früher für neuronale Netze maßgeschneiderte CG-Anpassungen vorgeschlagen, zum Beispiel der skalierter CG-Algorithmus (*Moller*, 1993).

8.6.3 BFGS

Der **Broyden-Fletcher-Goldfarb-Shanno-Algorithmus** (BFGS, engl. *Broyden-Fletcher-Goldfarb-Shanno algorithm*) versucht, einige der Vorteile des Newton-Verfahrens ohne dessen Rechenaufwand zu bieten. In dieser Hinsicht ähnelt BFGS dem CG-Verfahren. Allerdings folgt es einem direkteren Ansatz bei der Approximation des Newton-Updates. Das Newton-Update (ein aktualisierter Schritt im Newton-Verfahren) ergibt sich aus

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \quad (8.32)$$

wobei \mathbf{H} die Hesse-Matrix von J bezüglich $\boldsymbol{\theta}$ berechnet für $\boldsymbol{\theta}_0$ ist. Die wesentliche numerische Schwierigkeit beim Anwenden des Newton-Updates ist das Berechnen der Inversen zur Hesse-Matrix \mathbf{H}^{-1} . In Quasi-Newton-Verfahren (deren prominentester Vertreter der BFGS-Algorithmus ist) wird die Inverse anhand einer Matrix \mathbf{M}_t approximiert, die in niederrangigen Updates iterativ in Richtung einer besseren Approximation von \mathbf{H}^{-1} verfeinert wird.

Die BFGS-Approximation wird in vielen Lehrbüchern zur Optimierung näher beschrieben und hergeleitet, darunter *Luenberger* (1984).

Sobald die Approximation der inversen Hesse-Matrix \mathbf{M}_t aktualisiert wird, kann die Abstiegsrichtung $\boldsymbol{\rho}_t$ über $\boldsymbol{\rho}_t = \mathbf{M}_t \mathbf{g}_t$ bestimmt werden. Eine Liniensuche in dieser Richtung ermittelt die Schrittweite ϵ^* für diese Richtung. Die endgültige Parameteranpassung ergibt sich aus

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t. \quad (8.33)$$

Wie das Verfahren der konjugierten Gradienten iteriert der BFGS-Algorithmus eine Reihe von Liniensuchen für die Richtung der Informationen

zweiter Ordnung. Anders als das CG-Verfahren ist der Erfolg des Ansatzes jedoch nur in geringem Umfang davon abhängig, dass die Liniensuche einen Punkt auf der Linie in der Nähe des wahren Minimums findet. Somit bietet BFGS gegenüber CG den Vorteil, dass weniger Zeit für die Verfeinerung der einzelnen Liniensuchen benötigt wird. Andererseits muss der BFGS-Algorithmus die inverse Hesse-Matrix M speichern, was $O(n^2)$ an Speicherplatz kostet. Das führt dazu, dass BFGS für die meisten modernen Deep-Learning-Modelle mit Millionen von Parametern ungeeignet ist.

Limited-Memory-BFGS-Algorithmus (L-BFGS) Der Speicherbedarf des BFGS-Algorithmus kann deutlich verringert werden, wenn auf das Speichern der kompletten inversen Hesse-Approximation M verzichtet werden kann. Der L-BFGS-Algorithmus (BFGS-Algorithmus mit begrenztem Speicher) berechnet die Approximation M auf dieselbe Weise wie der BFGS-Algorithmus, allerdings unter der Annahme, dass $M^{(t-1)}$ die Einheitsmatrix ist. Somit wird auf das Speichern der Approximation zwischen den Schritten verzichtet. In Verbindung mit der exakten Liniensuche sind die mittels L-BFGS definierten Richtungen paarweise konjugiert. Allerdings verhält sich dieses Verfahren im Gegensatz zum CG-Verfahren auch dann noch gut, wenn das Minimum der Liniensuche nur näherungsweise erreicht wird. Der hier beschriebene L-BFGS-Algorithmus ohne Speichern lässt sich generalisieren, um mehr Informationen über die Hesse-Matrix einzubeziehen, indem einige der für die Aktualisierung von M verwendeteten Vektoren in jedem Zeitschritt gespeichert werden — das verursacht lediglich Kosten in Höhe von $O(n)$ pro Schritt.

8.7 Optimierungsverfahren und Meta-Algorithmen

Viele Optimierungsverfahren sind keine Algorithmen im eigentlichen Sinne, sondern vielmehr allgemeine Templates, die zum Erzeugen von Algorithmen angepasst werden können, oder Subroutinen, die sich in unterschiedlichen Algorithmen einsetzen lassen.

8.7.1 Batch-Normalisierung

Die Batch-Normalisierung (*Ioffe und Szegedy*, 2015) gehört zu den spannendsten Innovationen bei der Optimierung tiefer neuronaler Netze. Es handelt sich dabei nicht um einen Optimierungsalgorithmus, sondern um ein

Verfahren der adaptiven Reparametrisierung, das aus den Schwierigkeiten beim Trainieren sehr tiefer Modelle entstanden ist.

Bei sehr tiefen Modellen müssen mehrere Funktionen oder Schichten zusammengesetzt werden. Der Gradient gibt an, wie die einzelnen Parameter angepasst werden sollen, sofern sich die anderen Schichten nicht ändern. In der Praxis aktualisieren wir alle Schichten parallel. Wenn wir das Update vornehmen, kann es zu unerwarteten Ergebnissen kommen, da viele zusammengesetzte Funktionen zeitgleich geändert werden, und zwar mit Updates, die unter der Annahme berechnet wurden, dass die anderen Funktionen konstant bleiben. Ein einfaches Beispiel: Gegeben sei ein tiefes neuronales Netz, das nur eine Einheit je Schicht enthält und keine Aktivierungsfunktion für jede verdeckte Schicht einsetzt: $\hat{y} = xw_1w_2w_3 \dots w_l$. Hier stellt w_i das Gewicht für die Schicht i bereit. Die Ausgabe der Schicht i ist $h_i = h_{i-1}w_i$. Die Ausgabe \hat{y} ist eine lineare Funktion der Eingabe x , aber eine nichtlineare Funktion der Gewichte w_i . Angenommen, unsere Kostenfunktion belegt \hat{y} mit einem Gradienten von 1, wir möchten also \hat{y} geringfügig verringern. Der Backpropagation-Algorithmus kann nun einen Gradienten $\mathbf{g} = \nabla_{\mathbf{w}}\hat{y}$ berechnen. Beachten Sie, was geschieht, wenn wir ein Update $\mathbf{w} \leftarrow \mathbf{w} - \epsilon \mathbf{g}$ vornehmen. Die Approximation der Taylorreihe erster Ordnung von \hat{y} sagt vorher, dass der Wert von \hat{y} um $\epsilon \mathbf{g}^\top \mathbf{g}$ abnimmt. Wenn wir für \hat{y} eine Abnahme um 0,1 erzielen möchten, empfiehlt die Information erster Ordnung im Gradienten eine Lernrate ϵ von $\frac{0,1}{\mathbf{g}^\top \mathbf{g}}$. Allerdings enthält das Update Effekte zweiter und dritter Ordnung bis hin zu Effekten l -ter Ordnung. Der neue Wert für \hat{y} ergibt sich aus

$$x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l). \quad (8.34)$$

Ein Beispiel für einen Term zweiter Ordnung aus dieser Aktualisierung ist $\epsilon^2 g_1 g_2 \prod_{i=3}^l w_i$. Dieser Term mag vernachlässigbar sein, wenn $\prod_{i=3}^l w_i$ klein ist, oder exponentiell groß, wenn die Gewichte der Schichten 3 bis l größer als 1 sind. Die Wahl einer passenden Lernrate wird somit stark erschwert, denn die Effekte eines Updates auf die Parameter einer Schicht sind sehr stark von allen anderen Schichten abhängig. Optimierungsalgorithmen zweiter Ordnung berechnen daher ein Update, das diese Interaktionen zweiter Ordnung berücksichtigt. Doch wie wir sehen, können Interaktionen höherer Ordnung eine wichtige Rolle in sehr tiefen Netzen spielen. Selbst Optimierungsalgorithmen zweiter Ordnung sind aufwendig und erfordern meist eine Vielzahl von Approximationen, durch die verhindert wird, dass wirklich alle wesentlichen Interaktionen zweiter Ordnung berücksichtigt werden. Einen Optimierungsalgorithmus n -ter Ordnung für $n > 2$ zu konstruieren, erscheint hoffnungslos. Welche anderen Möglichkeiten gibt es?

Die Batch-Normalisierung bietet einen eleganten Weg zur Reparametrisierung nahezu jedes tiefen Netzes. Die Reparametrisierung reduziert das Problem der Koordinierung von Updates über viele Schichten hinweg deutlich. Die Batch-Normalisierung kann auf alle Eingabe- oder verdeckten Schichten in einem Netz angewandt werden. Sei \mathbf{H} ein Mini-Batch mit Aktivierungen der zu normalisierenden Schicht in Form einer Entwurfsmatrix, wobei die Aktivierungen für jedes Beispiel in einer Zeile der Matrix auftauchen. Für die Normalisierung von \mathbf{H} setzen wir stattdessen

$$\mathbf{H}' = \frac{\mathbf{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \quad (8.35)$$

ein, wobei $\boldsymbol{\mu}$ ein Vektor mit den Mittelwerten aller Einheiten ist und $\boldsymbol{\sigma}$ ein Vektor mit den Standardabweichungen aller Einheiten. Die Arithmetik hier beruht auf einem Broadcasting des Vektors $\boldsymbol{\mu}$ und des Vektors $\boldsymbol{\sigma}$ zur Anwendung in jeder Zeile der Matrix \mathbf{H} . In jeder Zeile erfolgt die Berechnung elementweise, sodass $H_{i,j}$ durch Subtrahieren von μ_j und Dividieren durch σ_j normalisiert wird. Der Rest des Netzes arbeitet dann mit \mathbf{H}' auf exakt dieselbe Weise wie das ursprüngliche Netz mit \mathbf{H} .

Zum Trainingszeitpunkt gilt

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \mathbf{H}_{i,:} \quad (8.36)$$

und

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\mathbf{H} - \boldsymbol{\mu})_i^2}, \quad (8.37)$$

wobei δ ein niedriger positiver Wert ist, beispielsweise 10^{-8} , der einen nicht definierten Gradienten von \sqrt{z} in $z = 0$ verhindern soll. Es ist entscheidend, dass wir zur Berechnung von Mittelwert und Standardabweichung sowie zur Anwendung dieser Werte für die Normalisierung von \mathbf{H} eine Backpropagation durch diese Operationen durchführen. Der Gradient schlägt somit niemals eine Operation vor, die lediglich dazu dient, die Standardabweichung oder den Mittelwert von h_i zu erhöhen; die Normalisierungsschritte entfernen den Effekt einer solchen Aktion und eliminieren die entsprechende Komponente aus dem Gradienten. Das war eine wichtige Innovation der Batch-Normalisierung. Bisherige Ansätze fügten Strafsterme zur Kostenfunktion hinzu, damit die Einheiten normalisierte Aktivierungsstatistiken aufwiesen, oder enthielten Eingriffe zur erneuten Normalisierung der Einheitenstatistiken nach jedem Schritt im Gradientenabstieg. Die erste Lösung führte meist zu einer unvollkommenen Normalisierung, die zweite zu unnützem Zeitaufwand,

da der Lernalgorithmus immer wieder Änderungen an Mittelwert und Varianz empfahl, woraufhin der Normalisierungsschritt diese Änderung immer wieder verwarf. Die Batch-Normalisierung führt eine Reparametrisierung des Modells durch, damit einige Einheiten nach Definition immer normalisiert sind. Dieser Kniff umgeht beide Probleme.

Zum Testzeitpunkt können μ und σ durch gleitende Mittelwerte ersetzt werden, die während der Trainingsdauer erfasst wurden. Dies gestattet es dem Modell, anhand eines einzelnen Beispiels evaluiert zu werden, ohne dass für μ und σ Definitionen eingeführt werden müssen, die von einem vollständigen Mini-Batch abhängig sind.

Kehren wir nochmals zum Beispiel $\hat{y} = xw_1w_2 \dots w_l$ zurück: Sie erkennen, dass die Schwierigkeiten beim Erlernen dieses Modells größtenteils durch Normalisieren von h_{l-1} überwunden werden können. Angenommen, x wird aus einer einheitlichen Normalverteilung gezogen. Dann entstammt auch h_{l-1} einer Normalverteilung, denn die Transformation von x in h_l ist linear. Allerdings ist der Mittelwert von h_{l-1} nicht länger gleich Null und verliert die Einheitsvarianz. Nach Anwenden der Batch-Normalisierung erhalten wir das normalisierte \hat{h}_{l-1} , das die Einheitsvarianz und einen Mittelwert gleich Null wiederherstellt. Bei nahezu allen Aktualisierungen der tieferen Schichten bleibt \hat{h}_{l-1} eine Standardnormalverteilung. Die Ausgabe \hat{y} kann dann als einfache lineare Funktion $\hat{y} = w_l \hat{h}_{l-1}$ erlernt werden. Das Lernen in diesem Modell wird dadurch sehr einfach, da die Parameter in den tieferen Schichten in den meisten Fällen keine Auswirkungen haben; ihre Ausgabe wird stets durch erneutes Normalisieren zu einer Standardnormalverteilung. In einigen Ausnahmefällen können die tieferen Schichten sich dennoch auswirken. Durch Ändern der Gewichte der tieferen Schichten auf 0 kann die Ausgabe degenerieren, durch Ändern des Vorzeichens eines der unteren Gewichte kann sich die Beziehung zwischen \hat{h}_{l-1} und y umkehren. Diese Fälle sind sehr selten. Ohne Normalisierung würde sich nahezu jedes Update stark auf die Statistik von h_{l-1} auswirken. Die Batch-Normalisierung erleichtert das Erlernen dieses Modells somit erheblich. Im Beispiel geht das leichtere Lernen zulasten der tieferen Schichten, die keinen Nutzen mehr bieten. In unserem linearen Beispiel zeigen die tieferen Schichten keine schädlichen Auswirkungen mehr, aber eben auch keine vorteilhaften Effekte. Das liegt daran, dass wir unsere Statistik erster und zweiter Ordnung normalisiert haben – und nur diese können von einem linearen Netz beeinflusst werden. In einem tiefen neuronalen Netz mit nichtlinearen Aktivierungsfunktionen können die tieferen Schichten auch nichtlineare Transformationen der Daten vornehmen, sodass sie weiterhin einen Nutzen bieten. Die Batch-Normalisierung normalisiert nur Mittelwert und Varianz der Einheiten, um

das Lernen zu stabilisieren. Aber sie ermöglicht auch eine Änderung der Beziehungen zwischen Einheiten und den nichtlinearen Statistiken einer einzelnen Einheit.

Da die letzte Schicht des Netzes eine lineare Transformation erlernen kann, möchten wir vielleicht alle linearen Beziehungen zwischen den Einheiten einer Schicht entfernen. Und genau diesen Ansatz verfolgen auch *Desjardins et al.* (2015), die zur Batch-Normalisierung angeregt haben. Leider ist das Eliminieren aller linearen Interaktionen sehr viel aufwendiger als das Normalisieren des Mittelwerts und der Standardabweichung jeder einzelnen Einheit, sodass die Batch-Normalisierung der bisher praktischste Ansatz bleibt.

Durch Normalisieren von Mittelwert und Standardabweichung einer Einheit kann die Aussagekraft des neuronalen Netzes, das die Einheit enthält, geschrämt werden. Um die Aussagekraft des Netzes beizubehalten, wird meist der Batch mit den Aktivierungen verdeckter Einheiten \mathbf{H} durch $\gamma\mathbf{H}' + \beta$ anstelle des normalisierten \mathbf{H}' ersetzt. Die Variablen γ und β sind erlernte Parameter, die der neuen Variablen erlauben, beliebige Mittelwerte und Standardabweichungen anzunehmen. Auf den ersten Blick erscheint das sinnlos – warum legen wir den Mittelwert auf $\mathbf{0}$ fest und führen dann einen Parameter ein, der das Setzen eines beliebigen Wertes β erlaubt? Die Antwort ist, dass die neue Parametrisierung dieselbe Funktionsfamilie für die Eingabe wie die alte Parametrisierung darstellen kann, aber eine andere Lerndynamik aufweist. In der alten Parametrisierung wurde der Mittelwert von \mathbf{H} über eine komplexe Interaktion zwischen den Parametern in den Schichten unter \mathbf{H} bestimmt. In der neuen Parametrisierung wird der Mittelwert von $\gamma\mathbf{H}' + \beta$ ausschließlich durch β bestimmt. Die neue Parametrisierung ist im Gradientenabstiegsverfahren deutlich einfacher zu erlernen.

Die meisten Schichten neuronaler Netze nehmen die Form $\phi(\mathbf{XW} + \mathbf{b})$ an, wobei ϕ eine bestimmte unveränderliche nichtlineare Aktivierungsfunktion ist, zum Beispiel die rektifizierte lineare Transformation. Stellt sich die Frage, ob wir die Batch-Normalisierung auf die Eingabe \mathbf{X} oder den transformierten Wert $\mathbf{XW} + \mathbf{b}$ anwenden sollten. *Ioffe und Szegedy* (2015) empfehlen Letzteres. Genauer formuliert, sollte $\mathbf{XW} + \mathbf{b}$ durch eine normalisierte Version von \mathbf{XW} ersetzt werden. Der Verzerrungsterm sollte weggelassen werden, da er redundant wird, wenn der β -Parameter auf die Reparametrisierung der Batch-Normalisierung angewandt wird. Die Eingabe einer Schicht ist normalerweise die Ausgabe einer nichtlinearen Aktivierungsfunktion, beispielsweise die rektifizierte lineare Funktion einer vorhergehenden Schicht. Die statistischen Größen der Eingabe sind daher

ehler nicht normalverteilt und weniger zugänglich für eine Standardisierung durch lineare Operationen.

In CNNs (siehe Kapitel 9) ist es wichtig, dieselben normalisierenden μ und σ für jede räumliche Position einer Merkmalskarte zu verwenden, damit die Statistiken der Merkmalskarte ungeachtet der räumlichen Position gleich bleiben.

8.7.2 Koordinatenabstieg

In einigen Fällen können wir ein Optimierungsproblem schnell lösen, indem wir es aufteilen. Wenn wir $f(\mathbf{x})$ bezüglich einer einzelnen Variable x_i minimieren und dann bezüglich einer weiteren Variable x_j usw. und dies dann mit allen Variablen wiederholt durchführen, erreichen wir in jedem Fall ein (lokales) Minimum. Dieses Verfahren wird als **Koordinatenabstieg** (engl. *coordinate descent*) bezeichnet, da wir jeweils nur eine Koordinate optimieren. Allgemein ausgedrückt bezeichnet **Block-Koordinatenabstieg** (engl. *block coordinate descent*) das gleichzeitige Minimieren bezüglich einer Teilmenge der Variablen. Der Begriff »Koordinatenabstieg« wird häufig sowohl für den Block-Koordinatenabstieg als auch den streng individuellen Koordinatenabstieg verwendet.

Der Koordinatenabstieg ist dann sinnvoll, wenn die unterschiedlichen Variablen im Optimierungsproblem trennscharf in Gruppen unterteilt werden können, die relativ isolierte Aufgaben haben, oder wenn die Optimierung bezüglich einer Variablengruppe deutlich effizienter als die Optimierung aller anderen Variablen ist. Betrachten Sie zum Beispiel diese Kostenfunktion:

$$J(\mathbf{H}, \mathbf{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (\mathbf{X} - \mathbf{W}^\top \mathbf{H})_{i,j}^2. \quad (8.38)$$

Sie beschreibt ein Lernproblem namens Sparse Coding, mit dem eine Gewichtungsmatrix \mathbf{W} bestimmt werden soll, die eine Matrix der Aktivierungswerte \mathbf{H} linear decodieren kann, um daraus die Trainingsdatenmenge \mathbf{X} zu rekonstruieren. Meist kommt bei der Anwendung des Sparse Codings auch Weight Decay oder eine Bedingung an die Norm der Spalten von \mathbf{W} zum Einsatz, um die pathologische Lösung mit extrem kleinem \mathbf{H} und großem \mathbf{W} zu verhindern.

Die Funktion J ist nicht konvex. Allerdings können wir die Eingaben für den Trainingsalgorithmus in zwei Mengen aufteilen: Dictionary-Parameter \mathbf{W} und Code-Darstellungen \mathbf{H} . Das Minimieren der Zielfunktion für eine dieser Variablenmengen ist ein konvexes Problem. Der Block-Koordinatenabstieg gibt uns ein Optimierungsverfahren an die Hand, mit der wir effiziente

konvexe Optimierungsalgorithmen einsetzen können, indem wir zunächst \mathbf{W} mit unveränderlichem \mathbf{H} optimieren und dann \mathbf{H} mit unveränderlichem \mathbf{W} .

Wenn der Wert einer Variable den optimalen Wert der anderen stark beeinflusst (z. B. in der Funktion $f(\mathbf{x}) = (x_1 - x_2)^2 + \alpha(x_1^2 + x_2^2)$ mit α als positiver Konstante), ist der Koordinatenabstieg kein geeignetes Verfahren. Dem ersten Term nach, sollten die beiden Variablen einen ähnlichen Wert annehmen, dem zweiten nach dagegen fast Null sein. Die Lösung besteht darin, beide auf Null zu setzen. Das Newton-Verfahren findet die Lösung in nur einem Schritt, da es sich um ein positiv definites quadratisches Problem handelt. Für kleine α macht der Koordinatenabstieg allerdings nur sehr langsam Fortschritte, da der erste Term nicht zulässt, dass eine einzelne Variable auf einen Wert geändert wird, der sich deutlich vom aktuellen Wert der anderen Variable unterscheidet.

8.7.3 Mittelwertbildung nach Polyak

Bei der Mittelwertbildung nach Polyak (*Polyak und Juditsky*, 1992) wird der Durchschnittswert mehrerer Punkte auf der Trajektorie eines Optimierungsalgorithmus durch den Parameterraum gebildet. Wenn t Iterationen des Gradientenabstiegsverfahrens die Punkte $\boldsymbol{\theta}^{(1)}, \dots, \boldsymbol{\theta}^{(t)}$ aufsuchen, ist die Ausgabe des Algorithmus für die Mittelwertbildung nach Polyak $\hat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t} \sum_i \boldsymbol{\theta}^{(i)}$. Bei einigen Problemklassen wie dem Gradientenabstiegsverfahren bei konvexen Problemen hat dieser Ansatz starke Konvergenzgarantien. Im Fall von neuronalen Netzen ist die Rechtfertigung eher heuristisch, aber die Ergebnisse sprechen für sich. Die Grundidee besagt, dass der Optimierungsalgorithmus mehrfach über ein Tal hin- und herspringen kann, ohne jemals einen Punkt in der Nähe der Talsohle aufzusuchen. Der Durchschnittswert aller Positionen auf beiden Seiten sollte jedoch in der Nähe der Talsohle liegen.

In nichtkonvexen Problemen kann der Pfad der Optimierungstrajektorie sehr komplex ausfallen und viele unterschiedliche Bereiche aufzusuchen. Das Einbeziehen von Punkten im Parameterraum aus der fernen Vergangenheit, die möglicherweise vom aktuellen Punkt durch große Schranken in der Kostenfunktion getrennt sind, scheint kein nützliches Verhalten zu sein. Wird die Mittelwertbildung nach Polyak auf nichtkonvexe Probleme angewendet, kommt im Ergebnis meist ein exponentiell abnehmender gleitender Mittelwert zum Tragen:

$$\hat{\boldsymbol{\theta}}^{(t)} = \alpha \hat{\boldsymbol{\theta}}^{(t-1)} + (1 - \alpha) \boldsymbol{\theta}^{(t)}. \quad (8.39)$$

Der gleitende Mittelwert wird in vielen Anwendungen genutzt; *Szegedy et al.* (2015) enthält ein jüngeres Beispiel.

8.7.4 Überwachtes Pretraining

Manchmal ist ein direktes Trainieren eines Modells zum Lösen einer bestimmten Aufgabe zu ambitioniert – insbesondere bei einem komplexen Modell, das schwierig zu optimieren ist, oder bei einer sehr schwierigen Aufgabe. Es kann dann effizienter sein, ein einfacheres Modell für die Lösung der Aufgabe zu trainieren, um das Modell im Anschluss komplexer zu machen. Oder das Modell wird zunächst für die Lösung einer einfacheren Aufgabe trainiert, bevor es an die eigentliche Aufgabe herangeführt wird. Diese Vorhensweise des Trainings einfacher Modelle für einfache Aufgaben vor dem Trainieren des eigentlichen Modells für die eigentliche Aufgabe werden in ihrer Gesamtheit als **Pretraining** (Vorabtraining) bezeichnet.

Greedy-Algorithmen unterteilen ein Problem in viele Komponenten und suchen dann für jede Komponente einzeln die optimale Lösung. Leider führt das Kombinieren der einzelnen optimalen Komponenten nicht unbedingt zu einer optimalen Gesamtlösung. Nichtsdestotrotz erweisen sich Greedy-Algorithmen manchmal als deutlich weniger aufwendig als Algorithmen, die nach der besten Komplettlösung suchen. Die Greedy-Lösung kann eine annehmbare, wenn auch nicht optimale, Qualität erreichen. An den Greedy-Algorithmus kann sich auch eine **Feinabstimmung** anschließen, in der ein übergreifender Optimierungsalgorithmus die optimale Lösung für das Gesamtproblem sucht. Die Initialisierung des übergreifenden Optimierungsalgorithmus durch eine Greedy-Lösung kann ersteren deutlich beschleunigen und die Qualität der gefundenen Lösung verbessern.

Algorithmen für das Pretraining und insbesondere für das Pretraining mit Greedy-Algorithmen sind im Deep Learning allgegenwärtig. In diesem Abschnitt beschäftigen wir uns vor allem mit solchen Pretraining-Algorithmen, die überwachte Lernprobleme in einfachere überwachte Lernprobleme aufteilen. Dieser Ansatz wird als **überwachtes Pretraining mit Greedy-Algorithmen** (engl. *greedy supervised pretraining*) bezeichnet.

In der ersten Version (*Bengio et al.*, 2007) des überwachten Pretrainings mit Greedy-Algorithmen bestand jede Phase aus einer Trainingsaufgabe für das überwachte Lernen, die nur eine Teilmenge der Schichten des endgültigen neuronalen Netzes enthielt. Ein Beispiel für überwachtes Pretraining mit Greedy-Algorithmen ist in Abbildung 8.7 dargestellt; hier wird jede neue verdeckte Schicht als Teil eines flachen überwachten mehrschichtigen Perzeptrons einem Pretraining unterzogen, wobei die Ausgabe der zuvor tra-

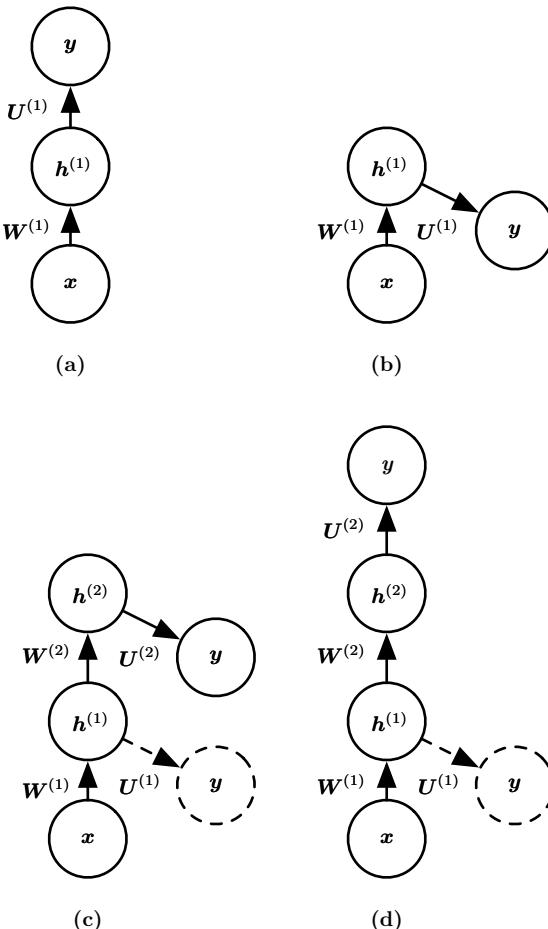


Abbildung 8.7: Darstellung einer Art des überwachten Pretrainings mit Greedy-Algorithmen (Bengio et al., 2007). (a) Zuerst wird eine hinreichend flache Architektur trainiert. (b) Eine weitere Zeichnung derselben Architektur. (c) Wir behalten nur die »Eingabe-zu-verdeckte-Schicht« des ursprünglichen Netzes und verwerfen die »Verdeckte-zu-Ausgabeschicht«. Wir nutzen die Ausgabe der ersten verdeckten Schicht als Eingabe für ein weiteres überwachtes MLP mit einer einzelnen verdeckten Schicht, das mit demselben Zielfunktion wie das erste Netz trainiert wird, um so eine zweite verdeckte Schicht hinzuzufügen. Dieser Schritt wird für beliebig viele Schichten wiederholt. (d) Eine weitere Zeichnung mit dem Ergebnis, in diesem Fall als Feedforward-Netz. Zur weiteren Verbesserung der Optimierung können wir alle Schichten einer übergreifenden Feinabstimmung unterziehen – entweder ganz am Ende oder in jeder Phase des Prozesses.

nierten verdeckten Schicht als Eingabe dient. Statt das Pretraining Schicht für Schicht durchzuführen, trainieren *Simonyan und Zisserman* (2015) ein tiefes CNN (elf Gewichtungsschichten) vorab und nutzen dann die ersten vier und die letzten drei Schichten dieses Netzes zur Initialisierung noch tieferer Netze (mit bis zu 19 Schichten mit Gewichten). Die mittleren Schichten des neuen sehr tiefen Netzes werden zufällig initialisiert. Anschließend wird das neue Netz als Ganzes trainiert. Eine weitere von *Yu et al.* (2010) untersuchte Möglichkeit besteht darin, die *Ausgaben* von zuvor trainierten mehrschichtigen Perzeptren sowie die unbearbeitete Eingabe (engl. *raw input*) als Input für jede neue Phase zu nutzen.

Inwiefern ist ein überwachtes Pretraining mit Greedy-Algorithmen hilfreich? Die zuerst von *Bengio et al.* (2007) behandelte Hypothese besagt, dass es die Richtung für die Zwischenstufen einer tiefen Hierarchie besser vorgibt. Allgemein kann Pretraining sowohl bei der Optimierung als auch der Generalisierung helfen.

Ein mit dem überwachten Pretraining verbundener Ansatz greift die Idee im Kontext des Transfer Learnings auf: *Yosinski et al.* (2014) führen ein Pretraining für ein tiefes CNN mit acht Gewichtungsschichten für eine Reihe von Aufgaben aus (eine Teilmenge der 1 000 ImageNet-Objektkategorien) und initialisieren anschließend ein Netz derselben Größe mit den ersten k Schichten des ersten Netzes. Alle Schichten des zweiten Netzes (dessen obere Schichten zufällig initialisiert werden) werden anschließend in ihrer Gesamtheit für eine andere Reihe von Aufgaben (eine weitere Teilmenge der 1 000 ImageNet-Objektkategorien) trainiert, allerdings mit weniger Trainingsbeispielen als für die erste Reihe von Aufgaben. Andere Ansätze beim Transfer Learning mit neuronalen Netzen werden in Abschnitt 15.2 behandelt.

Damit verwandt ist auch der **FitNets**-Ansatz (*Romero et al.*, 2015). Dabei wird zunächst ein Netz trainiert, dessen Tiefe gering genug und dessen Breite (Anzahl der Einheiten pro Schicht) groß genug ausfällt, um das Training leicht zu machen. Dieses Netz dient dann als **Lehrer** für ein zweites Netz, den **Schüler**. Das Schülernetz ist sehr viel tiefer und weniger breit (11 bis 19 Schichten) und wäre unter normalen Umständen mit dem stochastischen Gradientenabstiegsverfahren nur schwierig zu trainieren. Das Training des Schülernetzes wird einfacher, indem es lernt, nicht nur die Ausgabe der ursprünglichen Aufgabe vorherzusagen, sondern auch den Wert der mittleren Schicht des Lehrernetzes. Diese zusätzliche Aufgabe enthält eine Reihe von Hinweisen darüber, wie die verdeckten Schichten verwendet werden sollen und das Optimierungsproblem vereinfachen können.

Zusätzliche Parameter zur Regression der mittleren der fünf Schichten des Lehrernetzes anhand der mittleren Schicht des tieferen Schülernetzes werden eingeführt. Statt das endgültige Klassifizierungsziel vorherzusagen, soll jedoch die mittlere verdeckte Schicht des Lehrernetzes vorhergesagt werden. Die tieferen Schichten des Schülernetzes haben damit zwei Vorgaben: Sie müssen den Ausgaben des Schülernetzes dabei helfen, ihre Aufgaben zu erfüllen, und sie müssen die Zwischenschicht des Lehrernetzes vorhersagen. Zwar scheint ein dünnes und tiefes Netz schwieriger zu trainieren als ein breites und flaches Netz, aber das dünne und tiefe Netz kann besser generalisieren und bedeutet mit Sicherheit einen geringeren Berechnungsaufwand, wenn es dünn genug ist, um deutlich weniger Parameter aufzuweisen. Ohne die Hinweise in der verdeckten Schicht schneidet das Schülernetz in den Experimenten sehr schlecht ab – bei der Trainings- wie bei der Testdatenmenge. Hinweise in den mittleren Schichten sind somit ein Hilfsmittel zum Trainieren neuronaler Netze, die ansonsten nur schwer zu trainieren sind; doch auch andere Optimierungsverfahren oder Änderungen der Architektur können das Problem lösen.

8.7.5 Entwerfen von Modellen zur Unterstützung der Optimierung

Um die Optimierung zu verbessern, muss man nicht immer beim Optimierungsalgorithmus selbst ansetzen. Viele Verbesserungen bei der Optimierung tiefer Modelle sind durch die Verwendung von Modellen entstanden, die sich einfacher optimieren lassen.

Prinzipiell könnten wir Aktivierungsfunktionen nutzen, die ein zackenförmiges, nicht monotones Auf-und-ab-Muster zeigen. Das würde die Optimierung allerdings extrem schwierig machen. In der Praxis *ist es wichtiger, eine einfach zu optimierende Modellfamilie auszuwählen als einen leistungsstarken Optimierungsalgorithmus*. Die meisten Fortschritte im Lernen mit neuronalen Netzen der letzten 30 Jahre wurden mit Änderungen der Modellfamilie erzielt, nicht mit Änderungen des Optimierungsverfahrens. Das stochastische Gradientenabstiegsverfahren mit Momentum, das in den 1980ern zum Trainieren neuronaler Netze verwendet wurde, ist auch in modernen neuronalen Netzen an der Tagesordnung.

Vor allem spiegeln moderne neuronale Netze eine *Designentscheidung* für die Nutzung linearer Transformationen zwischen Schichten und Aktivierungsfunktionen wider, die fast überall differenzierbar sind und in großen Teilen ihrer Domäne eine markante Neigung aufweisen. Insbesondere Modell-Innovationen wie LSTM, ReLUs und Maxout-Einheiten haben sich stärker

in Richtung linearer Funktionen bewegt als frühere Modelle, beispielsweise tiefe Netze auf der Basis von sigmoiden Einheiten. Diese Modelle weisen gute Eigenschaften auf, die die Optimierung erleichtern. Der Gradient durchläuft viele Schichten, sofern die Jacobi-Matrix der linearen Transformation hinreichende Singulärwerte aufweist. Außerdem nehmen lineare Funktionen stetig in nur einer Richtung zu, sodass auch im Fall einer extrem stark abweichenden Ausgabe des Modells durch einfache Gradientenberechnung ermittelt werden kann, in welche Richtung die Ausgabe bewegen muss, um die Verlustfunktion zu reduzieren. Anders ausgedrückt: Moderne neuronale Netze sind so aufgebaut, dass ihre *lokale* Gradienteninformationen der Bewegung in Richtung einer entfernten Lösung recht gut entsprechen.

Andere Vorgehensweisen für den Modellentwurf können die Optimierung vereinfachen. Zum Beispiel reduzieren lineare Pfade oder Skip Connections zwischen Schichten die Länge des kürzesten Pfads zwischen den Parametern der tieferen Schicht und der Ausgabe, sodass das Problem der verschwindenden Gradienten abgeschwächt wird (*Srivastava et al.*, 2015). Das Hinzufügen zusätzlicher Kopien der Ausgabe, die an die verdeckten Zwischenschichten des Netzes angefügt werden, geht in die gleiche Richtung (z.B. GoogLeNet (*Szegedy et al.*, 2014a) und tief überwachte Netze (*Lee et al.*, 2014)). Diese »Behelfsköpfe« werden für dieselbe Aufgabe wie die primäre Ausgabe an der Spitze des Netzes trainiert, damit ein größerer Gradient in den tieferen Schichten vorliegt. Nach Abschluss des Trainings können die Behelfsköpfe verworfen werden. Es handelt sich um eine Alternative zu den Pretraining-Verfahren, die im vorigen Abschnitt vorgestellt wurden. Auf diese Weise können alle Schichten in einer Phase gemeinsam trainiert und dennoch die Architektur so geändert werden, dass die Zwischenschichten (insbesondere die tieferen) einige Hinweise über das von ihnen Erwartete auf kürzerem Wege erhalten. Diese Hinweise liefern ein Fehlersignal für die tieferen Schichten.

8.7.6 Fortsetzungsmethoden und Curriculum Learning

Wie in Abschnitt 8.2.7 gezeigt, entstehen viele der Herausforderungen bei der Optimierung aus der globalen Struktur der Kostenfunktion; diese können nicht einfach durch bessere Schätzwerte der lokalen Aktualisierungsrichtungen überwunden werden. Die vorherrschende Strategie für dieses Problem besteht darin, zu versuchen, die Parameter in einem Bereich zu initialisieren, der auf kurzem Wege im Parameterraum mit der Lösung verbunden ist und vom lokalen Abstieg gefunden werden kann.

Fortsetzungsmethoden (engl. *continuation methods*) sind eine Familie von Verfahren, die die Optimierung durch Wahl von Startpunkten vereinfachen kann, die dafür sorgen, dass die lokale Optimierung sich die meiste Zeit in angemessenen Bereichen des Raums aufhält. Mit Fortsetzungsmethoden soll eine Reihe von Zielfunktionen für dieselben Parameter konstruiert werden. Zum Minimieren einer Kostenfunktion $J(\boldsymbol{\theta})$ konstruieren wir neue Kostenfunktionen $\{J^{(0)}, \dots, J^{(n)}\}$. Diese Kostenfunktionen sind immer schwieriger aufgebaut: $J^{(0)}$ ist noch recht einfach zu minimieren, aber $J^{(n)}$ – die schwierigste – ist gleich der wahren Kostenfunktion $J(\boldsymbol{\theta})$ hinter dem gesamten Prozess. Wenn wir sagen, dass $J^{(i)}$ einfacher ist als $J^{(i+1)}$, bedeutet dies, dass sie sich in einem größeren Teil des $\boldsymbol{\theta}$ -Raums angemessen verhält. Eine Zufallsinitialisierung landet mit einer höheren Wahrscheinlichkeit in dem Bereich, in dem der lokale Abstieg die Kostenfunktion erfolgreich minimieren kann, da der Bereich größer ist. Die Reihen der Kostenfunktionen sind so konzipiert, dass die Lösung jeder Funktion ein guter Startpunkt für die nächste Funktion ist. Wir beginnen also mit einem einfachen Problem, verfeinern dessen Lösung dann, um inkrementell schwierigere Probleme zu lösen, bis wir die Lösung für die eigentliche Aufgabenstellung finden.

Klassische Fortsetzungsmethoden (bevor diese für das Trainieren neuronaler Netze eingesetzt wurden) beruhen meist auf dem Glätten der Zielfunktion. Wu (1997) enthält ein Beispiel für ein solches Verfahren und behandelt einige verwandte Verfahren. Fortsetzungsmethoden sind auch eng mit dem Simulated Annealing (simulierte Abkühlung) verwandt, bei dem den Parametern ein Rauschen hinzugefügt wird (Kirkpatrick et al., 1983). Diese Methoden haben sich in den letzten Jahren als extrem erfolgreich erwiesen. Mabohi und Fisher (2015) geben einen Überblick über neuere Literatur, insbesondere im KI-Kontext.

Fortsetzungsmethoden wurden in der Vergangenheit meist eingesetzt, um die Herausforderung lokaler Minima zu überwinden, insbesondere um ein globales Minimum zu erreichen, obwohl viele lokale Minima vorliegen. Dazu wurden einfachere Kostenfunktionen durch »Unschärfe« in der ursprünglichen Kostenfunktion kreiert. Diese Unschärfe-Operation lässt sich durch Approximieren von

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} J(\boldsymbol{\theta}') \quad (8.40)$$

mittels Stichprobenentnahme erzielen. Die Idee dahinter besagt, dass einige nichtkonvexe Funktionen näherungsweise konvex werden, wenn Unschärfe im Spiel ist. In vielen Fällen bleiben im Rahmen des Unschärfe-Verfahrens neue Informationen über die Lage eines globalen Minimums erhalten,

sodass es durch Lösen immer weniger unscharfer Versionen des Problems gefunden werden kann. Dieser Ansatz lässt sich auf drei Arten aufgliedern: Erstens könnte erfolgreich eine Reihe von Kostenfunktionen definiert werden, deren erste konvex ist und in denen das Optimum sich von einer Funktion zur nächsten erstreckt, bis das globale Minimum erreicht ist; allerdings sind eventuell so viele inkrementelle Kostenfunktionen erforderlich, dass der Gesamtaufwand für das Verfahren hoch bleibt. NP-schwere (engl. *NP-hard*) Optimierungsprobleme bleiben auch dann NP-schwer, wenn Fortsetzungsmethoden zum Einsatz kommen. Die anderen beiden Fortsetzungsmethoden versagen schon hinsichtlich der Anwendbarkeit. Erstens ist es möglich, dass die Funktion auch bei noch so viel Unschärfe nicht konvex wird. Betrachten Sie zum Beispiel die Funktion $J(\boldsymbol{\theta}) = -\boldsymbol{\theta}^\top \boldsymbol{\theta}$. Zweitens kann die Funktion durch die Unschärfe zwar konvex werden, aber das Minimum dieser unscharfen Funktion führt zu einem lokalen Minimum und nicht zu einem globalen Minimum der ursprünglichen Kostenfunktion.

Obwohl Fortsetzungsmethoden ursprünglich für das Problem lokaler Minima entwickelt wurden, herrscht mittlerweile die Überzeugung vor, dass lokale Minima nicht das Hauptproblem bei der Optimierung neuronaler Netze darstellen. Zum Glück sind Fortsetzungsmethoden weiterhin von Nutzen. Die einfacheren Zielfunktionen aus der Fortsetzungsmethode können flache Bereiche eliminieren, die Varianz in Gradientenschätzungen reduzieren, die Konditionierung der Hesse-Matrix verbessern und alles Mögliche tun, durch das sich lokale Updates leichter berechnen lassen oder durch das die Korrespondenz zwischen den Richtungen des lokalen Updates und dem Fortschritt in Richtung einer globalen Lösung erleichtert wird.

Bengio et al. (2009) haben festgestellt, dass ein als **Curriculum Learning** oder **Shaping** bezeichneter Ansatz als Fortsetzungsmethode betrachtet werden kann. Curriculum Learning (Lernen nach Lehrplan) beruht auf der Idee, einen Lernprozess so zu planen, dass zunächst einfache Konzepte erlernt werden, bevor der Vorgang mit komplexeren Konzepten fortgesetzt wird, die auf diesen einfachen Konzepten aufbauen. Dieses einfache Verfahren wurde zuvor bereits von Tiertrainern (*Skinner, 1958; Peterson, 2004; Krueger und Dayan, 2009*) und im Machine Learning (*Solomonoff, 1989; Elman, 1993; Sanger, 1994*) eingesetzt. *Bengio et al.* (2009) haben diesem Verfahren zu ihrer Berechtigung als Fortsetzungsmethode verholfen: Frühere $J^{(i)}$ werden durch Erhöhen des Einflusses simpler Beispiele vereinfacht (entweder durch Zuordnen höherer Koeffizienten zu ihren Beiträgen an der Kostenfunktion oder durch häufigeres Ziehen von Stichproben). Sie haben experimentell gezeigt, dass mithilfe eines Curriculums bessere Ergebnisse bei umfangreichen neuronalen Sprachmodellierungsaufgaben erzielt werden

konnten. Curriculum Learning wurde erfolgreich für viele Aufgaben im Bereich der natürlichen Sprache (*Spitkovsky et al.*, 2010; *Collobert et al.*, 2011a; *Mikolov et al.*, 2011b; *Tu und Honavar*, 2011) und der Computer Vision (*Kumar et al.*, 2010; *Lee und Grauman*, 2011; *Supancic und Ramanan*, 2013) eingesetzt. Außerdem hat sich gezeigt, dass das Curriculum Learning dem *Lehren* durch Menschen entspricht (*Khan et al.*, 2011): Lehrer beginnen mit einfachen und allgemeineren Beispielen und helfen den Schülern dann, ihren Horizont anhand weniger offensichtlicher Fälle zu erweitern. Curriculum-Verfahren sind beim Unterrichten von Menschen *effektiver* als Verfahren, die auf gleichförmigen Stichproben von Beispielen basieren. Außerdem können sie die Wirksamkeit anderer Unterrichtsverfahren stärken (*Basu und Christensen*, 2013).

Ein weiterer wichtiger Beitrag zur Forschung über das Curriculum Learning ergab sich im Zusammenhang mit dem Training von RNNs für die Erfassung langfristiger Abhängigkeiten: *Zaremba und Sutskever* (2014) fanden heraus, dass ein *stochastisches Curriculum* zu sehr viel besseren Ergebnissen führt, wenn dem Schüler eine zufällige Auswahl von einfachen und schwierigen Beispielen präsentiert wird und der durchschnittliche Anteil der schwierigeren Beispiele (in diesem Fall also jene mit langfristigen Abhängigkeiten) nach und nach erhöht wird. Bei einem deterministischen Curriculum konnte keine Verbesserung gegenüber der Baseline (normales Training anhand der vollständigen Trainingsdatenmenge) festgestellt werden.

Sie haben nun die grundlegende Familie von Modellen neuronaler Netze kennengelernt und etwas über deren Regularisierung und Optimierung erfahren. In den weiteren Kapiteln wenden wir uns Spezialisierungen unter den neuronalen Netzen zu, mit denen sehr große Netze konstruiert und Eingangsdaten mit spezieller Struktur verarbeitet werden können. Die in diesem Kapitel behandelten Optimierungsverfahren lassen sich in vielen Fällen ganz ohne oder mit geringen Änderungen für die spezialisierten Architekturen übernehmen.

9

CNNs

Convolutional Networks (*LeCun, 1989*), auch **Convolutional Neural Networks** oder kurz CNNs genannt, sind eine spezielle Form neuronaler Netze zur Verarbeitung von Daten mit einer bekannten rasterähnlichen Topologie. Beispiele dafür sind Zeitreihendaten, die ein 1-D-Raster mit Stichproben in regelmäßigen Zeitabständen darstellen, sowie Bilddaten, die ein 2-D-Raster aus Pixeln darstellen. CNNs haben gewaltige Erfolge in der Praxis gebracht. Die Bezeichnung »Convolutional Neural Network« deutet an, dass im Netz eine mathematische Operation namens **Faltung** (engl. *convolution*) genutzt wird. Die Faltung ist eine spezielle Form der linearen Operation. *CNNs sind nichts anderes als neuronale Netze, die in mindestens einer Schicht Faltung anstelle der allgemeinen Matrizenmultiplikation einsetzen.*

In diesem Kapitel erklären wir zunächst, was Faltung ist. Anschließend befassen wir uns mit der Motivation hinter dem Einsatz der Faltung in einem neuronalen Netz. Dann beschreiben wir eine Operation namens **Pooling**, die in nahezu allen CNNs genutzt wird. Für gewöhnlich entspricht die Operation in einem CNN nicht exakt der Definition einer Faltung aus anderen Fachbereichen, zum Beispiel dem Ingenieurwesen oder der reinen Mathematik. Wir beschreiben verschiedene Varianten der Faltungsfunktion, die in der Praxis neuronaler Netze weit verbreitet sind. Außerdem zeigen wir, wie die Faltung sich auf verschiedene Arten von Daten mit unterschiedlicher Dimensionalität anwenden lässt. Anschließend geht es um Wege für einen wirkungsvolleren Einsatz der Faltung. CNNs sind ein herausragendes Beispiel dafür, wie neurowissenschaftliche Grundlagen das Deep Learning beeinflussen. Wir beschreiben diese neurowissenschaftlichen Grundlagen, bevor wir gegen Ende noch die Rolle von CNNs in der Geschichte des Deep

Learnings aufzeigen. Ein Thema, das in diesem Kapitel fehlt, ist die Wahl der passenden Architektur für ein CNN. Wir möchten in diesem Kapitel zeigen, was mit CNNs möglich ist, welche Hilfsmittel sie uns also an die Hand geben. Allgemeine Anleitungen zur Wahl der richtigen Hilfsmittel für die jeweiligen Umstände finden Sie in Kapitel 11. Die Forschung im Bereich der Architekturen für CNNs schreitet so rasant voran, dass alle paar Wochen oder Monate eine neue »beste« Architektur für einen bestimmten Benchmark gekrönt wird. Damit ist jede schriftliche Abhandlung der besten Architektur bereits bei Erscheinen des Druckwerks veraltet. Nichtsdestotrotz sind bisher alle dieser besten Architekturen aus den hier beschriebenen Bausteinen zusammengesetzt.

9.1 Die Faltungsoperation

In ihrer allgemeinen Form ist die Faltung eine Operation für zwei Funktionen eines reellwertigen Arguments. Um die Grundidee hinter der Faltung zu verdeutlichen, sehen wir uns zunächst an, auf welche Funktionen sie angewandt werden kann.

Angenommen, wir verfolgen die Position eines Raumschiffs mit einem Lasersensor. Unser Lasersensor stellt eine einzelne Ausgabe $x(t)$ zur Verfügung, nämlich die Position des Raumschiffs zum Zeitpunkt t . Sowohl x als auch t sind reellwertig; wir können also jederzeit einen anderen Wert vom Lasersensor erhalten.

Was geschieht, wenn der Lasersensor rauschbehaftet ist? Um einen weniger rauschbehafteten Schätzwert der Position des Raumschiffs zu erhalten, möchten wir mehrere Messwerte mitteln. Natürlich sind neuere Messwerte von höherer Relevanz – wir benötigen also ein gewichtetes Mittel, bei dem neuere Messungen ein höheres Gewicht erhalten. Dazu benutzen wir eine Funktion $w(a)$, wobei a das Alter einer Messung ist. Wenn wir diese Operation des gewichteten Mittels auf jeden einzelnen Zeitpunkt anwenden, ergibt sich eine neue Funktion s , die einen geglätteten Schätzwert der Position des Raumschiffs angibt:

$$s(t) = \int x(a)w(t-a)da. \quad (9.1)$$

Diese Operation wird **Faltung** genannt. Die Faltungsoperation wird üblicherweise mit einem Sternchen dargestellt:

$$s(t) = (x * w)(t). \quad (9.2)$$

In unserem Beispiel muss w eine gültige Wahrscheinlichkeitsdichtefunktion sein. Ansonsten handelt es sich bei der Ausgabe nicht um ein gewichtetes Mittel. Außerdem muss w für alle negativen Argumente 0 sein, denn sonst würde es in die Zukunft blicken – und das übersteigt vermutlich unsere Fähigkeiten. Diese Einschränkungen gelten allerdings nur für unser Beispiel. Grundsätzlich ist die Faltung für alle Funktionen definiert, für die das obige Integral definiert ist, und kann auch für andere Zwecke als die Bildung eines gewichteten Mittels genutzt werden.

Im Rahmen von CNNs wird das erste Argument (hier die Funktion x) der Faltung häufig als **Eingabe** (engl. *input*) bezeichnet, das zweite Argument (hier die Funktion w) als **Kernel**. Die Ausgabe wird manchmal als **Merkmalskarte** (engl. *feature map*) bezeichnet.

Der in unserem Beispiel genannte Lasersensor, der zu jedem Zeitpunkt Messwerte bereitstellen kann, ist nicht realistisch. Normalerweise wird die Zeit bei der computergestützten Arbeit mit Daten diskretisiert; der Sensor gibt die Daten also in regelmäßigen Intervallen aus. Es wäre also realistischer, davon auszugehen, dass der Laser jede Sekunde einen Messwert ausgibt. Der Zeitindex t kann damit nur ganzzahlige Werte annehmen. Wenn wir nun annehmen, dass x und w nur für ganzzahlige t definiert sind, ergibt sich die diskrete Faltung:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \quad (9.3)$$

In Machine-Learning-Anwendungen ist die Eingabe normalerweise ein mehrdimensionales Array aus Daten und der Kernel ein mehrdimensionales Array aus Parametern, die durch den Lernalgorithmus angepasst werden. Wir bezeichnen diese mehrdimensionalen Arrays als Tensoren. Da jedes Element von Eingabe und Kernel ausdrücklich separat abgelegt werden muss, gehen wir für gewöhnlich davon aus, dass diese Funktionen überall Null sind – abgesehen von der endlichen Menge der Punkte, für die wir die Werte speichern. In der Praxis können wir somit die unendliche Aufsummierung als Aufsummierung über eine endliche Anzahl von Array-Elementen implementieren.

Schließlich nutzen wir Faltungen häufig über mehrere Achsen gleichzeitig. Ein Beispiel: Wenn wir ein zweidimensionales Bild I als Eingabe verwenden, soll vermutlich auch ein zweidimensionaler Kernel K zum Einsatz kommen:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n). \quad (9.4)$$

Die Faltung ist kommutativ, sodass die folgende Notation gleichwertig ist:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n). \quad (9.5)$$

Meist ist die zuletzt genannte Formel einfacher in einer Machine-Learning-Bibliothek zu implementieren, da es weniger Variationen im Bereich der gültigen Werte für m und n gibt.

Die kommutative Eigenschaft der Faltung entsteht, da wir den Kernel relativ zur Eingabe in dem Sinne **gedreht** haben, dass mit zunehmendem m der Index der Eingabe zunimmt, während der Index des Kernels abnimmt. Das Erzielen der kommutativen Eigenschaft ist der einzige Grund für das Drehen des Kernels (engl. *kernel flipping*). Obwohl die kommutative Eigenschaft beim Schreiben von Beweisen nützlich ist, handelt es sich allgemein nicht um eine wichtige Eigenschaft in der Implementierung neuronaler Netze. Stattdessen setzen viele Bibliotheken für neuronale Netze eine verwandte Funktion namens **Kreuzkorrelation** ein, die mit Ausnahme der Kerneldrehung der Faltung entspricht:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n). \quad (9.6)$$

Viele Machine-Learning-Bibliotheken implementieren Kreuzkorrelation, nennen die Operation aber Faltung. In diesem Text halten wir uns an dieses Übereinkommen und nennen beide Operationen Faltung. Daher geben wir explizit an, ob der Kernel gedreht werden soll oder nicht, wenn dies im jeweiligen Zusammenhang relevant ist. Im Machine-Learning-Kontext erlernt der Lernalgorithmus die geeigneten Werte des Kernels an den passenden Orten. Ein Algorithmus auf Basis der Faltung mit Kerneldrehung erlernt einen Kernel, der relativ zu dem Kernel, der von einem Algorithmus ohne Drehung erlernt wird, gedreht ist. Nur selten wird die Faltung als einziger Mechanismus im Machine Learning eingesetzt. Vielmehr kommen andere Funktionen parallel zum Einsatz, die in ihrer Gesamtheit nicht kommutativ sind, und zwar ungeachtet dessen, ob die Faltungsoperation den Kernel dreht oder nicht.

Abbildung 9.1 zeigt ein Beispiel der Faltung (ohne Kerneldrehung) für einen 2-D-Tensor.

Die diskrete Faltung lässt sich als Multiplikation mit einer Matrix betrachten, wobei die Bedingung an mehrere Matrizeneinträge ist, dass sie anderen Einträgen gleich sind. Beispielsweise ist bei eindimensionaler diskreter Faltung die Bedingung an jede Zeile der Matrix, dass sie gleich der um ein Element verschobenen Vorgängerzeile ist. Man spricht hier von

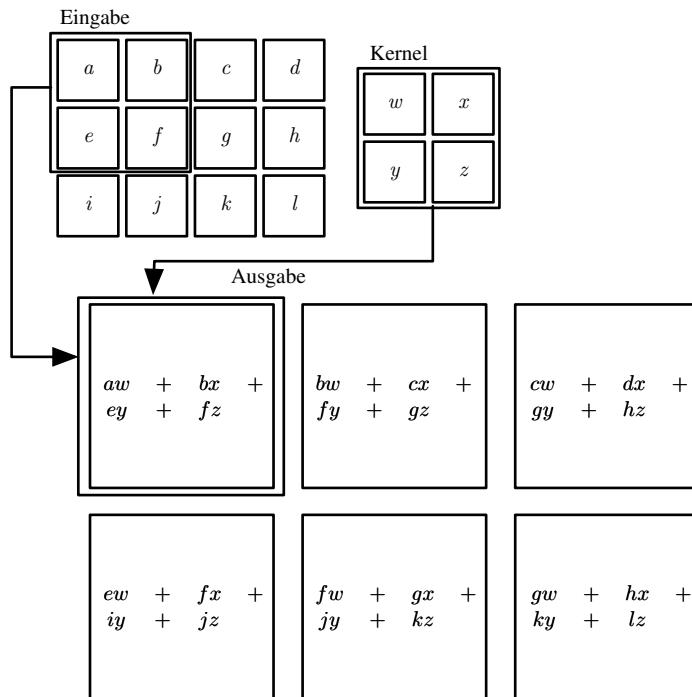


Abbildung 9.1: Beispiel einer 2-D-Faltung ohne Kerneldrehung. Wir beschränken die Ausgabe auf Positionen, für die der Kernel vollständig im Bild liegt. Dies wird in einigen Fällen als »gültige« (engl. *valid*) Faltung bezeichnet. Die Rahmen und Pfeile zeigen an, wie das obere linke Elemente des Ausgabetensors durch Anwenden des Kernels auf den entsprechenden oberen linken Bereich des Eingabetensors gebildet wird.

einer **Toeplitz-Matrix**. In zwei Dimensionen entspricht eine **doppelt zyklische Blockmatrix** (engl. *doubly block circulant matrix*) der Faltung. Neben diesen Bedingungen, dass mehrere Elemente einander gleich sein müssen, entspricht die Faltung normalerweise einer sehr dünnbesetzten Matrix (einer Matrix, deren Einträge größtenteils Null sind). Das liegt daran, dass der Kernel meist sehr viel kleiner als das Eingabebild ist. Jeder Algorithmus für neuronale Netze, der mit Matrizenmultiplikationen funktioniert und nicht von bestimmten Eigenschaften der Matrizenstruktur abhängig ist, sollte ohne Änderungen am neuronalen Netz auch mit Faltung funktionieren. Typische CNNs nutzen weitere Spezialisierungen, um große Eingabemengen effizient zu verarbeiten. Allerdings sind diese für die theoretische Betrachtung nicht unbedingt erforderlich.

9.2 Motivation

Die Faltung macht sich drei wichtige Konzepte zunutze, die beim Optimieren eines Machine-Learning-Systems helfen können: **spärliche Interaktionen** (engl. *sparse interaction*), **Parameter Sharing** und **äquivariante Repräsentationen** (engl. *equivariant representations*). Darüber hinaus bietet die Faltung eine Möglichkeit zur Arbeit mit Eingaben unterschiedlicher Größe. Im Folgenden wenden wir uns den einzelnen Konzepten zu.

Klassische Schichten in neuronalen Netzen verwenden die Matrizenmultiplikation mit einer Parameter-Matrix, die einen separaten Parameter zum Beschreiben der Interaktion zwischen den einzelnen Eingabe- und Ausgabeeinheiten aufweist. Jede Ausgabeeinheit interagiert also mit jeder Eingabeeinheit. CNNs weisen jedoch üblicherweise **spärliche Interaktionen** auf, die auch **spärliche Konnektivität** (engl. *sparse connectivity*) oder **dünnbesetzte Gewichte** (engl. *sparse weights*) genannt werden. Das liegt daran, dass der Kernel kleiner als die Eingabe ist. Bei der Bildverarbeitung kann das Eingabebild zum Beispiel aus Tausenden oder Millionen von Pixeln bestehen, aber wir können mit Kernen, die nur einige Dutzend oder Hundert Pixel umfassen, kleine aussagekräftige Merkmale wie Kanten aufspüren. Auf diese Weise müssen wir weniger Parameter speichern und reduzieren so den Speicherplatzbedarf des Modells, während wir gleichzeitig seine statistische Effizienz erhöhen. Außerdem kann die Ausgabe in weniger Operationen berechnet werden. Diese Verbesserungen der Effizienz sind meist enorm groß. Für m Eingaben und n Ausgaben erfordert die Matrizenmultiplikation $m \times n$ Parameter. Die in der Praxis eingesetzten Algorithmen weisen eine Laufzeit von $O(m \times n)$ auf (pro Beispiel). Wenn wir die Anzahl der Verbindungen für jede Ausgabe auf k einschränken, erfordert der Ansatz mit dünnbesetzten Verbindungen lediglich $k \times n$ Parameter und eine Laufzeit von $O(k \times n)$. In der Praxis ist es oft möglich, eine gute Leistung für die Machine-Learning-Aufgabe zu erzielen, wenn k mehrere Größenordnungen kleiner als m ist. Abbildungen 9.2 und 9.3 stellen die spärliche Konnektivität graphisch dar. In einem tiefen CNN können Einheiten in den tieferen Schichten auf *indirekte* Weise mit einem Großteil der Eingabe interagieren (vgl. Abbildung 9.4). Damit kann das Netz komplizierte Interaktionen zwischen vielen Variablen durch Konstruieren solcher Interaktionen aus einfachen Bausteinen, die jeder für sich nur spärliche Interaktionen beschreiben, effizient beschreiben.

Parameter Sharing bezeichnet die Nutzung desselben Parameters für mehr als eine Funktion im Modell. In einem klassischen neuronalen Netz wird jedes Element der Gewichtungsmatrix für die Berechnung der Ausgabe einer Schicht exakt einmal verwendet. Es wird mit einem Element der Eingabe

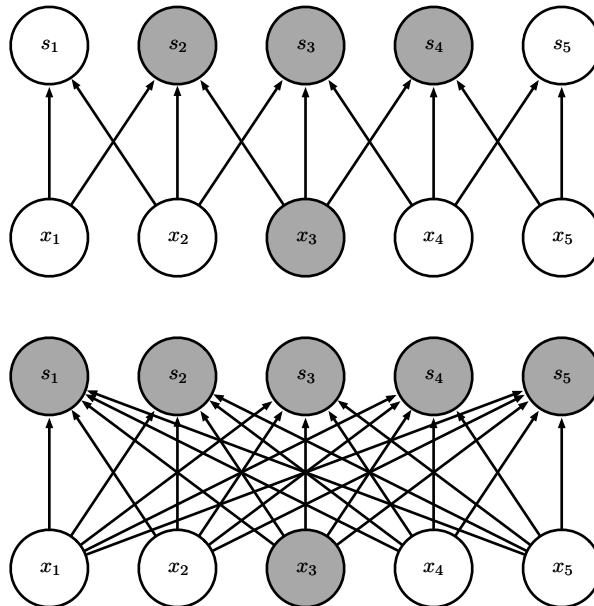


Abbildung 9.2: Spärliche Konnektivität, von unten betrachtet. Wir markieren eine Eingabeeinheit x_3 und die Ausgabeeinheit in s , die von dieser Einheit beeinflusst werden. (Oben) Wenn s durch Faltung mit einem Kernel der Breite 3 gebildet wird, wirkt x nur auf drei Ausgaben. (Unten) Wenn s durch Matrizenmultiplikation gebildet wird, ist die Konnektivität nicht länger spärlich, sodass sämtliche Ausgaben von x_3 beeinflusst werden.

multipliziert und dann nie mehr benötigt. Synonym zum Begriff Parameter Sharing kann man sagen, das Netz weist **gebundene Gewichte** (engl. *tied weights*) auf, da der Wert der auf eine Eingabe angewandten Gewichte an den Wert eines anderswo angewandten Gewichts gebunden ist. In einem CNN wird jedes Element des Kernels an jeder Position der Eingabe verwendet (ausgenommen vielleicht einige der Randpixel, je nach Designentscheidung bezüglich des Rands). Parameter Sharing während der Faltungsoperation bedeutet, dass nur eine Parametermenge anstelle vieler Mengen (nämlich für jede Position eine) erlernt wird. Das wirkt sich nicht auf die Laufzeit der Forward-Propagation aus – diese beträgt nach wie vor $O(k \times n)$ –, aber es reduziert den Speicherbedarf des Modells auf k Parameter. Sie haben bereits erfahren, dass k normalerweise mehrere Größenordnungen kleiner als m ist. Da m und n meist ungefähr gleich groß sind, ist k im Vergleich zu $m \times n$ praktisch unbedeutend. Die Faltung ist, was Speicherbedarf und statistische Effizienz betrifft, somit erheblich effizienter als die Multiplikation mit einer

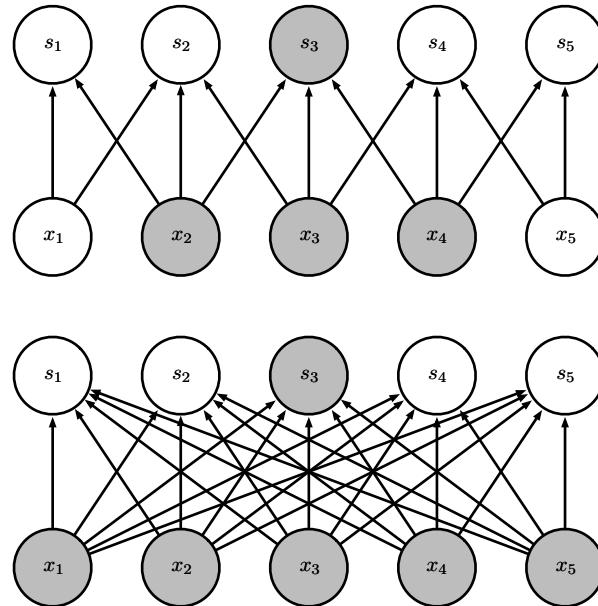


Abbildung 9.3: Spärliche Konnektivität, von oben betrachtet. Wir markieren eine Ausgabeeinheit s_3 und die Eingabeeinheiten in \mathbf{x} , die diese Einheit beeinflussen. Diese Einheiten werden als das **rezeptive Feld** von s_3 bezeichnet. (*Oben*) Wenn \mathbf{s} durch Faltung mit einem Kernel der Breite 3 gebildet wird, beeinflussen nur drei Eingaben s_3 . (*Unten*) Wenn \mathbf{s} durch Matrizenmultiplikation gebildet wird, gibt es keine spärliche Konnektivität mehr, sodass sämtliche Eingaben auf s_3 einwirken.

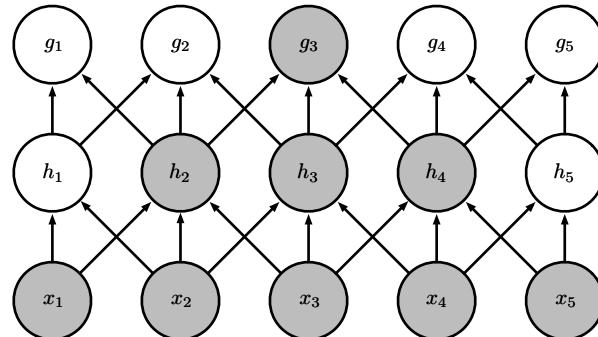


Abbildung 9.4: Das rezeptive Feld der Einheiten in den tieferen Schichten eines CNNs ist größer als das rezeptive Feld der Einheiten in den flachen Schichten. Dieser Effekt nimmt zu, wenn das Netz architektonische Merkmale wie Strided Convolution (Faltung mit erhöhter Schrittweite, Abbildung 9.12) oder Pooling (Abschnitt 9.3) enthält. Obwohl also *direkte* Verbindungen in einem CNN sehr spärlich sind, können Einheiten in den tieferen Schichten *indirekt* mit der Gesamtheit oder dem größten Teil des Eingabebildes verbunden sein.

vollbesetzten Matrix. Eine graphische Darstellung des Parameter Sharings finden Sie in Abbildung 9.5.

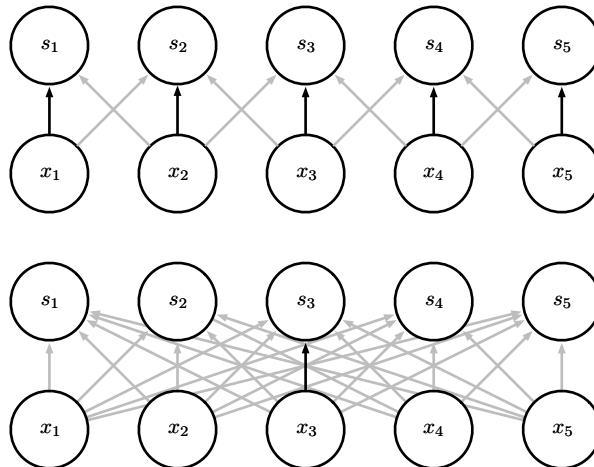


Abbildung 9.5: Parameter Sharing. Schwarze Pfeile zeigen in zwei verschiedenen Modellen die Verbindungen an, die einen bestimmten Parameter nutzen. (*Oben*) Die schwarzen Pfeile zeigen die Nutzung des zentralen Elements eines Kernels mit 3 Elementen in einem Modell mit Faltung an. Aufgrund des Parameter Sharings wird dieser einzelne Parameter an allen Eingabepositionen genutzt. (*Unten*) Der einzelne schwarze Pfeil zeigt die Nutzung des zentralen Elements der Gewichtungsmatrix in einem vollständig verbundenen Modell an. Da es in diesem Modell kein Parameter Sharing gibt, wird der Parameter nur einmal benutzt.

Ein Beispiel der beiden erstgenannten Konzepte zeigt Abbildung 9.6. Spärliche Konnektivität und Parameter Sharing können die Effizienz einer linearen Funktion zur Kantenerkennung in einem Bild deutlich steigern.

Bei der Faltung führt die besondere Art des Parameter Sharings dazu, dass die Schicht eine Eigenschaft namens **Äquivarianz** gegenüber der Verschiebung aufweist. Eine Funktion ist dann äquivariant, wenn eine Änderung der Eingabe zu einer gleichartigen Änderung der Ausgabe führt. Insbesondere ist eine Funktion $f(x)$ äquivariant zu einer Funktion g , wenn $f(g(x)) = g(f(x))$ ist. Wenn wir bei der Faltung eine Funktion g einsetzen, die die Eingabe verschiebt, dann ist die Faltungsfunktion äquivariant gegenüber g . Beispiel: I sei eine Funktion, die für ganzzahlige Koordinaten die Bildhelligkeit ermittelt. g sei eine Funktionszuordnung einer Bildfunktion zu einer anderen Bildfunktion, sodass $I' = g(I)$ die Bildfunktion mit $I'(x, y) = I(x - 1, y)$ ist. Damit wird jedes Pixel aus I um eine Einheit nach rechts verschoben. Wenn wir diese Transformation auf I anwenden und anschließend die Faltung nutzen, entspricht das Ergebnis einer Faltung

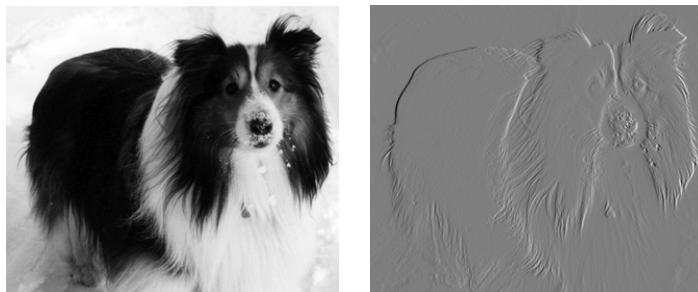


Abbildung 9.6: Effizienz der Kantenerkennung. Das rechte Bild entstand aus dem Originalbild (links), indem von jedem Pixel der Wert des jeweiligen linken benachbarten Pixels subtrahiert wurde. Dies zeigt die Stärke aller vertikal ausgerichteten Kanten im Eingabebild an – eine praktische Operation im Rahmen der Objekterkennung. Beide Bilder sind 280 Pixel hoch. Das Eingabebild ist 320 Pixel breit, das Ausgabebild dagegen nur 319 Pixel. Diese Transformation lässt sich durch einen Faltungskern mit zwei Elementen beschreiben. Sie benötigt $319 \times 280 \times 3 = 267\,960$ Gleitkommaoperationen (zwei Multiplikationen und eine Addition für jedes Ausgabepixel) für die Berechnung mittels Faltung. Um diese Transformation als Matrizenmultiplikation zu beschreiben, werden $320 \times 280 \times 319 \times 280$, also über acht Milliarden Einträge in der Matrix benötigt. Die Faltung ist also vier Milliarden Mal effizienter bei der Darstellung dieser Transformation. Der direkte Algorithmus für die Matrizenmultiplikation führt mehr als sechzehn Milliarden Gleitkommaoperationen aus; rechnerisch ist die Faltung in etwa 60 000 Mal effizienter. Natürlich sind die meisten Einträge der Matrix Null. Wenn wir nur die von Null verschiedenen Einträge der Matrix speichern würden, wäre die Anzahl der zu berechnenden Gleitkommaoperationen für Matrizenmultiplikation und Faltung gleich. Allerdings muss die Matrix dann noch immer $2 \times 319 \times 280 = 178\,640$ Einträge enthalten. Die Faltung ist ein extrem effizienter Weg zum Beschreiben von Transformationen, die dieselbe lineare Transformation auf einen kleinen, lokalen Bereich der gesamten Eingabe anwendet. Foto: Paula Goodfellow

von I' mit anschließender Transformation g der Ausgabe. Beim Verarbeiten von Zeitreihendaten erzeugt die Faltung also eine Art Zeitstrahl, auf dem wir ablesen können, wann unterschiedliche Merkmale in der Eingabe auftreten. Wenn wir ein Ereignis auf einen späteren Zeitpunkt in der Eingabe verschieben, kommt es in der Ausgabe zu exakt derselben Repräsentation des Ereignisses – nur später. Ebenso erzeugt die Faltung bei Bildern eine 2-D-Karte des Auftretens bestimmter Merkmale in der Eingabe. Wenn wir das Objekt in der Eingabe verschieben, verschiebt sich seine Repräsentation in der Ausgabe in denselben Maß. Das ist nützlich, wenn wir wissen, dass eine Funktion einer kleinen Anzahl benachbarter Pixel bei der Anwendung auf mehrere Eingabepositionen vorteilhaft ist. Ein Beispiel: In der Bild-

verarbeitung ist es nützlich, in der ersten Schicht eines CNNs Kanten zu erkennen. Dieselben Kanten erscheinen mehr oder weniger überall im Bild. Daraus folgt, dass das Teilen von Parametern über das gesamte Bild von Vorteil ist. In einigen Fällen möchten wir Parameter nicht über das gesamte Bild teilen. Wenn wir zum Beispiel Bilder verarbeiten, die so beschnitten sind, dass das Gesicht einer Person in der Mitte erscheint, möchten wir vermutlich unterschiedliche Merkmale an unterschiedlichen Positionen extrahieren: Der Teil des Netzes, der die obere Hälfte des Gesichts analysiert, soll nach Augenbrauen suchen, wohingegen der Teil für die untere Hälfte ein Kinn suchen soll.

Die Faltung ist nicht natürlich äquivariant zu einigen anderen Transformationen wie Maßstabsänderungen oder Bilddrehungen. Für solche Transformationen werden andere Mechanismen benötigt.

Es gibt auch einige Arten von Daten, die sich nicht mit neuronalen Netzen auf Basis von Matrizenmultiplikation mit unveränderlichen Matrixformen verarbeiten lassen. Solche Daten können mithilfe der Faltung verarbeitet werden. In Abschnitt 9.7 kommen wir darauf zurück.

9.3 Pooling

Eine typische Schicht in einem CNN umfasst drei Phasen (vgl. Abbildung 9.7). In der ersten Phase führt die Schicht mehrere Faltungen gleichzeitig durch, um eine Reihe linearer Aktivierungen zu erzeugen. In der zweiten Phase wird auf jede lineare Aktivierung eine nichtlineare Aktivierungsfunktion angewandt, wie beispielsweise bei der ReLU-Funktion (engl. *rectified linear activation function*). Diese Phase wird manchmal als **Erkennungsphase** (engl. *detector stage*) bezeichnet. In der dritten Phase nutzen wir eine sogenannte **Pooling-Funktion**, um die Ausgabe der Schicht weiter zu modifizieren.

Eine Pooling-Funktion ersetzt die Ausgabe des Netzes an einer bestimmten Stelle durch eine zusammengefasste statistische Größe der nahe gelegenen Ausgaben. So listet das **Max-Pooling** (Zhou und Chellappa, 1988) die maximale Ausgabe in einer rechteckigen Umgebung auf. Andere beliebte Pooling-Funktionen ermitteln den Durchschnittswert einer rechteckigen Umgebung, die L^2 -Norm einer rechteckigen Umgebung oder ein gewichtetes Mittel auf Grundlage des Abstands zu einem zentralen Pixel.

In allen Fällen hilft das Pooling dabei, die Repräsentation annähernd **invariant** gegenüber kleineren Verschiebungen der Eingabe zu machen. Invarianz gegenüber Verschiebung bedeutet, dass sich bei einer geringfügig-

gen Verschiebung der Eingabe die Werte der meisten zusammengefassten Ausgaben (engl. *pooled outputs*) nicht ändern. Abbildung 9.8 stellt die Funktionsweise an einem Beispiel dar. *Invarianz gegenüber lokaler Verschiebung kann eine nützliche Eigenschaft sein, wenn wir uns mehr dafür interessieren, ob ein bestimmtes Merkmal vorliegt, und nicht so sehr für seine exakte Lage.* Ein Beispiel: Wenn wir ermitteln, ob ein Bild ein Gesicht zeigt, müssen wir die Lage der Augen nicht auf Pixel genau kennen. Es reicht aus, zu wissen, dass es auf der rechten und linken Seite des Gesichts jeweils ein Auge gibt. In anderen Zusammenhängen ist es dagegen wichtiger, die Lage eines Merkmals beizubehalten. Ein Beispiel: Wenn wir den Eckpunkt suchen, in dem sich zwei Kanten mit einer bestimmten Ausrichtung treffen, müssen wir die Lage der Kanten genau genug bestimmen, um die Lage des Eckpunkts zu überprüfen.

Pooling lässt sich als eine unendlich starke A-priori-Wahrscheinlichkeit betrachten, dass die von der Schicht erlernte Funktion invariant gegenüber kleinen Verschiebungen sein muss. Ist diese Annahme korrekt, kann sie die statistische Effizienz des Netzes enorm steigern.

Das Pooling über räumliche Bereiche führt zu einer Invarianz gegenüber Verschiebungen. Wenn wir aber ein Pooling über die Ausgaben separat parametrisierter Faltungen durchführen, können die Merkmale lernen, gegenüber welchen Transformationen sie invariant werden müssen (vgl. Abbildung 9.9).

Da beim Pooling die Reaktionen einer vollständigen Umgebung zusammengefasst werden, können wir weniger Pooling-Einheiten als Erkennungseinheiten (engl. *detector units*) verwenden, indem wir zusammengefasste statistische Größen für Pooling-Bereiche mit einem Abstand von k Pixeln zueinander anstelle von einem Pixel auflisten. Abbildung 9.10 zeigt dafür ein Beispiel. Damit wird die Berechnungseffizienz des Netzes erhöht, denn die nächste Schicht muss in etwa k Eingaben weniger verarbeiten. Wenn die Anzahl der Parameter in der nächsten Schicht eine Funktion ihrer Eingabegröße ist (z. B. wenn die nächste Schicht vollständig verbunden ist und auf einer Matrizenmultiplikation beruht), kann eine derart verringerte Eingabegröße auch zu einer verbesserten statistischen Effizienz und einem geringeren Speicherbedarf für die Parameterablage führen.

Bei vielen Aufgaben ist Pooling für die Handhabung von Eingaben unterschiedlicher Größe unbedingt erforderlich. Wenn wir zum Beispiel Bilder unterschiedlicher Größe klassifizieren möchten, muss die Eingabe für die Klassifizierungsschicht eine feste Größe aufweisen. Dazu wird normalerweise die Größe eines Offsets zwischen den Pooling-Bereichen so verändert, dass die Klassifizierungsschicht stets dieselbe Anzahl zusammengefasster

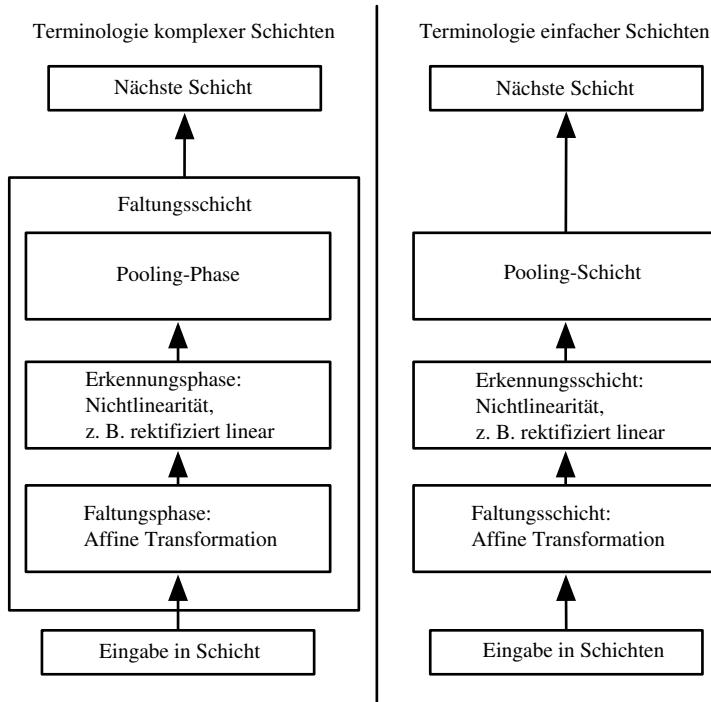


Abbildung 9.7: Die Komponenten einer typischen Schicht in einem CNN. Es gibt zwei übliche Terminologien zur Beschreibung dieser Schichten. (*Links*) In dieser Terminologie wird das CNN als eine kleine Anzahl relativ komplexer Schichten betrachtet. Dabei enthält jede Schicht viele »Phasen«. Hier liegt eine Eins-zu-eins-Zuordnung von Kernel-Tensoren zu Netzsichten vor. Es ist diese Terminologie, die wir in diesem Buch normalerweise verwenden. (*Rechts*) In dieser Terminologie wird das CNN als eine große Menge einfacher Schichten betrachtet; jeder einzelne Verarbeitungsschritt wird als separate Schicht angesehen. Das hat zur Folge, dass nicht jede »Schicht« über Parameter verfügt.

statistischer Größen empfängt – ungeachtet der Eingabegröße. So kann die endgültige Pooling-Schicht des Netzes so definiert sein, dass vier Mengen von zusammengefassten statistischen Größen ausgegeben werden, nämlich eine für jeden Bildquadranten, und das ungeachtet der Bildgröße.

Es gibt eine theoretische Untersuchung, die uns Hinweise gibt, welche Art von Pooling wir in verschiedenen Situationen einsetzen sollten (*Boureau et al., 2010*). Merkmale können auch dynamisch zu einem Pool zusammengefasst werden, zum Beispiel mittels Clustering-Algorithmus für die Positionen der interessanten Merkmale (*Boureau et al., 2011*). Dieser Ansatz führt zu einer anderen Menge von Pooling-Bereichen in jedem Bild. Ein weiterer Ansatz

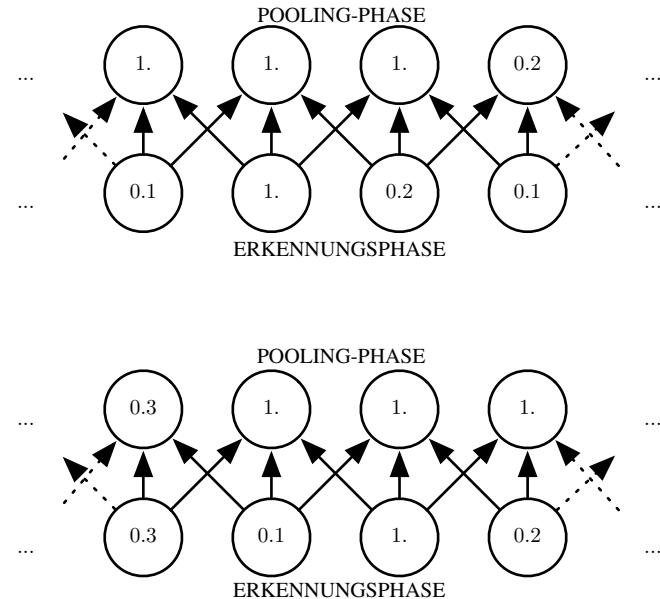


Abbildung 9.8: Max-Pooling führt zu Invarianz. (*Oben*) Die Abbildung zeigt den mittleren Bereich der Ausgabe einer Faltungsschicht. Die untere Zeile zeigt Ausgaben der Nichtlinearität. Die obere Zeile zeigt die Ausgaben nach dem Max-Pooling mit einer Schrittweite von einem Pixel zwischen den Pooling-Bereichen und drei Pixel breiten Pooling-Bereichen. (*Unten*) Eine Darstellung desselben Netzes, nachdem die Eingabe um ein Pixel nach rechts verschoben wurde. Jeder Wert in der unteren Zeile hat sich geändert, aber nur die Hälfte der Werte in der oberen Zeile, denn die Max-Pooling-Einheiten reagieren nur auf den Höchstwert in der Umgebung, nicht auf dessen exakte Lage.

besteht im *Erlernen* einer einzelnen Pooling-Struktur, die dann für alle Bilder verwendet wird (Jia et al., 2012).

Pooling kann bei einigen Architekturen neuronaler Netze, die Top-down-Informationen nutzen, zu Komplikationen führen; dazu zählen zum Beispiel Boltzmann-Maschinen und Autoencoder. Diese Probleme werden bei der Behandlung dieser Netze in Teil III noch genauer behandelt. Das Pooling in gefalteten Boltzmann-Maschinen wird in Abschnitt 20.6 vorgestellt. Die inversartigen Operationen für Pooling-Einheiten, die für einige differenzierbare Netze benötigt werden, sind ein Thema in Abschnitt 20.10.6.

Einige Beispiele vollständiger Architekturen für CNNs zur Klassifizierung mithilfe von Faltung und Pooling sind in Abbildung 9.11 dargestellt.

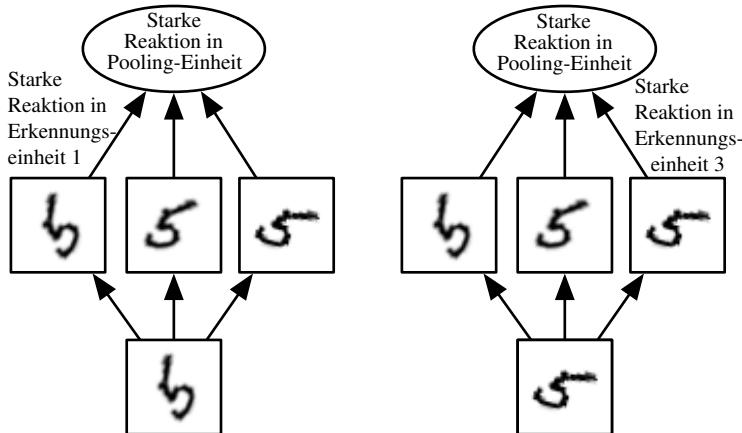


Abbildung 9.9: Beispiele für erlernte Invarianzen. Eine Pooling-Einheit, die das Pooling über mehrere Merkmale durchführt, die mit separaten Parametern angelernt wurden, kann eine Invarianz gegenüber Transformationen der Eingabe erlernen. Hier zeigen wir, wie eine Menge mit drei angelernten Filtern und einer Max-Pooling-Einheit die Invarianz gegenüber Drehungen erlernen kann. Alle drei Filter sollen eine handschriftliche 5 erkennen. Jeder Filter versucht, die 5 in einer leicht veränderten Ausrichtung zu erkennen. Wenn eine 5 in der Eingabe erscheint, passt der entsprechende Filter darauf und verursacht eine große Aktivierung in einer Erkennungseinheit. Die Max-Pooling-Einheit weist dann unabhängig davon, welche Erkennungseinheit aktiviert wurde, eine hohe Aktivierung auf. Wir zeigen hier, wie das Netz die beiden unterschiedlichen Einheiten verarbeitet, was dazu führt, dass zwei unterschiedliche Erkennungseinheiten aktiviert werden. Der Effekt auf die Pooling-Einheit ist in allen Fällen ungefähr gleich. Dieses Prinzip wird von Maxout-Netzen (Goodfellow et al., 2013a) und anderen CNNs genutzt. Max-Pooling über räumliche Positionen ist von Natur aus invariant gegenüber Verschiebung; dieser Multichannel-Ansatz ist nur für das Erlernen anderer Transformationen erforderlich.

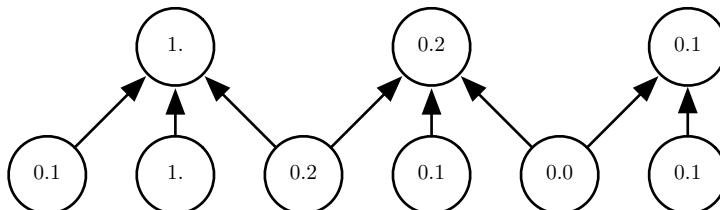


Abbildung 9.10: Pooling mit Downsampling. Hier nutzen wir Max-Pooling mit einem Pool der Breite 3 und Schrittweite 2 zwischen den Pools. Dadurch wird die Repräsentationsgröße um einen Faktor 2 reduziert, was wiederum den Rechenaufwand und die statistische Last in der nächsten Schicht reduziert. Beachten Sie, dass der Pooling-Bereich ganz rechts kleiner ist, aber dennoch eingeschlossen werden muss, wenn wir nicht einige der Erkennungseinheiten ignorieren wollen.

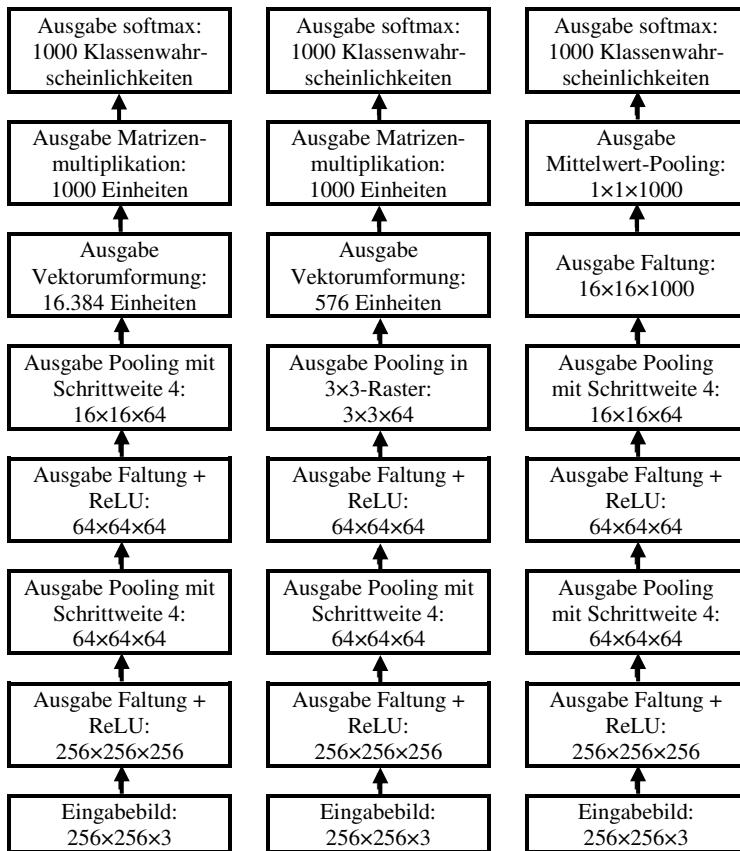


Abbildung 9.11: Beispiele für Architekturen zur Klassifizierung mit CNNs. Die spezifischen Schrittweiten und Tiefen in dieser Abbildung sind nicht praxistauglich. Sie wurden vielmehr gewählt, weil sie so flach sind, dass die Abbildung auf die Seite passt. Echte CNNs weisen häufig umfangreiche Verzweigungen auf – die Kettenstrukturen hier wurden aus Gründen der Einfachheit gewählt. (*Links*) Ein CNN zur Verarbeitung von Bildern mit fester Größe. Nach dem Wechsel zwischen Faltung und Pooling über einige Schichten hinweg wird der Tensor für die gefaltete Merkmalskarte so geformt, dass die räumlichen Dimensionen abgeflacht werden. Der Rest des Netzes ist ein klassisches Feedforward-Netz zur Klassifizierung (vgl. Kapitel 6). (*Mitte*) Ein CNN zur Verarbeitung von Bildern mit variablen Größen, das nach wie vor einen vollständig verbundenen Bereich enthält. Dieses Netz nutzt eine Pooling-Operation mit Pools veränderlicher Größe, aber stets derselben Anzahl von Pools; so wird ein Vektor mit fester Größe von 576 Einheiten für den vollständig verbundenen Teil des Netzes erreicht. (*Rechts*) Ein CNN ohne jegliche vollständig verbundene Gewichtungsschicht. Stattdessen gibt die letzte Faltungsschicht eine Merkmalskarte für jede Klasse aus. Das Modell erlernt vermutlich eine Karte der Wahrscheinlichkeiten, mit denen jede Klasse an jeder räumlichen Position auftritt. Durch Mitteln einer Merkmalskarte auf einen Einzelwert wird das Argument für den softmax-Klassifikator ganz oben bestimmt.

9.4 Faltung und Pooling als unendlich starke A-priori-Wahrscheinlichkeit

In Abschnitt 5.2 haben Sie das Konzept von **A-priori-Wahrscheinlichkeitsverteilungen** kennengelernt. Dies ist eine Wahrscheinlichkeitsverteilung über die Parameter eines Modells, die unsere Überzeugungen darüber repräsentiert, welche Modelle angemessen sind, und zwar, bevor wir irgendwelche Daten gesehen haben.

A-priori-Wahrscheinlichkeiten sind je nach Konzentration der Wahrscheinlichkeitsdichte entweder schwach oder stark. Eine schwache A-priori-Wahrscheinlichkeit ist eine A-priori-Verteilung mit hoher Entropie, zum Beispiel eine Normalverteilung mit großer Varianz. Solch eine A-priori-Wahrscheinlichkeit lässt eine mehr oder weniger freie Parameterverschiebung durch die Daten zu. Eine starke A-priori-Wahrscheinlichkeit weist eine sehr niedrige Entropie auf, zum Beispiel eine Normalverteilung mit geringer Varianz. Eine solche A-priori-Wahrscheinlichkeit spielt eine aktivere Rolle beim Bestimmen der Endposition der Parameter.

Eine unendlich starke A-priori-Wahrscheinlichkeit belegt einige Parameter mit der Wahrscheinlichkeit Null und besagt, dass diese Parameterwerte absolut verboten sind, ungeachtet dessen, wie sehr die Daten diese Werte unterstützen.

Sie können sich ein CNN ähnlich einem vollständig verbundenen Netz vorstellen, aber mit einer unendlich starken A-priori-Wahrscheinlichkeitsverteilung über seine Gewichte. Diese unendlich starke A-priori-Wahrscheinlichkeit besagt, dass die Gewichte für eine verdeckte Einheit den Gewichten ihrer Nachbarn entsprechen muss, allerdings im Raum verschoben ist. Die A-priori-Wahrscheinlichkeit besagt außerdem, dass die Gewichte Null sein müssen – außer in dem kleinen räumlich zusammenhängenden rezeptiven Feld (engl. *receptive field*), das der verdeckten Einheit zugewiesen ist. Insgesamt können Sie sich die Faltung als Einbringen einer unendlich starken A-priori-Wahrscheinlichkeitsverteilung über die Parameter einer Schicht vorstellen. Diese A-priori-Wahrscheinlichkeit besagt, dass die von der Schicht zu erlernende Funktion nur lokale Interaktionen enthält und äquivalent gegenüber der Verschiebung ist. Auf ähnliche Weise ist der Einsatz von Pooling eine unendlich starke A-priori-Wahrscheinlichkeit, dass jede Einheit invariant gegenüber kleinen Verschiebungen sein sollte.

Natürlich wäre es eine extreme Verschwendug von Rechenressourcen, würde man ein CNN als vollständig verbundenes Netz mit einer unendlich

starken A-priori-Wahrscheinlichkeit umsetzen. Doch die Vorstellung dieser Umsetzung gewährt einige Einblicke in die Funktionsweise von CNNs.

Eine wesentliche Erkenntnis besteht darin, dass Faltung und Pooling zu Unteranpassung führen können. Wie jede A-priori-Wahrscheinlichkeit sind Faltung und Pooling nur dann nützlich, wenn die dadurch gemachten Annahmen hinreichend genau sind. Wenn für eine Aufgabe exakte räumliche Informationen erhalten werden müssen, kann ein Pooling aller Merkmale den Trainingsfehler erhöhen. Einige Architekturen von CNNs (*Szegedy et al.*, 2014a) sind so aufgebaut, dass Pooling nur bei einigen Kanälen zum Einsatz kommt. So ergeben sich sowohl stark invariante Merkmale und Merkmale, die im Falle eines falschen Invarianz-Priors keine Unteranpassung zur Folge haben. Bezieht eine Aufgabe Informationen von weit entfernten Positionen in die Eingabe ein, kann eine durch Faltung auferlegte A-priori-Wahrscheinlichkeit unangemessen sein.

Eine weitere wichtige Erkenntnis ist, dass wir Modelle mit Faltung in Benchmarks der statistischen Lernleistung nur gegen andere Modelle mit Faltung antreten lassen sollten. Modelle ohne Faltung könnten sogar dann noch lernen, wenn wir alle Pixel des Bildes permutieren würden. Für viele Bilderdatensätze gibt es unterschiedliche Benchmarks für Modelle, die **invariant gegenüber Permutationen** sind und das Konzept der Topologie somit erlernen müssen, und für Modelle, in denen das Wissen über räumliche Beziehungen bereits fest hinterlegt ist.

9.5 Varianten der grundlegenden Faltungsfunktion

Wenn im Kontext neuronaler Netze von Faltung die Rede ist, meinen wir nur selten die übliche diskrete Faltungsoperation aus der Mathematik. Die in der Praxis verwendeten Funktionen weichen geringfügig ab. Wir beschreiben diese Unterschiede im Folgenden genauer und heben einige nützliche Eigenschaften der in neuronalen Netzen verwendeten Funktionen hervor.

Es sei zunächst gesagt, dass Faltung für neuronale Netze meist die Durchführung einer Operation bezeichnet, die viele Faltungen gleichzeitig vornimmt. Das ist der Fall, da die Faltung mit einem einzelnen Kernel nur eine Art von Merkmal extrahieren kann – dies allerdings an vielen räumlichen Positionen. Wir möchten normalerweise, dass jede Schicht in unserem Netz so viele Arten von Merkmalen an so vielen Positionen wie möglich extrahiert.

Außerdem handelt es sich bei der Eingabe für gewöhnlich nicht um ein Raster reeller Werte, sondern vielmehr um ein Raster vektorwertiger Beobachtungen. Zum Beispiel enthält ein farbiges Bild in jedem Pixel Intensitätswerte für Rot, Grün und Blau. In einem mehrschichtigen CNN ist die Eingabe der zweiten Schicht die Ausgabe der ersten Schicht; meist enthält diese die Ausgabe vieler unterschiedlicher Faltungen für jede Position. Im Zusammenhang mit Bildern stellen wir uns Eingabe und Ausgabe der Faltung allgemein als 3-D-Tensoren vor, wobei ein Index in die unterschiedlichen Kanäle einfließt und zwei Indizes in die räumlichen Koordinaten der einzelnen Kanäle. Softwareimplementierungen arbeiten meist im Batch-Modus, d. h. sie nutzen tatsächlich 4-D-Tensoren mit einer vierten Achse zur Indizierung der unterschiedlichen Beispiele im Batch. Allerdings lassen wir die Batch-Achse in unserer Beschreibung hier aus Gründen der Einfachheit weg.

Da CNNs normalerweise eine Multichannel-Faltung nutzen, sind die linearen Operationen, auf denen sie beruhen, nicht unbedingt kommutativ; das gilt auch, wenn die Kerneldrehung eingesetzt wird. Diese Mehrkanal-Operationen sind nur kommutativ, wenn jede Operation dieselbe Anzahl von Ausgabe- und Eingabekanälen aufweist.

Angenommen, ein 4-D-Kernel-Tensor \mathbf{K} mit Element $K_{i,j,k,l}$ gibt die Verbindungsstärke zwischen einer Einheit im Kanal i der Ausgabe und einer Einheit im Kanal j der Eingabe mit einem Offset von k Zeilen und l Spalten zwischen Ausgabe- und Eingabeeinheit an. Weiter angenommen, unsere Eingabe besteht aus beobachteten Daten \mathbf{V} mit Element $V_{i,j,k}$ für den Wert der Eingabeeinheit im Kanal i für die Zeile j und die Spalte k . Und wiederum angenommen, unsere Ausgabe besteht aus \mathbf{Z} im selben Format wie \mathbf{V} . Wenn \mathbf{Z} durch Faltung von \mathbf{K} über \mathbf{V} ohne Drehung von \mathbf{K} entsteht, dann gilt

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}, \quad (9.7)$$

wobei die Aufsummierung über l , m und n über alle Werte erfolgt, für die die Tensor-Indizierungsoperationen in der Aufsummierung gültig sind. In der linearen Algebra indizieren wir Anordnungen mit einer 1 für den ersten Eintrag. Für obige Formel wird somit -1 benötigt. Der Index in Programmiersprachen wie C und Python beginnt bei 0, sodass der obige Ausdruck noch weiter vereinfacht wird.

Wir können einige Positionen im Kernel überspringen, um den Berechnungsaufwand zu senken (wobei dann die Merkmale nicht ganz so fein extrahiert werden). Das ähnelt dem Downsampling (Bündeln der Reprä-

sentationsgröße) der Ausgabe der vollständigen Faltung. Wenn wir nur jedes s -te Pixel in jeder Richtung der Ausgabe ziehen möchten, können wir eine Faltungsfunktion mit Downsampling c so definieren, dass sich folgende Formel ergibt:

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} [V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n}]. \quad (9.8)$$

Wir bezeichnen s als die **Schrittweite** der Faltung mit Downsampling. Es ist auch möglich, eine separate Schrittweite für jede Bewegungsrichtung anzugeben. Abbildung 9.12 stellt dies dar.

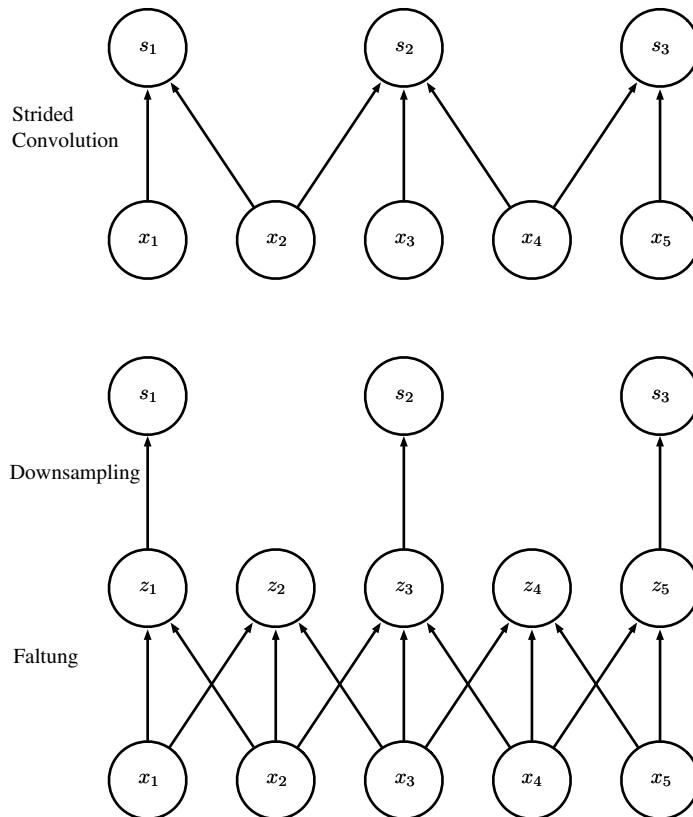


Abbildung 9.12: Strided Convolution. In diesem Beispiel beträgt die Schrittweite 2. (*Oben*) Faltung mit einer Schrittweite von 2 in einer einzigen Operation. (*Unten*) Die Faltung mit einer Schrittweite größer als ein Pixel ist mathematisch äquivalent zur Faltung mit einfacher Schrittweite und anschließendem Downsampling. Offensichtlich ist ein Ansatz mit Schrittweite 2 und Downsampling rechnerisch zu aufwendig, da viele Werte berechnet werden, die anschließend verworfen werden.

Eine wesentliche Funktion bei der Implementierung jedes CNNs ist die Fähigkeit, ein implizites Zero Padding (dt. *Auffüllen mit Nullen*) der Eingabe \mathbf{V} durchführen zu können, um sie zu verbreitern. Ohne diese Funktion verringert sich die Breite der Repräsentation gegenüber der Kernelbreite in jeder Schicht um ein Pixel. Das Zero Padding erlaubt es uns, die Kernelbreite und die Größe der Ausgabe unabhängig voneinander zu steuern. Ohne Zero Padding müssten wir uns entweder mit einem rasant abnehmenden räumlichen Ausmaß des Netzes oder mit kleinen Kernen abfinden – beides Szenarios, die die Aussagekraft des Netzes stark einschränken. Abbildung 9.13 zeigt ein Beispiel.

Es gibt drei Sonderfälle beim Zero Padding, die wir erwähnen sollten. Einer ist der Extremfall, in dem keinerlei Auffüllung stattfindet und der Faltungskern (engl. *convolution kernel*) nur Positionen aufsuchen darf, an denen der gesamte Kernel vollständig im Bild liegt. In der MATLAB-Sprache ist dies eine **valid** convolution (gültige Faltung). In diesem Fall sind sämtliche Pixel der Ausgabe eine Funktion derselben Anzahl von Pixeln der Eingabe, sodass das Verhalten eines Ausgabepixel eher geregelt ist. Allerdings nimmt die Ausgabegröße mit jeder Schicht ab. Wenn das Eingabebild die Breite m aufweist und der Kernel die Breite k , ist die Ausgabe $m - k + 1$ breit. Bei großen Kernen kann die Verkleinerungssrate enorm hoch ausfallen. Da die Verkleinerung größer als 0 ist, ist die Anzahl der im Netz möglichen Faltungsschichten eingeschränkt. Mit jeder weiteren hinzugefügten Schicht nimmt die räumliche Dimension des Netzes ab, bis sie irgendwann auf 1×1 fällt; ab diesem Punkt können weitere Schichten zweckmäßig nicht mehr als gefaltet betrachtet werden. Ein weiterer Sonderfall beim Zero Padding liegt vor, wenn gerade genug Nullen hinzugefügt werden, um die Ausgabegröße und Eingabegröße identisch zu halten. MATLAB spricht hier von einer **same** convolution (identischen Faltung). In diesem Fall kann das Netz so viele Faltungsschichten enthalten, wie die verfügbare Hardware unterstützt, da die Faltungsoperation sich nicht auf die architektonischen Möglichkeiten der nächsten Schicht auswirkt. Allerdings beeinflussen die Eingabepixel in Randnähe weniger Ausgabepixel als die Eingabepixel, die weiter innen liegen. Dadurch werden die Randpixel im Modell möglicherweise unterrepräsentiert. Diese Erkenntnis führt zu einem weiteren extremen Fall, den MATLAB als **full** convolution (vollständige Faltung) bezeichnet. Darin werden so viele Nullen hinzugefügt, dass jedes Pixel in jeder Richtung k Mal besucht wird; die Breite des Ausgabebildes ist dann $m + k - 1$. In diesem Fall sind die Ausgabepixel in Randnähe eine Funktion von weniger Pixeln als die Ausgabepixel weiter innen. Dadurch kann das Erlernen eines einzelnen Kernels, der an allen Positionen in der gefalteten Merkmalskarte

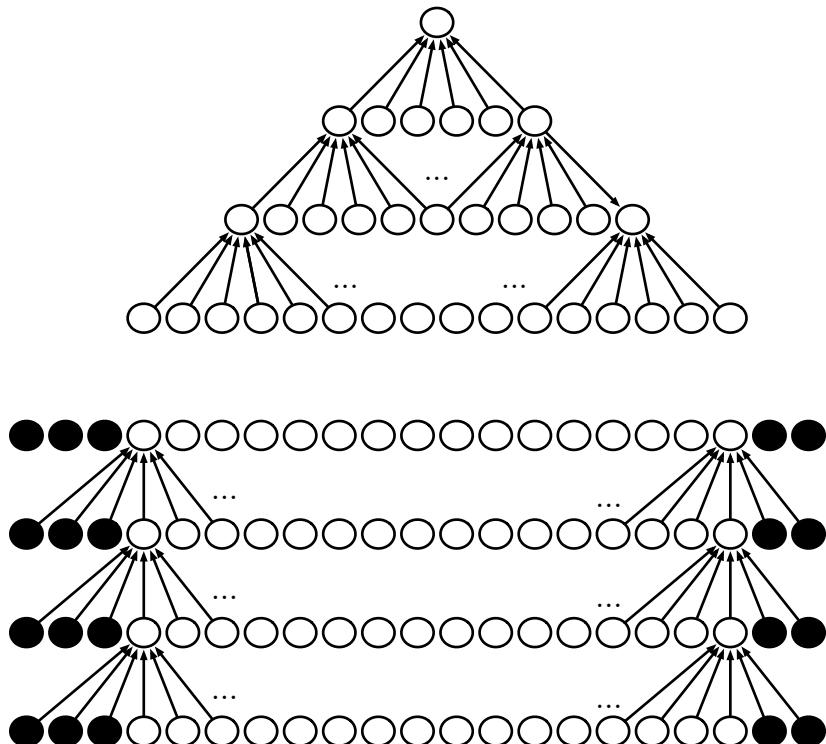


Abbildung 9.13: Die Wirkung des Zero Paddings auf die Netzgröße. Betrachten wir ein CNN mit einem Kernel der Breite 6 in jeder Schicht. In diesem Beispiel verwenden wir kein Pooling, sodass nur die Faltungsoperation selbst die Netzgröße verringert. (*Oben*) In diesem CNN verwenden wir kein implizites Zero Padding. Dadurch verkleinert sich die Repräsentation in jeder Schicht um fünf Pixel. Mit einer Eingabe, die aus 16 Pixeln besteht, können wir nur drei Faltungsschichten einsetzen, von denen die letzte den Kernel niemals verschiebt, sodass letztlich nur zwei der Schichten wirklich gefaltet sind. Die Verkleinerungsrate lässt sich durch den Einsatz kleinerer Kernel abschwächen, aber kleinere Kernel sind weniger ausdrucksstark; in dieser Architektur ist eine gewisse Verkleinerung unausweichlich. (*Unten*) Durch Hinzufügen von fünf impliziten Nullen zu jeder Schicht verhindern wir eine Verkleinerung der Repräsentation mit zunehmender Tiefe. Damit kann unser CNN beliebig tief werden.

gut funktioniert, erschwert werden. Meist liegt die optimale Wahl für das Zero Padding (bezüglich der Korrektklassifikationsrate der Testdatenmenge) irgendwo zwischen den Faltungstypen `valid` und `same`.

In einigen Fällen möchten wir gar keine Faltung einsetzen, sondern stattdessen lokal verbundene Schichten verwenden (*LeCun*, 1986, 1989). Dann ist die Adjazenzmatrix im Graphen unseres mehrschichtigen Perzeptrons dieselbe, aber jede Verbindung weist ein eigenes Gewicht auf, die durch einen 6-D-Tensor \mathbf{W} spezifiziert wird. Die Indizes für \mathbf{W} sind respektive i (Ausgabekanal), j (Ausgabezeile), k (Ausgabespalte), l (Eingabekanal), m (Zeilenversatz in der Eingabe) und n (Spaltenversatz in der Eingabe). Der lineare Teil einer lokal verbundenen Schicht ergibt sich damit aus

$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n}] . \quad (9.9)$$

Manchmal wird dies als **ungeteilte Faltung** (engl. *unshared convolution*) bezeichnet, da die Operation der diskreten Faltung mit einem kleinen Kernel ohne Parameter Sharing (also geteilte Parameter) über Positionen hinweg ähnelt. Abbildung 9.14 vergleicht lokale Verbindungen, Faltung und vollständige Verbindungen miteinander.

Lokal verbundene Schichten sind nützlich, wenn wir wissen, dass jedes Merkmal eine Funktion eines kleinen Raumabschnitts sein muss, aber es keinen Grund dafür gibt, zu glauben, dass dasselbe Merkmal im gesamten Raum wiederholt auftritt. Ein Beispiel: Um zu ermitteln, ob ein Bild ein Gesicht zeigt, müssen wir lediglich in der unteren Bildhälfte nach einem Mund suchen.

Auch das Erstellen von Versionen der Faltung oder der lokal verbundenen Schichten, in denen die Konnektivität noch weiter eingeschränkt wird, kann nützlich sein, zum Beispiel, um jedem Ausgabekanal i die Bedingung aufzuerlegen, dass er eine Funktion einer Teilmenge der Eingabekanäle l ist. Dazu werden üblicherweise die ersten m Ausgabekanäle exklusiv mit den ersten n Eingabekanälen verbunden, die zweiten m Ausgabekanäle mit den zweiten n Eingabekanälen usw. Abbildung 9.15 zeigt ein Beispiel. Das Modellieren von Interaktionen zwischen wenigen Kanälen ermöglicht Netze mit weniger Parametern, einem geringeren Speicherbedarf, einer höheren statistischen Effizienz und einem geringeren Rechenaufwand für die Forward- und Backpropagation. Diese Ziele werden ohne jede Reduzierung der Anzahl verdeckter Einheiten erreicht.

Die **Kachel-Faltung** (engl. *tiled convolution*) (*Gregor und LeCun*, 2010a; *Le et al.*, 2010) stellt einen Kompromiss zwischen einer Faltungsschicht und einer lokal verbundenen Schicht dar. Statt für jede Position im Raum eine

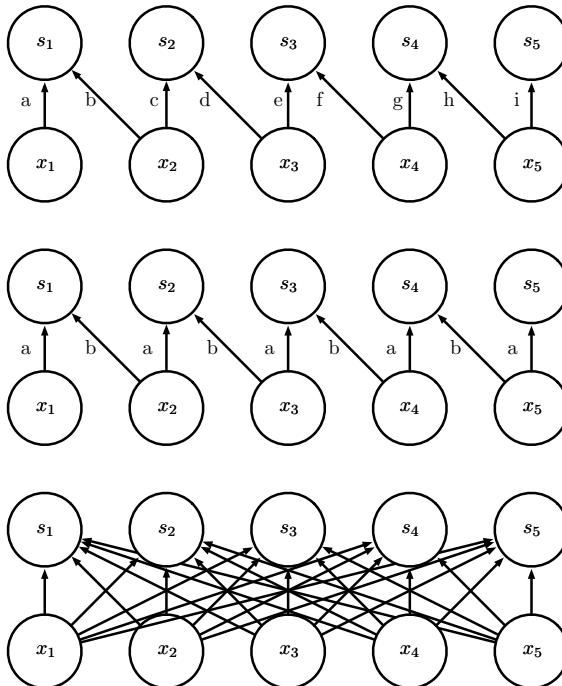


Abbildung 9.14: Vergleich von lokalen Verbindungen, Faltung und vollständigen Verbindungen. (*Oben*) Eine lokal verbundene Schicht mit einer Bereichsgröße von zwei Pixeln. Jede Kante ist mit einem eindeutigen Buchstaben gekennzeichnet, der angeibt, dass jede Kante einem eigenen Gewichtungsparameter zugewiesen ist. (*Mitte*) Eine Faltungsschicht mit einer Kernelbreite von zwei Pixeln. Dieses Modell weist exakt dieselbe Konnektivität wie die lokal verbundenen Schicht auf. Der Unterschied liegt nicht darin, welche Einheiten miteinander interagieren, sondern wie die Parameter geteilt werden. In der lokal verbundenen Schicht gibt es kein Parameter Sharing. Die Faltungsschicht verwendet dieselben zwei Gewichte wiederholt über die gesamte Eingabe; das wird durch die Wiederholung der Buchstaben an den Kanten angezeigt. (*Unten*) Eine vollständig verbundene Schicht ähnelt einer lokal verbundenen Schicht insofern, als jede Kante über eigene Parameter verfügt (in dieser Abbildung sind zu viele Parameter, so dass sie hier nicht alle einzeln gekennzeichnet werden können). Allerdings gibt es hier anders als in der lokal verbundenen Schicht keine eingeschränkte Konnektivität.

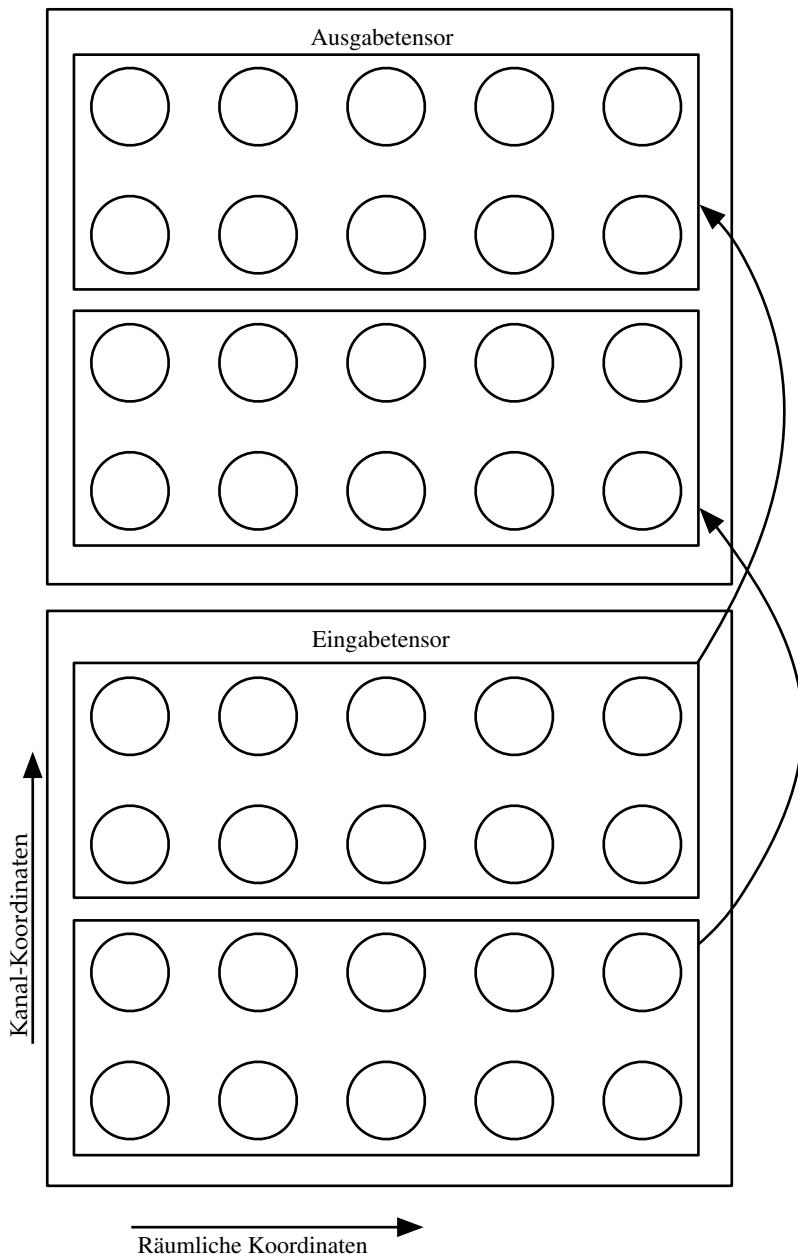


Abbildung 9.15: Ein CNN, in dem die ersten beiden Ausgabekanäle exklusiv mit den ersten beiden Eingabekanälen verbunden sind und die nächsten beiden Ausgabekanäle exklusiv mit den nächsten beiden Eingabekanälen.

eigene Menge von Gewichten zu erlernen, wird eine Menge von Kerneln erlernt, die dann während der Bewegung durch den Raum durchgewechselt werden. Damit weisen direkt benachbarte Positionen (oder Kacheln) unterschiedliche Filter auf (wie in einer lokal verbundenen Schicht), aber der Speicherplatzbedarf zum Vorhalten der Parameter steigt nur um den Faktor der Größe dieser Kernelmengen an, nicht um die Größe der gesamten Ausgabe-Merkmalskarte. Abbildung 9.16 vergleicht lokal verbundene Schichten, Kachel-Faltung und die normale Faltung miteinander.

Um eine Kachel-Faltung algebraisch zu definieren, sei k ein 6-D-Tensor, bei dem zwei der Dimensionen den unterschiedlichen Positionen in der Ausgabekarte entsprechen. Statt für jede Position der Ausgabekarte einen eigenen Index zu verwenden, wechseln die Ausgabepositionen durch eine Reihe von t verschiedenen Kernelstapel-Optionen pro Richtung. Ist t gleich der Ausgabebreite, gibt es keinen Unterschied zu einer lokal verbundenen Schicht.

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j \% t + 1, k \% t + 1}, \quad (9.10)$$

wobei Prozent für die Modul-Operation steht mit $t \% t = 0$, $(t+1) \% t = 1$ usw. Diese Gleichung kann problemlos für andere Kachelbereiche in jeder Dimension verwendet werden.

Lokal verbundene Schichten und Kachel-Faltungsschichten weisen eine interessante Interaktion mit Max-Pooling auf: Die Erkennungseinheiten dieser Schichten werden durch unterschiedliche Filter getrieben. Wenn die Filter lernen, unterschiedlich transformierte Versionen derselben grundlegenden Merkmale zu erkennen, werden die einem Max-Pooling unterzogenen Einheiten invariant gegenüber den erlernten Transformationen (vgl. Abbildung 9.9). Faltungsschichten sind speziell gegenüber Verschiebungen invariant.

Für die Implementierung eines CNNs sind meist noch andere Operationen neben der Faltung erforderlich. Für das Lernen muss der Gradient relativ zum Kernel berechnet werden (gegeben ist der Gradient relativ zu den Ausgaben). In einigen einfacheren Fällen kann diese Operation mit der Faltungsoperation durchgeführt werden, aber häufig – beispielsweise bei einer Schrittweite größer als 1 – ist diese Eigenschaft nicht gegeben.

Denken Sie daran, dass die Faltung eine lineare Operation ist und somit als Matrizenmultiplikation beschrieben werden kann (sofern wir den Eingabetensor zunächst in einen flachen Vektor umformen). Die beteiligte Matrix ist eine Funktion des Faltungskerns. Die Matrix ist spärlich und jedes Element des Kernels wird in mehrere Elemente der Matrix kopiert.

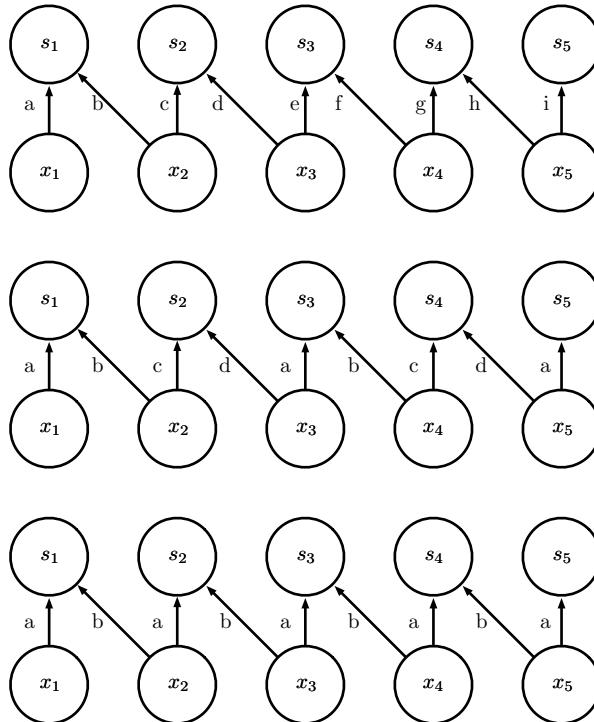


Abbildung 9.16: Ein Vergleich von lokal verbundenen Schichten, Kachel-Faltung und normaler Faltung. Alle drei weisen dieselben Verbindungen zwischen den Einheiten auf, sofern dieselbe Kernelgröße genutzt wird. Die Abbildung verwendet einen Kernel mit einer Breite von 2 Pixeln. Die Unterschiede zwischen den Verfahren liegen darin, wie die Parameter geteilt werden. (*Oben*) Eine lokal verbundene Schicht nutzt gar kein Sharing. Um anzusehen, dass jede Verbindung ein eigenes Gewicht nutzt, ist jede der Kanten mit einem eindeutigen Buchstaben gekennzeichnet. (*Mitte*) Die Kachel-Faltung verwendet eine Menge aus t unterschiedlichen Kerneln. Hier dargestellt für $t = 2$. Einer der Kernel weist Kanten mit den Kennzeichnungen **a** und **b** auf, der andere die Kanten **c** und **d**. Sobald wir uns in der Ausgabe ein Pixel nach rechts bewegen, wechseln wir zu einem anderen Kernel. So weisen die Nachbareinheiten in der Ausgabe wie in der lokal verbundenen Schicht unterschiedliche Parameter auf. Anders als in der lokal verbundenen Schicht fahren wir nach der Verwendung aller t Kernel jedoch wieder mit dem ersten Kernel fort. Wenn zwei Ausgabeeinheiten durch ein Vielfaches von t Schritten voneinander getrennt sind, teilen sie die Parameter miteinander. (*Unten*) Die klassische Faltung entspricht der Kachel-Faltung mit $t = 1$. Es gibt also nur einen Kernel, der überall angewandt wird. Das wird in der Abbildung durch die mit **a** und **b** gekennzeichneten Gewichte deutlich.

Diese Betrachtungsweise hilft uns bei der Herleitung einiger der anderen Operationen, die für die Implementierung eines CNNs notwendig sind.

Die Multiplikation mit der Transponierten der mittels Faltung definierten Matrix ist eine solche Operation. Sie wird benötigt, um eine Backpropagation der Ableitungen der Fehlerfunktion durch eine Faltungsschicht zu ermöglichen, somit also auch zum Trainieren von CNNs, die mehr als eine verdeckte Schicht nutzen. Dieselbe Operation wird auch benötigt, wenn die sichtbaren Einheiten aus den verdeckten Einheiten wiederhergestellt werden sollen (*Simard et al.*, 1992). Das Wiederherstellen der sichtbaren Einheiten ist eine häufig genutzte Operation bei den Modellen, die in Teil III dieses Buchs beschrieben werden, darunter Autoencoder, RBMs (Restricted Boltzmann Machines) und Sparse Coding. Die transponierte Faltung wird benötigt, um gefaltete Versionen dieser Modelle zu konstruieren. Wie die Kernel-Gradientenoperation kann auch diese Eingabe-Gradientenoperation manchmal mittels Faltung umgesetzt werden, aber im Allgemeinen wird dafür eine dritte Operation benötigt. Außerdem muss die Transposition sorgfältig auf die Forward-Propagation abgestimmt werden. Die Größe der Ausgabe, die von der Transposition zurückgegeben werden sollte, ist abhängig vom Zero Padding und der Schrittweite der Forward-Propagation sowie der Größe der Ausgabekarte der Forward-Propagation. In manchen Fällen können mehrere Größen der Eingabe für die Forward-Propagation zur selben Größe der Ausgabekarte führen, sodass Sie der Transposition explizit mitteilen müssen, welche Größe die ursprüngliche Eingabe hatte.

Diese drei Operationen – Faltung, Backpropagation von der Ausgabe zu den Gewichten sowie Backpropagation von der Ausgabe zu den Eingaben – reichen aus, um sämtliche Gradienten für das Trainieren beliebig tiefer Feedforward-CNNs zu berechnen. Ebenso reichen sie zum Trainieren von CNNs mit Rekonstruktionsfunktionen auf Basis der Transponierten der Faltung aus. *Goodfellow* (2010) enthält die komplette Herleitung dieser Gleichungen für den vollständig allgemeinen mehrdimensionalen Fall mit mehreren Beispielen. Um ein Gespür für die Funktionsweise dieser Gleichungen zu bekommen, stellen wir hier die zweidimensionale Version mit einem Beispiel vor:

Es soll ein CNN mit Strided Convolution des Kernelstapel **K** für das Multichannel-Bild **V** mit Schrittweite s , definiert durch $c(\mathbf{K}, \mathbf{V}, s)$ trainiert werden (vgl. Gleichung 9.8). Angenommen, unser Ziel ist die Minimierung einer Verlustfunktion $J(\mathbf{V}, \mathbf{K})$. Während der Forward-Propagation müssen wir c selbst für die Ausgabe **Z** nutzen, die dann durch den Rest des Netzes propagiert wird und beim Berechnen der Kostenfunktion J zum Einsatz

kommt. Während der Backpropagation erhalten wir einen Tensor \mathbf{G} für den gilt $G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(\mathbf{V}, \mathbf{K})$.

Für das Trainieren des Netzes müssen wir die Ableitungen bezüglich der Gewichte im Kernel bestimmen. Dazu können wir folgende Funktion nutzen:

$$g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j,(m-1)\times s+k, (n-1)\times s+l}. \quad (9.11)$$

Ist diese Schicht nicht die unterste Schicht im Netz, müssen wir den Gradienten bezüglich \mathbf{V} berechnen, um eine weitere Backpropagation des Fehlers vorzunehmen. Das ist mit dieser Funktion möglich:

$$\begin{aligned} h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} &= \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K}) \\ &= \sum_{\substack{l,m \\ \text{s.t.} \\ (l-1)\times s+m=j}} \sum_{\substack{n,p \\ \text{s.t.} \\ (n-1)\times s+p=k}} \sum_q K_{q,i,m,p} G_{q,l,n}. \end{aligned} \quad (9.12)$$

Die in Kapitel 14 beschriebenen Autoencoder-Netze sind Feedforward-Netze, die darauf trainiert werden, ihre Eingabe in die Ausgabe zu kopieren. Ein einfaches Beispiel ist der Algorithmus für die Hauptkomponentenanalyse (engl. *principal components analysis*, PCA), der seine Eingabe \mathbf{x} mittels der Funktion $\mathbf{W}^\top \mathbf{W} \mathbf{x}$ in eine approximative Rekonstruktion \mathbf{r} kopiert. In allgemeineren Autoencodern kommt häufig die Multiplikation mit der Transponierten der Gewichtungsmatrix zum Einsatz – wie in der PCA. Damit derartige Modelle gefaltet werden, können wir die Funktion h verwenden, um die Transposition der Faltungsoperation vorzunehmen. Angenommen, die verdeckten Einheiten \mathbf{H} liegen im selben Format wie \mathbf{Z} vor, und wir definieren eine Rekonstruktion

$$\mathbf{R} = h(\mathbf{K}, \mathbf{H}, s). \quad (9.14)$$

Zum Trainieren des Autoencoders erhalten wir den Gradienten bezüglich \mathbf{R} als einen Tensor \mathbf{E} . Zum Trainieren des Decoders müssen wir den Gradienten bezüglich \mathbf{K} ermitteln. Dies ergibt sich aus $g(\mathbf{H}, \mathbf{E}, s)$. Zum Trainieren des Encoders müssen wir den Gradienten bezüglich \mathbf{H} ermitteln. Dies ergibt sich aus $c(\mathbf{K}, \mathbf{E}, s)$. Wir können mit c und h durch g differenzieren, aber diese Operationen sind für den Backpropagation-Algorithmus in normalen Netzarchitekturen nicht erforderlich.

Grundsätzlich verwenden wir nicht nur eine lineare Operation für die Transformation von den Eingaben zu den Ausgaben in einer Faltungsschicht.

Normalerweise addieren wir auch einen Verzerrungsterm zu jeder Ausgabe, bevor wir die Nichtlinearität anwenden. Da stellt sich die Frage, wie Parameter Sharing Verzerrungs-übergreifend erfolgt. Für lokal verbundene Schichten erhält jede Einheit ihre eigene Verzerrung, wohingegen in der Kachel-Faltung die Verzerrungen mit demselben Kachelmuster wie die Kernel geteilt werden. In Faltungsschichten gibt es üblicherweise eine Verzerrung pro Ausgabekanal, die über alle Positionen in den einzelnen Faltungskarten geteilt wird. Ist die Eingabe jedoch eine bekannte unveränderliche Größe, kann für jede Position der Ausgabekarte auch eine separate Verzerrung erlernt werden. Separate Verzerrungen können die statistische Effizienz des Modells geringfügig reduzieren, ermöglichen dem Modell aber andererseits, Unterschiede in der Bildstatistik an unterschiedlichen Positionen zu korrigieren. Wenn Sie zum Beispiel implizites Zero Padding verwenden, erhalten die Erkennungseinheiten am Bildrand eine insgesamt geringere Eingabe und benötigen im Gegenzug höhere Verzerrungen.

9.6 Strukturierte Ausgaben

CNNs ermöglichen die Ausgabe eines hochdimensionalen strukturierten Objekts, anstatt nur ein Klassen-Label für eine Klassifizierungsaufgabe oder einen reellen Wert für eine Regressionsaufgabe vorherzusagen. Typischerweise ist dieses Objekt lediglich ein Tensor einer normalen Faltungsschicht. Für das Modell kann es zum Beispiel einen Tensor \mathbf{S} geben, bei dem $S_{i,j,k}$ die Wahrscheinlichkeit ausdrückt, dass das Pixel (j, k) der Eingabe in das Netz zur Klasse i gehört. Das erlaubt dem Modell, jedes Pixel in einem Bild mit einem dazugehörigen Label zu kennzeichnen und exakte Masken zu zeichnen, die den Umrissen einzelner Objekte folgen.

Ein häufig auftretendes Problem ist, dass die Ausgabeebene kleiner als die Eingabeebene sein kann (vgl. Abbildung 9.13). In den Architekturvarianten, die üblicherweise zur Klassifizierung eines einzelnen Bildobjekts eingesetzt werden, entsteht die größte Reduzierung in den räumlichen Dimensionen des Netzes aus dem Einsatz von Pooling-Schichten mit großer Schrittweite. Um eine Ausgabekarte zu erzeugen, die ähnlich groß wie die Eingabe ist, kann man vollständig auf das Pooling verzichten (*Jain et al.*, 2007). Eine andere Vorgehensweise ist es, Label einfach mit geringerer Auflösung zurückzugeben (*Pinheiro und Collobert*, 2014, 2015). Und schließlich könnte man auch einen Pooling-Operator mit einfacher Schrittweite nutzen.

Ein Verfahren, Bilder Pixel für Pixel mit einem Label zu kennzeichnen, besteht darin, zunächst eine Schätzung der Bild-Label vorzunehmen und

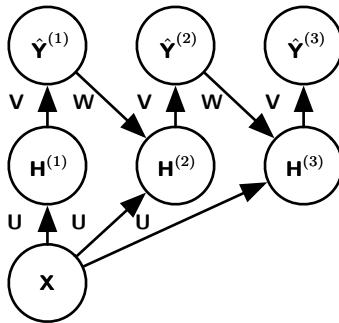


Abbildung 9.17: Ein Beispiel für ein RNN mit Faltung, bei dem Pixel mit einem Label gekennzeichnet werden. Die Eingabe ist ein Bildtensor \mathbf{X} , dessen Achsen den Bildzeilen, Bildspalten und Kanälen (Rot, Grün, Blau) entsprechen. Ausgegeben werden soll ein Tensor mit Labeln \hat{Y} , mit einer Wahrscheinlichkeitsverteilung über die Label der einzelnen Pixel. Die Achsen dieses Tensors entsprechen den Bildzeilen, Bildspalten und den unterschiedlichen Klassen. Statt \hat{Y} in nur einem Schritt auszugeben, verfeinert das RNN seinen Schätzwert \hat{Y} iterativ; dazu wird ein früherer Schätzwert für \hat{Y} als Eingabe zum Erzeugen des neuen Schätzwerts verwendet. Für jeden aktualisierten Schätzwert werden dieselben Parameter verwendet – der Schätzwert kann beliebig oft verfeinert werden. Der Tensor der Faltungskerne \mathbf{U} wird in jedem Schritt zum Berechnen der verdeckten Repräsentation anhand des Eingabebildes genutzt. Der Kerneltensor \mathbf{V} dient zum Erzeugen eines Schätzwerts für die Label anhand der verdeckten Werte. In allen Schritten (bis auf den ersten) werden die Kernel \mathbf{W} über \hat{Y} gefaltet, um die Eingabe für die verdeckte Schicht zu erzeugen. Im ersten Zeitschritt wird dieser Term durch Null ersetzt. Da in jedem Schritt dieselben Parameter zum Einsatz kommen, handelt es sich hier um ein Beispiel für ein RNN (siehe Beschreibung in Kapitel 10).

diese erste Schätzung mithilfe der Interaktionen zwischen benachbarten Pixeln zu verfeinern. Die mehrfache Wiederholung dieses Verfeinerungsschritts entspricht der Verwendung derselben Faltungen in jeder Phase, wodurch die Gewichte für die letzten Schichten des tiefen Netzes gemeinsam verwendet werden (*Jain et al.*, 2007). So wird die Abfolge der Berechnungen, die von den aufeinanderfolgenden Faltungsschichten mit über die Schichtgrenzen hinweg geteilten Gewichten durchgeführt werden, zu einer Sonderform eines RNNs (*Pinheiro und Collobert*, 2014, 2015). Abbildung 9.17 zeigt die Architektur eines solchen RNNs.

Sobald eine Vorhersage für jedes Pixel erfolgt ist, können diese Vorhersagen mit diversen Verfahren verarbeitet werden, um eine Aufteilung des Bildes in Bereiche vorzunehmen (*Briggman et al.*, 2009; *Turaga et al.*, 2010; *Farabet et al.*, 2013). Die Grundidee geht davon aus, dass große Gruppen zusammenhängender Pixel dazu tendieren, demselben Label zugeordnet

zu werden. Graphische Modelle können die probabilistischen Beziehungen zwischen benachbarten Pixeln beschreiben. Alternativ kann das CNN für eine maximale Approximation des Trainingsziels des graphischen Modells trainiert werden (*Ning et al., 2005; Thompson et al., 2014*).

9.7 Datentypen

Die in einem CNN genutzten Daten enthalten normalerweise mehrere Kanäle. Jeder dieser Kanäle steht für die Beobachtung einer anderen Größe an einem Punkt in Raum oder Zeit. Tabelle 9.1 enthält Beispiele für Datentypen unterschiedlicher Dimensionalität und Kanalanzahl.

Ein Beispiel für CNNs in Verbindung mit Videos findet sich in *Chen et al. (2010)*.

	Single Channel	Multichannel
1-D	Audio-Wellenform: Die Faltungsachse entspricht der Zeit. Wir diskretisieren die Zeit und messen die Amplitude der Wellenform einmal pro Zeitschritt.	Drahtgitteranimationsdaten (auch Skelettanimation genannt): Animationen von Computergerenderten 3-D-Charakteren durch Verändern der Haltung eines »Skeletts« im Laufe der Zeit. Die Haltung des Charakters wird zu jedem Zeitpunkt anhand der Winkel der einzelnen Gelenke im Skelett (Drahtgitter) des Charakters beschrieben. Jeder Kanal in den Daten, die wir an das Modell mit Faltung übergeben, steht für den Winkel über einer Achse eines Gelenks.

- 2-D Audiodaten, die zuvor einer Fourier-Transformation unterzogen wurden:
Wir können die Audio-Wellenform in einen 2-D-Tensor umwandeln, der unterschiedliche Zeilen für die unterschiedlichen Frequenzen und unterschiedliche Spalten für die unterschiedlichen Zeitpunkte aufweist. Durch Faltung in der Zeit wird das Modell äquivariant gegenüber zeitlichen Verschiebungen. Mittels Faltung über der Frequenzachse wird das Modell äquivariant gegenüber der Frequenz, sodass eine Melodie, die in einer anderen Oktave wiedergegeben wird, dieselbe Repräsentation – allerdings in einer anderen Höhe – in der Ausgabe des Netzes bewirkt.
- Farbbilddaten:
Ein Kanal enthält die roten, ein weiterer die grünen und einer die blauen Pixel. Der Faltungskern bewegt sich sowohl auf der horizontalen als auch der vertikalen Bildachse, was zu einer Äquivarianz gegenüber Verschiebungen in beiden Richtungen führt.
- 3-D Volumetrische Daten:
Eine häufige Quelle solcher Daten sind bildgebende Verfahren aus der Medizin, zum Beispiel CT-Aufnahmen.
- Farbvideodaten:
Eine Achse bildet die Zeit ab, eine die Höhe des Videobildes und eine die Breite des Videobildes.

Tabelle 9.1: Beispiele für unterschiedliche Datenformate, die mit CNNs genutzt werden können

Bisher haben wir nur den Fall behandelt, in dem jedes Beispiel in den Trainings- und Testdaten dieselben räumlichen Abmessungen aufweist. Allerdings besteht ein Vorteil von CNNs gerade darin, dass sie auch Eingaben mit variabler räumlicher Ausdehnung verarbeiten können. Diese Eingaben lassen sich nicht als klassische neuronale Netze auf Basis von Matrizenmultiplikationen darstellen. Das ist ein triftiger Grund für den Einsatz von CNNs auch dort, wo Berechnungsaufwand und Überanpassung keine nennenswerten Probleme darstellen.

Betrachten wir zum Beispiel eine Bildersammlung, in der alle Bilder unterschiedlich hoch und breit sind. Wie sollte man solche Eingaben mit einer Gewichtungsmatrix unveränderlicher Größe modellieren? Für die Faltung ist das kein Problem: Der Kernel wird ganz einfach abhängig von der Größe der Eingabe mehrfach angewandt und die Ausgabe der Faltungsoperation wird entsprechend skaliert. Die Faltung kann als Matrizenmultiplikation betrachtet werden; derselbe Faltungskern führt zu einer anderen Größe der doppelt zyklischen Blockmatrix (engl. *doubly block circulant matrix*) für jede Eingabegröße. Manchmal dürfen sowohl Aus- als auch Eingabe des Netzes eine variable Größe aufweisen, zum Beispiel bei der Zuordnung eines Klassen-Labels zu jedem Pixel der Eingabe. In diesem Fall sind keine weiteren Anpassungen erforderlich. In anderen Fällen muss das Netz eine ganz bestimmte Ausgabegröße aufweisen, zum Beispiel bei der Zuordnung eines einzelnen Klassen-Labels zum gesamten Bild. In diesem Fall sind noch Anpassungsschritte erforderlich, beispielsweise in Form einer Pooling-Schicht, deren Pooling-Bereiche proportional zur Eingabegröße skaliert werden, so dass eine feste Größe gepoolter Ausgaben beibehalten wird. Abbildung 9.11 zeigt einige Beispiele für dieses Verfahren.

Beachten Sie, dass der Einsatz der Faltung zur Verarbeitung unterschiedlich großer Eingaben nur dann sinnvoll ist, wenn die Größe der Eingaben aufgrund unterschiedlicher Beobachtungsmengen derselben Art schwankt – unterschiedliche Längen zeitlicher Aufzeichnungen, unterschiedliche Breiten räumlicher Beobachtungen usw. Die Faltung ist nicht sinnvoll, wenn die Eingabe unterschiedlich groß ist, weil sie optional unterschiedliche Arten von Beobachtungen enthalten kann. Wenn wir zum Beispiel Hochschulanmeldungen bearbeiten und unsere Merkmale sowohl Noten als auch Punktzahlen für standardisierte Prüfungen enthalten, aber nicht alle Bewerber an den standardisierten Prüfungen teilgenommen haben, ist es nicht sinnvoll, dieselben Merkmalsgewichte für Noten und Punktzahlen zu falten.

9.8 Effiziente Faltungsalgorithmen

Moderne Anwendungen für CNNs beinhalten häufig Netze mit über einer Million Einheiten. Leistungsstarke Implementierungen unter Zuhilfenahme parallelisierter Berechnungsressourcen, die in Abschnitt 12.1 behandelt werden, sind hier eine Grundvoraussetzung. In vielen Fällen ist es aber auch möglich, die Faltung durch Wahl eines geeigneten Faltungsalgorithmus zu beschleunigen.

Die Faltung entspricht der Umwandlung von Eingabe und Kernel in den Frequenzbereich anhand einer Fourier-Transformation, einer punktweisen Multiplikation der beiden Signale und der Rückkonvertierung in den Zeitbereich anhand einer inversen Fourier-Transformation. Für manche Problemgrößen gelingt dies schneller als mit der naiven Implementierung der diskreten Faltung.

Wenn ein d -dimensionaler Kernel als äußeres Produkt von d Vektoren dargestellt werden kann (ein Vektor pro Dimension), wird er als **separierbar** (engl. *separable*) bezeichnet. Ist ein Kernel separierbar, dann ist die naive Faltung ineffizient. Der Vorgang entspricht der Komposition von d eindimensionalen Faltungen mit jedem dieser Vektoren. Der Kompositionsansatz ist deutlich schneller als die Durchführung einer d -dimensionalen Faltung mit dem zugehörigen äußeren Produkt. Der Kernel benötigt auch weniger Parameter zur Darstellung als Vektoren. Ist der Kernel in jeder Dimension w Elemente breit, benötigt die naive mehrdimensionale Faltung $O(w^d)$ für Laufzeit und Speicherplatz für die Parameter, wohingegen die separierbare Faltung $O(w \times d)$ Laufzeit und Speicherplatz für die Parameter benötigt. Natürlich lässt sich nicht jede Faltung auf diese Weise darstellen.

Die Entwicklung schnellerer Möglichkeiten für die Faltung oder approximative Faltung ohne Beeinträchtigung der Korrektklassifikationsrate des Modells ist Thema aktueller Forschungen. Selbst Verfahren, die nur die Effizienz der Forward-Propagation verbessern, sind nützlich, da im kommerziellen Umfeld mehr Ressourcen auf die Umsetzung als das Training eines Netzes angewandt wird.

9.9 Zufällige oder unüberwachte Merkmale

Generell ist das Erlernen der Merkmale der aufwendigste Teil beim Trainieren eines CNNs. Die Ausgabeschicht ist meist recht einfach umzusetzen, da hier nur eine geringe Anzahl der Merkmale eingegeben wird, nachdem die Daten mehrere Pooling-Schichten durchlaufen haben. Bei einem überwachten Training mit Gradientenabstiegsverfahren muss für jeden Gradientenschritt ein vollständiger Lauf der Forward-Propagation und Backpropagation durch das gesamte Netz erfolgen. Eine Möglichkeit zum Reduzieren des Aufwands beim Trainieren von CNNs besteht darin, Merkmale einzusetzen, die nicht auf überwachte Weise trainiert werden.

Es gibt grundsätzlich drei Verfahren zum Erzeugen von Faltungskernen ohne überwachtes Training. Eins davon besteht ganz einfach in einer zufälligen Initialisierung. Ein weiteres ist eine Konstruktion von Hand, indem

beispielsweise jeder Kernel für die Erkennung von Kanten einer bestimmten Ausrichtung oder Größe angepasst wird. Und das letzte Verfahren lernt die Kernel mit einem unüberwachten Kriterium. Zum Beispiel nutzen *Coates et al.* (2011) *k*-Means-Clustering für kleine Bildbereiche und verwenden anschließend jeden erlernten Zentroid als Faltungskern. In Teil III beschreiben wir viele weitere Ansätze für unüberwachtes Lernen. Das Erlernen von Merkmalen mit einem unüberwachten Kriterium ermöglicht es, diese getrennt von der Klassifizierungsschicht ganz oben in der Architektur zu bestimmen. Anschließend lassen sich die Merkmale der gesamten Trainingsdatenmenge einmalig extrahieren, sodass praktisch eine neue Trainingsdatenmenge für die letzte Schicht entsteht. Das Erlernen der letzten Schicht ist dann üblicherweise ein konvexes Optimierungsproblem, sofern die letzte Schicht zum Beispiel eine logistische Regression oder eine SVM ist.

Zufällige Filter (engl. *random filters*) funktionieren oft erstaunlich gut in CNNs (*Jarrett et al.*, 2009; *Saxe et al.*, 2011; *Pinto et al.*, 2011; *Cox und Pinto*, 2011). *Saxe et al.* (2011) haben gezeigt, dass Schichten, in denen Pooling auf Faltung folgt, auf natürliche Weise frequenzselektiv und verschiebungsinvariant werden, wenn ihnen zufällige Gewichte zugeordnet werden. Sie argumentieren, dass dies eine wenig aufwendige Möglichkeit zur Auswahl der Architektur eines CNNs sei: Zuerst wird die Leistung mehrerer Netzarchitekturen mit Faltung durch Trainieren der jeweils letzten Schicht beurteilt, dann wird die beste dieser Architekturen für ein Training der gesamten Architektur mit einem aufwendigeren Ansatz ausgewählt.

Ein Zwischenstopp ist das Erlernen der Merkmale anhand von Verfahren, für die keine vollständige Forward- oder Backpropagation bei jedem Gradientenschritt erforderlich ist. Wie beim mehrschichtigen Perzeptron nutzen wir ein schichtweises Pretraining mit Greedy-Algorithmen, um die erste Schicht isoliert zu trainieren. Dann extrahieren wir einmalig sämtliche Merkmale aus der ersten Schicht. Anschließend wird die zweite Schicht isoliert anhand dieser Merkmale trainiert usw. In Kapitel 8 haben wir gezeigt, wie das überwachte Pretraining mit Greedy-Algorithmen funktioniert. In Teil III erweitern wir dieses Konzept zu einem schichtweisen Pretraining mit Greedy-Algorithmen anhand eines unüberwachten Kriteriums in jeder Schicht. Das kanonische Beispiel für ein schichtweises Pretraining mit Greedy-Algorithmen eines Modells mit Faltung ist das Deep-Belief-Netz (*Lee et al.*, 2009). CNNs bieten uns häufig die Chance, bei den Verfahren für das Pretraining einen Schritt weiter zu gehen, als dies beim mehrschichtigen Perzeptron möglich wäre. Statt in jedem Schritt eine vollständige Faltungsschicht zu trainieren, können wir ein Modell für einen kleinen Bereich trainieren, wie es *Coates et al.* (2011) mit *k*-Means tun. Dann können wir die Parameter

aus diesem bereichsbasierten Modell zum Definieren der Kernel einer Faltungsschicht verwenden. So ist es möglich, unüberwachtes Lernen für das Training eines CNNs einzusetzen, *ohne jemals während des Trainingsverfahrens die Faltung nutzen zu müssen*. Mit diesem Ansatz lassen sich sehr große Modelle trainieren, wobei nur während des Durchführens der Inferenz ein hoher Rechenaufwand entsteht (*Ranzato et al.*, 2007b; *Jarrett et al.*, 2009; *Kavukcuoglu et al.*, 2010; *Coates et al.*, 2013). Dieser Ansatz war zwischen etwa 2007 und 2013 beliebt, als mit Labels gekennzeichnete Datensätze klein und die Rechenleistung eher eingeschränkt war. Heutzutage werden die meisten CNNs vollständig überwacht trainiert – mit vollständiger Forward- und Backpropagation über das gesamte Netz bei jeder Trainingsiteration.

Wie bei anderen Ansätzen für ein unüberwachtes Pretraining ist es schwierig, die Ursache für einige der Vorteile, die sich daraus ergeben, eindeutig zu benennen. Unüberwachtes Pretraining kann eine gewisse Regularisierung gegenüber überwachtem Training bieten oder lediglich das Trainieren viel größerer Architekturen erlauben, da der Berechnungsaufwand für die Lernregel geringer ausfällt.

9.10 Die neurowissenschaftliche Basis für CNNs

CNNs dürften zu den größten Erfolgen der von der Biologie inspirierten Künstlichen Intelligenz gehören. Obschon CNNs auch Anleihen aus vielen anderen Bereichen nutzen, kommen einige der wesentlichen Prinzipien neuronaler Netze aus der Neurowissenschaft.

Die Geschichte der CNNs beginnt lange vor der Entwicklung der relevanten Berechnungsmodelle mit neurowissenschaftlichen Experimenten. Die Neurophysiologen David Hubel und Torsten Wiesel arbeiteten mehrere Jahre gemeinsam an der Bestimmung der grundlegenden Fakten über die Funktionsweise des visuellen Systems bei Säugetieren (*Hubel und Wiesel*, 1959, 1962, 1968). Ihre Arbeiten wurden letztlich mit einem Nobelpreis gewürdigt. Ihre Ergebnisse mit dem größten Einfluss auf die zeitgenössischen Deep-Learning-Modelle beruhten auf der Aufzeichnung der Aktivität der einzelnen Neuronen bei Katzen. Sie haben beobachtet, wie die Neuronen im Katzenhirn auf Bilder reagierten, die auf exakte Positionen auf einem Bildschirm vor der Katze projiziert wurden. Ihre große Entdeckung war, dass Neuronen im frühen visuellen System am stärksten auf ganz spezielle Lichtmuster reagierten, zum Beispiel auf exakt ausgerichtete Balken, aber kaum auf alle anderen Muster.

Ihre Arbeit hat dazu beigetragen, viele Aspekte der Hirnfunktion zu charakterisieren, die den Umfang dieses Buchs sprengen würden. Vom Standpunkt des Deep Learnings können wir uns auf eine vereinfachte Darstellung der Hirnfunktion beschränken.

In dieser vereinfachten Darstellung konzentrieren wir uns auf den V1 genannten Teil des Gehirns, den **primären visuellen Cortex**. Der V1 ist der erste Bereich des Gehirns, der mit einer sehr fortschrittlichen Verarbeitung der visuellen Informationen beginnt. In unserer Darstellung werden Bilder durch das in das Auge fallende Licht gebildet, das die Retina – die lichtempfindliche Netzhaut hinten im Auge – reizt. Die Neuronen in der Retina führen eine einfache Vorverarbeitung des Bildes durch, nehmen aber keine wesentlichen Änderungen an seiner Darstellung vor. Das Bild wird dann durch den Sehnerv und eine *Corpus geniculatum laterale*(CGL, dt. *seitlicher Kniehöcker*) genannte Hirnregion geleitet. Die für uns bedeutende Hauptaufgabe der beiden anatomischen Bereiche besteht in erster Linie darin, das Signal vom Auge zum V1 im hinteren Bereich des Kopfes zu transportieren.

Eine gefaltete Netzsicht soll drei Eigenschaften des V1 erfassen:

1. V1 ist als räumliche Karte aufgebaut. Die eigentliche Struktur ist zweidimensional und spiegelt so den Aufbau des Bildes auf der Retina wider. Zum Beispiel beeinflusst auf die untere Hälfte der Retina treffendes Licht nur die entsprechende Hälfte von V1. CNNs erfassen diese Eigenschaft, indem ihre Merkmale in Form von zweidimensionalen Karten definiert werden.
2. V1 enthält viele **einfache Zellen**. Die Aktivität einer einfachen Zelle lässt sich bis zu einem gewissen Grad als lineare Funktion des Bildes in einem kleinen, räumlich lokalisierten rezeptiven Feld charakterisieren. Die Erkennungseinheiten eines CNNs sind so konstruiert, dass diese Eigenschaften einfacher Zellen emuliert werden.
3. V1 enthält darüber hinaus viele **komplexe Zellen**. Diese Zellen reagieren auf Merkmale, die den von einfachen Zellen erkannten ähneln, aber dabei sind komplexe Zellen invariant gegenüber kleinen Verschiebungen in der Position des Merkmals. Dadurch sind die Pooling-Einheiten von CNNs inspiriert. Komplexe Zellen sind außerdem invariant gegenüber einigen Änderungen in der Beleuchtung, die sich nicht einfach durch Pooling über räumliche Positionen erfassen lassen. Diese Invarianzen haben als Inspiration für einige der kanalübergreifenden

den Pooling-Verfahren in CNNs gedient, bspw. für Maxout-Einheiten (*Goodfellow et al.*, 2013a).

Obschon wir den V1 nahezu vollständig kennen, wird allgemein geglaubt, dass dieselben Prinzipien für andere Bereiche des visuellen Systems gelten. In unserer vereinfachten Darstellung des visuellen Systems wird das grundlegende Verfahren aus Erkennung und darauffolgendem Pooling wiederholt angewandt, während wir tiefer ins Gehirn vordringen. Während wir so mehrere anatomische Hirnschichten hinter uns lassen, gelangen wir irgendwann zu Zellen, die auf ein bestimmtes Konzept reagieren und invariant gegenüber vielen Transformationen der Eingabe sind. Diese Zellen werden auch als »Großmutterneuronen« bezeichnet. Dahinter steckt die Idee, dass jemand über ein Neuron verfügen könnte, das beim Anblick der eigenen Großmutter feuert – unabhängig davon, ob sie links oder rechts im Bild zu finden ist, ob es sich um eine Nahaufnahme des Gesichts oder eine Totale der gesamten Großmutter handelt, ob sie in strahlendem Sonnenschein oder im Schatten steht usw.

Es wurde bewiesen, dass diese Großmutterzellen im menschlichen Gehirn tatsächlich existieren, und zwar in einer Region namens *medialer Schläfenlappen* (*Quiroga et al.*, 2005). Forscher haben untersucht, ob einzelne Neuronen auf Fotos berühmter Persönlichkeiten reagieren. Dabei stießen sie auf das sogenannte »Halle-Berry-Neuron«, das durch das Konzept »Halle Berry« aktiviert wird. Dieses Neuron feuert, wenn jemand ein Foto oder eine Zeichnung von Halle Berry sieht oder einen Text mit den Wörtern »Halle Berry« liest. Das hat natürlich nichts mit Halle Berry selbst zu tun, denn es gibt andere Neuronen, die auf die Präsenz von Bill Clinton, Jennifer Aniston und so weiter reagieren.

Diese Neuronen im medialen Schläfenlappen sind etwas allgemeiner als moderne CNNs, die beim Lesen eines Namens oder einer Bezeichnung nicht automatisch auf die Identifikation einer Person oder eines Objekts generalisieren. Das ähnlichste Analogon zur letzten Merkmalsschicht eines CNNs ist eine Hirnregion namens *inferotemporaler Cortex* (ITC). Beim Betrachten eines Objekts fließen Informationen von der Retina über den CGL zum V1 und von dort weiter zum V2, V4 und schließlich zum ITC. Das geschieht innerhalb der ersten 100 ms nach dem ersten Erfassen eines Objekts. Kann sich eine Person das Objekt längere Zeit ansehen, beginnen die Informationen in Gegenrichtung zu fließen, während das Gehirn Top-down-Feedback einsetzt, um die Aktivierungen in den niedrigeren Hirnregionen zu aktualisieren. Wenn wir den Blick der Person jedoch unterbrechen und nur die Feuerrate beobachten, die sich aus den ersten 100 ms der primären

Feedforward-Aktivierung ergibt, ähnelt der ITC einem CNN. CNNs können die ITC-Feuerraten vorhersagen und ähnlich wie (zeitlich eingeschränkte) Menschen bei der Objekterkennung abschneiden (*DiCarlo*, 2013).

Natürlich gibt es viele Unterschiede zwischen CNNs und dem Sehapparat bei Säugetieren. Einige dieser Unterschiede sind in der Computational Neuroscience wohlbekannt, gehen aber über den Rahmen dieses Buchs hinaus. Einige andere dieser Unterschiede sind noch nicht bekannt, da viele grundlegende Fragen über die Funktionsweise des Sehapparats bei Säugetieren noch nicht beantwortet sind. Hier eine kurze Aufstellung:

- Das menschliche Auge arbeitet größtenteils mit einer sehr niedrigen Auflösung. Das gilt nicht für einen winzigen Bereich, die **Fovea**. Die Fovea erfasst lediglich einen Bereich von der Größe eines in Armlänge gehaltenen Daumennagels. Wir haben zwar das Gefühl, den ganzen betrachteten Bereich in hoher Auflösung zu sehen, aber dabei handelt es sich nur um eine Illusion, die vom Unterbewusstsein erzeugt wird, das die kurzen Blicke auf viele kleine Bereiche aneinander legt. Die meisten CNNs arbeiten tatsächlich mit großen Fotografien in voller Auflösung als Eingabe. Das menschliche Gehirn macht mehrere **Sakkaden** genannte Augenbewegungen, mit denen die visuell charakteristischen oder für eine Aufgabe relevanten Teile einer Szene erblickt werden. Die Einbindung ähnlicher Aufmerksamkeitsmechanismen in Deep-Learning-Modelle ist ein aktuelles Forschungsgebiet. Im Deep-Learning-Kontext haben sich Aufmerksamkeitsmechanismen insbesondere in der Verarbeitung natürlicher Sprache (siehe Abschnitt 12.4.5.1) als erfolgreich erwiesen. Mehrere optische Modelle mit Fovea-Mechanismen wurden entwickelt, aber bisher hat sich noch kein dominanter Ansatz herausgebildet (*Larochelle und Hinton*, 2010; *Denil et al.*, 2012).
- Das menschliche visuelle System weist Verknüpfungen zu vielen anderen Sinnen auf, darunter der Hörsinn, und zu Faktoren wie unserer Stimmung und unseren Gedanken. CNNs arbeiten bisher rein visuell.
- Das menschliche visuelle System ist für viel mehr als lediglich die Erkennung von Objekten verantwortlich. Es kann ganze Szenen begreifen, darunter auch solche mit vielen Objekten und Beziehungen zwischen diesen Objekten. Gleichzeitig verarbeitet es die umfangreichen geometrischen 3-D-Informationen, die unsere Körper für die Interaktion mit der Welt um uns herum benötigen. CNNs wurden auf viele dieser Probleme angesetzt, aber diese Anwendungen stecken noch in den Kinderschuhen.

- Selbst einfache Hirnregionen wie V1 hängen stark vom Feedback übergeordneter Ebenen ab. Feedback wurde für Modelle neuronaler Netze eingehend untersucht, aber es haben sich hierdurch bisher keine zwingenden Verbesserungen gezeigt.
- Obwohl die Feedforward-ITC-Feuerraten viele derselben Informationen erfassen, die auch von Merkmalen von CNNs erfasst werden, ist noch nicht klar, wie ähnlich die Zwischenrechnungen sind. Das Gehirn verwendet vermutlich ganz andere Aktivierungs- und Pooling-Funktionen. Die Aktivierung eines einzelnen Neurons lässt sich mit einer einzelnen Linear-Filter-Reaktion vermutlich nicht gut abbilden. Ein jüngeres Modell von V1 umfasst mehrere quadratische Filter für jedes Neuron (Rust *et al.*, 2005). Tatsächlich könnte unsere vereinfachende Aufteilung in »einfache Zellen« und »komplexe Zellen« eine Unterscheidung darstellen, die es vielleicht gar nicht gibt; einfache und komplexe Zellen könnten dieselbe Art von Zelle sein und sich nur hinsichtlich der »Parameter« unterscheiden, die ein Kontinuum von Verhaltensweisen ermöglichen, das wir als »einfach« oder »komplex« ansehen.

Interessant ist auch, dass die Neurowissenschaft uns relativ wenige Informationen über das *Trainieren* von CNNs gegeben hat. Modellstrukturen mit Parameter Sharing über mehrere räumliche Positionen gehen auf frühe konnektionistische Modelle des Sehens zurück (Marr und Poggio, 1976), aber diese Modelle haben den modernen Backpropagation-Algorithmus und das Gradientenabstiegsverfahren nicht genutzt. Zum Beispiel kennt das Neocognitron (Fukushima, 1980) die meisten Designelemente moderner CNNs, verlässt sich jedoch auf einen schichtweise unüberwachten Clustering-Algorithmus.

Lang und Hinton (1988) haben die Backpropagation erstmals zum Trainieren **zeitverzögerter neuronaler Netze** (engl. *time-delay neural networks*, TDNN) genutzt. Unter Verwendung der aktuellen Terminologie sind TDNNs eindimensionale CNNs, die auf Zeitreihen angewandt werden. Die Backpropagation im Rahmen dieser Modelle wurde nicht durch irgendwelche neurowissenschaftlichen Beobachtungen inspiriert und wird von einigen Akteuren als biologisch unstimmig erklärt. Nach dem Erfolg des Trainings von TDNNs mittels Backpropagation entwickelten LeCun *et al.* (1989) das moderne CNN durch Verwendung desselben Trainingsalgorithmus für die 2-D-Faltung von Bildern.

Bisher haben wir gezeigt, wie die einfachen Zellen grob linear und selektiv auf bestimmte Merkmale reagieren, während komplexe Zellen ein eher

nichtlineares Verhalten an den Tag legen und invariant gegenüber einigen Transformationen der Merkmale dieser einfachen Zellen werden. Außerdem haben wir gesehen, dass Stapel aus Schichten, die zwischen Selektivität und Invarianz wechseln, Großmutterzellen für spezifische Phänomene hervorbringen können. Wir haben aber noch nicht ausgeführt, was genau diese einzelnen Zellen erkennen. In einem tiefen, nichtlinearen Netz ist ein Verständnis der Funktionen einzelner Zellen schwer zu erlangen. Einfache Zellen in der ersten Schicht lassen sich einfacher untersuchen, da ihre Reaktion einer linearen Funktion unterliegt. In einem künstlichen neuronalen Netz können wir einfach ein Bild des Faltungskerns anzeigen, um zu ermitteln, worauf der jeweilige Kanal einer Faltungsschicht reagiert. In einem biologischen neuronalen Netz haben wir keine Möglichkeit, auf die Gewichte selbst zuzugreifen. Stattdessen bringen wir eine Elektrode in dem Neuron an, präsentieren der Retina des Tieres mehrere Beispielbilder mit weißem Rauschen und zeichnen auf, wie jedes dieser Beispiele zu einer Aktivierung des Neurons führt. Anschließend können wir ein lineares Modell auf diese Reaktionen legen, um so eine Approximation der Gewichte des Neurons zu erhalten. Dieser Ansatz wird als **Reverse Correlation** bezeichnet (*Ringach und Shapley, 2004*).

Die Reverse Correlation zeigt uns, dass die meisten V1-Zellen Gewichte aufweisen, die mittels **Gaborfunktionen** beschrieben werden. Die Gaborfunktion beschreibt die Gewichte als 2-D-Punkt im Bild. Sie können sich ein Bild als eine Funktion $I(x, y)$ der 2-D-Koordinaten vorstellen. Ebenso können Sie sich vorstellen, dass eine einfache Zelle dem Bild an einer Reihe von Positionen Stichproben entnimmt, die durch eine Menge von x -Koordinaten \mathbb{X} und eine Menge von y -Koordinaten \mathbb{Y} definiert sind, und dann Gewichte anwendet, die ebenfalls eine Funktion $w(x, y)$ der Position sind. So betrachtet ergibt sich die Reaktion einer einfachen Zelle auf ein Bild aus

$$s(I) = \sum_{x \in \mathbb{X}} \sum_{y \in \mathbb{Y}} w(x, y) I(x, y). \quad (9.15)$$

Insbesondere nimmt $w(x, y)$ die Form einer Gaborfunktion an:

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right) \cos(f x' + \phi), \quad (9.16)$$

wobei

$$x' = (x - x_0) \cos(\tau) + (y - y_0) \sin(\tau) \quad (9.17)$$

ist und

$$y' = -(x - x_0) \sin(\tau) + (y - y_0) \cos(\tau). \quad (9.18)$$

Hier sind α , β_x , β_y , f , ϕ , x_0 , y_0 und τ Parameter zur Steuerung der Eigenschaften der Gaborfunktion. Abbildung 9.18 zeigt einige Beispiele von Gaborfunktionen mit unterschiedlichen Einstellungen dieser Parameter.

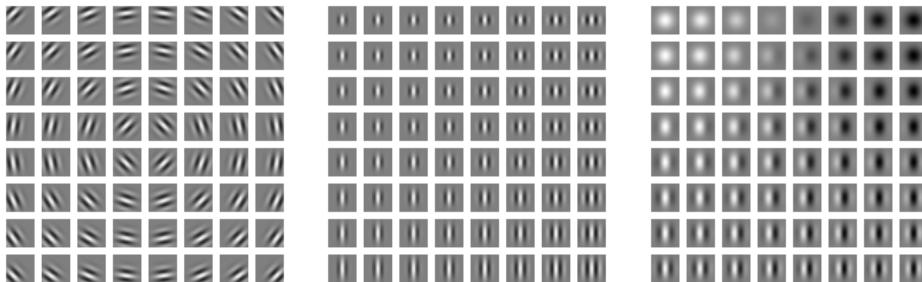


Abbildung 9.18: Gaborfunktionen mit unterschiedlichen Parametereinstellungen. Weiß steht für ein stark positives Gewicht, Schwarz für ein stark negatives Gewicht. Das Hintergrundgrau steht für das Gewicht Null. (*Links*) Gaborfunktionen mit unterschiedlichen Parameterwerten zur Steuerung des Koordinatensystems: x_0 , y_0 und τ . Jeder Gaborfunktion in diesem Raster wird ein Wert x_0 und y_0 zugewiesen, der proportional zur Position im Raster ist; τ wird so gewählt, dass jeder Gaborfilter empfindlich auf die Richtung reagiert, die von der Rastermitte nach außen strahlt. Für die anderen beiden Plots sind x_0 , y_0 und τ stets Null. (*Mitte*) Gaborfunktionen mit unterschiedlichen Maßstabsparametern β_x und β_y der Normalverteilung. Gaborfunktionen sind im Gitter von links nach rechts nach zunehmender Breite sortiert (abnehmendes β_x) und von oben nach unten nach zunehmender Höhe (abnehmendes β_y). Für die beiden anderen Plots sind die β -Werte stets das Anderthalbfache der Bildbreite. (*Rechts*) Gaborfunktionen mit unterschiedlichen Sinusparametern f und ϕ . Von oben nach unten nimmt f ab, von links nach rechts nimmt ϕ zu. Für die anderen beiden Plots ist ϕ stets 0 und f stets das Fünffache der Bildbreite.

Die Parameter x_0 , y_0 und τ spannen ein Koordinatensystem auf. Wir verschieben und drehen x und y mit dem Ergebnis x' und y' . Gerade die einfache Zelle reagiert auf Bildmerkmale, die im Punkt zentriert sind (x_0, y_0) ; außerdem reagiert sie auf Änderungen der Helligkeit, wenn wir uns entlang einer um τ Radian von der Horizontalen gedrehten Linie bewegen.

Als eine Funktion von x' und y' reagiert die Funktion w dann auf Helligkeitsveränderungen entlang der Achse x' . Es sind zwei wichtige Faktoren im Spiel: einerseits eine Gauß-Funktion, andererseits eine Kosinusfunktion.

Der Faktor $\alpha \exp(-\beta_x x'^2 - \beta_y y'^2)$ kann als eine Art Tor angesehen werden, das sicherstellt, dass die einfache Zelle nur auf Werte reagiert, wenn sowohl x' als auch y' nahezu Null sind, die Position also in der Nähe des Zentrums des rezeptiven Felds liegt. Der Skalierungsfaktor α passt die

Gesamtstärke der Reaktion der einfachen Zelle an, während β_x und β_y regeln, wie schnell das rezeptive Feld abfällt.

Der Kosinusfaktor $\cos(fx' + \phi)$ regelt, wie die einfache Zelle auf die sich ändernde Helligkeit entlang der Achse x' reagiert. Der Parameter f steuert die Kosinusfrequenz, ϕ steuert den Phasenversatz.

Insgesamt zeigt diese Darstellung der einfachen Zellen, dass eine einfache Zelle auf eine bestimmte räumliche Helligkeitsfrequenz in einer bestimmten Richtung an einer bestimmten Position reagiert. Einfache Zellen werden am stärksten angeregt, wenn die Welle der Helligkeit im Bild dieselbe Phase wie die Gewichte aufweist. Das geschieht, wenn das Bild bei positiven Gewichten hell ist und dunkel bei negativen Gewichten. Einfache Zellen werden am stärksten gehemmt, wenn die Welle der Helligkeit gegenüber den Gewichten vollständig phasenverschoben ist – das Bild also bei positiven Gewichten dunkel und bei negativen Gewichten hell ist.

Unsere vereinfachte Darstellung einer komplexen Zelle besagt, dass diese die L^2 -Norm des 2-D-Vektors berechnet, der Reaktionen von zwei einfachen Zellen enthält: $c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$. Ein wichtiger Sonderfall tritt ein, wenn s_1 bis auf ϕ dieselben Parameter wie s_0 ausweist und ϕ so eingestellt ist, dass s_1 um einen Viertelzyklus phasenverschoben zu s_0 ist. In diesem Fall bilden s_0 und s_1 ein **Quadraturpaar**. Eine derart definierte komplexe Zelle reagiert, wenn das neu gewichtete normalverteilte Bild $I(x, y) \exp(-\beta_{xx'}^2 - \beta_{yy'}^2)$ eine sinusförmige Welle mit hoher Amplitude aufweist, deren Frequenz f in Richtung τ nahe (x_0, y_0) liegt, *und zwar ungeachtet des Phasenversatzes dieser Welle*. Mit anderen Worten: Die komplexe Zelle ist invariant gegenüber kleinen Verschiebungen des Bildes in der Richtung τ oder einem Negativ des Bildes (Austausch von Schwarz gegen Weiß und umgekehrt).

Einige der bemerkenswertesten Übereinstimmungen zwischen der Neurawissenschaft und dem Machine Learning ergeben sich aus dem visuellen Vergleich von Machine-Learning-Modellen erlernerter Merkmale mit denen des V1. Olshausen und Field (1996) haben gezeigt, dass ein einfacher unüberwachter Lernalgorithmus, nämlich Sparse Coding, Merkmale mit rezeptiven Feldern erlernt, die denen einfacher Zellen ähneln. Seitdem haben wir festgestellt, dass eine extrem hohe Zahl statistischer Lernalgorithmen Merkmale mit Gabor-ähnlichen Funktionen erlernt, wenn sie auf natürliche Bilder angesetzt werden. Darunter fallen die meisten Deep-Learning-Algorithmen, die diese Merkmale in ihrer ersten Schicht erlernen. Abbildung 9.19 zeigt einige Beispiele. Da so viele unterschiedliche Lernalgorithmen in der Lage sind, Kantendetektoren zu erlernen, ist es schwierig, nur aufgrund der erlernten

Merkmale zu schließen, dass ein bestimmter Lernalgorithmus das »richtige« Modell des Gehirns darstellt (obwohl es gewiss ein schlechtes Zeichen ist, wenn ein Algorithmus *keinerlei* Kantendetektor erlernt, wenn er mit natürlichen Bildern zu tun hat). Diese Merkmale sind ein wichtiger Teil der statistischen Struktur natürlicher Bilder und können auf verschiedene Weise mittels statistischer Modellierung wiederhergestellt werden. *Hyvärinen et al.* (2009) behandeln die Statistik hinsichtlich natürlicher Bilder.

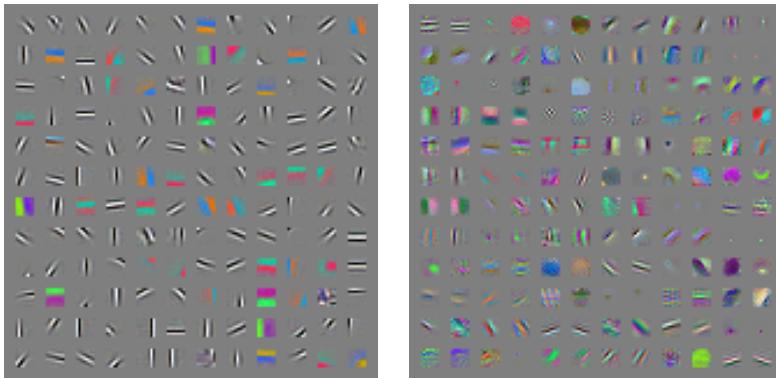


Abbildung 9.19: Viele Machine-Learning-Algorithmen erlernen Merkmale, die Kanten oder Kanten einer besonderen Farbe erkennen, wenn man sie auf natürliche Bilder anwendet. Diese Merkmalsdetektoren erinnern an die Gaborfunktionen, die im primären visuellen Cortex nachgewiesen wurden. (*Links*) Gewichte werden anhand eines unüberwachten Lernalgorithmus (Spike and Slab Sparse Coding) für kleine Bildbereiche erlernt. (*Rechts*) Von der ersten Schicht oder einem vollständig überwachten Maxout-CNN erlernte Faltungskerne. Benachbarte Filterpaare steuern dieselbe Maxout-Einheit.

9.11 CNNs und die Geschichte des Deep Learnings

CNNs haben eine wichtige Rolle in der Geschichte des Deep Learnings gespielt. Sie sind ein bedeutendes Beispiel für die erfolgreiche Anwendung von Erkenntnissen, die bei der Untersuchung des Gehirns für Machine-Learning-Anwendungen gewonnen wurden. Außerdem gehörten sie zu den ersten gut funktionierenden tiefen Modellen, lange bevor Modelle beliebiger Tiefe als realisierbar angesehen wurden. CNNs gehörten auch zu den ersten neuronalen Netzen, mit denen wichtige kommerzielle Anwendungen gelöst werden können – und auch heute sind sie Wegbereiter für das Deep Learning im kommerziellen Umfeld. So hat in den 1990er-Jahren die Forschungs-

gruppe für neuronale Netze bei AT&T ein CNN zum Lesen von Schecks entwickelt (*LeCun et al.*, 1998b). Gegen Ende der 1990er wurde dieses von NEC entwickelte System bei mehr als 10 Prozent aller Scheckeinreichungen in den USA eingesetzt. Später setzte Microsoft mehrere OCR- und Handschrifterkennungssysteme auf Basis von CNNs ein (*Simard et al.*, 2003). Kapitel 12 enthält Einzelheiten zu derartigen Anwendungen und moderneren Einsatzgebieten für CNNs. *LeCun et al.* (2010) behandeln die Geschichte der CNNs bis ins Jahr 2010 detailliert.

CNNs schafften es auch in vielen Wettbewerben auf den ersten Platz. Das derzeit starke kommerzielle Interesse am Deep Learning nahm seinen Anfang, als *Krizhevsky et al.* (2012) den ImageNet-Objekterkennungswettbewerb gewannen, aber CNNs wurden bereits Jahre zuvor in anderen Wettbewerben rund um Machine Learning und Computer Vision eingesetzt, allerdings mit weniger weitreichenden Folgen.

CNNs gehörten außerdem zu den ersten funktionierenden tiefen Netzen, die mittels Backpropagation trainiert wurden. Es ist nicht ganz klar, warum CNNs dieser Erfolg beschieden war, wo doch allgemeine Backpropagation-Netze als Fehlschlag betrachtet wurden. Vielleicht lag es einfach daran, dass CNNs rechnerisch effizienter als vollständig verbundene Netze waren und es daher ohne große Probleme möglich war, mehr Experimente damit auszuführen und ihre Implementierung und Hyperparameter abzustimmen. Größere Netze scheinen auch einfacher zu trainieren zu sein. Mit moderner Hardware kommen große, vollständig verbundene Netze scheinbar gut mit vielen Aufgaben zurecht, auch wenn Datensätze und Aktivierungsfunktionen verwendet werden, die bereits verfügbar und beliebt waren, als man vollständig verbundene Netze noch für untauglich hielt. Vielleicht waren die Hinderungsgründe für den Erfolg neuronaler Netze auch rein psychologischer Natur (Entwickler glaubten nicht daran, dass neuronale Netze funktionieren würden, also unternahm man auch keine nennenswerten Anstrengungen in dieser Hinsicht). Was auch immer der Grund war: Zum Glück hat sich vor Jahrzehnten gezeigt, dass CNNs gute Leistungen erbringen. Sie haben auf manche Weise den Weg für das restliche Feld des Deep Learnings bereitet und für die verbreitete Akzeptanz neuronaler Netze gesorgt.

CNNs bieten eine Möglichkeit, neuronale Netze auf Daten mit rasterförmiger Topologie zuzuschneiden und solche Modelle sehr stark zu skalieren. Dieser Ansatz hat sich für zweidimensionale Bildtopologien als der erfolgreichste erwiesen. Zur Verarbeitung eindimensionaler sequenzieller Daten wenden wir uns einer weiteren leistungsstarken Variante neuronaler Netze zu, den RNNs.

10

Sequenzmodellierung: RNNs und rekursive Netze

Rekurrente neuronale Netze, auch rückgekoppelte neuronale Netze oder kurz RNNs genannt (*Rumelhart et al.*, 1986a), sind eine Familie neuronaler Netze zur Verarbeitung sequenzieller Daten. So wie ein CNN ein neuronales Netz ist, das auf die Verarbeitung eines Werterasters \mathbf{X} (beispielsweise Bilder) spezialisiert ist, handelt es sich bei einem RNN um ein neuronales Netz, das auf die Verarbeitung einer Sequenz von Werten $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}$ spezialisiert ist. Wie CNNs Bilder mit großer Breite und Höhe problemlos skalieren und einige CNNs Bilder mit wechselnden Größen verarbeiten können, sind RNNs in der Lage, sehr viel längere Sequenzen als Netze ohne Sequenzspezialisierung zu verarbeiten. Die meisten RNNs können außerdem Sequenzen variabler Länge verarbeiten.

Für den Schritt von mehrschichtigen zu rekurrenten Netzen greifen wir auf eine frühe Idee aus dem Bereich des Machine Learnings und der statistischen Modelle der 1980er-Jahre zurück: das Parameter Sharing über unterschiedliche Teile des Modells hinweg. Parameter Sharing ermöglicht es, das Modell auf Beispiele mit anderen Formen (hier: unterschiedliche Länge) zu erweitern und anzuwenden und sie zu generalisieren. Gäbe es separate Parameter für jeden Wert im Zeitindex, könnten wir weder über Sequenzlängen generalisieren, denen wir im Training nicht begegnet sind, noch statistische Aussagekraft über verschiedene Sequenzlängen und verschiedene Positionen in der Zeit hinweg weitergeben. Eine solches Sharing ist ganz besonders wichtig, wenn eine bestimmte Information an mehreren Stellen der Sequenz auftauchen kann. Ein solches Sharing ist ganz besonders wichtig, wenn eine bestimmte Information an mehreren Stellen der Sequenz auftauchen kann.

Nehmen wir als Beispiel die beiden folgenden Sätze: »Ich besuchte Nepal im Jahr 2009«, und »Im Jahr 2009 besuchte ich Nepal«. Wenn wir ein Machine-Learning-Modell anweisen, die Sätze einzulesen und das Jahr zu nennen, in dem der Erzähler Nepal besuchte, soll es in beiden Fällen die Antwort 2009 liefern – ungeachtet dessen, ob diese Information als sechstes oder drittes Wort im Satz zu finden ist. Angenommen, wir haben ein Feedforward-Netz trainiert, das Sätze mit einer bestimmten Länge verarbeitet. Ein klassisches vollständig verbundenes Feedforward-Netz würde für jedes Eingabemerkmal andere Parameter nutzen, d. h., es müsste sämtliche Regeln einer Sprache für jede Stelle im Satz separat erlernen. Im Gegensatz dazu verwendet ein RNN dieselben Gewichtungen über mehrere Zeitschritte hinweg.

Ein ähnliches Konzept ist der Einsatz der Faltung über eine zeitliche 1-D-Abfolge. Dieser Ansatz mit Faltung bildet die Basis für zeitverzögerte neuronale Netze (*Lang und Hinton*, 1988; *Waibel et al.*, 1989; *Lang et al.*, 1990). Die Faltungsoperation ermöglicht Parameter Sharing im Netz über zeitliche Grenzen hinweg, ist aber flach. Das Ergebnis der Faltung ist eine Sequenz, in der jedes Element der Ausgabe eine Funktion einer kleinen Anzahl benachbarter Elemente der Eingabe ist. Die Idee von Parameter Sharing manifestiert sich in der Anwendung desselben Faltungskerns (engl. *convolution kernel*) in jedem Zeitschritt. RNNs teilen die Parameter auf andere Weise. Jedes Element der Ausgabe ist eine Funktion der vorhergehenden Elemente der Ausgabe. Jedes Element der Ausgabe wird anhand derselben Update-Regel erzeugt, die auf die vorhergehenden Ausgaben angewandt wird. Dieser rekurrente Ansatz führt zu Parameter Sharing im gesamten Verlauf eines sehr tiefen Berechnungsgraphen.

Aus Gründen der Einfachheit sagen wir bei RNNs, dass sie eine Sequenz verarbeiten, die Vektoren $\mathbf{x}^{(t)}$ mit einem Zeitschritt-Index t von 1 bis τ enthält. In der Praxis werden meist Mini-Batches solcher Sequenzen mit RNNs verwendet, wobei für jedes Element des Mini-Batches eine andere Sequenzlänge τ genutzt wird. Für eine übersichtlichere Notation verzichten wir auf die Angabe der Indizes für Mini-Batches. Außerdem muss der Zeitschritt-Index nicht unbedingt das Verstreichen von Zeit in der Realität darstellen. Manchmal bezeichnet er lediglich die Position in der Sequenz. RNNs können auch in zwei Dimensionen über räumliche Daten angewandt werden, zum Beispiel für Bilder. Selbst wenn sie für Daten mit Zeitbezug genutzt werden, kann es Verbindungen geben, die in der Zeit zurückgehen, sofern die gesamte Sequenz beobachtet wird, bevor sie an das Netz übergeben wird.

Dieses Kapitel greift das Konzept eines Berechnungsgraphen auf und ergänzt es um Zyklen. Diese Zyklen stellen den Einfluss des aktuellen

Werts einer Variable auf ihren eigenen Wert in einem späteren Zeitschritt dar. Derartige Berechnungsgraphen erlauben die Definition von RNNs. Anschließend beschreiben wir viele unterschiedliche Möglichkeiten zum Konstruieren, Trainieren und Verwenden von RNNs.

Wenn Sie noch tiefer in RNNs einsteigen möchten, empfehlen wir das Fachbuch von *Graves* (2012).

10.1 Auffalten von Berechnungsgraphen

Ein Berechnungsgraph ermöglicht die Formalisierung der Struktur einer Reihe von Berechnungen, beispielsweise bei der Zuordnung von Eingaben und Parametern zu Ausgaben und Verlust. Eine allgemeine Einführung finden Sie in Abschnitt 6.5.1. In diesem Abschnitt erklären wir das Konzept der **Auffaltung** (engl. *unfolding*) einer rekursiven oder rekurrenten Berechnung zu einem Berechnungsgraphen, der eine repetitive Struktur aufweist, die im Normalfall einer Kette von Ereignissen entspricht. Durch Auffalten des Graphen werden die Parameter über eine tiefe Netzstruktur geteilt.

Als Beispiel dient die klassische Form eines dynamischen Systems:

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \boldsymbol{\theta}), \quad (10.1)$$

mit $\mathbf{s}^{(t)}$ als Zustand des Systems.

Gleichung 10.1 ist rekurrent, da die Definition von \mathbf{s} zum Zeitpunkt t auf dieselbe Definition zum Zeitpunkt $t - 1$ zurückgreift.

Für eine endliche Anzahl von Zeitschritten τ kann der Graph aufgefaltet werden; hierzu wird die Definition $\tau - 1$ Male angewandt. Wenn wir Gleichung 10.1 zum Beispiel für $\tau = 3$ Zeitschritte auffalten, erhalten wir

$$\mathbf{s}^{(3)} = f(\mathbf{s}^{(2)}; \boldsymbol{\theta}) \quad (10.2)$$

$$= f(f(\mathbf{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta}). \quad (10.3)$$

Dieses Auffalten der Gleichung durch wiederholtes Anwenden der Definition hat uns zu einem Ausdruck geführt, der keine Rekurrenz enthält. Ein solcher Ausdruck kann nun als klassischer gerichteter azyklischer Berechnungsgraph dargestellt werden. Der aufgefaltete Berechnungsgraph für Gleichung 10.1 und Gleichung 10.3 ist in Abbildung 10.1 dargestellt.

Als weiteres Beispiel betrachten wir ein dynamisches System mit einem Eingang für ein externes Signal $\mathbf{x}^{(t)}$,

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}), \quad (10.4)$$

wobei wir sehen, dass der Zustand nun Angaben über die gesamte bisherige Sequenz enthält.

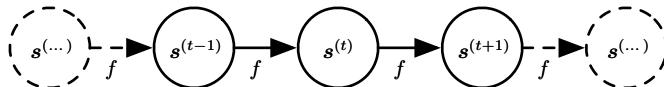


Abbildung 10.1: Das durch Gleichung 10.1 beschriebene klassische dynamische System, hier dargestellt als aufgefalteter Berechnungsgraph. Jeder Knoten steht für den Zustand zu einem Zeitpunkt t , und die Funktion f ordnet den Zustand zum Zeitpunkt t dem Zustand zum Zeitpunkt $t + 1$ zu. Dieselben Parameter (derselbe Wert von θ zur Parametrisierung von f) werden für alle Zeitschritte verwendet.

RNNs können auf unterschiedliche Weise aufgebaut werden. Wie praktisch jede Funktion als neuronales Feedforward-Netz betrachtet werden kann, lässt sich auch jede Funktion mit Rekurrenz als RNN betrachten.

Viele RNNs verwenden Gleichung 10.5 oder eine ähnliche Gleichung zum Definieren der Werte ihrer verdeckten Einheiten. Um anzuzeigen, dass der Zustand den verdeckten Einheiten des Netzes entspricht, notieren wir Gleichung 10.4 unter Verwendung der Variable \mathbf{h} für den Zustand neu:

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta), \quad (10.5)$$

dargestellt in Abbildung 10.2; typische RNNs fügen zusätzliche architektonische Merkmale hinzu, beispielsweise Ausgabeschichten, die Informationen aus dem Zustand \mathbf{h} auslesen, um Vorhersagen zu machen.

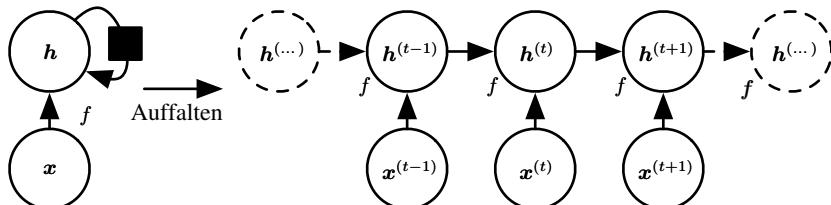


Abbildung 10.2: Ein RNN ohne Ausgaben. Dieses RNN verarbeitet lediglich Informationen der Eingabe \mathbf{x} durch Einbinden derselben in den Zustand \mathbf{h} , der in mehreren Zeitschritten nach vorn durchgereicht wird. (*Links*) Ablaufplan. Das schwarze Quadrat steht für eine Verzögerung, die einen Zeitschritt lang ist. (*Rechts*) Dasselbe Netz als aufgefalteter Berechnungsgraph, in dem nun jedem Knoten eine bestimmte Zeitinstanz zugewiesen ist.

Wenn das RNN für eine Aufgabe trainiert wird, bei der die Zukunft anhand der Vergangenheit vorhergesagt werden muss, lernt es üblicherweise, $\mathbf{h}^{(t)}$ als eine Art verlustbehaftete Zusammenfassung der aufgabenrelevanten

Aspekte der vergangenen Sequenz von Eingaben bis zum Zeitpunkt t zu verwenden. Diese Zusammenfassung ist im Allgemeinen zwangsläufig verlustbehaftet, da sie eine Sequenz $(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)})$ beliebiger Länge einem Vektor $\mathbf{h}^{(t)}$ fester Länge zuordnet. Je nach Trainingskriterium kann diese Zusammenfassung selektiv einige Aspekte der vergangenen Sequenz präziser behalten als andere Aspekte. Wenn das RNN zum Beispiel für die statistische Modellierung der Sprache eingesetzt wird, bei der üblicherweise das nächste Wort anhand der bisherigen Wörter vorhergesagt wird, ist es vielleicht gar nicht erforderlich, alle Informationen in der Eingabesequenz bis zum Zeitpunkt t zu speichern – es reicht aus, nur die Menge an Informationen zu speichern, die für die Vorhersage des restlichen Satzinhalts benötigt wird. Die anspruchsvollste Situation liegt vor, wenn wir fordern, dass $\mathbf{h}^{(t)}$ umfassend genug ist, um damit die Eingabesequenz näherungsweise wiederherstellen zu können; ein Beispiel dafür sind Autoencoder-Frameworks (Kapitel 14).

Gleichung 10.5 kann auf zwei Arten dargestellt werden. Eine Möglichkeit zur Darstellung des RNNs besteht in einem Diagramm, das für jede Komponente, die in einer physischen Implementierung des Modells (z. B. ein biologisches neuronales Netz) existieren könnte, einen Knoten enthält. In dieser Ansicht definiert das Netz einen Kreislauf, der in Echtzeit arbeitet und physische Bestandteile enthält, deren aktueller Zustand ihren künftigen Zustand beeinflussen kann (vgl. Ablaufplan auf der linken Seite in Abbildung 10.2). In diesem Kapitel verwenden wir ein schwarzes Quadrat, um in einem Ablaufplan anzusehen, dass eine Interaktion um einen Zeitschritt verzögert stattfindet (vom Zustand zum Zeitpunkt t zum Zustand zum Zeitpunkt $t + 1$). Die andere Darstellung eines RNNs ist ein aufgefalteter Berechnungsgraph, in dem jede Komponente durch viele verschiedene Variablen dargestellt wird, und zwar eine Variable pro Zeitschritt. Diese stehen für den Zustand der Komponente zum jeweiligen Zeitpunkt. Jede Variable für jeden Zeitschritt wird als separater Knoten des Berechnungsgraphen dargestellt (vgl. Abbildung 10.2). Als Auffaltung bezeichnen wir die Operation, bei der ein Kreislauf (linke Seite der Abbildung) einem Berechnungsgraphen mit wiederholten Teilstücken (rechte Seite) zugeordnet wird. Die Größe des aufgefalteten Graphen richtet sich nach der Sequenzlänge.

Wir können die aufgefaltete Rekurrenz nach t Schritten als Funktion $g^{(t)}$ angeben:

$$\mathbf{h}^{(t)} = g^{(t)}(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}) \quad (10.6)$$

$$= f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta). \quad (10.7)$$

Die Funktion $g^{(t)}$ nimmt die gesamte bisherige Sequenz $(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)})$ als Eingabe und erzeugt den aktuellen Zustand; mit der aufgefalteten rekurrenten Struktur können wir $g^{(t)}$ in wiederholter Anwendung einer Funktion f faktorisieren. Die Auffaltung bietet somit zwei wesentliche Vorteile:

1. Ungeachtet der Sequenzlänge weist das gelernte Modell stets dieselbe Eingabegröße auf, da es als Übergang von einem Zustand in einen anderen spezifiziert wird, nicht als eine Zustandshistorie wechselnder Länge.
2. Es ist möglich, in jedem Zeitschritt *dieselbe* Übergangsfunktion f mit denselben Parametern zu verwenden.

Diese beiden Faktoren machen es möglich, ein einzelnes Modell f zu erlernen, das für sämtliche Zeitschritte und alle Sequenzlängen funktioniert; es muss also nicht für alle möglichen Zeitschritte jeweils ein eigenes Modell $g^{(t)}$ erlernt werden. Das Erlernen eines einzelnen gemeinsamen Modells erlaubt die Generalisierung für Sequenzlängen, die in der Trainingsdatenmenge nicht enthalten waren. Außerdem kann das Modell mit deutlich weniger Trainingsbeispielen geschätzt werden, als ohne Parameter Sharing erforderlich wären.

Sowohl der rekurrente Graph als auch der aufgefaltete Graph haben ihre Einsatzbereiche. Der rekurrente Graph ist kurz und prägnant. Der aufgefaltete Graph bietet eine explizite Beschreibung der durchzuführenden Berechnungen. Der aufgefaltete Graph hilft uns auch, die Idee des Informationsflusses vorwärts durch die Zeit (Berechnung von Ausgaben und Verlusten) bzw. rückwärts durch die Zeit (Gradientenberechnung) darzustellen, indem der Pfad, entlang dem die Informationen fließen, explizit gezeigt wird.

10.2 RNNs

Mit dem Auffalten des Graphen und dem Parameter Sharing aus Abschnitt 10.1 können wir nun eine Vielzahl von RNNs konstruieren.

Beispiele für wichtige Design Patterns für RNNs sind zum Beispiel:

- RNNs, die für jeden Zeitschritt eine Ausgabe erzeugen und rekurrente Verbindungen zwischen verdeckten Einheiten aufweisen (vgl. Abbildung 10.3)

- RNNs, die für jeden Zeitschritt eine Ausgabe erzeugen und nur rekurrente Verbindungen zwischen der Ausgabe eines Zeitschritts und den verdeckten Einheiten des nächsten Zeitschritts aufweisen (vgl. Abbildung 10.4)
- RNNs, die rekurrente Verbindungen zwischen den verdeckten Einheiten aufweisen, die eine komplette Sequenz einlesen und anschließend eine einzelne Ausgabe erzeugen (vgl. Abbildung 10.5)

Abbildung 10.3 ist ein ziemlich repräsentatives Beispiel, auf das wir im Laufe des Kapitels immer wieder zurückgreifen werden.

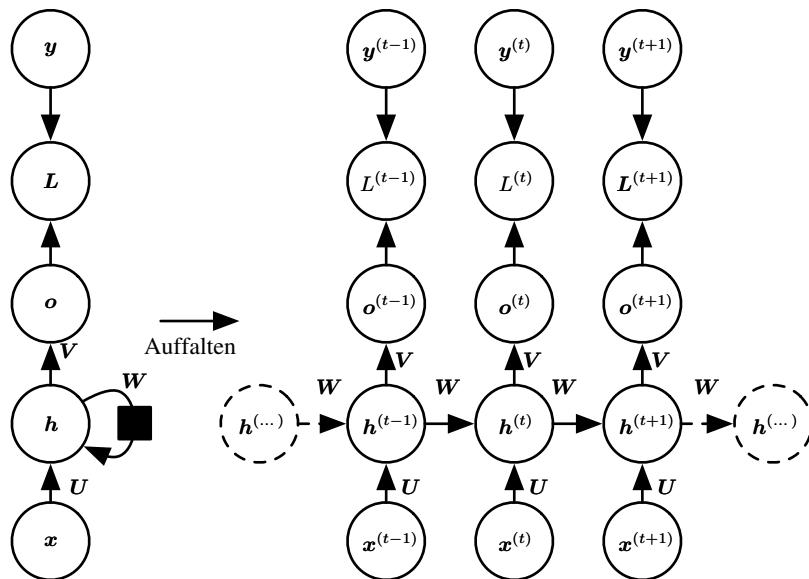


Abbildung 10.3: Der Berechnungsgraph zum Berechnen des Trainingsverlusts eines RNNs, das eine Eingabesequenz mit \mathbf{x} Werten einer entsprechenden Sequenz mit \mathbf{o} -Werten für die Ausgabe zuordnet. Ein Verlust L bestimmt, wie weit jedes \mathbf{o} vom jeweiligen Trainingsziel \mathbf{y} entfernt ist. Bei softmax-Ausgaben nehmen wir an, dass \mathbf{o} die nicht normalisierten Log-Wahrscheinlichkeiten darstellt. Der Verlust L berechnet intern den Ausdruck $\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$ und vergleicht das Ergebnis mit dem Zielwert \mathbf{y} . Im RNN gibt es Eingaben für verdeckte Verbindungen, die durch eine Gewichtungsmatrix \mathbf{U} parametrisiert werden, rekurrente Verdeckt-zu-Verdeckt-Verbindungen, die durch eine Gewichtungsmatrix \mathbf{W} parametrisiert werden, und Verdeckt-zu-Ausgabe-Verbindungen, die durch eine Gewichtungsmatrix \mathbf{V} parametrisiert werden. Gleichung 10.8 definiert die Forward-Propagation in diesem Modell. (*Links*) Das RNN und sein Verlust, dargestellt mit rekurrenten Verbindungen. (*Rechts*) Dasselbe Netz als zeitliche Auffaltung eines Berechnungsgraphen, in dem nun jedem Knoten eine bestimmte Zeitinstanz zugewiesen ist.

Das RNN aus Abbildung 10.3 und Gleichung 10.8 ist insofern universell, als jede Funktion, die von einer Turing-Maschine berechnet werden kann, auch von solch einem RNN unendlicher Größe berechenbar ist. Die Ausgabe kann dem RNN nach einer Anzahl von Zeitschritten entnommen werden, die asymptotisch linear zur Anzahl der Zeitschritte der Turing-Maschine und asymptotisch linear zur Länge der Eingabe ist (*Siegelmann und Sontag*, 1991; *Siegelmann*, 1995; *Siegelmann und Sontag*, 1995; *Hyotyniemi*, 1996). Die von einer Turing-Maschine berechenbaren Funktionen sind diskret; damit beziehen sich die Ergebnisse auf eine exakte Implementierung der Funktion, nicht auf Approximationen. Das RNN nimmt, wenn es als Turing-Maschine verwendet wird, eine binäre Sequenz als Eingabe; seine Ausgaben müssen diskretisiert werden, um eine binäre Ausgabe zu erzeugen. Es ist möglich, alle Funktionen in diesem Kontext mit einem einzelnen spezifischen RNN endlicher Größe zu berechnen (*Siegelmann und Sontag* 1995 nutzen 886 Einheiten). Die »Eingabe« der Turing-Maschine ist eine Spezifikation der zu berechnenden Funktion, sodass das für die Simulation dieser Turing-Maschine genutzte Netz für alle Probleme hinreichend ist. Das für den Beweis eingesetzte theoretische RNN kann einen uneingeschränkten Stapel simulieren, indem dessen Aktivierungen und Gewichte als rationale Zahlen unbegrenzter Genauigkeit dargestellt werden.

Wir entwickeln nun die Gleichungen für die Forward-Propagation im RNN aus Abbildung 10.3. Die Abbildung gibt keine Aktivierungsfunktion für die verdeckten Einheiten vor. Für diesen Fall nehmen wir den Tangens hyperbolicus an. Außerdem gibt die Abbildung nicht an, welche Form die Ausgabe und die Verlustfunktion genau annehmen sollen. Wir gehen von einer diskreten Ausgabe aus, als würde das RNN zum Vorhersagen von Wörtern oder Zeichen benutzt. Eine natürliche Methode zur Darstellung diskreter Variablen besteht darin, anzunehmen, dass die Ausgabe \mathbf{o} nicht normalisierte Log-Wahrscheinlichkeiten für jeden möglichen Wert der diskreten Variable angibt. Wir können nun die softmax-Operation als Nachverarbeitungsschritt anwenden, um einen Vektor $\hat{\mathbf{y}}$ der normalisierten Wahrscheinlichkeiten über die Ausgabe zu bestimmen. Die Forward-Propagation beginnt mit einer Spezifikation des anfänglichen Zustands $\mathbf{h}^{(0)}$. Dann wenden wir in jedem Zeitschritt von $t = 1$ bis $t = \tau$ die folgenden Update-Gleichungen an:

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}, \quad (10.8)$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}), \quad (10.9)$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}, \quad (10.10)$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)}), \quad (10.11)$$

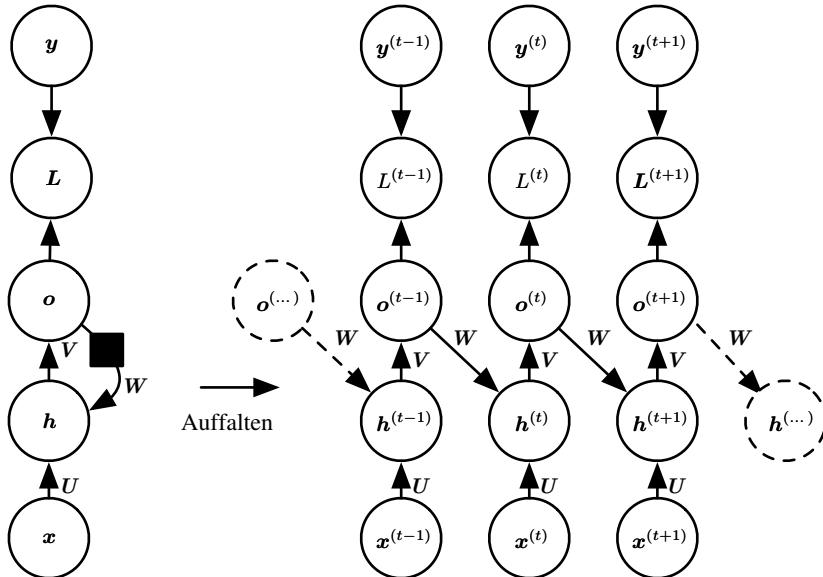


Abbildung 10.4: Ein RNN, dessen einzige Rekurrenz die Feedback-Verbindung von der Ausgabe- zur verdeckten Schicht ist. In jedem Zeitschritt t ist die Eingabe \mathbf{x}_t , die Aktivierungen der verdeckten Schicht sind $\mathbf{h}^{(t)}$, die Ausgaben sind $\mathbf{o}^{(t)}$, die Zielwerte sind $\mathbf{y}^{(t)}$ und der Verlust ist $L^{(t)}$. (Links) Ablaufplan. (Rechts) Aufgefalteter Berechnungsgraph. Ein solches RNN ist weniger leistungsstark (kann nur eine kleinere Menge von Funktionen ausdrücken) als jene aus der durch Abbildung 10.3 dargestellten Familie. Das RNN in Abbildung 10.3 kann beliebige Informationen über die Vergangenheit als Eingabe für die eigene verdeckte Repräsentation \mathbf{h} nutzen und \mathbf{h} in die Zukunft übertragen. Das RNN in dieser Abbildung ist darauf trainiert, einen bestimmten Ausgabewert in \mathbf{o} einzugeben; \mathbf{o} ist die einzige Angabe, die es in die Zukunft übertragen kann. Es gibt keine direkten Verbindungen von \mathbf{h} nach vorn. Das vorhergehende \mathbf{h} ist mit dem aktuellen nur indirekt über die Vorhersagen, die mit ihm erzeugt wurden, verbunden. Sofern \mathbf{o} nicht extrem hochdimensional und umfassend ist, fehlen im Normalfall wichtige Informationen aus der Vergangenheit. Daher ist das RNN aus dieser Abbildung weniger leistungsstark. Allerdings kann es einfacher zu trainieren sein, da jeder Zeitschritt isoliert von den anderen trainiert werden kann, was einen höheren Grad an Parallelisierung während des Trainings erlaubt (siehe Abschnitt 10.2.1).

wobei die Parameter die Verzerrungsvektoren \mathbf{b} und \mathbf{c} samt der Gewichtungsmatrizen \mathbf{U} , \mathbf{V} bzw. \mathbf{W} für Eingabe-zu-Verdeckt-, Verdeckt-zu-Ausgabe- und Verdeckt-zu-Verdeckt-Verbindungen sind. Dieses Beispiel zeigt, wie ein RNN eine Eingabesequenz einer Ausgabesequenz derselben Länge zuordnet. Der Gesamtverlust einer gegebenen Sequenz mit \mathbf{x} Werten gepaart mit einer Sequenz mit \mathbf{y} Werten wäre dann einfach die Summe der Verluste in allen Zeitschritten. Ein Beispiel: Wenn $L^{(t)}$ die negative Log-Likelihood von $y^{(t)}$ auf Basis von $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}$ ist, dann gilt

$$L\left(\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}\}, \{\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}\}\right) \quad (10.12)$$

$$= \sum_t L^{(t)} \quad (10.13)$$

$$= - \sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}\right), \quad (10.14)$$

wobei $p_{\text{model}}\left(y^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}\right)$ sich durch Lesen des Eintrags für $y^{(t)}$ aus dem Ausgabevektor $\hat{\mathbf{y}}^{(t)}$ des Modells ergibt. Den Gradienten dieser Verlustfunktion bezüglich der Parameter zu berechnen, ist eine aufwendige Operation. Sie umfasst eine Forward-Propagation von links nach rechts durch unsere Darstellung des aufgefalteten Graphen aus Abbildung 10.3,

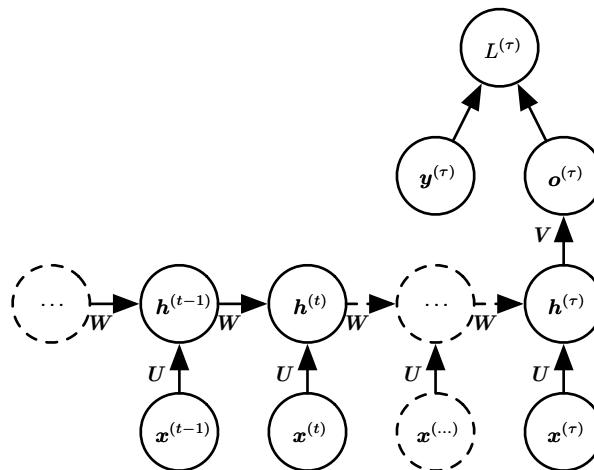


Abbildung 10.5: Zeitliche Auffaltung eines RNNs mit einer einzelnen Ausgabe am Ende der Sequenz. Ein solches Netz kann zum Zusammenfassen einer Sequenz und zum Erzeugen einer Repräsentation mit fester Größe, die als Eingabe für die weitere Verarbeitung dient, eingesetzt werden. Möglicherweise gibt es direkt am Ende einen Zielwert (wie in dieser Darstellung) oder der Gradient der Ausgabe $\mathbf{o}^{(t)}$ lässt sich durch Backpropagation in nachgelagerten Modulen ermitteln.

gefolgt von einer Backpropagation von rechts nach links durch diesen Graphen. Die Laufzeit ist $O(\tau)$. Eine Verkürzung durch Parallelisierung ist nicht möglich, da der Graph der Forward-Propagation in sich sequenziell ist: Jeder Zeitschritt kann erst nach dem vorherigen berechnet werden. Die in der Vorwärtsberechnung ermittelten Zustände müssen gespeichert werden, bis sie während der Rückwärtsberechnung erneut benötigt werden – damit liegt auch der Speicherbedarf bei $O(\tau)$. Der Backpropagation-Algorithmus für den aufgefalteten Graphen mit einem Aufwand von $O(\tau)$ wird **Backpropagation Through Time** (BPTT) genannt und in Abschnitt 10.2.2 genauer behandelt. Das Netz mit Rekurrenz zwischen verdeckten Einheiten ist somit sehr leistungsstark, aber auch sehr aufwendig beim Training. Gibt es eine Alternative?

10.2.1 Teacher Forcing und Netze mit Ausgaberekurrenz

Das Netz, das lediglich rekurrente Verbindungen zwischen der Ausgabe eines Zeitschritts und den verdeckten Einheiten des nächsten Zeitschritts aufweist (siehe Abbildung 10.4), ist deutlich weniger leistungsstark, da ihm die rekurrenten Verdeckt-zu-Verdeckt-Verbindungen fehlen. Es kann zum Beispiel keine universelle Turing-Maschine simulieren. Da diesem Netz die Verdeckt-zu-Verdeckt-Rekurrenz fehlt, müssen die Ausgabeeinheiten sämtliche Informationen über die Vergangenheit erfassen, die es zur Vorhersage der Zukunft benötigt. Da die Ausgabeeinheiten explizit bezüglich der Ziele der Trainingsdatenmenge trainiert werden, ist es unwahrscheinlich, dass die benötigten Informationen über die vergangene Historie der Eingabe erfasst werden – es sei denn, der Anwender kann den vollständigen Zustand des Systems beschreiben und liefert diese Zustandsbeschreibung im Rahmen der Ziele für die Trainingsdatenmenge mit. Durch das Weglassen der Verdeckt-zu-Verdeckt-Rekurrenz ergibt sich der Vorteil, dass für jede Verlustfunktion, die auf einem Vergleich der Vorhersage zum Zeitpunkt t mit dem Trainingsziel zum Zeitpunkt t beruht, alle Zeitschritte entkoppelt sind. Das Training kann somit parallelisiert und der Gradient für jeden Schritt t isoliert berechnet werden. Es ist nicht notwendig, zunächst die Ausgabe des vorherigen Zeitschritts zu berechnen, da die Trainingsdatenmenge den idealen Wert dieser Ausgabe liefert.

Modelle mit rekurrenten Verbindungen von den eigenen Ausgaben zurück ins Modell können durch sogenanntes **Teacher Forcing** trainiert werden. Teacher Forcing ist ein Verfahren, das aus dem Kriterium der Maximum Likelihood entstanden ist; dabei erhält das Modell die bekannte Ausgabe $y^{(t)}$ als Eingabe zum Zeitpunkt $t+1$. Bei der Betrachtung einer Sequenz aus zwei

Zeitschritten wird dies deutlich. Das Kriterium der bedingten Maximum Likelihood ist

$$\log p(\mathbf{y}^{(1)}, \mathbf{y}^{(2)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) \quad (10.15)$$

$$= \log p(\mathbf{y}^{(2)} | \mathbf{y}^{(1)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) + \log p(\mathbf{y}^{(1)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}). \quad (10.16)$$

In diesem Beispiel wird das Modell zum Zeitpunkt $t = 2$ für eine maximale bedingte Wahrscheinlichkeit von $\mathbf{y}^{(2)}$ anhand der bisherigen \mathbf{x} -Sequenz und des vorherigen \mathbf{y} -Werts aus der Trainingsdatenmenge trainiert. Die Maximum Likelihood gibt somit an, dass während des Trainings die Ausgabe des Modells nicht wieder eingespeist wird, sondern den Verbindungen vielmehr die Zielwerte zugeführt werden, die angeben, was die korrekte Ausgabe sein sollte. Das ist in Abbildung 10.6 dargestellt.

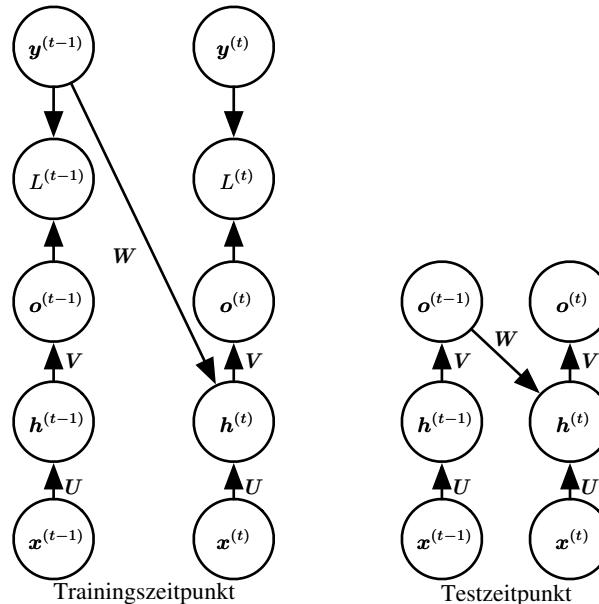


Abbildung 10.6: Darstellung des Teacher Forcings. Teacher Forcing ist eine Trainingsmethode für RNNs, die Verbindungen zwischen der Ausgabe und den verborgenen Zuständen des nächsten Zeitschritts aufweisen. (*Links*) Zum Trainingszeitpunkt führen wir $\mathbf{h}^{(t+1)}$ die *korrekte Ausgabe* $\mathbf{y}^{(t)}$ aus der Trainingsdatenmenge als Eingabe zu. (*Rechts*) Bei der Modellkonzeption ist die wahre Ausgabe meist unbekannt. In diesem Fall approximieren wir die korrekte Ausgabe $\mathbf{y}^{(t)}$ mit der Ausgabe des Modells $\mathbf{o}^{(t)}$ und speisen die Ausgabe wieder in das Modell ein.

Teacher Forcing war ursprünglich eine Möglichkeit zum Vermeiden der Backpropagation Through Time (BPTT) in Modellen ohne Verdeckt-zu-

Verdeckt-Verbindungen. Es kann auch auf Modelle mit Verdeckt-zu-Verdeckt-Verbindungen angewandt werden, sofern diese zwischen der Ausgabe in einem Zeitschritt und den im nächsten Zeitschritt berechneten Werten Verbindungen aufweisen. Sobald die verdeckten Einheiten eine Funktion früherer Zeitschritte werden, muss jedoch der BPTT-Algorithmus verwendet werden. Einige Modelle können daher sowohl mit Teacher Forcing als auch mit BPTT trainiert werden.

Der Nachteil eines strikten Teacher Forcings zeigt sich, wenn das Netz später im sogenannten **Open-Loop-Modus** genutzt werden soll, bei dem Ausgaben des Netzes (oder Stichproben aus der Ausgabeverteilung) wieder als Eingabe zurückgespeist werden. In diesem Fall kann sich die Art der Eingaben, die das Netz während des Trainings sieht, erheblich von den Eingaben zum Testzeitpunkt unterscheiden. Eine Möglichkeit, dieses Problem zu verringern, besteht darin, sowohl Eingaben mit Teacher Forcing als auch freie Eingaben für das Training zu verwenden, zum Beispiel indem anhand der aufgefalteten Ausgabe-zu-Eingabe-Pfade im Vorhinein das korrekte Ziel vorausgesagt wird. So kann das Netz lernen, Eingabebedingungen zu berücksichtigen (z. B. jene, die es selbst im freien Modus generiert), die im Training nicht auftreten, und wie sich ein Zustand rückwärts einem anderen zuordnen lässt, der nach einigen Schritten zum Generieren korrekter Ausgaben führt. Ein weiterer Ansatz (*Bengio et al.*, 2015b) zum Verringern des Abstands zwischen den Eingaben während des Trainings und den Eingaben während des Tests besteht darin, zufällig zwischen erzeugten Werten oder tatsächlichen Datenwerten als Eingabe auszuwählen. Dieser Ansatz nutzt ein Curriculum-Learning-Verfahren, das nach und nach mehr erzeugte Werte als Eingabe nutzt.

10.2.2 Gradientenberechnung in einem RNN

Die Gradientenberechnung in einem RNN ist recht einfach und gelingt mithilfe des generalisierten Backpropagation-Algorithmus aus Abschnitt 6.5.6 für den aufgefalteten Berechnungsgraphen. Es werden keine weiteren speziellen Algorithmen benötigt. Mittels Backpropagation bestimmte Gradienten können dann mit einer beliebigen allgemeinen Methode auf Gradientenbasis zum Trainieren eines RNNs verwendet werden.

Um ein Gespür für das Verhalten des BPTT-Algorithmus zu entwickeln, geben wir ein Beispiel für die Gradientenberechnung mittels BPTT für die vorgenannten RNN-Gleichungen (Gleichung 10.8 und Gleichung 10.12). Die Knoten unseres Berechnungsgraphen enthalten die Parameter \mathbf{U} , \mathbf{V} , \mathbf{W} , \mathbf{b} und \mathbf{c} sowie die Sequenz der mit t indizierten Knoten für $\mathbf{x}^{(t)}$, $\mathbf{h}^{(t)}$, $\mathbf{o}^{(t)}$

und $L^{(t)}$. Für jeden Knoten \mathbf{N} müssen wir den Gradienten $\nabla_{\mathbf{N}} L$ rekursiv bestimmen, und zwar auf Basis des für die im Graphen darauf folgenden Knoten berechneten Gradienten. Wir beginnen die Rekursion mit den Knoten direkt vor dem letzten Verlust:

$$\frac{\partial L}{\partial L^{(t)}} = 1. \quad (10.17)$$

In dieser Ableitung gehen wir davon aus, dass die Ausgaben $\mathbf{o}^{(t)}$ als Argument für die softmax-Funktion verwendet werden, um den Vektor $\hat{\mathbf{y}}$ der Wahrscheinlichkeiten über die Ausgabe zu ermitteln. Auf Grundlage der bisherigen Eingabe nehmen wir auch an, dass der Verlust die negative Log-Likelihood des tatsächlichen Ziels $y^{(t)}$ ist. Der Gradient $\nabla_{\mathbf{o}^{(t)}} L$ auf den Ausgaben im Zeitschritt t , für alle i, t , ist

$$(\nabla_{\mathbf{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}. \quad (10.18)$$

Wir arbeiten uns vom Ende der Sequenz nach vorne. Im letzten Zeitschritt τ weist $\mathbf{h}^{(\tau)}$ nur $\mathbf{o}^{(\tau)}$ als Nachfolger aus; der Gradient ist entsprechend einfach:

$$\nabla_{\mathbf{h}^{(\tau)}} L = \mathbf{V}^\top \nabla_{\mathbf{o}^{(\tau)}} L. \quad (10.19)$$

Wir können dann rückwärts durch die Zeit iterieren, um Gradienten entlang der Zeitachse einer Backpropagation zu unterziehen, und zwar von $t = \tau - 1$ bis hinab zu $t = 1$. Beachten Sie, dass $\mathbf{h}^{(t)}$ (für $t < \tau$) als Nachfolger sowohl $\mathbf{o}^{(t)}$ als auch $\mathbf{h}^{(t+1)}$ aufweist. Der zugehörige Gradient ergibt sich somit aus

$$\nabla_{\mathbf{h}^{(t)}} L = \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\nabla_{\mathbf{h}^{(t+1)}} L) + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\nabla_{\mathbf{o}^{(t)}} L) \quad (10.20)$$

$$= \mathbf{W}^\top (\nabla_{\mathbf{h}^{(t+1)}} L) \operatorname{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right) + \mathbf{V}^\top (\nabla_{\mathbf{o}^{(t)}} L), \quad (10.21)$$

wobei $\operatorname{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right)$ die Diagonalmatrix mit den Elementen $1 - (h_i^{(t+1)})^2$ angibt. Dies ist die Jacobi-Matrix des Tangens hyperbolicus für die verdeckte Einheit i zum Zeitpunkt $t + 1$.

Sobald die Gradienten der inneren Knoten des Berechnungsgraphen bestimmt sind, können wir die Gradienten auf den Parameterknoten ermitteln. Da die Parameter über viele Zeitschritte hinweg genutzt werden, müssen wir beim Markieren von Rechenoperationen, die diese Variablen nutzen, sorgfältig vorgehen. Die Gleichungen, die wir implementieren möchten, verwenden

die `bprop`-Methode aus Abschnitt 6.5.6, die den Beitrag einer einzelnen Kante zum Berechnungsgraphen des Gradienten berechnet. Der Operator $\nabla_{\mathbf{W}} f$ aus der Analysis berücksichtigt jedoch den Beitrag von \mathbf{W} zum Wert von f auf Basis *aller* Kanten im Berechnungsgraphen. Um diese Unklarheit zu beseitigen, führen wir Scheinvariablen $\mathbf{W}^{(t)}$ ein, die als Kopien von \mathbf{W} definiert sind, jedoch so, dass jedes $\mathbf{W}^{(t)}$ nur im Zeitschritt t verwendet wird. Anschließend können wir $\nabla_{\mathbf{W}^{(t)}}$ als Bezeichnung der Gewichte im Zeitschritt t für den Gradienten verwenden.

Mithilfe dieser Notation ergibt sich der Gradient für die restlichen Parameter aus

$$\nabla_{\mathbf{c}} L = \sum_t \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{c}} \right)^{\top} \nabla_{\mathbf{o}^{(t)}} L = \sum_t \nabla_{\mathbf{o}^{(t)}} L, \quad (10.22)$$

$$\nabla_{\mathbf{b}} L = \sum_t \left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}^{(t)}} \right)^{\top} \nabla_{\mathbf{h}^{(t)}} L = \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) \nabla_{\mathbf{h}^{(t)}} L, \quad (10.23)$$

$$\nabla_{\mathbf{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\mathbf{V}} o_i^{(t)} = \sum_t (\nabla_{\mathbf{o}^{(t)}} L) \mathbf{h}^{(t)\top}, \quad (10.24)$$

$$\begin{aligned} \nabla_{\mathbf{W}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{W}^{(t)}} h_i^{(t)} \\ &= \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) (\nabla_{\mathbf{h}^{(t)}} L) \mathbf{h}^{(t-1)\top}, \end{aligned} \quad (10.25)$$

$$\nabla_{\mathbf{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\mathbf{U}^{(t)}} h_i^{(t)} \quad (10.27)$$

$$= \sum_t \text{diag} \left(1 - (\mathbf{h}^{(t)})^2 \right) (\nabla_{\mathbf{h}^{(t)}} L) \mathbf{x}^{(t)\top}, \quad (10.28)$$

Wir müssen den Gradienten bezüglich $\mathbf{x}^{(t)}$ im Training nicht berechnen, da er keine Vorgängerparameter im Berechnungsgraphen hat, die den Verlust definieren.

10.2.3 RNNs als gerichtete graphische Modelle

Im bisher entwickelten Beispiel-RNN waren die Verluste $L^{(t)}$ Kreuzentropien zwischen den Trainingszielen $\mathbf{y}^{(t)}$ und Ausgaben $\mathbf{o}^{(t)}$. Wie in einem Feedforward-Netz ist es auch in einem RNN prinzipiell möglich, nahezu jeden beliebigen Verlust zu verwenden. Der Verlust muss passend zur Aufgabe gewählt werden. Ebenfalls wie bei einem Feedforward-Netz möchten wir die Ausgabe des RNN normalerweise als Wahrscheinlichkeitsverteilung

interpretieren und verwenden die mit dieser Verteilung verbundene Kreuzentropie, um den Verlust zu definieren. Der mittlere quadratische Fehler ist der Kreuzentropieverlust, der einer Ausgabeverteilung zugeordnet ist, die beispielsweise eine Standardnormalverteilung ist – ebenfalls wie in einem Feedforward-Netz.

Wenn wir ein Trainingsziel mit prädiktiver Log-Likelihood verwenden, zum Beispiel Gleichung 10.12, trainieren wir das RNN darauf, die bedingte Verteilung des nächsten Elements $\mathbf{y}^{(t)}$ der Sequenz anhand der bisherigen Eingaben zu schätzen. Das kann eine Maximierung der Log-Likelihood

$$\log p(\mathbf{y}^{(t)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}) \quad (10.29)$$

bedeuten, oder – sofern das Modell Verbindungen von der Ausgabe in einem Zeitschritt zum nächsten Zeitschritt enthält

$$\log p(\mathbf{y}^{(t)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}, \mathbf{y}^{(1)}, \dots, \mathbf{y}^{(t-1)}). \quad (10.30)$$

Die Zerlegung der multivariaten Verteilung über die Sequenz mit \mathbf{y} Werten als Serie probabilistischer Einzelschritt-Vorhersagen ist eine Möglichkeit zum Erfassen der vollständigen multivariaten Verteilung für die gesamte Sequenz. Wenn wir bisherige \mathbf{y} -Werte nicht als Eingaben verwenden, die die Vorhersage des nächsten Schritts einschränken, enthält das gerichtete graphische Modell keine Kanten von einem vergangenen $\mathbf{y}^{(i)}$ zum aktuellen $\mathbf{y}^{(t)}$. In diesem Fall sind die Ausgaben \mathbf{y} bedingt unabhängig von der Sequenz der \mathbf{x} -Werte. Wenn wir die tatsächlichen \mathbf{y} -Werte (nicht ihre Vorhersagewerte, sondern die tatsächlich beobachteten oder erzeugten Werte) wieder in das Netz einspeisen, enthält das gerichtete graphische Modell Kanten von allen bisherigen $\mathbf{y}^{(i)}$ -Werten zum aktuellen $\mathbf{y}^{(t)}$ -Wert.

Betrachten wir als einfaches Beispiel den Fall, in dem das RNN nur eine Sequenz skalarer Zufallsvariablen $\mathbb{Y} = \{y^{(1)}, \dots, y^{(\tau)}\}$ ohne zusätzliche Eingaben \mathbf{x} modelliert. Die Eingabe im Zeitschritt t ist ganz einfach die Ausgabe von Zeitschritt $t - 1$. Das RNN definiert dann ein gerichtetes graphisches Modell über die y -Variablen. Wir parametrisieren die multivariate Verteilung dieser Beobachtungen anhand der Kettenregel (Gleichung 3.6) für bedingte Wahrscheinlichkeiten:

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} | \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \dots, \mathbf{y}^{(1)}), \quad (10.31)$$

wobei die rechte Seite des Strichs für $t = 1$ natürlich leer ist. Somit ist die negative Log-Likelihood einer Menge von Werten $\{y^{(1)}, \dots, y^{(\tau)}\}$ gemäß einem solchen Modell

$$L = \sum_t L^{(t)}, \quad (10.32)$$

mit

$$L^{(t)} = -\log P(y^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, \dots, y^{(1)}). \quad (10.33)$$

Die Kanten in einem graphischen Modell geben an, welche Variablen direkt von anderen Variablen abhängig sind. Viele graphische Modelle versuchen, statistische Effizienz und Berechnungseffizienz durch Weglassen von Kanten zu erreichen, die keine starken Interaktionen darstellen. Zum Beispiel ist es übliche Praxis, der Markow-Annahme zu folgen, die besagt, dass das graphische Modell nur Kanten von $\{y^{(t-k)}, \dots, y^{(t-1)}\}$ zu $y^{(t)}$ enthalten sollte, und nicht die Kanten der gesamten Historie. In einigen Fällen sind wir jedoch der Meinung, dass alle bisherigen Eingaben das nächste Element der Sequenz beeinflussen sollten. RNNs sind nützlich, wenn die Verteilung über $y^{(t)}$ vermutlich von einem Wert für $y^{(i)}$ aus der fernen Vergangenheit in einer Art abhängig ist, die sich durch den Effekt von $y^{(i)}$ auf $y^{(t-1)}$ nicht erfassen lässt.

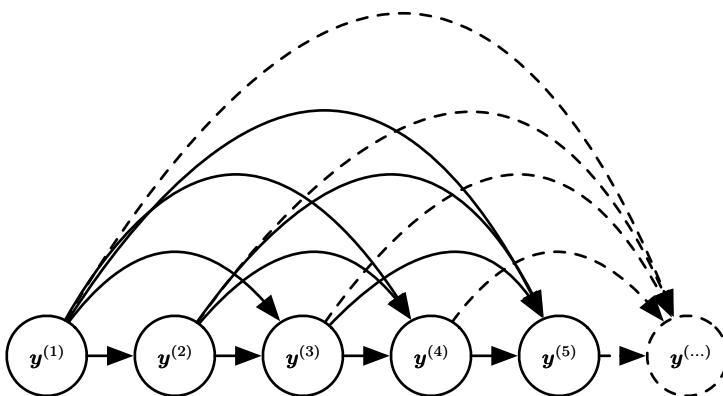


Abbildung 10.7: Vollständig verbundenes graphisches Modell einer Sequenz $y^{(1)}, y^{(2)}, \dots, y^{(t)}, \dots$. Jede vergangene Beobachtung $y^{(i)}$ kann die bedingte Verteilung eines $y^{(t)}$ (für $t > i$) beeinflussen, das sich aus den vorhergehenden Werten ergibt. Die direkte Parametrisierung des graphischen Modells anhand dieses Graphen (vgl. Gleichung 10.6) könnte extrem ineffizient sein, da die Anzahl der Eingaben und Parameter für jedes Element der Sequenz immer weiter wächst. RNNs führen zur selben vollständigen Verbindung bei effizienter Parametrisierung, wie Abbildung 10.8 zeigt.

Eine Möglichkeit zum Interpretieren eines RNNs als graphisches Modell besteht darin, das RNN als Definition eines graphischen Modells zu betrachten, dessen Struktur der vollständige Graph ist und das somit die direkten Abhängigkeiten zwischen beliebigen y-Wertepaaren darstellen kann. Das graphische Modell über die y-Werte mit der vollständigen Graphenstruktur ist in Abbildung 10.7 dargestellt. Die Interpretation des vollständigen Graphen

des RNNs beruht auf dem Ignorieren der verdeckten Einheiten $\mathbf{h}^{(t)}$, indem man sie aus dem Modell entfernt.

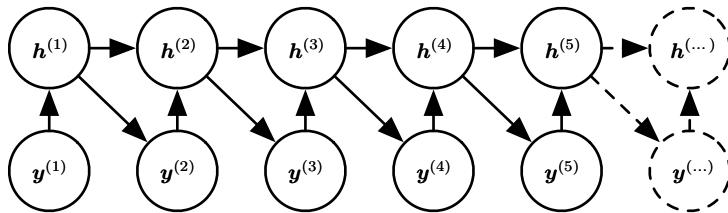


Abbildung 10.8: Das Einführen der Zustandsvariable in das graphische Modell des RNNs – obwohl es sich um eine deterministische Funktion seiner Eingaben handelt – hilft dabei, zu erkennen, wie eine sehr effiziente Parametrisierung erreicht werden kann (als Grundlage dient Gleichung 10.5). Jede Phase in der Sequenz (für $\mathbf{h}^{(t)}$ und $\mathbf{y}^{(t)}$) bezieht dieselbe Struktur ein (dieselbe Anzahl von Eingaben in jedem Knoten) und kann dieselben Parameter mit anderen Phasen teilen.

Es ist interessanter, die Struktur graphischer Modelle von RNNs zu betrachten, die aus der Betrachtung der verdeckten Einheiten $\mathbf{h}^{(t)}$ als Zufallsvariablen entstehen.¹ Das Einschließen der verdeckten Einheiten in das graphische Modell zeigt, dass das RNN eine effiziente Parametrisierung der multivariaten Verteilung über die Beobachtungen erreicht. Angenommen, wir stellen eine beliebige multivariate Verteilung über diskrete Werte als Tabelle dar – ein Array mit jeweils einem anderen Eintrag für jede mögliche Wertzuordnung, wobei der Wert der Einträge sich aus der Wahrscheinlichkeit ergibt, dass diese Zuordnung auftritt. Wenn y k verschiedene Werte annehmen kann, weist die Tabelle $O(k^T)$ Parameter auf. Im Gegensatz dazu beträgt die Anzahl der Parameter im RNN bei Nutzung von Parameter Sharing $O(1)$ als eine Funktion der Sequenzlänge. Die Anzahl der Parameter im RNN kann zum Steuern der Modellkapazität angepasst werden, muss aber nicht mit der Sequenzlänge skaliert werden. Gleichung 10.5 zeigt, dass das RNN langfristige Beziehungen zwischen Variablen auf effiziente Weise parametrisiert, indem dieselbe Funktion f und dieselben Parameter θ in jedem Zeitschritt rekurrent angewandt werden. Abbildung 10.8 zeigt die Interpretation des graphischen Modells. Durch Einbinden der $\mathbf{h}^{(t)}$ -Knoten in das graphische Modell werden Vergangenheit und Zukunft entkoppelt; sie dienen also als Zwischengröße. Eine Variable $y^{(i)}$ in der fernen Vergangenheit kann eine Variable $y^{(t)}$ mithilfe ihrer Auswirkung auf \mathbf{h} beeinflussen. Die Struktur dieses Graphen zeigt, dass das Modell in jedem Zeitschritt

¹ Die bedingte Verteilung über diese Variablen ist auf Grundlage ihrer Eltern deterministisch. Es ist vollkommen legitim, wenn auch selten, ein graphisches Modell mit solch deterministischen verdeckten Einheiten zu konzipieren.

effizient mittels derselben bedingten Wahrscheinlichkeitsverteilungen parametrisiert werden kann und dass bei Beobachtung sämtlicher Variablen die Wahrscheinlichkeit der multivariaten Zuordnung aller Variablen effizient berechnet werden kann.

Selbst bei einer effizienten Parametrisierung des graphischen Modells stellen einige Operationen eine rechnerische Herausforderung dar. Zum Beispiel ist es schwierig, fehlende Werte in der Mitte der Sequenz vorherzusagen.

Der Preis, den RNNs für die geringere Parameteranzahl zahlen, ist die möglicherweise schwierige *Optimierung* der Parameter.

Parameter Sharing in RNNs beruht auf der Annahme, dass dieselben Parameter für unterschiedliche Zeitschritte verwendet werden können. Ebenso gilt die Annahme, dass die bedingte Wahrscheinlichkeitsverteilung über die Variablen im Zeitpunkt $t + 1$ auf Grundlage der Variablen im Zeitpunkt t **stationär** ist, dass also die Beziehung zwischen dem vorhergehenden Zeitschritt und dem nächsten Zeitschritt nicht abhängig von t ist. Im Prinzip wäre es möglich, t in jedem Zeitschritt als zusätzliche Eingabe zu verwenden und den Klassifikator jegliche Zeitabhängigkeit erkennen zu lassen, während er so viele Informationen wie möglich zwischen den verschiedenen Zeitschritten teilt. Das wäre bereits sehr viel besser als der Einsatz unterschiedlicher bedingter Wahrscheinlichkeitsverteilungen für jeden Zeitpunkt t , aber das Netz müsste anschließend für neue Werte von t extrapolieren.

Um unsere Betrachtung eines RNNs als graphisches Modell abzuschließen, müssen wir beschreiben, wie aus diesem Modell Stichproben gezogen werden. Die wesentliche Operation, die wir ausführen müssen, ist einfach in jedem Zeitschritt das Ziehen von Stichproben aus der bedingten Verteilung. Allerdings gibt es eine weitere Komplikation. Das RNN muss über einen Mechanismus verfügen, mit dem sich die Sequenzlänge bestimmen lässt. Hierfür bieten sich verschiedene Möglichkeiten an.

Wenn die Ausgabe ein aus einem Vokabular entnommenes Symbol ist, können wir ein spezielles Symbol für das Ende der Sequenz hinzufügen (*Schmidhuber*, 2012). Wird dieses Symbol erzeugt, endet die Stichprobenentnahme. In der Trainingsdatenmenge fügen wir dieses Symbol als zusätzliches Element der Sequenz in jedem Trainingsbeispiel direkt nach $\mathbf{x}^{(\tau)}$ hinzu.

Eine weitere Option besteht im Einführen einer zusätzlichen Bernoulli-Ausgabe zum Modell, die für jeden Zeitschritt die Entscheidung darstellt, entweder mit der Generierung fortzufahren oder diese zu beenden. Dieser Ansatz ist allgemeiner als das Hinzufügen eines zusätzlichen Symbols zum Vokabular, da er sich auf beliebige RNNs anwenden lässt, nicht nur auf solche, die eine Sequenz von Symbolen ausgeben. Zum Beispiel lässt

sich dieser Ansatz für ein RNN verwenden, das eine Sequenz mit reellen Zahlen ausgibt. Die neue Ausgabeeinheit ist meist eine sigmoide Einheit, die mit dem Kreuzentropieverlust trainiert wurde. Bei diesem Ansatz wird die Sigmoidfunktion auf eine Maximierung der Log-Wahrscheinlichkeit der korrekten Vorhersage trainiert, mit der für jeden Zeitschritt angegeben wird, ob die Sequenz endet oder fortgeführt wird.

Eine weitere Möglichkeit zum Bestimmen der Sequenzlänge τ ist das Hinzufügen einer weiteren Ausgabe zum Modell, die die ganze Zahl τ selbst vorhersagt. Das Modell kann Stichproben eines Werts von τ ziehen und die Stichprobenentnahme anschließend τ Datenschritte lang durchführen. Dieser Ansatz benötigt in jedem Zeitschritt eine zusätzliche Eingabe für das rekurrente Update, damit das Update »weiß«, ob das Ende der erzeugten Sequenz nahe ist. Diese zusätzliche Eingabe kann entweder aus dem Wert für τ oder $\tau - t$ – der Anzahl verbleibender Schritte – bestehen. Ohne die zusätzliche Eingabe könnte das RNN abrupt endende Sequenzen erzeugen, zum Beispiel einen Satz, der unvollständig abbricht. Der Ansatz beruht auf der Zerlegung

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}) = P(\tau)P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)} | \tau). \quad (10.34)$$

Das Verfahren zur direkten Vorhersage von τ wird beispielsweise von *Goodfellow et al.* (2014d) genutzt.

10.2.4 Modellieren von kontextabhängigen Sequenzen mit RNNs

Im vorhergehenden Abschnitt haben wir beschrieben, wie ein RNN einem gerichteten graphischen Modell über einer Sequenz von Zufallsvariablen $y^{(t)}$ ohne Eingaben \mathbf{x} entsprechen könnte. Natürlich enthielt unsere Entwicklung von RNNs laut Gleichung 10.8 eine Sequenz aus Eingaben $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(\tau)}$. Generell ermöglichen RNNs die Erweiterung des graphischen Modells nicht nur zur Darstellung einer multivariaten Verteilung über die y -Variablen, sondern auch einer bedingten Verteilung über y auf Basis von \mathbf{x} . Wie im Zusammenhang mit Feedforward-Netzen in Abschnitt 6.2.1.1 erläutert, kann jedes Modell, das eine Variable $P(\mathbf{y}; \boldsymbol{\theta})$ repräsentiert, als Modell neu interpretiert werden, das eine bedingte Verteilung $P(\mathbf{y}|\boldsymbol{\omega})$ mit $\boldsymbol{\omega} = \boldsymbol{\theta}$ repräsentiert. Wir können ein solches Modell für die Repräsentation einer Verteilung $P(\mathbf{y} | \mathbf{x})$ erweitern, indem wir dasselbe $P(\mathbf{y} | \boldsymbol{\omega})$ wie zuvor verwenden, dabei allerdings $\boldsymbol{\omega}$ zu einer Funktion von \mathbf{x} machen. Im Falle eines RNNs lässt sich dies auf unterschiedliche Weise erreichen. Wir stellen hier die gängigsten vor.

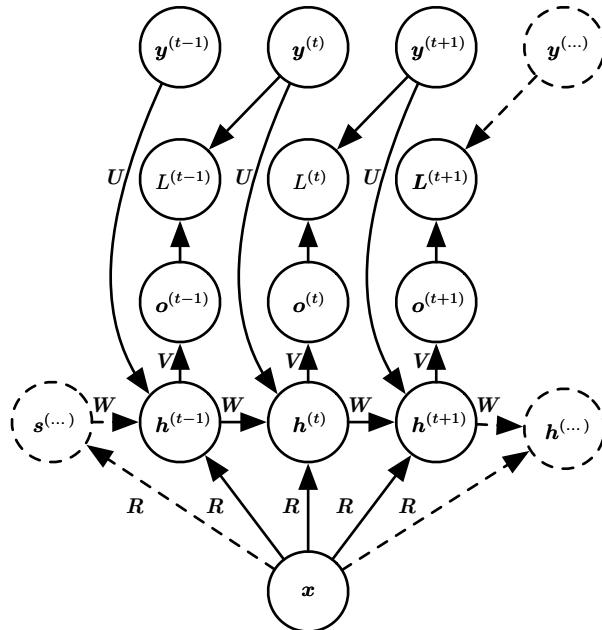


Abbildung 10.9: Ein RNN, das einen Vektor \mathbf{x} mit fester Länge einer Verteilung über die Sequenzen \mathbf{Y} zuordnet. Dieses RNN ist für Aufgaben wie die Bildbeschriftung geeignet, in denen ein Einzelbild als Modelleingabe dient, aus der eine Sequenz von Wörtern erzeugt wird, die den Bildinhalt beschreibt. Jedes Element $y^{(t)}$ der beobachteten Ausgabesequenz dient sowohl als Eingabe (für den aktuellen Zeitschritt) und – während des Trainings – als Ziel (für den vorhergehenden Zeitschritt).

Zuvor haben wir RNNs betrachtet, die eine Sequenz von Vektoren $\mathbf{x}^{(t)}$ für $t = 1, \dots, \tau$ als Eingabe nutzen. Eine andere Option verwendet nur einen einzelnen Vektor \mathbf{x} als Eingabe. Wenn es sich bei \mathbf{x} um einen Vektor mit fester Größe handelt, können wir daraus einfach eine zusätzliche Eingabe für das RNN machen, das die \mathbf{y} -Sequenz erzeugt. Einige gängige Arten zum Bereitstellen einer zusätzlichen Eingabe für ein RNN sind diese:

1. Verwenden einer zusätzlichen Eingabe für jeden Zeitschritt
2. Verwenden der Eingabe als anfänglichen Zustand $\mathbf{h}^{(0)}$
3. Kombination der beiden

Der erste und üblichste Ansatz ist in Abbildung 10.9 dargestellt. Die Interaktion zwischen der Eingabe \mathbf{x} und jedem verdeckten Einheitsvektor $\mathbf{h}^{(t)}$ wird durch eine neu eingeführte Gewichtungsmatrix \mathbf{R} parametrisiert, die im Modell, das nur die Sequenz der y -Werte enthält, nicht vorkommt. Dasselbe Produkt $\mathbf{x}^\top \mathbf{R}$ wird in jedem Zeitschritt als zusätzliche Eingabe

für die verdeckten Einheiten addiert. Sie können sich die Wahl von \boldsymbol{x} als Bestimmung des Werts von $\boldsymbol{x}^\top \boldsymbol{R}$ vorstellen, der effektiv ein neuer Verzerrungsparameter für jede der verdeckten Einheiten ist. Die Gewichte bleiben unabhängig von der Eingabe. Stellen Sie sich dieses Modell so vor, dass die Parameter $\boldsymbol{\theta}$ des nicht bedingten Modells in $\boldsymbol{\omega}$ umgewandelt werden, wobei die Verzerrungsparameter in $\boldsymbol{\omega}$ nun eine Funktion der Eingabe darstellen.

Statt nur einen einzelnen Vektor \boldsymbol{x} als Eingabe zu erhalten, kann das RNN eine Reihe von Vektoren $\boldsymbol{x}^{(t)}$ als Eingabe erhalten. Das in Gleichung 10.8 beschriebene RNN entspricht einer bedingten Verteilung $P(\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(\tau)} | \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)})$, die eine bedingte Unabhängigkeitsannahme trifft, die diese Verteilung faktorisiert als

$$\prod_t P(\boldsymbol{y}^{(t)} | \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}). \quad (10.35)$$

Um die bedingte Unabhängigkeitsannahme zu entfernen, können wir Verbindungen von der Ausgabe im Zeitpunkt t zur verdeckten Einheit im Zeitpunkt $t + 1$ hinzufügen (vgl. Abbildung 10.10). Das Modell kann beliebige Wahrscheinlichkeitsverteilungen über die \boldsymbol{y} -Sequenz darstellen. Diese Art Modell zur Darstellung einer Verteilung über eine Sequenz aufgrund einer weiteren Sequenz weist noch immer eine Einschränkung auf: Die Länge beider Sequenzen muss gleich sein. Wie sich diese Einschränkung aufheben lässt, erfahren Sie in Abschnitt 10.4.

10.3 Bidirektionale RNNs

Alle bisher betrachteten RNNs weisen eine »kausale« Struktur auf. Das bedeutet, dass der Zustand im Zeitpunkt t nur Informationen aus der Vergangenheit erfasst, nämlich $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t-1)}$ sowie die aktuelle Eingabe $\boldsymbol{x}^{(t)}$. Einige der behandelten Modelle lassen auch zu, dass Informationen der letzten \boldsymbol{y} -Werte den aktuellen Zustand beeinflussen, sofern die \boldsymbol{y} -Werte verfügbar sind.

In vielen Anwendungen möchten wir allerdings eine Vorhersage für $\boldsymbol{y}^{(t)}$ treffen, die möglicherweise von *der gesamten Eingabesequenz* abhängig ist. Ein Beispiel: In der Spracherkennung kann die korrekte Interpretation des aktuellen Lauts als Phonem aufgrund der Koartikulation von einigen darauffolgenden Phonemen abhängig sein; sogar eine Abhängigkeit von den nächsten Wörtern ist infolge der linguistischen Abhängigkeiten in einer Wortgruppe möglich: Wenn es zwei Interpretationen des aktuellen Worts gibt, die beide akustisch plausibel sind, müssen wir eventuell weit in die

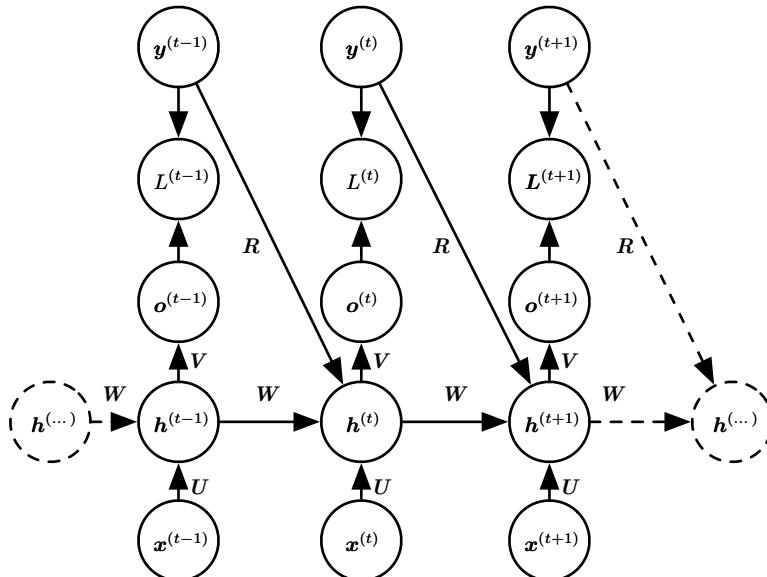


Abbildung 10.10: Ein bedingtes RNN, das eine Sequenz von x -Werten mit veränderlicher Länge einer Verteilung über Sequenzen der y -Werte identischer Länge zuordnet. Im Gegensatz zu Abbildung 10.3 enthält dieses RNN Verbindungen von der vorhergehenden Ausgabe zum aktuellen Zustand. Diese Verbindungen ermöglichen diesem RNN das Modellieren einer beliebigen Verteilung über Sequenzen von y anhand der Sequenzen von x mit identischer Länge. Das RNN aus Abbildung 10.3 kann nur Verteilungen darstellen, in denen die y -Werte auf Grundlage der x -Werte bedingt unabhängig voneinander sind.

Zukunft (und Vergangenheit) blicken, um sie zu diambiguieren. Das gilt auch für die Handschrifterkennung und viele weitere Sequenz-zu-Sequenz-Lernaufgaben, die im nächsten Abschnitt beschrieben werden.

Bidirektionale RNNs (oder BRNNs) wurden genau aus diesem Grund erfunden (*Schuster und Paliwal*, 1997). Sie haben sich als extrem erfolgreich (*Graves*, 2012) in entsprechenden Anwendungen erwiesen, darunter bei der Handschrifterkennung (*Graves et al.*, 2008; *Graves und Schmidhuber*, 2009), der Spracherkennung (*Graves und Schmidhuber*, 2005; *Graves et al.*, 2013) und in der Bioinformatik (*Baldi et al.*, 1999).

Wie der Name schon andeutet, kombinieren bidirektionale RNNs ein RNN, das sich vorwärts durch die Zeit bewegt (also am Anfang der Sequenz beginnt), mit einem weiteren RNN, das sich rückwärts durch die Zeit bewegt (also am Ende der Sequenz beginnt). Abbildung 10.11 zeigt ein typisches Beispiel für ein bidirektionales RNN. Dabei steht $h^{(t)}$ für den Zustand des Sub-RNNs, das sich vorwärts durch die Zeit bewegt, und

$\mathbf{g}^{(t)}$ für den Zustand des Sub-RNNs, das sich rückwärts durch die Zeit bewegt. So können die Ausgabeeinheiten $\mathbf{o}^{(t)}$ eine Repräsentation berechnen, die von der Vergangenheit UND der Zukunft abhängig ist, aber ebenfalls sehr empfindlich auf die Eingabewerte rund um den Zeitpunkt t reagiert, ohne dass ein Bereich fester Größe um t herum definiert wäre (wie es bei einem Feedforward-Netz, einem CNN oder einem regulären RNN mit einem Vorschaupuffer mit fester Größe (engl. *fixed-size look-ahead buffer*) der Fall wäre).

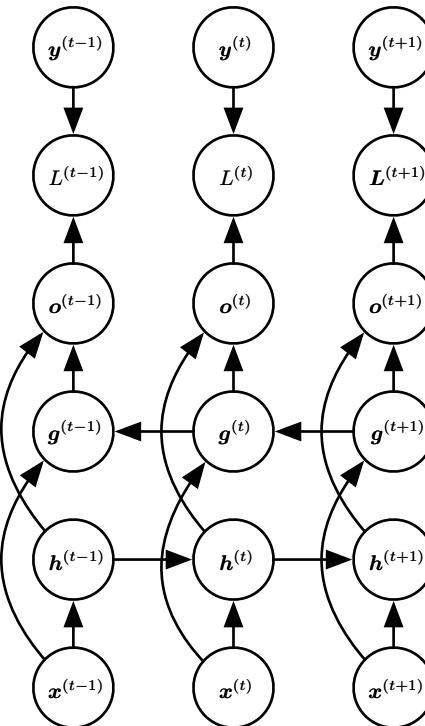


Abbildung 10.11: Berechnung eines typischen bidirektionalen RNNs, das lernen soll, Eingabesequenzen \mathbf{x} entsprechenden Zielsequenzen \mathbf{y} zuzuordnen, mit einem Verlust $L^{(t)}$ in jedem Zeitschritt t . Die \mathbf{h} -Rekurrenz propagiert Informationen in der Zeit vorwärts (nach rechts), die \mathbf{g} -Rekurrenz dagegen rückwärts (nach links). Somit können in jedem Punkt t die Ausgabeeinheiten $\mathbf{o}^{(t)}$ von einer relevanten Zusammenfassung der Vergangenheit in ihrer Eingabe $\mathbf{h}^{(t)}$ profitieren und von einer relevanten Zusammenfassung der Zukunft in ihrer Eingabe $\mathbf{g}^{(t)}$.

Diese Idee lässt sich ganz natürlich auf zweidimensionale Eingaben wie Bilder erweitern, indem vier RNNs eingesetzt werden – eines für jede der vier Kardinalrichtungen Aufwärts, Abwärts, Links und Rechts. In jedem Punkt (i, j) eines 2-D-Rasters könnte eine Ausgabe $O_{i,j}$ dann eine Repräsentation

berechnen, die primär lokale Informationen erfasst, aber gleichzeitig auch von langfristigeren Eingaben abhängig ist, sofern das RNN lernen kann, diese Informationen mitzuführen. Im Vergleich zu CNNs sind RNNs für die Anwendung auf Bilder meist aufwendiger, ermöglichen aber langfristige laterale Interaktionen zwischen Merkmalen derselben Merkmalskarte (*Visin et al.*, 2015; *Kalchbrenner et al.*, 2015). Tatsächlich lassen sich die Gleichungen für die Forward-Propagation in solchen RNNs in einer Form notieren, die zeigt, dass sie eine Faltung verwenden, die die Eingabe einer Schicht von unten nach oben berechnet, bevor die rekurrente Propagation in der Merkmalskarte erfolgt, die die lateralen Interaktionen einbezieht.

10.4 Sequenz-zu-Sequenz-Architekturen

In Abbildung 10.5 ist zu sehen, wie ein RNN eine Eingabesequenz einem Vektor mit fester Größe zuordnet. In Abbildung 10.9 ordnet ein RNN umgekehrt einen Vektor mit fester Größe einer Sequenz zu. Und in den Abbildungen 10.3, 10.4, 10.10 und 10.11 wird gezeigt, wie ein RNN eine Eingabesequenz einer Ausgabesequenz mit identischer Länge zuordnen kann.

Hier behandeln wir, wie ein RNN so trainiert werden kann, dass es eine Eingabesequenz einer Ausgabesequenz zuordnen kann, die nicht zwingend dieselbe Länge aufweisen muss. Diese Notwendigkeit entsteht in vielen Anwendungen, darunter Spracherkennung, maschinelle Übersetzung und Antwortsysteme, denn die Ein- und Ausgabesequenzen in der Trainingsdatenmenge sind nur selten gleich lang (obwohl die Längen zusammenhängen können).

Wir bezeichnen die Eingabe für ein RNN häufig als »Kontext«. Für diesen Kontext C möchten wir eine Repräsentation erzeugen. Beim Kontext C kann es sich um einen Vektor oder eine Sequenz von Vektoren handeln, der oder die die Eingabesequenz $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n_x)})$ zusammenfasst.

Die einfachste RNN-Architektur zum Zuordnen einer Sequenz mit variabler Länge zu einer anderen Sequenz mit variabler Länge wurde zuerst von *Cho et al.* (2014a) und kurz darauf von *Sutskever et al.* (2014) vorgeschlagen, die diese Architektur unabhängig voneinander entwickelt hatten und auf dieser Grundlage Wegbereiter für aktuelle Übersetzungen waren. Das erstgenannte System basiert auf Bewertungs- oder Scoring-Vorschlägen, die von einem anderen System für maschinelle Übersetzungen erzeugt werden, das zweite verwendet ein autonomes RNN zum Erzeugen der Übersetzungen. Die Autoren haben diese in Abbildung 10.12 dargestellte Architektur als Encoder-Decoder- bzw. Sequenz-zu-Sequenz-Architektur bezeichnet. Die

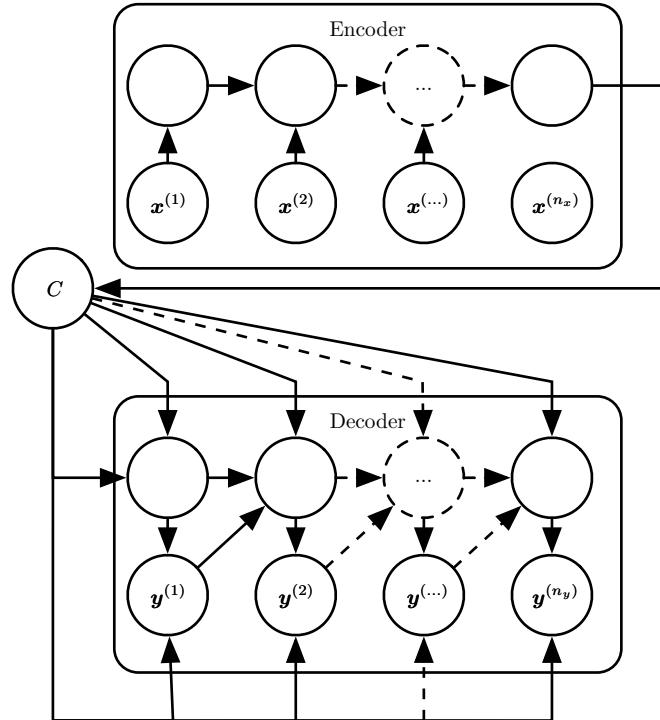


Abbildung 10.12: Beispiel für ein RNN mit Encoder-Decoder- oder Sequenz-zu-Sequenz-Architektur, das lernen soll, eine Ausgabesequenz $(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)})$ für eine Eingabesequenz $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n_x)})$ zu erzeugen. Es besteht aus einem Encoder-RNN, das die Eingabesequenz einliest, sowie einem Decoder-RNN, das die Ausgabesequenz erzeugt (bzw. die Wahrscheinlichkeit für eine bestimmte Ausgabesequenz berechnet). Der endgültige verdeckte Zustand des Encoder-RNNs dient zum Berechnen einer Kontextvariable C , die generell eine feste Größe aufweist, eine semantische Zusammenfassung der Eingabesequenz darstellt und als Eingabe für das Decoder-RNN dient.

Grundidee ist recht einfach: (1) Ein **Encoder** oder **Leser** oder **Eingabe**-RNN verarbeitet die Eingabesequenz. Der Encoder gibt den Kontext C aus, meist in Form einer einfachen Funktion seines endgültigen verdeckten Zustands. (2) Ein **Decoder** oder **Schreiber** oder **Ausgabe**-RNN wird für diesen Vektor mit fester Länge so festgelegt (wie in Abbildung 10.9), dass die Ausgabesequenz $\mathbf{Y} = (\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)})$ erzeugt wird. Die Innovation bei dieser Architekturform gegenüber den zuvor in diesem Kapitel vorgestellten Architekturen besteht darin, dass die Längen n_x und n_y voneinander abweichen können – in den anderen Architekturen galt stets $n_x = n_y = \tau$. In einer Sequenz-zu-Sequenz-Architektur werden die beiden RNNs gemeinsam so tra-

niert, dass der Durchschnittswert von $\log P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n_x)})$ über alle Paare aus den \mathbf{x} - und \mathbf{y} -Sequenzen in der Trainingsdatenmenge maximiert wird. Der letzte Zustand \mathbf{h}_{n_x} des Encoder-RNNs wird normalerweise als Repräsentation C der Eingabesequenz als Eingabe an das Decoder-RNN übergeben.

Wenn es sich beim Kontext C um einen Vektor handelt, ist das Decoder-RNN lediglich ein Vektor-zu-Sequenz-RNN, das in Abschnitt 10.2.4 beschrieben wird. Wie wir gesehen haben, gibt es mindestens zwei Möglichkeiten zur Eingabe für ein Vektor-zu-Sequenz-RNN: Die Eingabe kann als anfänglicher Zustand des RNNs erfolgen oder in jedem Zeitschritt mit den verdeckten Einheiten verbunden werden. Auch eine Kombination dieser beiden Wege ist möglich.

Es gibt keine Bedingung, dass der Encoder eine verdeckte Schicht der selben Größe wie der Decoder aufweisen muss.

Eine offensichtliche Einschränkung dieser Architektur tritt zutage, wenn der Kontext C auf der Ausgabeseite des Encoder-RNNs eine Dimension aufweist, die zu klein ist, um eine lange Sequenz ordentlich zusammenzufassen. Dieses Phänomen wurde von *Bahdanau et al.* (2015) im Zusammenhang mit maschinellen Übersetzungen festgestellt. Sie haben vorgeschlagen, C zu einer Sequenz mit variabler Länge zu machen und keinen Vektor mit fester Größe zu verwenden. Außerdem haben sie einen **Aufmerksamkeitsmechanismus** (engl. *attention mechanism*) eingeführt, der lernt, Elemente der Sequenz C mit Elementen der Ausgabesequenz zu verknüpfen. Weitere Informationen finden Sie in Abschnitt 12.4.5.1.

10.5 Tiefe RNNs

Die Berechnung in den meisten RNNs kann in drei Parameterblöcke und zugehörige Transformationen zerlegt werden:

1. von der Eingabe zum verdeckten Zustand
2. vom vorhergehenden zum nachfolgenden verdeckten Zustand
3. vom verdeckten Zustand zur Ausgabe

In der RNN-Architektur aus Abbildung 10.3 ist jeder dieser drei Blöcke mit einer einzelnen Gewichtungsmatrix verknüpft. Anders formuliert: Wenn das Netz aufgefaltet wird, entspricht jeder dieser Blöcke einer flachen Transformation. Als *flache Transformation* bezeichnen wir eine Transformation,

die in einem tiefen mehrschichtigen Perzeptron als einzelne Schicht repräsentiert würde. Normalerweise handelt es sich um eine Transformation, die durch eine erlernte affine Transformation gefolgt von einer unveränderlichen Nichtlinearität dargestellt wird.

Wäre es von Vorteil, in jeder dieser Operationen für mehr Tiefe zu sorgen? Experimentelle Nachweise (*Graves et al.*, 2013; *Pascanu et al.*, 2014a) deuten stark darauf hin. Der experimentelle Nachweis stimmt mit der Idee überein, dass wir zum Durchführen der erforderlichen Zuordnungen eine hinreichende Tiefe benötigen. In *Schmidhuber* (1992), *El Hihi und Bengio* (1996) oder *Jaeger* (2007a) finden Sie frühe Arbeiten zu tiefen RNNs.

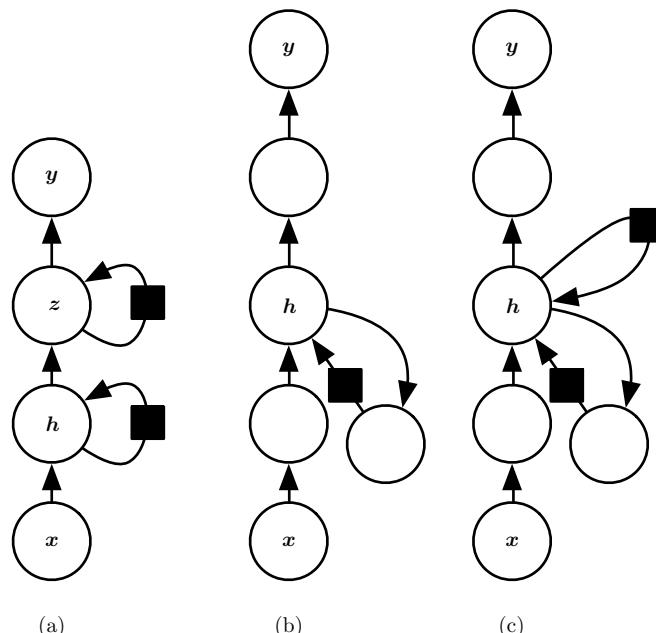


Abbildung 10.13: Ein RNN kann auf vielfältige Weise tief gestaltet werden (*Pascanu et al.*, 2014a). (a) Der verdeckte rekurrente Zustand kann in hierarchisch organisierte Gruppen unterteilt werden. (b) Es kann eine tiefere Berechnung (z. B. ein MLP) in den Teilen Eingabe-zu-Verdeckt, Verdeckt-zu-Verdeckt und Verdeckt-zu-Ausgabe eingeführt werden. Dadurch wird eventuell der kürzeste Pfad zwischen unterschiedlichen Zeitschritten verlängert. (c) Der Pfadverlängerungseffekt lässt sich durch Einführen von Skip Connections abschwächen.

Graves et al. (2013) zeigten als Erste einen nennenswerten Vorteil durch Zerlegung des Zustands eines RNNs in mehrere Schichten (vgl. den linken Teil von Abbildung 10.13). Wir können uns vorstellen, dass die unteren Schichten in der Hierarchie aus Abbildung 10.13a eine Rolle bei der Transformation der unbearbeiteten Eingabe (engl. *raw input*) in eine für die

höheren Ebenen des verdeckten Zustands passendere Repräsentation spielen. *Pascanu et al.* (2014a) gehen noch einen Schritt weiter und schlagen ein separates mehrschichtiges Perzeptron (möglicherweise tief) für jeden der drei oben aufgeführten Blöcke vor (siehe Abbildung 10.13b). Überlegungen hinsichtlich der repräsentativen Kapazität legen nahe, ausreichende Kapazitäten in jedem der drei Schritte zu reservieren; wenn dies jedoch anhand zusätzlicher Tiefe erfolgt, kann die Lernleistung sinken, da die Optimierung schwieriger wird. Grundsätzlich ist es einfacher, flachere Architekturen zu optimieren; die zusätzliche Tiefe in Abbildung 10.13b lässt den kürzesten Pfad von einer Variable im Zeitschritt t zu einer Variable im Zeitschritt $t + 1$ länger werden. Ein Beispiel: Wird ein mehrschichtiges Perzeptron mit nur einer verdeckten Schicht für den Zustand-zu-Zustand-Übergang verwendet, verdoppelt sich die Länge des kürzesten Pfads zwischen Variablen in zwei unterschiedlichen Zeitschritten gegenüber dem gewöhnlichen RNN aus Abbildung 10.3. Allerdings, so auch *Pascanu et al.* (2014a), kann dieser Effekt durch Einführen von Skip Connections im Verdeckt-zu-Verdeckt-Pfad abgeschwächt werden (vgl. Abbildung 10.13c).

10.6 Rekursive neuronale Netze

Rekursive neuronale Netze² repräsentieren noch eine weitere Generalisierung von RNNs, mit einer anderen Art von Berechnungsgraph, der als tiefer Baum strukturiert ist, im Gegensatz zu einer kettenförmigen Struktur in einem RNN.

Der typische Berechnungsgraph eines rekursiven Netzes ist in Abbildung 10.14 dargestellt. Rekursive neuronale Netze wurden von *Pollack* (1990) eingeführt. Die Möglichkeit, diese einzusetzen, um zu lernen, Schlüsse zu ziehen, wurde in *Bottou* (2011) beschrieben. Rekursive Netze wurden erfolgreich für die Verarbeitung von *Datenstrukturen* als Eingabe für neuronale Netze eingesetzt (*Frasconi et al.*, 1997, 1998), aber auch für die Verarbeitung natürlicher Sprache (engl. *natural language processing*, NLP) (*Socher et al.*, 2011a,c, 2013a) und in der Computer Vision (*Socher et al.*, 2011b).

Ein klarer Vorteil rekursiver Netze gegenüber RNNs besteht darin, dass die Tiefe (gemessen als Anzahl der Kompositionen nichtlinearer Operationen) für eine Sequenz identischer Länge τ wesentlich von τ auf $O(\log \tau)$ reduziert werden kann, was insbesondere im Zusammenhang mit langfristi-

² Wir raten davon ab, den Begriff »rekursives neuronales Netz« als »RNN« abzukürzen, um Verwechslungen mit dem Begriff »rekurrentes neuronales Netz« zu vermeiden.

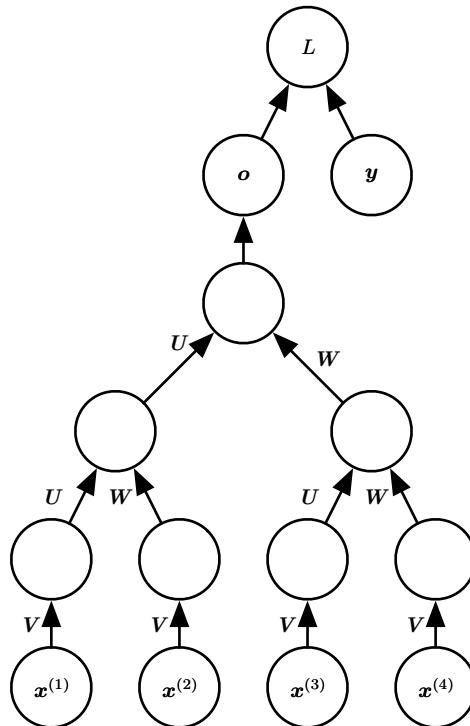


Abbildung 10.14: Ein rekursives Netz weist einen Berechnungsgraphen auf, der den des RNNs von einer Kette zu einem Baum generalisiert. Eine Sequenz $x^{(1)}, x^{(2)}, \dots, x^{(t)}$ variabler Größe kann einer Repräsentation mit unveränderlicher Größe (der Ausgabe \mathbf{o}) mittels einer festen Parametermenge (den Gewichtungsmatrizen $\mathbf{U}, \mathbf{V}, \mathbf{W}$) zugeordnet werden. Die Abbildung zeigt den Fall beim überwachten Lernen, in dem ein Zielwert \mathbf{y} angegeben wird, das mit der gesamten Sequenz verbunden ist.

gen Abhängigkeiten hilfreich sein kann. Es bleibt die Frage, wie der Baum am besten aufgebaut sein sollte. Eine Option ist eine Baumstruktur, die unabhängig von den Daten ist, zum Beispiel ein balancierter Binärbaum. In einigen Anwendungsbereichen können externe Verfahren bei der Wahl der geeigneten Baumstruktur helfen. Werden zum Beispiel Sätze in natürlicher Sprache verarbeitet, kann die Baumstruktur für das rekursive Netz der Struktur des Syntaxbaums des Satzes entsprechen, der von einem NLP-Parser geliefert wird (*Socher et al., 2011a, 2013a*). Idealerweise soll der Klassifikator selbst die passende Baumstruktur für jede beliebige Eingabe erkennen und ableiten können; siehe auch *Bottou (2011)*.

Das Konzept des rekursiven Netzes ist in vielen Varianten denkbar. Zum Beispiel verknüpfen *Frasconi et al. (1997, 1998)* die Daten mit einer Baum-

struktur und die Eingaben und Zielwerte mit einzelnen Knoten des Baums. Die in jedem Knoten ausgeführte Berechnung muss keine klassische Berechnung eines künstlichen Neurons sein (affine Transformation aller Eingaben gefolgt von einer monotonen Nichtlinearität). *Socher et al.* (2013a) schlagen beispielsweise vor, Tensor-Operationen und bilineare Formen zu verwenden, die sich schon bei der Modellierung von Beziehungen zwischen Konzepten als hilfreich erwiesen haben (*Weston et al.*, 2010; *Bordes et al.*, 2012), wenn die Konzepte durch stetige Vektoren (Embeddings, dt. *Einbettungen*) dargestellt werden.

10.7 Die Herausforderung langfristiger Abhängigkeiten

Die mathematische Herausforderung beim Erlernen langfristiger Abhängigkeiten in RNNs wird in Abschnitt 8.2.5 vorgestellt. Das Grundproblem ist, dass über viele Phasen propagierte Gradienten dazu neigen, entweder zu verschwinden (dies geschieht in den meisten Fällen) oder zu explodieren (selten, aber mit großem Schaden für die Optimierung). Selbst wenn wir annehmen, dass die Parameter so gewählt sind, dass das RNN stabil ist (wir können Erinnerungen ablegen, ohne dass die Gradienten explodieren), entsteht die Schwierigkeit mit langfristigen Abhängigkeiten aus den exponentiell kleineren Gewichten für langfristige Interaktionen (die an der Multiplikation vieler Jacobi-Matrizen beteiligt sind) gegenüber denen kurzfristiger Interaktionen. Viele weitere Quellen gehen näher darauf ein (*Hochreiter*, 1991; *Doya*, 1993; *Bengio et al.*, 1994; *Pascanu et al.*, 2013). In diesem Abschnitt beschreiben wir das Problem genauer. Die verbleibenden Abschnitte beschreiben Ansätze für den Umgang mit diesem Problem.

RNNs nutzen die wiederholte Komposition derselben Funktion – einmal pro Zeitschritt. Diese Kompositionen können ein extrem nichtlineares Verhalten hervorrufen (vgl. Abbildung 10.15).

Insbesondere die Funktionskomposition in RNNs ähnelt in gewisser Weise einer Matrizenmultiplikation. Wir können uns die Rekurrenzrelation

$$\mathbf{h}^{(t)} = \mathbf{W}^\top \mathbf{h}^{(t-1)} \quad (10.36)$$

als sehr einfaches RNN ohne nichtlineare Aktivierungsfunktion und ohne die Eingaben \mathbf{x} vorstellen. Wie in Abschnitt 8.2.5 beschrieben, beschreibt diese Rekurrenzrelation im Wesentlichen die Potenzmethode. Sie kann vereinfacht werden zu

$$\mathbf{h}^{(t)} = (\mathbf{W}^t)^\top \mathbf{h}^{(0)}; \quad (10.37)$$

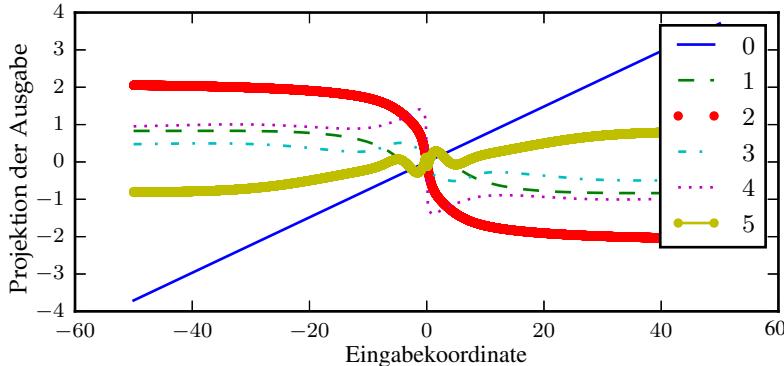


Abbildung 10.15: Wiederholte Komposition von Funktionen. Werden viele nichtlineare Funktionen zusammengesetzt (wie die hier abgebildete Schicht linear-tanh), ist das Ergebnis hochgradig nichtlinear, wobei im Normalfall die meisten Werte mit einer winzigen Ableitung und einige Werte mit einer großen Ableitung verknüpft sind sowie viele Wechsel zwischen Zu- und Abnahme vorliegen. Hier plotten wir eine lineare Projektion eines 100-dimensionalen verdeckten Zustands in eine Dimension (Plot auf der y -Achse). Die x -Achse ist die Koordinate des anfänglichen Zustands entlang einer zufälligen Richtung im 100-dimensionalen Raum. Wir können den Plot daher als linearen Querschnitt einer hochdimensionalen Funktion betrachten. Die Plots zeigen die Funktion nach jedem Zeitschritt bzw. äquivalent jeder Anzahl der Male, die die Übergangsfunktion zusammengesetzt wurde.

sofern \mathbf{W} eine Eigenwertzerlegung der Form

$$\mathbf{W} = \mathbf{Q} \boldsymbol{\Lambda} \mathbf{Q}^\top \quad (10.38)$$

mit orthogonalem \mathbf{Q} erlaubt, lässt sich die Rekurrenz weiter vereinfachen zu

$$\mathbf{h}^{(t)} = \mathbf{Q}^\top \boldsymbol{\Lambda}^t \mathbf{Q} \mathbf{h}^{(0)}. \quad (10.39)$$

Die Eigenwerte werden mit t potenziert, wodurch die Eigenwerte mit einem Betrag unter 1 zu 0 zerfallen und Eigenwerte mit einem Betrag über 1 explodieren. Jede Komponente von $\mathbf{h}^{(0)}$, die nicht am größten Eigenvektor ausgerichtet ist, wird irgendwann verworfen.

Dies ist ein spezielles Problem bei RNNs. Stellen Sie sich für den Skalarfall vor, dass ein Gewicht w mehrfach mit sich selbst multipliziert wird. Das Produkt w^t wird je nach dem Betrag von w verschwinden oder explodieren. Wenn wir ein nichtrekurrentes Netz verwenden, das in jedem Zeitschritt ein anderes Gewicht $w^{(t)}$ aufweist, stellt sich die Sache anders dar. Ergibt sich der anfängliche Zustand aus 1, dann ergibt sich der Zustand zum Zeitpunkt t aus $\prod_t w^{(t)}$. Nehmen wir an, die $w^{(t)}$ -Werte werden zufällig und unabhängig

voneinander erzeugt mit Mittelwert gleich Null und Varianz v . Die Varianz des Produkts beträgt $O(v^n)$. Um eine gewünschte Varianz v^* zu erreichen, können wir die individuellen Gewichte mit der Varianz $v = \sqrt[n]{v^*}$ auswählen. Sehr tiefe Feedforward-Netze mit sorgfältig ausgewählter Skalierung können somit das Problem der verschwindenden und explodierenden Gradienten umgehen, wie *Sussillo* (2014) zeigt.

Das Problem der verschwindenden und explodierenden Gradienten in RNNs wurde von mehreren Forschern unabhängig voneinander entdeckt (*Hochreiter*, 1991; *Bengio et al.*, 1993, 1994). Man könnte hoffen, dass es sich einfach dadurch vermeiden lässt, dass man in einem Bereich des Parameterraums bleibt, in dem die Gradienten weder verschwinden noch explodieren. Leider muss das RNN zum Speichern von Erinnerungen auf eine Weise, die sich gegenüber kleinen Störungen als robust erweist, Bereiche des Parameterraums betreten, in denen die Gradienten verschwinden (*Bengio et al.*, 1993, 1994). Insbesondere dann, wenn das Modell in der Lage ist, langfristige Abhängigkeiten darzustellen, weist der Gradient einer langfristigen Interaktion einen exponentiell kleineren Betrag auf als der Gradient einer kurzfristigen Interaktion. Das bedeutet nicht, dass das Lernen unmöglich wäre, sondern dass es sehr lange dauern kann, langfristige Abhängigkeiten zu erlernen, da das Signal über diese Abhängigkeiten dazu neigt, in den kleinsten Schwankungen unterzugehen, die aus den kurzfristigen Abhängigkeiten entstehen. In der Praxis zeigen die Experimente aus *Bengio et al.* (1994), dass die Optimierung auf Gradientenbasis mit zunehmender Bandbreite der Abhängigkeiten, die erfasst werden müssen, immer schwieriger wird und sich die Wahrscheinlichkeit eines erfolgreichen Trainings eines klassischen RNNs mittels stochastischen Gradientenabstiegsverfahrens (SGD, engl. *stochastic gradient descent*) rasant 0 annähert, wenn die Länge der Sequenzen nur 10 oder 20 beträgt.

Eine weiterführende Abhandlung von RNNs als dynamische Systeme findet sich in *Doya* (1993), *Bengio et al.* (1994) und *Siegelmann und Sontag* (1995), eine Besprechung in *Pascanu et al.* (2013). Die restlichen Abschnitte dieses Kapitels befassen sich mit diversen Ansätzen, die vorgeschlagen wurden, um die Schwierigkeit beim Erlernen langfristiger Abhängigkeiten zu reduzieren (sodass ein RNN in einigen Fällen Abhängigkeiten über Hunderte von Schritten erlernen kann). Dennoch bleibt das Erlernen langfristiger Abhängigkeiten eine der größten Herausforderungen für das Deep Learning.

10.8 Echo-State-Netze

Die Zuordnung der rekurrenten Gewichte von $\mathbf{h}^{(t-1)}$ zu $\mathbf{h}^{(t)}$ und die Zuordnung von Eingabegewichten von $\mathbf{x}^{(t)}$ zu $\mathbf{h}^{(t)}$ gehören zu den schwierigsten Parametern, die es in einem RNN zu erlernen gibt. Ein vorgeschlagener Ansatz zur Vermeidung dieser Schwierigkeit (Jaeger, 2003; Maass et al., 2002; Jaeger und Haas, 2004; Jaeger, 2007b) besteht darin, die rekurrenten Gewichte so einzustellen, dass die rekurrenten verdeckten Einheiten die Historie vergangener Eingaben gut erfassen können und *nur die Ausgabegewichte erlernen*. Diese Idee wurde unabhängig für sogenannte **Echo-State-Netze** (kurz ESNs) (Jaeger und Haas, 2004; Jaeger, 2007b) und **Liquid State Machines** (Maass et al., 2002) vorgeschlagen. Letztere sind ähnlich, allerdings werden Spiking-Neuronen (mit binären Ausgaben) anstelle der stetigwertigen verdeckten Einheiten aus ESNs verwendet. Sowohl ESNs als auch Liquid State Machines werden als **Reservoir Computing** (Lukoševičius und Jaeger, 2009) bezeichnet, da die verdeckten Einheiten ein Reservoir temporaler Merkmale bilden, das unterschiedliche Aspekte der Historie der Eingaben zu erfassen vermag.

Man kann sich diese Reservoir-Computing-RNNs ähnlich wie Kernel-Maschinen vorstellen: Sie ordnen eine beliebig lange Sequenz (die Historie der Eingaben bis zum Zeitpunkt t) einem Vektor mit fester Länge zu (dem rekurrenten Zustand $\mathbf{h}^{(t)}$), auf den ein linearer Prädiktor (meist eine lineare Regression) zum Lösen des gestellten Problems angewandt werden kann. Das Trainingskriterium kann dann ganz einfach konvex als eine Funktion der Ausgabegewichte konzipiert werden. Ein Beispiel: Wenn die Ausgabe aus einer linearen Regression von den verdeckten Einheiten zu den Zielwerten der Ausgabe besteht und das Trainingskriterium ein mittlerer quadratischer Fehler ist, ist sie konvex und kann zuverlässig mit einfachen Lernalgorithmen gelöst werden (Jaeger, 2003).

Die entscheidende Frage lautet daher wie folgt: Wie stellen wir die Eingabe- und die rekurrenten Gewichte so ein, dass im Zustand des RNNs eine ausreichende Menge an auf der Vergangenheit beruhender Erfahrung repräsentiert werden kann? Die in der Literatur zum Reservoir Computing vorgeschlagene Antwort lautet, dass man das RNN als dynamisches System betrachten und daher Eingabe- und rekurrente Gewichte so setzen solle, dass sich das dynamische System in der Nähe des Randes der Stabilität befindet.

Die ursprüngliche Idee bestand darin, die Eigenwerte der Jacobi-Matrix der Zustand-zu-Zustand-Übergangsfunktion beinahe 1 zu machen. Wie in Abschnitt 8.2.5 erläutert wird, ist eine wichtige Eigenschaft von RNNs

das Eigenwert-Spektrum der Jacobi-Matrizen $\mathbf{J}^{(t)} = \frac{\partial s^{(t)}}{\partial s^{(t-1)}}$. Von besonderer Bedeutung ist der **Spektralradius** von $\mathbf{J}^{(t)}$, der als Maximum der Absolutbeträge seiner Eigenwerte definiert ist.

Um den Effekt des Spektralradius zu verstehen, betrachten Sie den einfachen Fall der Backpropagation mit einer Jacobi-Matrix \mathbf{J} , die sich nicht mit t verändert. Das geschieht zum Beispiel, wenn das Netz rein linear ist. Angenommen, \mathbf{J} hat einen Eigenvektor \mathbf{v} mit zugehörigem Eigenwert λ . Was geschieht, wenn wir einen Gradientenvektor rückwärts durch die Zeit propagieren? Wenn wir mit einem Gradientenvektor \mathbf{g} beginnen, erhalten wir nach einem Schritt der Backpropagation $\mathbf{J}\mathbf{g}$ und nach n Schritten $\mathbf{J}^n\mathbf{g}$. Aber was geschieht, wenn wir stattdessen eine verzerrte Version von \mathbf{g} für die Backpropagation nutzen? Beginnend mit $\mathbf{g} + \delta\mathbf{v}$ erhalten wir nach einem Schritt $\mathbf{J}(\mathbf{g} + \delta\mathbf{v})$. Nach n Schritten erhalten wir $\mathbf{J}^n(\mathbf{g} + \delta\mathbf{v})$. Wir stellen also fest, dass die Backpropagation mit Ausgangspunkt \mathbf{g} und die mit Ausgangspunkt $\mathbf{g} + \delta\mathbf{v}$ nach n Schritten um $\delta\mathbf{J}^n\mathbf{v}$ voneinander abweichen. Wird \mathbf{v} als Einheits-Eigenvektor von \mathbf{J} mit dem Eigenwert λ gewählt, dann skaliert eine Multiplikation mit der Jacobi-Matrix die Differenz mit jedem Schritt. Die beiden Ausführungen der Backpropagation sind um den Abstand $\delta|\lambda|^n$ voneinander getrennt. Wenn \mathbf{v} dem größten Wert von $|\lambda|$ entspricht, erreicht diese Störung die größtmögliche Trennung einer anfänglichen Störung der Größe δ .

Für $|\lambda| > 1$ nimmt der Abweichungsbetrag $\delta|\lambda|^n$ exponentiell zu. Für $|\lambda| < 1$ wird der Abweichungsbetrag exponentiell klein.

Natürlich setzt dieses Beispiel voraus, dass die Jacobi-Matrix in jedem Zeitschritt identisch ist, was einem RNN ohne Nichtlinearität entspricht. Liegt eine Nichtlinearität vor, erreicht die Ableitung der Nichtlinearität in vielen Zeitschritten Null und hilft so, eine Explosion als Folge eines großen Spektralradius zu verhindern. Tatsächlich spricht sich die neueste Arbeit zu ESNs dafür aus, einen sehr viel größeren Spektralradius als Eins zu verwenden (Yildiz et al., 2012; Jaeger, 2012).

Alles bisher Gesagte über die Backpropagation mittels wiederholter Matrizenmultiplikation gilt ebenso für die Forward-Propagation in einem Netz ohne Nichtlinearitäten mit dem Zustand $\mathbf{h}^{(t+1)} = \mathbf{h}^{(t)\top} \mathbf{W}$.

Wenn eine lineare Abbildung \mathbf{W}^\top stets zur Abnahme von \mathbf{h} als Maß der L^2 -Norm führt, dann ist diese Abbildung **kontrahierend** (engl. *contractive*). Ist der Spektralradius kleiner als Eins, ist die Zuordnung von $\mathbf{h}^{(t)}$ zu $\mathbf{h}^{(t+1)}$ kontrahierend, sodass eine kleine Änderung nach jedem Zeitschritt kleiner wird. Das führt zwangsläufig dazu, dass das Netz Informationen über die

Vergangenheit vergisst, wenn wir ein endliches Genauigkeitsniveau (z.B. 32-Bit-Ganzzahlen) zum Speichern des Zustandsvektors verwenden.

Die Jacobi-Matrix gibt an, wie eine kleine Änderung von $\mathbf{h}^{(t)}$ einen Schritt vorwärts propagiert oder – äquivalent – wie der Gradient auf $\mathbf{h}^{(t+1)}$ während der Backpropagation einen Schritt rückwärts propagiert. Beachten Sie, dass weder \mathbf{W} noch \mathbf{J} symmetrisch sein müssen (obwohl sie quadratisch und reell sind), sodass sie komplexwertige Eigenwerte und Eigenvektoren mit imaginären Komponenten aufweisen können, die einem potenziell oszillierenden Verhalten entsprechen (sofern sie identisch zur iterativ angewandten Jacobi-Matrix sind). Obschon $\mathbf{h}^{(t)}$ oder eine kleine Variation von $\mathbf{h}^{(t)}$, die in der Backpropagation relevant sind, reellwertig sind, können sie in einer solch komplexwertigen Basis dargestellt werden. Es kommt darauf an, was mit dem Betrag (komplexer Absolutbetrag) dieser möglicherweise komplexwertigen Basis-Koeffizienten geschieht, wenn wir die Matrix mit dem Vektor multiplizieren. Ein Eigenwert mit einem Betrag größer Eins entspricht einer Vergrößerung (exponentielles Wachstum, sofern iterativ angewandt) oder Abnahme (exponentielle Verringerung, sofern iterativ angewandt).

Bei einer nichtlinearen Abbildung kann sich die Jacobi-Matrix mit jedem Schritt ändern. Die Dynamik wird dadurch viel komplexer. Es gilt jedoch weiterhin, dass eine kleine anfängliche Variation nach einigen Schritten zu einer großen Variation werden kann. Ein Unterschied zwischen dem rein linearen Fall und dem nichtlinearen Fall ist, dass die Verwendung einer zwingenden Nichtlinearität wie tanh dazu führen kann, dass die rekurrente Dynamik eingeschränkt wird. Beachten Sie, dass die Backpropagation die uneingeschränkte Dynamik auch dann beibehalten kann, wenn die Forward-Propagation eine eingeschränkte Dynamik aufweist, beispielsweise, wenn eine Sequenz von tanh-Einheiten sich vollständig in der Mitte ihres linearen Bereichs befindet und über Gewichtungsmatrizen mit einem Spektralradius größer als 1 verbunden ist. Nichtsdestotrotz ist es selten, dass alle tanh-Einheiten gleichzeitig in ihrem linearen Aktivierungspunkt liegen.

Das Verfahren der Echo-State-Netze besteht ganz einfach darin, die Gewichte auf einem bestimmten Spektralradius festzuhalten, zum Beispiel 3, in dem Informationen vorwärts durch die Zeit transportiert werden, ohne zu explodieren, da die sättigenden Nichtlinearitäten wie tanh eine stabilisierende Wirkung entfalten.

Kürzlich wurde gezeigt, dass Verfahren zum Festlegen der Gewichte in ESNs auch zum *Initialisieren* der Gewichte in einem vollständig trainierbaren RNN eingesetzt werden könnten (wobei die rekurrenten Verdeckt-zu-Verdeckt-Gewichte mittels Backpropagation im Zeitverlauf trainiert werden);

das ist für das Erlernen langfristiger Abhängigkeiten hilfreich (*Sutskever*, 2012; *Sutskever et al.*, 2013). In diesem Fall funktioniert ein anfänglicher Spektralradius von 1,2 in Verbindung mit dem in Abschnitt 8.4 beschriebenen Verfahren zur dünnbesetzten Initialisierung gut.

10.9 Leaky-Einheiten und andere Verfahren für mehrere Zeitskalen

Eine Möglichkeit für den Umgang mit langfristigen Abhängigkeiten ist das Entwerfen eines Modells, das mit mehreren Zeitskalen arbeitet, sodass einige Teile des Modells mit feinen Zeitskalen an kleinen Details arbeiten, während andere mit groben Zeitskalen Informationen aus der fernen Vergangenheit effizienter in die Gegenwart übertragen können. Es sind diverse Vorgehensweisen zum Konstruieren dieser feinen und groben Zeitskalen möglich. Dazu gehören das Hinzufügen sogenannte Skip Connections über die Zeit, »Leaky-Einheiten« (engl. *leaky units*), die Signale mit unterschiedlichen Zeitkonstanten integrieren, sowie das Entfernen einiger Verbindungen, die zum Modellieren feiner Zeitskalen eingesetzt werden.

10.9.1 Hinzufügen von Skip Connections über die Zeit

Eine Möglichkeit zum Schaffen grober Zeitskalen besteht im Hinzufügen direkter Verbindungen von Variablen in der fernen Vergangenheit zu Variablen in der Gegenwart. Die Idee solcher Skip Connections geht auf *Lin et al.* (1996) zurück und gründet auf der Idee, Verzögerungen in neuronale Feed-forward-Netze einzubinden (*Lang und Hinton*, 1988). In einem gewöhnlichen RNN verknüpft eine rekurrente Verbindung eine Einheit im Zeitpunkt t mit einer Einheit im Zeitpunkt $t + 1$. Es ist möglich, RNNs mit längeren Verzögerungen zu konstruieren (*Bengio*, 1991).

Wie wir in Abschnitt 8.2.5 gezeigt haben, können Gradienten exponentiell *zur Anzahl der Zeitschritte* verschwinden oder explodieren. *Lin et al.* (1996) hat rekurrente Verbindungen mit einer Zeitverzögerung d vorgestellt, die dieses Problem verringern. Gradienten verschwinden nun exponentiell als Funktion von $\frac{\tau}{d}$ statt τ . Da es sowohl verzögerte als auch Einzelschrittverbindungen gibt, können Gradienten in τ noch immer exponentiell explodieren. Auf diese Weise kann der Lernalgorithmus längere Abhängigkeiten erfassen; allerdings werden möglicherweise nicht alle langfristigen Abhängigkeiten auf diese Weise gut dargestellt.

10.9.2 Leaky-Einheiten und ein Spektrum unterschiedlicher Zeitskalen

Eine weitere Option für Pfade, auf denen das Produkt der Ableitungen nahezu Eins ist, besteht in Einheiten mit *linearen* Selbstverbindungen (engl. *linear self-connections*) und einem Gewicht von nahezu Eins auf diesen Verbindungen.

Wenn wir einen gleitenden Mittelwert $\mu^{(t)}$ eines Wertes $v^{(t)}$ anhand des Updates $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1 - \alpha)v^{(t)}$ akkumulieren, ist der α -Parameter ein Beispiel für eine lineare Selbstverbindung von $\mu^{(t-1)}$ zu $\mu^{(t)}$. Ist α nahezu Eins, erinnert sich der gleitende Mittelwert lange Zeit an Informationen über die Vergangenheit; ist α nahezu Null, werden Informationen über die Vergangenheit sehr schnell verworfen. Verdeckte Einheiten mit linearen Selbstverbindungen können sich ähnlich wie solche gleitenden Mittelwerte verhalten. Man nennt derartige verdeckte Einheiten **Leaky-Einheiten** (engl. *leaky units*).

Skip Connections über d Zeitschritte sind eine Möglichkeit, sicherzustellen, dass eine Einheit jederzeit lernen kann, von einem Wert beeinflusst zu werden, der d Zeitschritte zurückliegt. Die Nutzung einer linearen Selbstverbindung mit einem Gewicht nahe Eins ist eine weitere Möglichkeit, sicherzustellen, dass eine Einheit auf Werte aus der Vergangenheit zugreifen kann. Die lineare Selbstverbindung erlaubt eine feinere und flexiblere Anpassung dieses Effekts durch Ändern des reellwertigen α anstelle der ganzzahligen Sprunglänge.

Diese Ideen wurden von *Mozer* (1992) und *El Hihi und Bengio* (1996) vorgeschlagen. Leaky-Einheiten haben sich auch im Rahmen von Echo-State-Netzen als nützlich erwiesen (*Jaeger et al.*, 2007).

Es gibt zwei grundlegende Verfahren zum Setzen der Zeitkonstanten, die von Leaky-Einheiten genutzt werden. Eine besteht darin, sie von Hand auf Werte festzulegen, die konstant bleiben, zum Beispiel durch einmalige Entnahme von Wertbeispielen aus einer Verteilung zum Initialisierungszeitpunkt. Die andere macht die Zeitkonstanten zu freien Parametern und erlernt diese. Das Vorhandensein von Leaky-Einheiten auf unterschiedlichen Zeitskalen scheint hilfreich für langfristige Abhängigkeiten zu sein (*Mozer*, 1992; *Pascanu et al.*, 2013).

10.9.3 Entfernen von Verbindungen

Ein weiterer Ansatz zur Handhabung langfristiger Abhängigkeiten besteht darin, den Zustand des RNNs auf mehreren Zeitskalen zu organisieren

(*El Hihi und Bengio*, 1996), wobei die Informationen auf den langsameren Zeitskalen ungehinderter über längere Strecken fließen können.

Diese Idee unterscheidet sich vom vorgenannten Konzept der Skip Connections über die Zeit, da hier Verbindungen der Länge Eins aktiv *entfernt* und durch längere Verbindungen ersetzt werden. Derart modifizierte Einheiten werden zur Arbeit auf einer längeren Zeitskala gezwungen. Skip Connections über die Zeit *fügen Kanten hinzu*. Einheiten, die solche neuen Verbindungen erhalten, können ggf. das Arbeiten auf einer langen Zeitskala erlernen, können sich aber auch auf ihre anderen, kurzfristigen Verbindungen konzentrieren.

Es gibt verschiedene Möglichkeiten, mit denen eine Gruppe rekurrenter Einheiten gezwungen werden kann, auf unterschiedlichen Zeitskalen zu arbeiten. Eine besteht darin, die rekurrenten Einheiten zu Leaky-Einheiten zu machen, dabei jedoch den unterschiedlichen, festen Zeitskalen unterschiedliche Einheitengruppen zuzuweisen. Dieser Vorschlag stammt aus *Mozer* (1992) und wurde in *Pascanu et al.* (2013) erfolgreich eingesetzt. Eine weitere Möglichkeit sind explizite und diskrete Updates zu unterschiedlichen Zeitpunkten mit einer unterschiedlichen Häufigkeit für die einzelnen Einheitengruppen. Dies ist der Ansatz aus *El Hihi und Bengio* (1996) und aus *Koutnik et al.* (2014). Er hat sich bei einer Reihe von Benchmark-Datensätzen bewährt.

10.10 Das Long Short-Term Memory und andere Gated RNNs

Zum Zeitpunkt der Abfassung gibt es in praktischen Anwendungen besonders effektive Sequenzmodelle, die als **Gated RNNs** bezeichnet werden. Dazu gehören das **Long Short-Term Memory** (LSTM) und Netze auf Grundlage der **Gated Recurrent Unit**.

Wie Leaky-Einheiten basieren auch Gated RNNs auf der Idee, Pfade in der Zeit zu schaffen, die Ableitungen aufweisen, die weder verschwinden noch explodieren. Leaky-Einheiten verwenden dazu Verbindungsgewichte, die entweder manuell ausgewählte Konstanten oder aber Parameter sind. Gated RNNs generalisieren diesen Ansatz auf Verbindungsgewichte, die sich in jedem Zeitschritt verändern können.

Leaky-Einheiten erlauben dem Netz, über einen langen Zeitraum Informationen zu *akkumulieren*, darunter Nachweise für ein besonderes Merkmal oder eine besondere Kategorie. Sobald die Informationen verwendet wurden,

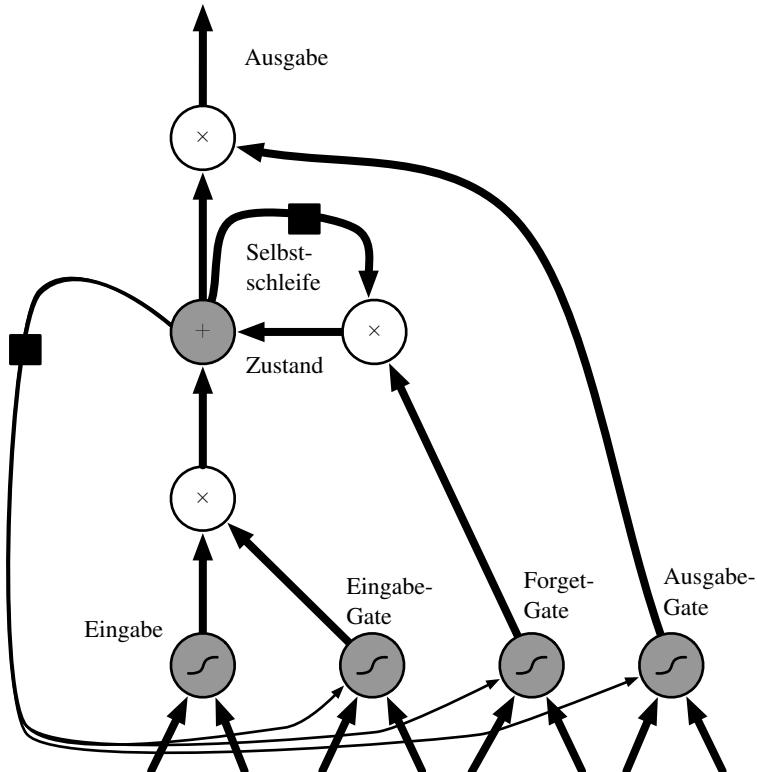


Abbildung 10.16: Schaubild einer LSTM-RNN-Zelle. Zellen sind rekurrent miteinander verbunden und ersetzen die üblichen verdeckten Einheiten normaler RNNs. Ein Eingabemerkmal wird anhand einer gewöhnlichen künstlichen neuronalen Einheit berechnet. Sein Wert kann im Zustand akkumuliert werden, sofern das sigmoide Eingabe-Gate es zulässt. Die Zustandseinheit weist eine lineare Selbstschleife auf, deren Gewicht über das Forget-Gate gesteuert wird. Die Ausgabe der Zelle kann durch das Ausgabe-Gate abgeschaltet werden. Alle Gating-Einheiten weisen eine sigmoide Nichtlinearität auf; die Eingabeeinheiten können eine zwingende Nichtlinearität aufweisen. Die Zustandseinheit kann außerdem als zusätzliche Eingabe für die Gating-Einheiten verwendet werden. Das schwarze Quadrat steht für eine Verzögerung, die einen Zeitschritt lang ist.

kann es für das neuronale Netz jedoch sinnvoll sein, den alten Zustand zu vergessen. Wenn eine Sequenz zum Beispiel aus Subsequenzen besteht und wir möchten, dass eine Leaky-Einheit Nachweise in jeder Sub-Subsequenz sammelt, benötigen wir einen Mechanismus, der den alten Zustand durch Setzen auf Null vergisst. Statt manuell über den Zeitpunkt für das Löschen des Zustands zu entscheiden, soll das neuronale Netz selbst lernen, wann der richtige Zeitpunkt gekommen ist. Genau das tun Gated RNNs.

10.10.1 LSTM

Das geschickte Konzept der Selbstschleifen (engl. *self-loops*) zum Erzeugen von Pfaden, auf denen der Gradient lange Zeit laufen kann, ist ein fundamentaler Baustein für das anfängliche **LSTM-Modell** (*Hochreiter und Schmidhuber, 1997*). Eine wichtige Ergänzung war es, das Gewicht der Selbstschleife kontextabhängig anstatt unveränderlich zu gestalten (*Gers et al., 2000*). Durch ein (durch eine andere verdeckte Einheit) »gated« Gewicht dieser Selbstschleife kann die Zeitskala der Integration dynamisch geändert werden. In diesem Fall heißt das, dass selbst bei einem LSTM mit unveränderlichen Parametern die Zeitskala der Integration auf Grundlage der Eingabesequenz geändert werden kann, da die Zeitkonstanten vom Modell selbst ausgegeben werden. Das LSTM hat sich in vielen Bereichen als extrem erfolgreich erwiesen, darunter in der uneingeschränkten Handschrifterkennung (engl. *unconstrained handwriting recognition*) (*Graves et al., 2009*), der Spracherkennung (*Graves et al., 2013; Graves und Jaitly, 2014*), der Erzeugung von Handschriften (*Graves, 2013*), der maschinellen Übersetzung (*Sutskever et al., 2014*), der Bildbeschriftung (*Kiros et al., 2014b; Vinyals et al., 2014b; Xu et al., 2015*) und beim Parsing (*Vinyals et al., 2014a*).

Abbildung 10.16 zeigt ein Schaubild eines LSTMs. Die entsprechenden Gleichungen der Forward-Propagation sind im Folgenden für die flache Architektur eines RNNs angegeben. Auch tiefere Architekturen wurden erfolgreich eingesetzt (*Graves et al., 2013; Pascanu et al., 2014a*). Statt einer Einheit, die einfach nur eine elementweise Nichtlinearität auf die affine Transformation der Eingaben und rekurrenten Einheiten anwendet, verfügen rekurrente LSTM-Netze über »LSTM-Zellen«, die neben der äußeren Rekurrenz des RNNs eine innere Rekurrenz (eine Selbstschleife) aufweisen. Jede Zelle verfügt über dieselben Eingaben und Ausgaben wie in einem gewöhnlichen RNN, hat aber weitere Parameter und ein System aus Gating-Einheiten, das den Informationsfluss steuert. Die wichtigste Komponente ist die Zustandseinheit $s_i^{(t)}$, die ähnlich den Leaky-Einheiten aus dem vorherigen Abschnitt eine lineare Selbstschleife aufweist. Allerdings wird das

Gewicht der Selbstschleife (oder der zugehörigen Zeitkonstanten) hier über eine sogenannte **Forget-Gate**-Einheit $f_i^{(t)}$ (für Zeitschritt t und Zelle i) gesteuert, die dieses Gewicht auf einen Wert zwischen 0 und 1 einstellt, und zwar über eine sigmoide Einheit:

$$f_i^{(t)} = \sigma \left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} \right), \quad (10.40)$$

mit $\mathbf{x}^{(t)}$ als aktuellem Eingabevektor und $\mathbf{h}^{(t)}$ als aktuellem Vektor der verdeckten Schicht, der die Ausgaben aller LSTM-Zellen enthält, und \mathbf{b}^f , \mathbf{U}^f , \mathbf{W}^f als Verzerrungen, Eingabegewichten bzw. rekurrenten Gewichten für die Forget-Gates. Der interne Zustand der LSTM-Zellen wird somit wie folgt aktualisiert, aber mit einem bedingten Selbstschleifen-Gewicht $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right), \quad (10.41)$$

mit \mathbf{b} , \mathbf{U} und \mathbf{W} als Verzerrungen, Eingabegewichte bzw. rekurrenten Gewichten für die LSTM-Zelle. Die **externe Eingabe-Gate**-Einheit $g_i^{(t)}$ wird auf ähnliche Weise wie das Forget-Gate (mit einer sigmoiden Einheit zur Ermittlung des Gating-Werts zwischen 0 und 1) berechnet, allerdings mit eigenen Parametern:

$$g_i^{(t)} = \sigma \left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right). \quad (10.42)$$

Die Ausgabe $h_i^{(t)}$ der LSTM-Zelle kann auch abgeschaltet werden, nämlich über das **Ausgabe-Gate** $q_i^{(t)}$, das ebenfalls eine sigmoide Einheit für das Gating nutzt:

$$h_i^{(t)} = \tanh(s_i^{(t)}) q_i^{(t)}, \quad (10.43)$$

$$q_i^{(t)} = \sigma \left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right), \quad (10.44)$$

mit den Parametern \mathbf{b}^o , \mathbf{U}^o , \mathbf{W}^o für Verzerrungen, Eingabegewichte bzw. rekurrente Gewichte. Unter den Varianten besteht die Möglichkeit, den Zellzustand $s_i^{(t)}$ als zusätzliche Eingabe (mit dem zugehörigen Gewicht) für die drei Gates der i -ten Einheit zu nutzen (vgl. Abbildung 10.16). Dafür werden drei weitere Parameter benötigt.

Es hat sich gezeigt, dass LSTM-Netze langfristige Abhängigkeiten im Gegensatz zu einfachen rekurrenten Architekturen leichter erlernen können: zunächst an künstlichen Datensätzen, die speziell zum Testen des Lernvermögens langfristiger Abhängigkeiten entwickelt wurden (*Bengio et al.*, 1994; *Hochreiter und Schmidhuber*, 1997; *Hochreiter et al.*, 2001), dann für komplexe Sequenzverarbeitungsaufgaben, in denen eine Leistung auf dem aktuellen Stand der Technik erzielt wurde (*Graves*, 2012; *Graves et al.*, 2013; *Sutskever et al.*, 2014). Bereits untersuchte und verwendete Varianten und Alternativen zum LSTM werden im Folgenden behandelt.

10.10.2 Andere Gated RNNs

Welche Teile der LSTM-Architektur werden wirklich benötigt? Welche anderen erfolgreichen Architekturen könnten entwickelt werden, um dem Netz eine dynamische Kontrolle der Zeitskala und des Vergessenverhaltens unterschiedlicher Einheiten zu ermöglichen?

Einige Antworten auf diese Fragen liefern jüngere Erkenntnisse aus der Arbeit mit Gated RNNs, deren Einheiten auch als Gated Recurrent Units oder kurz GRUs bezeichnet werden (*Cho et al.*, 2014b; *Chung et al.*, 2014, 2015a; *Jozefowicz et al.*, 2015; *Chrupala et al.*, 2015). Der wesentliche Unterschied beim LSTM ist, dass eine einzelne Gating-Einheit gleichzeitig den Vergessensfaktor und die Entscheidung hinsichtlich der Aktualisierung der Zustandseinheit steuert. Die Aktualisierungsgleichungen lauten wie folgt:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right), \quad (10.45)$$

wobei \mathbf{u} für das »Update«-Gate und \mathbf{r} für das »Reset«-Gate steht. Ihr Wert wird wie üblich definiert:

$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right) \quad (10.46)$$

und

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right). \quad (10.47)$$

Die Reset- und Update-Gates können Teile des Zustandsvektors einzeln »ignorieren«. Die Update-Gates agieren wie bedingte Leaky-Integratoren, die jede Dimension linear be- bzw. überwachen können, um sie zum Beispiel

(im einen Extrem des Sigmoids) zu kopieren oder (im anderen Extrem) vollständig zu ignorieren und durch den neuen »Zielzustandswert« zu ersetzen (gegen den der Leaky-Integrator konvergiert). Die Reset-Gates steuern, welche Teile des Zustands beim Berechnen des nächsten Zielzustands verwendet werden, und führen so einen zusätzlichen nichtlinearen Effekt in die Beziehung zwischen dem vergangenen und dem zukünftigen Zustand ein.

Rund um dieses Konzept lassen sich viele weitere Varianten entwickeln. Zum Beispiel könnte die Ausgabe des Reset-Gates (oder Forget-Gates) über mehrere verdeckte Einheiten geteilt werden. Alternativ könnte das Produkt eines globalen Gates (das eine ganze Einheitengruppe, z. B. eine ganz Schicht) und eines lokalen Gates (pro Einheit) verwendet werden, um globale und lokale Kontrolle zu kombinieren. Mehrere Untersuchungen zu architektonischen Variationen von LSTM und GRU haben jedoch keine Variante gefunden, die diesen beiden bei einer Vielzahl von Aufgaben überlegen wäre (*Greff et al.*, 2015; *Jozefowicz et al.*, 2015). *Greff et al.* (2015) haben festgestellt, dass das Forget-Gate ein wesentlicher Bestandteil ist, während *Jozefowicz et al.* (2015) zeigen, dass eine Verzerrung von 1 für das LSTM-Forget-Gate (eine Praxis, die auch von *Gers et al.* (2000) verfochten wird) dazu führt, dass das LSTM so stark wie die besten bisher untersuchten architektonischen Varianten wird.

10.11 Optimierung für langfristige Abhängigkeiten

In den Abschnitten 8.2.5 und 10.7 wurde das Problem der verschwindenden und explodierenden Gradienten beim Optimieren von RNNs über viele Zeitschritte beschrieben.

Ein interessanter Gedanke von *Martens und Sutskever* (2011) besagt, dass die zweiten Ableitungen zeitgleich mit den ersten Ableitungen verschwinden könnten. Optimierungsalgorithmen zweiter Ordnung lassen sich grob als Division der ersten durch die zweite Ableitung auffassen (in höherer Dimension: Multiplikation des Gradienten mit der inversen Hesse-Matrix). Falls die zweite Ableitung in einem ähnlichen Maß abnimmt wie die erste Ableitung, dann bleibt das Verhältnis von erster und zweiter Ableitung eventuell relativ konstant. Leider haben Verfahren zweiter Ordnung viele Nachteile, darunter den hohen Berechnungsaufwand, das Erfordernis eines großen Mini-Batches und eine Tendenz hin zu Sattelpunkten. *Martens und Sutskever* (2011) haben mithilfe von Verfahren zweiter Ordnung vielversprechende Ergebnisse entdeckt. Später stellten *Sutskever et al.* (2013) fest,

dass einfachere Verfahren wie das Nesterow-Momentum bei sorgfältiger Initialisierung ähnliche Ergebnisse erzielen könnten. Weitere Informationen finden Sie in *Sutskever* (2012). Beide Ansätze wurden mittlerweile durch einfaches Anwenden des SGDs (auch ohne Momentum) auf LSTMs ersetzt. Das ist Teil einer anhaltenden Bewegung im Machine Learning: Häufig ist es einfacher, ein Modell zu konzipieren, das sich leicht optimieren lässt, als einen leistungsstärkeren Optimierungsalgorithmus zu entwickeln.

10.11.1 Gradienten-Clipping

Wie in Abschnitt 8.2.4 beschrieben, neigen stark nichtlineare Funktionen wie die durch ein RNN über viele Zeitschritte berechneten, zu Ableitungen, die entweder sehr groß oder sehr klein werden. Das ist in Abbildung 8.3 und Abbildung 10.17 dargestellt: Die Zielfunktion (als Funktion der Parameter) weist eine »Landschaft« mit »Klippen« auf – breite und recht flache Bereiche, die durch winzige Bereiche unterbrochen werden, in denen die Zielfunktion in kurzer Zeit starken Änderungen unterworfen ist und so eine Art Klippe bildet.

Bei sehr großem Parametergradienten entsteht das Problem, dass eine Parameteranpassung im Gradientenabstiegsverfahren die Parameter sehr weit in einen Bereich schleudern kann, in dem die Zielfunktion größer ist, sodass ein Großteil der bisherigen Arbeit zum Erreichen der aktuellen Lösung verloren geht. Der Gradient gibt die Richtung an, die dem steilsten Abstieg in einem infinitesimalen Bereich rund um die aktuellen Parameter entspricht. Außerhalb dieses infinitesimalen Bereichs kann die Kostenfunktion sich wieder als konkav erweisen. Die Aktualisierung muss klein genug gewählt werden, um eine zu starke Aufwärtskrümmung zu vermeiden. Wir nutzen normalerweise Lernraten, die langsam genug abnehmen, sodass aufeinanderfolgende Schritte ungefähr dieselbe Lernrate aufweisen. Eine Schrittweite, die für einen relativ linearen Bereich des Verlaufs geeignet ist, ist in »hüglicheren« Bereichen des Verlaufs häufig ungeeignet und kann im nächsten Schritt zu einer Aufwärtsbewegung führen.

Das sogenannte **Gradienten-Clipping** stellt eine einfache Lösung dar, die sich im Praxiseinsatz schon viele Jahre bewährt hat. Es gibt verschiedene Darstellungen dieser Idee (*Mikolov*, 2012; *Pascanu et al.*, 2013). Eine Möglichkeit besteht darin, den Parametergradienten *elementweise* von einem Mini-Batch abzuschneiden (engl. *to clip*) (*Mikolov*, 2012), und zwar direkt vor der Parameteranpassung. Eine weitere *beschneidet die Norm $\|\mathbf{g}\|$ des Gradienten \mathbf{g}* (*Pascanu et al.*, 2013) direkt vor der Parameteranpassung:

$$\text{if } \|\mathbf{g}\| > v \quad (10.48)$$

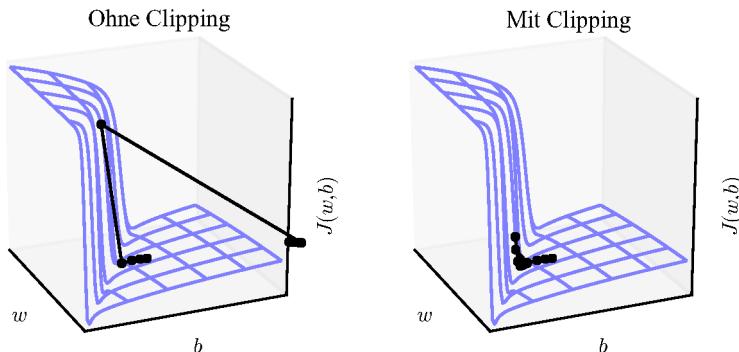


Abbildung 10.17: Beispiel der Auswirkung des Gradienten-Clippings in einem RNN mit den beiden Parametern w und b . Das Gradienten-Clipping kann das Gradientenabstiegsverfahren in der Nähe extrem steiler Klippen »zügeln«. Diese steilen Klippen treten häufig in RNNs dort auf, wo diese sich näherungsweise linear verhalten. Die Klippe ist über eine Reihe von Zeitschritten exponentiell steil, da die Gewichtungsmatrix in jedem Zeitschritt einmal mit sich selbst multipliziert wird. (*Links*) Der Gradientenabstieg ohne Gradienten-Clipping schießt über die Sohle dieser kleinen Schlucht hinaus und erhält dann von der Klippenwand einen sehr großen Gradienten. Der große Gradient schleudert die Parameter auf katastrophale Weise von den Plotachsen weg. (*Rechts*) Im Gradientenabstiegsverfahren mit Gradienten-Clipping fällt die Reaktion auf die Klippe deutlich moderater aus. Es kommt noch immer zu einem Anstieg an der Klippenwand, aber die Schrittweite ist so eingeschränkt, dass kein Wegschleudern aus dem steilen Bereich in Lösungsnähe möglich ist. (Abbildung mit freundlicher Genehmigung von Pascanu et al. (2013) angepasst)

$$\mathbf{g} \leftarrow \frac{\mathbf{g}v}{\|\mathbf{g}\|}, \quad (10.49)$$

wobei v der Schwellenwert für die Norm ist und \mathbf{g} zur Parameteranpassung dient. Da der Gradient aller Parameter (einschließlich unterschiedlicher Parametergruppen wie Gewichten und Verzerrungen) mit einem einzelnen Skalierungsfaktor gemeinsam erneut normalisiert wird, hat das letztgenannte Verfahren den Vorteil, dass jeder Schritt garantiert in Gradientenrichtung erfolgt. Experimente deuten jedoch darauf hin, dass beide Formen ähnlich gut funktionieren. Obwohl die Parameteranpassung dieselbe Richtung wie der echte Gradient aufweist (mit Clipping der Norm des Gradienten), ist die Vektornorm der Parameteranpassung nun eingeschränkt. Dieser eingeschränkte Gradient verhindert einen nachteiligen Schritt im Falle explodierender Gradienten. Wenn die Größe des Gradienten einen Schwellenwert überschreitet, funktioniert ein *Zufallsschritt* fast genauso gut. Ist die Explosion so stark, dass der Gradient numerisch `Inf` oder `Nan` (sprich unendlich

oder »Not a Number«) wird, kann ein Zufallsschritt der Größe v meist zu einem Wegbewegen von der numerisch instabilen Konfiguration führen. Das Clipping der Norm des Gradienten pro Mini-Batch ändert die Gradientenrichtung für einen einzelnen Mini-Batch nicht. Der Durchschnittswert eines durch die Norm beschnittenen Gradienten aus vielen Mini-Batches ist allerdings nicht gleichwertig zum Clipping der Norm des wahren Gradienten (der Gradient, der aus allen Beispielen gebildet wird). Der Beitrag von Beispielen mit großer Norm des Gradienten sowie Beispielen aus demselben Mini-Batch hinsichtlich der endgültigen Richtung ist vermindert. Das widerspricht dem klassischen Mini-Batch-Gradientenabstieg, bei dem die wahre Gradientenrichtung gleich dem Durchschnittswert aller Mini-Batch-Gradienten ist. Anders ausgedrückt: Das klassische, stochastische Gradientenabstiegsverfahren nutzt eine erwartungstreue Gradientenschätzung, wohingegen das Gradientenabstiegsverfahren mit Norm-Clipping eine heuristische Verzerrung einführt, die sich empirisch als nützlich erwiesen hat. Beim elementweisen Clipping stimmt die Update-Richtung nicht mit dem wahren oder dem Mini-Batch-Gradienten überein, aber es handelt sich dennoch um einen Abstieg. Es wurde auch vorgeschlagen (Graves, 2013), den Gradienten aus der Backpropagation zu beschneiden (relativ zu den verdeckten Einheiten), aber bisher wurde kein Vergleich zwischen diesen Varianten veröffentlicht. Wir vermuten, dass sich all diese Verfahren ähnlich verhalten.

10.11.2 Regularisierung für einen besseren Informationsfluss

Das Gradienten-Clipping hilft zwar beim Problem der explodierenden Gradienten, nicht aber bei verschwindenden Gradienten. Hierfür und zur besseren Erfassung langfristiger Abhängigkeiten haben wir angedacht, Pfade im Berechnungsgraphen der aufgefalteten rekurrenten Architektur zu schaffen, entlang derer das Produkt der Gradienten für die Bögen nahezu 1 ist. Ein Ansatz hierfür liegt in LSTMs und anderen Selbstschleifen sowie Gating-Mechanismen, die in Abschnitt 10.10 beschrieben werden. Eine andere Idee ist die Regularisierung oder Beschränkung der Parameter auf eine Weise, die den »Informationsfluss« fördert. Wir möchten vor allem für eine Backpropagation des Gradientenvektors $\nabla_{\mathbf{h}^{(t)}} L$ sorgen, um dessen Größe beizubehalten, und zwar auch dann, wenn die Verlustfunktion nur die Ausgabe am Ende der Sequenz bestraft. Formell soll also

$$(\nabla_{\mathbf{h}^{(t)}} L) \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \quad (10.50)$$

so groß sein wie

$$\nabla_{\mathbf{h}^{(t)}} L. \quad (10.51)$$

Für dieses Ziel schlagen *Pascanu et al.* (2013) folgenden Regularisierer vor:

$$\Omega = \sum_t \left(\frac{\left| \left| (\nabla_{\mathbf{h}^{(t)}} L) \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \right| \right|^2}{\left| \left| \nabla_{\mathbf{h}^{(t)}} L \right| \right|} - 1 \right)^2. \quad (10.52)$$

Die Gradientenberechnung für diesen Regularisierer erscheint vielleicht kompliziert, aber *Pascanu et al.* (2013) schlagen eine Approximation vor, in der wir für die backpropagierten Vektoren $\nabla_{\mathbf{h}^{(t)}} L$ annehmen, dass es sich um Konstanten handelt (für den Zweck dieses Regularisierers; es muss also keine Backpropagation durch sie hindurch erfolgen). Die Experimente mit diesem Regularisierer deuten darauf hin, dass er in Verbindung mit der Heuristik zum Norm-Clipping (die sich um die Gradientenexplosion kümmert) die Bandbreite der Abhängigkeiten, die von einem RNN erlernt werden können, deutlich steigern kann. Da das Gradienten-Clipping die RNN-Dynamik am Rand der explosiven Gradienten hält, ist es besonders wichtig. Ohne Gradienten-Clipping verhindert die Gradientenexplosion ein erfolgreiches Lernen.

Eine wesentliche Schwäche dieses Ansatzes ist die gegenüber einem LSTM geringere Effektivität bei Aufgaben mit sehr vielen Daten, zum Beispiel dem Modellieren von Sprache.

10.12 Explizites Gedächtnis

Intelligenz erfordert Wissen, und der Erwerb von Wissen kann durch Lernen erfolgen, was die Entwicklung umfangreicher tiefer Architekturen angeregt hat. Allerdings gibt es viele verschiedene Arten von Wissen. Manches Wissen ist implizit, unterbewusst und nur schwer in Worte zu fassen – wie man läuft, zum Beispiel, oder worin sich ein Hund optisch von einer Katze unterscheidet. Anderes Wissen ist explizit, deklarativ und relativ einfach in Worte zu fassen – Alltagswissen oder der gesunde Menschenverstand, zum Beispiel »eine Katze ist ein Tier«, aber auch ganz spezielle Fakten, die zum Erreichen der eigenen Ziele benötigt werden, zum Beispiel »Besprechung mit dem Vertriebsteam um 15:00 Uhr in Raum 141«.

Neuronale Netze können implizites Wissen hervorragend speichern, aber beim Memorieren von Fakten haben sie Probleme. Das stochastische Gradientenabstiegsverfahren benötigt viele Beispiele für dieselbe Eingabe, bevor diese in Form von Parametern für das neuronale Netz gespeichert werden

kann. Und selbst dann wird diese Eingabe nicht gerade präzise abgelegt. *Graves et al.* (2014b) vermuteten, dass dies daran liegt, dass es neuronalen Netzen am Gegenstück zum **Arbeitsgedächtnis** mangelt, in dem wir Menschen explizit jene Informationshappen aufbewahren und verändern können, die für ein bestimmtes Ziel von Bedeutung sind. Solche Bausteine für ein explizites Gedächtnis würden unseren Systemen nicht nur das blitzschnelle und »absichtliche« Ablegen und Abrufen bestimmter Fakten ermöglichen, sondern auch ein sequenzielles Schließen daraus. Das Bedürfnis nach neuronalen Netzen, die Informationen in einer Abfolge von Schritten verarbeiten können, wird schon lange als wichtig für die Fähigkeit, Schlüsse zu ziehen, anstatt automatisch und intuitiv auf Eingaben zu reagieren, erachtet (*Hinton*, 1990).

Als Lösungsansatz stellten *Weston et al.* (2014) **Memory-Netze** vor, die eine Reihe von Speicherzellen enthalten, die einen adressierten Zugriff erlauben. Memory-Netze erforderten ursprünglich ein Überwachungssignal, das ihnen Anweisungen zur Verwendung der Speicherzellen gibt. *Graves et al.* (2014b) stellten die **neuronale Turing-Maschine** vor, die das Lesen und Schreiben beliebiger Inhalte aus den und in die Speicherzellen ohne explizite Überwachung der auszuführenden Aktionen erlernen kann, sodass ein Ende-zu-Ende-Training ohne dieses Überwachungssignal möglich ist; dabei wird ein inhaltsbasierter, weicher Aufmerksamkeitsmechanismus eingesetzt (siehe *Bahdanau et al.* [2015] und Abschnitt 12.4.5.1). Diese weiche Adressierung wurde auch für ähnliche Architekturen zum Standard, die algorithmische Mechanismen auf eine Art und Weise emulieren, die noch immer eine Optimierung auf Gradientenbasis erlaubt (*Sukhbaatar et al.*, 2015; *Joulin und Mikolov*, 2015; *Kumar et al.*, 2015; *Vinyals et al.*, 2015a; *Grefenstette et al.*, 2015).

Jede Speicherzelle ähnelt einer Erweiterung der Speicherzellen in LSTMs und GRUs. Der Unterschied besteht darin, dass das Netz einen internen Zustand ausgibt, der auswählt, aus welcher Zelle gelesen bzw. in welche Zelle geschrieben wird, so wie Speicherzugriffe für Lese- und Schreibvorgänge in einem digitalen Rechner mithilfe bestimmter Adressen erfolgen.

Es ist schwierig, Funktionen zu optimieren, die exakt ganzzahlige Adressen erzeugen. Angesichts dieses Problems nutzen neuronale Turing-Maschinen beim Lesen oder Schreiben tatsächlich viele Speicherzellen gleichzeitig. Beim Lesen wird das gewichtete Mittel vieler Zellen gebildet. Beim Schreiben werden mehrere Zellen in unterschiedlichem Umfang geändert. Die Koeffizienten dieser Operationen werden so gewählt, dass nur eine kleine Anzahl von Zellen betroffen ist, zum Beispiel mithilfe einer softmax-Funktion. Die Verwendung dieser Gewichtungen mit von Null verschiedenen Ableitungen

ermöglicht den Funktionen die mittels Gradientenabstiegsverfahren optimierte Zugriffskontrolle auf den Speicher (das Gedächtnis). Der Gradient dieser Koeffizienten gibt an, ob diese jeweils erhöht oder verringert werden sollten, aber der Gradient ist normalerweise nur für jene Speicheradressen groß, die einen großen Koeffizienten erhalten.

Diese Speicherzellen sind üblicherweise so verbessert, dass sie einen Vektor enthalten, nicht nur einen einzelnen Skalar wie bei den LSTM- oder GRU-Speicherzellen. Es gibt zwei Gründe, die für diese Vergrößerung der Speicherzellen sprechen: Erstens haben wir den Aufwand für den Zugriff auf eine Speicherzelle erhöht. Wir wenden den Berechnungsaufwand auf, um einen Koeffizienten für viele Zellen zu erzeugen, aber wir erwarten, dass diese Koeffizienten sich auf eine kleine Anzahl von Zellen konzentrieren. Durch Auslesen eines Vektorwerts anstelle eines Skalarwerts können wir einen Teil dieses Aufwands zurückgewinnen. Außerdem nutzen wir vektorwertige Speicherzellen, weil diese eine **Inhaltsbasierte Adressierung** ermöglichen, in der die Gewichte zum Lesen oder Schreiben einer Zelle eine Funktion dieser Zelle darstellt. Mit vektorwertigen Zellen können wir einen vollständig vektorwertigen Speicher abrufen, wenn wir ein Muster erzeugen können, das einigen – aber nicht allen – seiner Elemente entspricht. Das entspricht Ihrer Fähigkeit, sich an einen Liedtext zu erinnern, wenn Sie nur wenige Wörter daraus hören. Eine inhaltsbasierte Leseanweisung könnte zum Beispiel so lauten: »Rufe den Text für das Lied mit dem Refrain *We all live in a yellow submarine* ab.« Die inhaltsbasierte Adressierung ist hilfreicher, wenn die abzurufenden Objekte groß sind – wäre jeder Buchstabe des Liedtextes in einer eigenen Speicherzelle abgelegt, könnten wir nicht auf diese Weise danach suchen. Im Gegensatz dazu darf die **ortsbasierte Adressierung** nicht auf den Speicherinhalt verweisen. Ein Beispiel für eine ortsbasierte Leseanweisung lautet: »Rufe den Text für das Lied im Speicherplatz 347 ab.« Generell, aber insbesondere auch bei sehr kleinen Speicherzellen, hat die ortsbasierte Adressierung daher ihre Berechtigung.

Wird der Inhalt einer Speicherzelle in den meisten Zeitschritten kopiert (nicht vergessen), dann können die darin enthaltenen Informationen vorwärts durch die Zeit propagiert werden, während die Gradienten rückwärts durch die Zeit propagiert werden, ohne dass es zu einem Verschwinden oder einer Explosion kommt.

Abbildung 10.18 zeigt ein explizites Gedächtnis, in dem ein »neuronales Aufgabennetz« mit einem Speicherteil verbunden ist. Obschon das neuronale Aufgabennetz in Gestalt eines RNNs oder eines Feedforward-Netzes auftreten kann, ist das Gesamtsystem stets ein RNN. Das Aufgabennetz kann bestimmte Speicheradressen zum Lesen oder Schreiben frei wählen. Mit

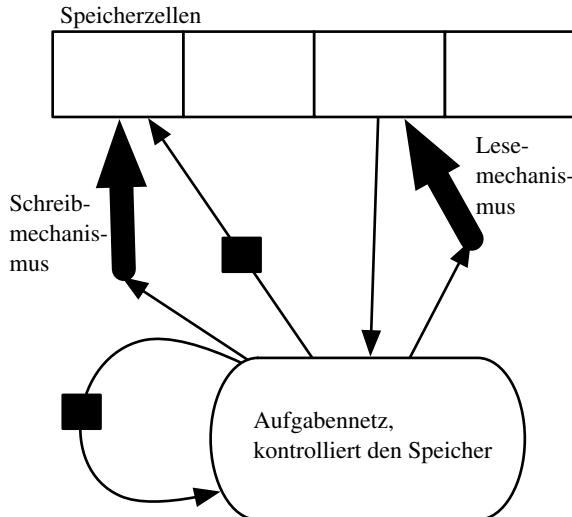


Abbildung 10.18: Aufbau eines Netzes mit explizitem Gedächtnis, das einige der wesentlichen Designelemente der neuronalen Turing-Maschine aufweist. In dieser Abbildung unterscheiden wir zwischen dem »Repräsentationsteil« des Modells (dem »Aufgabennetz«, hier ein RNN im unteren Teil) und dem »Speicher-« oder »Gedächtnisteil« des Modells (der Zellreihe) zum Speichern der Fakten. Das Aufgabennetz lernt, den Speicher zu »kontrollieren« und entscheidet, wo im Speicher jeweils gelesen und geschrieben wird (anhand der Lese- und Schreibmechanismen, die durch fette Pfeile auf die jeweiligen Adressen dargestellt sind).

einem expliziten Gedächtnis scheinen Modelle in der Lage zu sein, Aufgaben zu erlernen, die gewöhnlichen RNNs oder LSTM-RNNs verschlossen bleiben. Ein Grund für diesen Vorteil ist möglicherweise, dass Informationen und Gradienten über sehr lange Zeiträume propagiert werden können (vorwärts bzw. rückwärts durch die Zeit).

Als Alternative zur Backpropagation mittels gewichteter Mittel von Speicherzellen können wir den Koeffizienten für die Speicheradressierung als Wahrscheinlichkeiten interpretieren und stochastisch nur eine Zelle lesen (Zaremba und Sutskever, 2015). Für die Optimierung von Modellen, die diskrete Entscheidungen treffen, werden spezielle Optimierungsalgorithmen benötigt. Diese werden in Abschnitt 20.9.1 beschrieben. Bisher ist das Trainieren dieser stochastischen Architekturen, die diskrete Entscheidungen treffen, noch schwieriger als das Trainieren deterministischer Algorithmen, die weiche Entscheidungen treffen.

Ob nun weich (Backpropagation ist erlaubt) oder stochastisch und hart: Der Mechanismus zum Auswählen einer Adresse ist der Form nach identisch

zum **Aufmerksamkeitsmechanismus** (engl. *attention mechanism*), der zuvor im Rahmen der maschinellen Übersetzung vorgestellt wurde (*Bahdanau et al.*, 2015) und in Abschnitt 12.4.5.1 behandelt wird. Das Konzept der Aufmerksamkeitsmechanismen für neuronale Netze kam noch früher im Rahmen der Erzeugung von Handschriften auf (*Graves*, 2013), mit der Bedingung an den Aufmerksamkeitsmechanismus, sich nur zeitlich vorwärts durch die Sequenz zu bewegen. Im Fall der maschinellen Übersetzung und der Memory-Netze kann der Aufmerksamkeitsfokus in jedem Schritt frei auf eine ganz andere Stelle als im vorhergehenden Schritt bewegt werden.

RNNs bieten eine Möglichkeit, Deep Learning auch für sequenzielle Daten zu nutzen. Sie sind das neueste der großen Hilfsmittel in unserem Deep-Learning-Werkzeugkasten. Wir widmen uns nun der Auswahl und Nutzung dieser Werkzeuge und ihrer Anwendung in der Praxis.

11

Praxisorientierte Methodologie

Für eine erfolgreiche Anwendung von Deep-Learning-Techniken ist mehr nötig als nur eine gute Kenntnis der verfügbaren Algorithmen und ihrer Funktionsweise. Sie müssen auch wissen, wie ein Algorithmus für eine bestimmte Anwendung ausgewählt wird, wie in Experimenten das Feedback analysiert wird und wie entsprechende Anpassungen erfolgen, um ein Machine-Learning-System zu verbessern. Bei der täglichen Entwicklung von Machine-Learning-Systemen müssen Sie als Entwickler entscheiden, ob Sie mehr Daten benötigen, die Modellkapazität erhöhen oder verringern, regularisierende Merkmale hinzufügen oder entfernen, die Modelloptimierung verbessern, die approximative Inferenz in einem Modell verbessern oder ein Debugging der Softwareimplementierung des Modells durchführen sollten. All diese Operationen durchzuführen, ist zumindest zeitaufwendig, daher ist es wichtig, die richtige Vorgehensweise auszumachen, anstatt blind zu raten.

Im Großteil dieses Buchs geht es um die verschiedenen Machine-Learning-Modelle, Trainingsalgorithmen und Zielfunktionen. Das mag vielleicht den Eindruck erwecken, dass es für Machine-Learning-Experten vor allem darauf ankommt, möglichst viele Machine-Learning-Verfahren zu kennen und mathematisch vielseitig begabt zu sein. In der Praxis ist es meist Erfolg versprechender, einen gängigen Algorithmus korrekt anzuwenden, als einen obskuren Algorithmus nachlässig einzusetzen. Die korrekte Anwendung eines Algorithmus steht und fällt mit der Umsetzung einer recht einfachen Methodologie. Anregungen für viele der Empfehlungen in diesem Kapitel sind *Ng* (2015) entnommen.

Wir empfehlen Ihnen, den folgenden praxisorientierten Designprozess einzuhalten:

- Legen Sie Ihre Ziele fest: Welche Fehlerkennzahlen sollen verwendet werden? Was ist der betreffende Zielwert? Diese Ziele und Fehlerkennzahlen müssen natürlich zu dem Problem passen, das die Anwendung lösen soll.
- Legen Sie so früh wie möglich einen funktionsfähigen Ende-zu-Ende-Prozess fest, der auch die Schätzung geeigneter Performance-Kriterien umfasst.
- Konfigurieren Sie das System gut, um Engpässe bei der Leistung aufzuspüren. Diagnostizieren Sie, welche Komponenten schlechter als erwartet abschneiden – und ob diese Minderleistung eine Folge von Überanpassung, Unteranpassung oder Mängeln in den Daten oder der Software ist.
- Nehmen Sie wiederholt geringfügige Änderungen aufgrund der Ergebnisse der Konfiguration vor: Sammeln Sie zum Beispiel neue Daten, passen Sie die Hyperparameter an oder ändern Sie die Algorithmen.

Wir verwenden als Beispiel das Transkriptionssystem von Street View für Hausnummern (*Goodfellow et al.*, 2014d). Diese Anwendung soll Gebäude zu Google Maps hinzufügen. Street-View-Fahrzeuge fotografieren die Gebäude und zeichnen dabei für jedes Foto die GPS-Koordinaten auf. Ein CNN erkennt die Hausnummern in den Fotos, sodass die Google-Maps-Datenbank die Adresse korrekt verorten kann. Die Geschichte der Entstehung dieser kommerziellen Anwendung ist ein nützliches Beispiel für das Befolgen unserer Methodologie.

Es folgt eine Beschreibung der einzelnen Prozessschritte.

11.1 Performance-Kriterien

Das Festlegen der Ziele anhand von Fehlerkennzahlen ist ein notwendiger erster Schritt, da diese Fehlerkennzahlen als Richtschnur für alle weiteren Aktionen dienen. Sie müssen auch wissen, welches Leistungsniveau in etwa erwünscht ist.

Bedenken Sie, dass es in den meisten Anwendungen unmöglich ist, den absoluten Null-Fehler-Punkt zu erreichen. Der Bayes-Fehler definiert die kleinste Fehlerquote, die Sie bestenfalls erreichen können, selbst wenn Ihnen

unendlich viele Trainingsbeispiele und die wahre Wahrscheinlichkeitsverteilung zur Verfüzung stehen. Der Grund dafür ist, dass Ihre Eingabemerkmale eventuell nicht alle Informationen über die AusgabevARIABLE enthalten oder das System intrinsisch stochastisch ist. Auch sind die verfügbaren Trainingsdaten endlich.

Der Umfang der Trainingsdatenmenge kann aus verschiedenen Gründen eingeschränkt sein: Wenn Sie das bestmögliche praxistaugliche Produkt oder eine ebensolche Dienstleistung erstellen möchten, können Sie meist mehr Daten sammeln. Dennoch gilt es abzuwägen, welchen Preis (für die Erfassung weiterer Daten) eine weitere Reduzierung des Fehlers hat. Daten zu erfassen kostet Zeit, Geld oder menschliches Leid (zum Beispiel im Rahmen invasiver medizinischer Eingriffe). Wenn Sie die wissenschaftliche Frage beantworten möchten, welcher Algorithmus hinsichtlich eines festen Benchmarks besser abschneidet, entscheidet die Spezifikation dieses Benchmarks normalerweise über die Trainingsdatenmenge, sodass Sie keine zusätzlichen Daten sammeln dürfen.

Wie lässt sich ein angemessenes Leistungsniveau bestimmen? Im akademischen Umfeld kennen wir meist einen Schätzwert für die Fehlerquote auf Grundlage zuvor veröffentlichter Benchmark-Ergebnisse. Im Praxisalltag haben wir eine ungefähre Vorstellung von der Fehlerquote, die eine sichere wirtschaftliche oder für Kunden ansprechende Anwendung ermöglicht. Sobald Sie eine realistische Wunschfehlerquote bestimmt haben, zielen Ihre Designentscheidungen auf das Erreichen dieser Fehlerquote ab.

Ein weiterer wichtiger Aspekt neben dem Zielwert für die Performance-Kriterien ist die Wahl des zu verwendenden Leistungsmaßes. Sie können die Wirksamkeit einer kompletten Anwendung, die Machine-Learning-Komponenten enthält, mit mehreren Kennzahlen beurteilen. Diese unterscheiden sich üblicherweise von der Kostenfunktion, die zum Trainieren des Modells genutzt wird. Wie in Abschnitt 5.1.2 beschrieben, wird meist die Korrektklassifikationsrate – oder äquivalent die Fehlerquote – eines Systems gemessen.

Allerdings spielen in vielen Anwendungen komplexere Kriterien eine Rolle.

Manchmal ist eine Art von Fehler kostspieliger und aufwendiger als ein anderer. Ein Beispiel: Ein Spamfilter für E-Mails kann zulässige E-Mails fälschlicherweise als Spam einstufen oder eine Spammachricht im Posteingang anzeigen. Es ist schlimmer, eine zulässige E-Mail zu blockieren, als eine fragwürdige Nachricht freizugeben. Statt nun die Fehlerquote des Spamfilters zu messen, interessieren wir uns eher für einen Gesamtaufwand (Kosten),

bei dem das Blockieren zulässiger Nachrichten härter bestraft wird als das Zulassen von Spammnachrichten.

Manchmal möchten wir einen binären Klassifikator trainieren, der ein seltenes Ereignis erkennen soll. Es könnte sich zum Beispiel um einen medizinischen Test auf eine seltene Krankheit handeln. Nehmen wir einmal an, diese Krankheit taucht nur einmal bei einer Million Menschen auf. Mit einem einfachen Kunstgriff erreichen wir eine Korrektklassifikationsrate von 99,9999 Prozent: Wir müssen den Klassifikator lediglich anweisen, stets auszugeben, dass die Krankheit nicht vorliegt. Damit ist die Korrektklassifikationsrate als Leistungsmaß für ein solches System nicht brauchbar. Wir können das Problem aber durch Bestimmen von **Genauigkeit** und **Trefferquote** (engl. *precision and recall*) lösen. Die Genauigkeit ist der Anteil der vom Modell gemeldeten Erkennungen, die korrekt waren, die Trefferquote dagegen der Anteil der wahren Ereignisse, die erkannt wurden. Ein Detektor, der einfach behauptet, dass niemand an der Krankheit leidet, hätte zwar eine perfekte Genauigkeit, aber eine Trefferquote von Null. Ein Detektor, der behauptet, dass alle Personen an der Krankheit leiden, hat eine perfekte Trefferquote – aber die Genauigkeit wäre gleich dem Anteil der Menschen, die an der Krankheit leiden (in unserem Beispiel einer Krankheit, die ein Mensch von einer Million hat, demzufolge 0,0001 Prozent). In Verbindung mit Genauigkeit und Trefferquote wird meist ein **PR-Diagramm** (Abkürzung für den englischen Begriff *precision-recall curve*) erstellt; die Genauigkeit ist auf der y -Achse eingetragen, die Trefferquote auf der x -Achse. Der Klassifikator erzeugt ein Maß, das höher ist, wenn das zu erkennende Ereignis aufgetreten ist. Ein Beispiel: Ein Feedforward-Netz zur Erkennung einer Krankheit gibt $\hat{y} = P(y = 1 | \mathbf{x})$ aus und schätzt so die Wahrscheinlichkeit, dass eine Person, deren medizinische Ergebnisse durch das Merkmal \mathbf{x} beschrieben wird, an der Krankheit leidet. Wir möchten eine Erkennung melden, sobald dieses Maß einen bestimmten Schwellenwert überschreitet. Durch Variieren des Schwellenwerts können wir zwischen Genauigkeit und Trefferquote abwägen. In vielen Fällen möchten wir die Leistung des Klassifikators als einzelnen Wert angeben, nicht als Kurve. Dazu können wir die Genauigkeit p und die Trefferquote r mit folgender Formel in ein **F-Maß** umrechnen:

$$F = \frac{2pr}{p + r}. \quad (11.1)$$

Eine weitere Möglichkeit besteht darin, die Gesamtfläche unter dem PR-Diagramm anzugeben.

In einigen Anwendungen kann es vorkommen, dass das Machine-Learning-System eine Entscheidung verweigert. Das ist nützlich, wenn der Machine-

Learning-Algorithmus abschätzen kann, wie sicher er hinsichtlich einer Entscheidung sein sollte, vor allem dann, wenn eine falsche Entscheidung Nachteile mit sich bringen kann und in diesem Fall einem Menschen überlassen werden soll. Das Transkriptionssystem von Street View hält auch hierfür ein Beispiel bereit. Es soll Hausnummern aus einem Foto transkribieren, um den Standort, an dem das Foto aufgenommen wurde, der korrekten Adresse in einer Karte zuzuordnen. Da der Wert der Karte erheblich abnimmt, wenn sie fehlerhaft ist, dürfen Adressen nur dann hinzugefügt werden, wenn die Transkription korrekt erfolgt ist. Wenn das Machine-Learning-System der Meinung ist, die korrekte Transkription mit einer geringeren Wahrscheinlichkeit als ein Mensch zu bestimmen, dann ist es nur logisch, die Transkription einem Menschen zu überlassen. Natürlich besteht der Sinn eines Machine-Learning-Systems darin, die Menge der Fotos, die von menschlichen Bearbeitern geprüft werden müssen, erheblich zu reduzieren. Ein häufig verwendetes Leistungsmaß in diesem Zusammenhang ist die **Deckung** (engl. *coverage*). Sie gibt den Anteil der Beispiele an, für die das Machine-Learning-System eine Antwort liefern kann. Es ist möglich, Deckung und Korrektklassifikationsrate gegeneinander abzuwägen: Für eine Korrektklassifikationsrate von 100 Prozent muss einfach nur die Verarbeitung aller Beispiele abgelehnt werden – aber das hat eine Deckung von 0 Prozent zur Folge. Im Street-View-Kontext sollte eine Korrektklassifikationsrate der Transkription auf menschlichem Niveau bei gleichzeitiger Deckung von 95 Prozent erreicht werden. Menschen erledigen diese Aufgabe mit einer Korrektklassifikationsrate von 98 Prozent.

Es sind noch viele weitere Kriterien denkbar. Wir können zum Beispiel Click-Through-Raten (auch: Klickraten) messen, Zufriedenheitsumfragen sammeln usw. Viele spezielle Anwendungsbereiche weisen ebenso spezielle Kriterien auf.

Es ist wichtig, im Vorfeld zu ermitteln, welches Performance-Kriterium (oder Leistungsmaß) verbessert werden muss, und sich dann darauf zu konzentrieren. Ohne klar definierte Ziele lässt sich nur schwer sagen, welche Änderungen zu einer positiven Entwicklung des Machine-Learning-Systems führen und welche nicht.

11.2 Default-Baseline-Modell

Nach der Wahl der Performance-Kriterien und Ziele ist der nächste Schritt in der Praxis, für Ihre Anwendung so früh wie möglich ein angemessenes durchgängiges System aufzubauen. In diesem Abschnitt geben wir für unter-

schiedliche Gegebenheiten Empfehlungen dafür, welche Algorithmen Sie bei der Herangehensweise an Ihre erste Baseline verwenden können. Bedenken Sie, dass die Deep-Learning-Forschung rasante Fortschritte macht; es wird also vermutlich schon bessere Standardalgorithmen geben, sobald diese Zeilen geschrieben sind.

Abhängig von der Komplexität Ihres Problems können Sie im ersten Ansatz sogar ganz ohne Deep Learning arbeiten. Wenn das Problem durch die passende Auswahl weniger linearer Gewichte gelöst werden kann, reicht vielleicht ein einfaches statistisches Modell wie die logistische Regression.

Wenn Sie wissen, dass Ihr Problem in die Kategorie der »harten KI« fällt, also Objekterkennung, Spracherkennung, maschinelle Übersetzung usw., dann dürfte ein geeignetes Deep-Learning-Modell ein passender Ausgangspunkt sein.

Wählen Sie zunächst die allgemeine Modellkategorie auf Basis der Datenstruktur. Wenn Sie ein überwachtes Lernen mit Vektoren mit fester Größe als Eingabe durchführen möchten, passt ein Feedforward-Netz mit vollständig verbundenen Schichten. Wenn Sie wissen, dass die Eingabe bekannte topologische Strukturen aufweist (es sich zum Beispiel um ein Bild handelt), greifen Sie zu einem CNN. In diesen Fällen sollten Sie mit einer Art stückweise linearer Einheit (ReLUs oder ihren Generalisierungen wie Leaky ReLUs, PReLUs oder Maxout) beginnen. Handelt es sich bei Ein- und Ausgabe um Sequenzen, wählen Sie ein Gated RNN (LSTM oder GRU).

Eine sinnvolle Wahl beim Optimierungsalgorithmus ist das stochastische Gradientenabstiegsverfahren (SGD) mit Momentum und einer Verringerung der Lernrate (beliebte Verringerungs- bzw. Decay-Konzepte, die je nach Problemstellung besser oder schlechter abschneiden, sind eine lineare Verringerung bis zum Erreichen einer festen Mindestlernrate, eine exponentielle Verringerung oder eine Verringerung der Lernrate um einen Faktor 2–10, sobald der Validierungsfehler abflacht). Eine andere gute Alternative ist der Adam-Algorithmus (siehe Abschnitt 8.5.3). Die Batch-Normalisierung kann sich erheblich auf die Optimierungsleistung auswirken, insbesondere in CNNs und Netzen mit sigmoiden Nichtlinearitäten. Es ist zwar durchaus vernünftig, die Batch-Normalisierung von der ersten Baseline auszunehmen, aber sie sollte ebenso schnell verwendet werden, wenn die Optimierung ein Problem darzustellen scheint.

Sofern Ihre Trainingsdatenmenge nicht zig Millionen Beispiele enthält, sollten Sie von Anfang an eine schwache Form der Regularisierung verwenden. Der frühe Abbruch passt eigentlich immer und überall. Dropout ist ein hervorragender Regularisierer, der sich leicht implementieren lässt und

mit vielen Modellen und Trainingsalgorithmen kompatibel ist. Auch die Batch-Normalisierung kann manchmal den Generalisierungsfehler reduzieren, sodass auf Dropout verzichtet werden kann, denn der Schätzwert der statistischen Größe für die Normalisierung jeder Variable ist rauschbehaftet.

Ahnelt Ihre Aufgabe einer bereits ausführlich untersuchten Aufgabe, dann tun Sie gut daran, zunächst das beste bekannte Modell samt Algorithmus zu übernehmen. Sie können sogar ein im Rahmen jener Aufgabe trainiertes Modell kopieren. Es ist zum Beispiel üblich, die Merkmale aus einem anhand von ImageNet trainierten CNN zu kopieren, um andere Aufgaben im Bereich der Computer Vision zu lösen (*Girshick et al.*, 2015).

Oft wird die Frage gestellt, ob man von Beginn an unüberwachtes Lernen einsetzen sollte (eine Beschreibung finden Sie in Teil III). Das kommt auf das Anwendungsgebiet an. Zum Teil, zum Beispiel bei der Verarbeitung natürlicher Sprache, bringt unüberwachtes Lernen – beispielsweise das Erlernen unüberwachter Wort-Embeddings – erhebliche Vorteile mit sich. In anderen Gebieten wie der Computer Vision bieten aktuelle Verfahren für unüberwachtes Lernen keine Vorteile. Davon ausgenommen ist das halbüberwachte Lernen, sofern die Anzahl der mit Labels gekennzeichneter Beispiele sehr klein ist (*Kingma et al.*, 2014; *Rasmus et al.*, 2015). Wenn unüberwachtes Lernen im Kontext Ihrer Anwendung als wichtig erkannt wurde, sollten Sie es in der ersten Ende-zu-Ende-Baseline berücksichtigen. Ansonsten verwenden Sie im ersten Versuch nur dann unüberwachtes Lernen, wenn die zu lösende Aufgabe ebenfalls unüberwacht ist. Sie können es später immer noch mit unüberwachtem Lernen probieren, falls Sie feststellen, dass die anfängliche Baseline zur Überanpassung führt.

11.3 Prüfen, ob mehr Daten gesammelt werden sollen

Nachdem das erste Ende-zu-Ende-System geschaffen ist, müssen Sie die Leistung des Algorithmus messen und herausfinden, wie sie gesteigert werden kann. Viele Einsteiger ins Machine Learning versuchen, Verbesserungen durch Ausprobieren vieler unterschiedlicher Algorithmen vorzunehmen. Allerdings ist es oft sinnvoller, weitere Daten zu sammeln, als Verbesserungen am Lernalgorithmus vorzunehmen.

Wie entscheidet man, ob man mehr Daten sammeln sollte? Bestimmen Sie zunächst, welche Leistung auf der Trainingsdatenmenge annehmbar ist. Ist die Leistung auf der Trainingsdatenmenge schlecht, nutzt der Lernalgorithmus die bereits verfügbaren Trainingsdaten nicht – es gibt also

keinen Grund dafür, zusätzliche Daten zu erfassen. Versuchen Sie stattdessen, die Modellgröße durch Hinzufügen weiterer Schichten oder zusätzlicher verdeckter Einheiten in den Schichten zu erhöhen. Versuchen Sie auch, den Lernalgorithmus zu verbessern – zum Beispiel durch Abstimmen des Hyperparameters für die Lernrate. Falls große Modelle und sorgfältig abgestimmte Optimierungsalgorithmen nicht gut funktionieren, liegt das Problem möglicherweise in der *Qualität* der Trainingsdaten. Vielleicht sind diese zu verrauscht oder enthalten nicht die richtigen Eingaben zum Vorhersagen der gewünschten Ausgaben. Dann beginnen Sie am besten von vorn und sammeln bessere Daten oder solche mit mehr Merkmalen.

Wenn die Leistung auf der Trainingsdatenmenge annehmbar ist, dann messen Sie die Leistung auf einer Testdatenmenge. Ist die Leistung auch auf der Testdatenmenge annehmbar, bleibt nicht mehr viel zu tun. Schneidet das System dagegen mit der Testdatenmenge sehr viel schlechter ab als mit der Trainingsdatenmenge, ist das Sammeln zusätzlicher Daten eine der wirkungsvollsten Lösungen. Die wesentlichen Überlegungen dabei sind der Aufwand und die Durchführbarkeit des Sammelns weiterer Daten, der Aufwand und die Durchführbarkeit einer Reduzierung des Testfehlers auf anderem Wege sowie die Menge der Daten, die vermutlich benötigt wird, um die Leistung hinsichtlich der Testdatenmenge deutlich zu steigern. Bei großen Internetunternehmen mit Millionen oder gar Milliarden von Nutzern ist das Erfassen großer Datensätze ein gangbarer Weg – und die Kosten hierfür sind häufig deutlich geringer als bei den Alternativen. Daher sind in diesem Umfeld mehr Trainingsdaten fast immer die Antwort. So war die Entwicklung großer mit Labels gekennzeichneter Datensätze einer der wichtigsten Faktoren bei der Lösung der Objekterkennung. In anderen Bereichen, zum Beispiel für medizinische Anwendungen, kann das Erfassen weiterer Daten sehr teuer oder quasi unmöglich sein. Eine einfache Alternative zum Sammeln zusätzlicher Daten ist das Reduzieren der Modellgröße oder das Verbessern der Regularisierung durch Anpassen der Hyperparameter, beispielsweise der Koeffizienten für den Weight Decay, oder das Hinzufügen von Regularisierungsverfahren wie Dropout. Wenn die Lücke zwischen der Leistung anhand der Trainings- und Testdaten auch nach der Abstimmung der Hyperparameter zur Regularisierung noch immer zu groß ist, ist es ratsam, mehr Daten zu sammeln.

In diesem Zusammenhang stellt sich auch die Frage, wie groß die Menge zusätzlich erfasster Daten sein sollte. Es ist hilfreich, die Beziehung zwischen der Größe der Trainingsdatenmenge und dem Generalisierungsfehler wie in Abbildung 5.4 zu plotten. Durch Extrapolieren der Kurven lässt sich vorhersagen, wie viele Trainingsdaten noch benötigt werden, um ein gewisses

Leistungsniveau zu erreichen. Meist wirkt es sich nicht auf den Generalisierungsfehler aus, wenn man nur einen Bruchteil der vorhandenen Beispiele hinzufügt. Sie sollten die Trainingsdatenmenge daher logarithmisch angehen, zum Beispiel indem Sie die Anzahl der Beispiele zwischen den einzelnen Experimenten jeweils verdoppeln.

Wenn das Erfassen von mehr Daten nicht möglich ist, bleibt zum Verbessern des Generalisierungsfehlers nur eine Verbesserung des Lernalgorithmus selbst. Dies ist allerdings ein Forschungsfeld und somit nicht Teil unserer Tipps für die Praxis.

11.4 Auswählen von Hyperparametern

Die meisten Deep-Learning-Algorithmen weisen mehrere Hyperparameter auf, die das Verhalten des Algorithmus in vielen Aspekten beeinflussen. Einige dieser Hyperparameter beeinflussen den Zeit- und Speicherbedarf für das Ausführen des Algorithmus. Andere beeinflussen die im Rahmen des Trainingsverfahrens entstehende Modellqualität und dessen Fähigkeit, für neue Eingaben auf korrekte Ergebnisse zu schließen.

Es gibt zwei grundlegende Ansätze für die Wahl dieser Hyperparameter: Man kann sie manuell oder automatisch auswählen. Für die manuelle Auswahl von Hyperparametern ist ein Verständnis ihrer Funktion sowie der Abläufe für eine gute Fähigkeit zur Generalisierung eines Machine-Learning-Modells erforderlich. Mit automatischen Algorithmen zur Auswahl der Hyperparameter kann man auf viel Vorwissen verzichten, allerdings sind diese meist deutlich rechenaufwendiger.

11.4.1 Manuelle Anpassung der Hyperparameter

Für das manuelle Setzen von Hyperparametern müssen Sie den Zusammenhang zwischen den Hyperparametern, dem Trainingsfehler, dem Generalisierungsfehler und den Berechnungsressourcen (Speicher und Laufzeit) verstehen. Sie benötigen also eine solide Grundlage hinsichtlich der fundamentalen Konzepte und Ideen, die die tatsächliche Kapazität eines Lernalgorithmus betreffen (vgl. Kapitel 5).

Das Ziel der manuellen Suche nach Hyperparametern ist es normalerweise, den niedrigsten Generalisierungsfehler für eine bestimmte Laufzeit und einen verfügbaren Speicherplatz zu finden. Wir gehen hier nicht darauf ein, wie die verschiedenen Hyperparameter die Laufzeit und den Speicherbedarf beeinflussen, da diese Aspekte stark plattformabhängig sind.

Primäres Ziel bei der manuellen Hyperparameter-Suche ist eine Anpassung der tatsächlichen Kapazität des Modells an die Komplexität der Aufgabe. Die tatsächliche Kapazität wird durch drei Faktoren eingeschränkt: die repräsentative Kapazität des Modells, die Fähigkeit des Lernalgorithmus zur erfolgreichen Minimierung der für das Modelltraining verwendeten Kostenfunktion sowie der Grad, zu dem die Kostenfunktion und das Trainingsverfahren das Modell regularisieren. Ein Modell mit mehr Schichten und mehr verdeckten Einheiten pro Schicht hat eine höhere repräsentative Kapazität – kann also kompliziertere Funktionen darstellen. Es ist aber nicht zwingend in der Lage, all diese Funktionen zu erlernen, denn vielleicht kann der Trainingsalgorithmus nicht erkennen, dass bestimmte Funktionen einen wertvollen Beitrag zum Minimieren der Kostenfunktion leisten, oder Regularisierungsterme wie Weight Decay verbieten einige dieser Funktionen.

Der Generalisierungsfehler folgt für gewöhnlich einer U-förmigen Kurve, wenn er als Funktion eines der Hyperparameter geplottet wird (vgl. Abbildung 5.3). Im einen Extrem entspricht der Wert des Hyperparameters einer niedrigen Kapazität und der Generalisierungsfehler ist aufgrund eines hohen Trainingsfehlers ebenfalls hoch. Das ist der Unteranpassung geschuldet. Im anderen Extrem entspricht der Wert des Hyperparameters einer hohen Kapazität und der Generalisierungsfehler ist aufgrund einer großen Lücke zwischen Trainings- und Testfehler hoch. Irgendwo dazwischen liegt die optimale Modellkapazität, die zum niedrigsten möglichen Generalisierungsfehler führt, indem eine mittelgroße Generalisierungslücke zu einem mittleren Trainingsfehler addiert wird.

Bei einigen Hyperparametern kommt es zur Überanpassung, wenn ihr Wert groß ist. Die Anzahl der verdeckten Einheiten in einer Schicht ist ein Beispiel dafür, denn mehr verdeckte Einheiten führen zu einer höheren Modellkapazität. Bei anderen Hyperparametern kommt es zur Überanpassung, wenn der Wert klein ist. Zum Beispiel entspricht der kleinste zulässige Koeffizient für den Weight Decay von Null einer höheren tatsächlichen Kapazität des Lernalgorithmus.

Nicht alle Hyperparameter können über die gesamte U-förmige Kurve hinweg gesteuert werden. Viele Hyperparameter sind diskret, beispielsweise die Anzahl der Einheiten in einer Schicht oder die Anzahl der linearen Stücke einer Maxout-Einheit, sodass nur bestimmte Punkte auf der Kurve besetzt werden können. Einige Hyperparameter sind binär. Dabei handelt es sich normalerweise um Schalter, die angeben, ob eine optionale Komponente des Lernalgorithmus genutzt wird oder nicht, zum Beispiel ein vorbereitender Schritt, der die Eingabemerkmale durch Subtraktion ihres Mittelwerts und Division durch ihre Standardabweichung normalisiert. Diese Hyperparameter

können nur zwei Punkte der Kurve aufsuchen. Andere Hyperparameter weisen einen Minimal- oder Maximalwert auf, der einige Bereiche der Kurve für sie verschließt. Zum Beispiel kann der Koeffizient für den Weight Decay nicht unter Null fallen. Wenn das Modell also bei einem Weight Decay von Null eine Unteranpassung aufweist, bleibt uns die Überanpassung durch Modifizierung des Koeffizienten für den Weight Decay verschlossen. Anders ausgedrückt: Einige Hyperparameter eignen sich nur dazu, die Kapazität zu verringern.

Die Lernrate dürfte der wohl wichtigste Hyperparameter sein. Wenn Sie nur Zeit für das Abstimmen eines Hyperparameters haben, sollten Sie es mit der Lernrate versuchen. Sie steuert die tatsächliche Kapazität des Modells auf komplexere Weise als andere Hyperparameter – die tatsächliche Kapazität des Modells ist dann am größten, wenn die Lernrate für das Optimierungsproblem *korrekt* ist – und nicht etwa dann, wenn die Lernrate besonders groß oder klein ist. Die Lernrate weist für den *Trainingsfehler* eine U-Form auf (vgl. Abbildung 11.1). Wenn die Lernrate zu groß ist, kann das Gradientenabstiegsverfahren unbeabsichtigt zu einem Anstieg des Trainingsfehlers führen, nicht zu dessen Verringerung. Im idealisierten quadratischen Fall geschieht dies, wenn die Lernrate mindestens doppelt so groß wie ihr optimaler Wert ist (LeCun et al., 1998a). Ist die Lernrate zu klein, verlangsamt dies das Training und kann sogar zu einem Festhängen bei einem hohen Trainingsfehler führen. Woran das liegt, ist noch nicht bekannt – bei einer konvexen Verlustfunktion geschieht dies nicht.

Das Abstimmen anderer Parameter als der Lernrate bedarf der Überwachung von Trainings- und Testfehler, um herauszufinden, ob eine Überanpassung oder Unteranpassung des Modells vorliegt; anschließend muss die Kapazität entsprechend angepasst werden.

Falls Ihr Fehler für die Trainingsdatenmenge größer als die gewünschte Fehlerquote ist, bleibt Ihnen nur eine Erhöhung der Kapazität. Falls Sie keine Regularisierung verwenden und sicher sind, dass der Optimierungsalgorithmus korrekt arbeitet, müssen Sie das Netz um weitere Schichten ergänzen oder weitere verdeckte Einheiten hinzufügen. Leider steigt damit auch der Berechnungsaufwand für das Modell.

Ist der Fehler für die Testdatenmenge größer als die gewünschte Fehlerquote, haben Sie zwei Möglichkeiten. Der Testfehler ist die Summe von Trainingsfehler und Lücke zwischen Trainings- und Testfehler. Der optimale Testfehler ergibt sich durch Abwägung dieser Größen. Neuronale Netze funktionieren meist am besten, wenn der Trainingsfehler sehr gering ist (und somit die Kapazität hoch) und der Testfehler in erster Linie eine Funktion

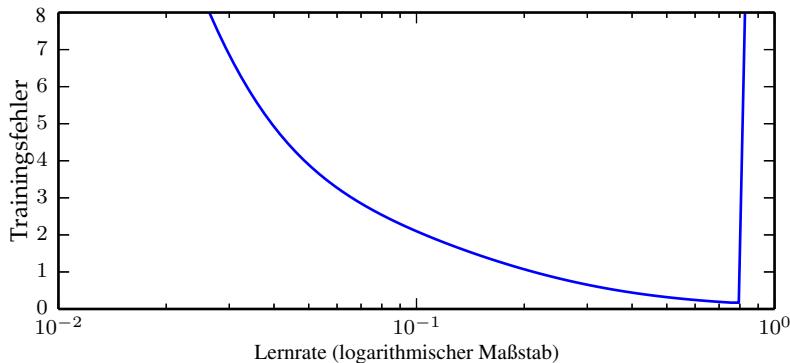


Abbildung 11.1: Typischer Zusammenhang zwischen Lernrate und Trainingsfehler. Beachten Sie den steilen Anstieg im Fehler, wenn die Lernrate einen optimalen Wert übersteigt. Dies gilt für eine feste Trainingsdauer, da eine kleinere Lernrate das Training manchmal nur um einen Faktor verlangsamt, der proportional zu ihrem Rückgang ist. Der Generalisierungsfehler kann dieser Kurve folgen oder durch Regularisierungseffekte kompliziert werden, die sich aus einer zu hohen oder niedrigen Lernrate ergeben können, da eine schlechte Optimierung bis zu einem gewissen Grad die Überanpassung reduzieren oder verhindern kann; sogar Punkte mit äquivalentem Trainingsfehler können einen unterschiedlichen Generalisierungsfehler aufweisen.

der Lücke zwischen Trainings- und Testfehler ist. Ihr Ziel ist es, diese Lücke zu verkleinern, ohne dass der Trainingsfehler schneller anwächst, als die Lücke abnimmt. Zum Reduzieren der Lücke ändern Sie die Regularisierungs-Hyperparameter, um so die effektive Modellkapazität zu reduzieren, beispielsweise durch Hinzufügen von Dropout oder Weight Decay. Die beste Leistung ergibt sich für gewöhnlich aus einem großen Modell, das gut regularisiert ist – zum Beispiels mittels Dropout.

Für die Einstellung der meisten Hyperparameter reicht es aus zu überlegen, ob sie die Modellkapazität erhöhen oder verringern. Tabelle 11.1 zeigt einige Beispiele.

Verlieren Sie bei der manuellen Hyperparameter-Abstimmung aber nicht das eigentliche Ziel aus dem Blick: eine gute Leistung auf der Testdatenmenge. Eine Möglichkeit, dieses Ziel zu erreichen, besteht in der Regularisierung. Sofern der Trainingsfehler klein ist, können Sie den Generalisierungsfehler stets durch Erfassen weiterer Trainingsdaten reduzieren. Die Brute-Force-Methode für einen praktisch garantierten Erfolg besteht darin, die Modellkapazität und die Größe der Trainingsdatenmenge stetig zu erhöhen, bis die Aufgabe gelöst ist. Dadurch steigt natürlich der Berechnungsaufwand für das Training und die Inferenz. Wählen Sie diesen Weg also nur dann,

Hyper-parameter	Erhöht Kapazi-tät, wenn...	Grund	Wichtige Hinweise
Anzahl der verdeckten Einheiten	erhöht	Die Erhöhung der Anzahl der verdeckten Einheiten erhöht die repräsentative Kapazität des Modells.	Die Erhöhung der Anzahl der verdeckten Einheiten erhöht sowohl den Zeit- als auch den Speicherbedarf praktisch jeder Operation im Modell.
Lernrate	optimal abge-stimmt	Eine falsche Lernrate – gleich ob zu hoch oder zu niedrig – führt infolge eines Optimierungsfehlers zu einem Modell mit einer geringen tatsächlichen Kapazität.	
Breite des Faltungskerns	erhöht	Die Erhöhung der Kernelbreite erhöht die Anzahl der Parameter im Modell.	Ein breiterer Kernel führt zu einer schmaleren Ausgabedimension und reduziert die Modellkapazität, sofern Sie nicht mittels impliziten Zero Padding gegensteuern. Breitere Kernel benötigen mehr Speicherplatz für die Parameter und erhöhen die Laufzeit, aber eine schmalere Ausgabe reduziert den Speicherbedarf.
Implizites Zero Padding	erhöht	Das Hinzufügen impliziter Nullen vor der Faltung sorgt für eine weiterhin hohe Repräsentationsgröße.	Zeit- und Speicherbedarf der meisten Operationen werden erhöht.
Koeffizient für den Weight Decay	verringert	Ein kleinerer Koeffizient für den Weight Decay lässt die Modellparameter größer werden.	
Dropout-Rate	verringert	Durch das Entfernen von Einheiten haben die Einheiten oft mehr Möglichkeiten, miteinander zu »konspirieren«, um zur Trainingsdatenmenge zu passen.	

Tabelle 11.1: Die Auswirkung der verschiedenen Hyperparameter auf die Modellkapazität

wenn ausreichend Ressourcen zur Verfügung stehen. Prinzipiell kann der Ansatz aufgrund von Schwierigkeiten bei der Optimierung auch fehlschlagen, aber bei vielen Problemen scheint die Optimierung keine große Rolle zu spielen, solange das Modell passend gewählt wurde.

11.4.2 Algorithmen zur automatischen Hyperparameter-Optimierung

Der ideale Lernalgorithmus nimmt einfach einen Datensatz und gibt eine Funktion aus, ohne dass die Hyperparameter von Hand abgestimmt werden müssen. Die Beliebtheit einiger Lernalgorithmen wie der logistischen Regression und der Support Vector Machines (SVMs) gründet sich zum Teil auf deren Fähigkeit, mit nur einem oder zwei abgestimmten Hyperparametern gut zurechtzukommen. Neuronale Netze können manchmal mit einer geringen Anzahl abgestimmter Hyperparameter gut abschneiden, profitieren aber häufig stark von der Abstimmung von 40 oder mehr Parametern. Die manuelle Hyperparameter-Abstimmung ist eine sehr gute Wahl, wenn Sie über einen guten Ausgangspunkt verfügen, der vielleicht von anderen Personen ermittelt wurde, die an derselben Anwendung und Architektur gearbeitet haben. Auch viele Monate oder gar Jahre Erfahrung in der Untersuchung von Hyperparameter-Werten für neuronale Netze für ähnliche Aufgaben können hilfreich sein. Allerdings sind diese Voraussetzungen für viele Anwendungen nicht gegeben. In diesen Fällen können automatisierte Algorithmen nützliche Werte für die Hyperparameter ermitteln.

Wenn wir uns vor Augen führen, wie der Anwender eines Lernalgorithmus gute Werte für die Hyperparameter sucht, stellen wir fest, dass es sich hierbei um eine Optimierung handelt: Wir versuchen, einen Wert für die Hyperparameter zu finden, der eine Zielfunktion optimiert, beispielsweise den Validierungsfehler, manchmal unter Nebenbedingungen (darunter die verfügbare Trainingsdauer, der verfügbare Speicherplatz oder die Erkennungsdauer). Es ist also prinzipiell möglich, Algorithmen für die **Optimierung der Hyperparameter** zu entwickeln, die einen Lernalgorithmus umhüllen und dessen Hyperparameter auswählen; dabei werden die Hyperparameter des Lernalgorithmus effektiv vor dem Anwender verborgen. Leider setzen Algorithmen für die Optimierung der Hyperparameter oft eigene Hyperparameter voraus, zum Beispiel einen Wertebereich, der für jeden einzelnen Hyperparameter des Lernalgorithmus untersucht werden muss. Diese sekundären Hyperparameter sind meist einfacher auszuwählen, da eine annehmbare Leistung für eine große Anzahl von Aufgaben mit denselben sekundären Hyperparametern für alle Aufgaben erzielt werden kann.

11.4.3 Rastersuche

Wenn es drei oder weniger Hyperparameter gibt, ist eine **Rastersuche** (engl. *grid search*) das gängige Verfahren. Der Anwender wählt für jeden Hyperparameter eine kleine endliche Wertemenge aus, die untersucht werden soll. Der Algorithmus für die Rastersuche trainiert dann das Modell für jede übergreifende Spezifikation der Hyperparameter-Werte im kartesischen Produkt der Wertemenge für jeden einzelnen Hyperparameter. Das Experiment mit dem besten Validierungsdatenfehler wird dann als Experiment gewählt, in dem die besten Hyperparameter ermittelt wurden. Die linke Seite von Abbildung 11.2 zeigt beispielhaft ein Raster mit Hyperparameter-Werten.

Wie sollten die Wertelisten für die Suche zusammengestellt werden? Im Fall numerischer (geordneter) Hyperparameter werden das kleinste und größte Element jeder Liste konservativ aufgrund vorheriger Erfahrungen in ähnlichen Experimenten ausgewählt, damit der optimale Wert mit hoher Wahrscheinlichkeit im angegebenen Wertebereich liegt. Meist werden für die Rastersuche Werte auf einer annähernd *logarithmischen Skala* ausgewählt, zum Beispiel eine Lernrate in der Menge $\{0, 1, 0, 01, 10^{-3}, 10^{-4}, 10^{-5}\}$ oder eine Zahl verdeckter Einheiten aus der Menge $\{50, 100, 200, 500, 1000, 2000\}$.

Die Rastersuche funktioniert normalerweise dann am besten, wenn sie wiederholt ausgeführt wird. Nehmen Sie zum Beispiel an, dass wir eine Rastersuche für einen Hyperparameter α mit den Werten $\{-1, 0, 1\}$ ausführen. Wenn der beste gefundene Wert 1 ist, dann haben wir den Bereich, in dem der beste Wert α liegt, unterschätzt und müssen das Raster verschieben, bevor wir eine erneute Suche durchführen, die α zum Beispiel in der Menge $\{1, 2, 3\}$ sucht. Ergibt sich für α ein bester Wert von 0, dann können wir unsere Schätzung verfeinern und eine erneute Rastersuche für die Menge $\{-0, 1, 0, 0, 1\}$ durchführen.

Das offensichtliche Problem bei der Rastersuche ist, dass der Berechnungsaufwand mit der Anzahl der Hyperparameter exponentiell ansteigt. Für m Hyperparameter mit jeweils maximal n Werten wächst die Anzahl der Trainings- und Bewertungsversuche bereits nach dem Muster $O(n^m)$. Die Versuche können parallel durchgeführt werden und so eine grobe Parallelisierung nutzen (wobei kaum Kommunikation zwischen den Maschinen, auf denen die Suche ausgeführt wird, erfolgen muss). Leider führt – aufgrund des exponentiellen Aufwands für die Rastersuche – selbst die Parallelisierung nicht unbedingt zu einer zufriedenstellenden Suchgröße.

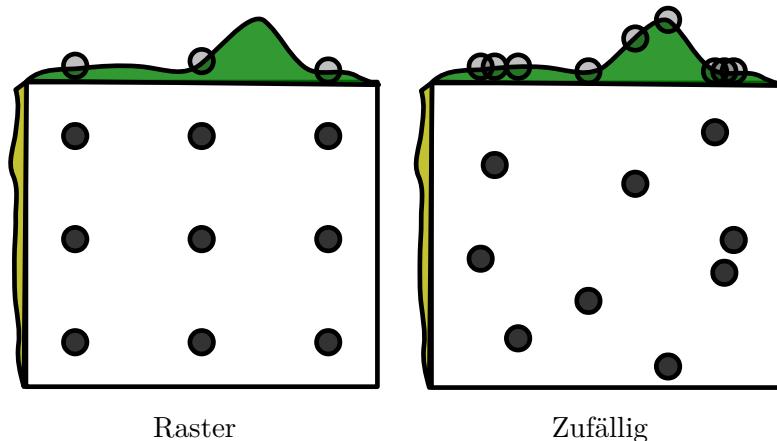


Abbildung 11.2: Vergleich von Rastersuche und Zufallssuche (engl. *random search*). Zur Veranschaulichung zeigen wir nur zwei Hyperparameter. In der Praxis gibt es meist sehr viel mehr. (*Links*) Für die Rastersuche stellen wir für jeden Hyperparameter eine Wertemenge bereit. Der Suchalgorithmus führt für die verbundenen Hyperparameter-Konfigurationen im Kreuzprodukt dieser Mengen ein Training durch. (*Rechts*) Bei der Zufallssuche stellen wir eine Wahrscheinlichkeitsverteilung für die verbundenen Hyperparameter-Konfigurationen bereit. Die meisten dieser Hyperparameter sind normalerweise voneinander unabhängig. Zu den klassischen Verteilungen über einem einzelnen Hyperparameter gehören die Gleichverteilung und die logarithmische Gleichverteilung (um Stichproben aus einer logarithmischen Gleichverteilung zu entnehmen, wird der \exp einer Stichprobe aus einer Gleichverteilung verwendet). Der Suchalgorithmus schlägt dann zufällige verbundene Hyperparameter-Konfigurationen vor und führt mit jeder davon ein Training durch. Bei beiden Formen der Suche (Raster und Zufall) wird der Validierungsdatenfehler evaluiert und die beste Konfiguration wird benannt. Die Abbildung zeigt den typischen Fall, in dem nur einige Hyperparameter einen starken Einfluss auf das Ergebnis haben. In der Abbildung führen nur Hyperparameter auf der horizontalen Achse zu einem nennenswerten Effekt. Die Rastersuche verschwendet Berechnungsressourcen exponentiell in der Anzahl der Hyperparameter ohne Einfluss auf das Ergebnis, die Zufallssuche testet dagegen einen einzigartigen Wert für jeden »einflussreichen« Hyperparameter in fast jedem Versuch. (Abbildung mit freundlicher Genehmigung von Bergstra und Bengio (2012))

11.4.4 Zufallssuche

Zum Glück gibt es eine Alternative zur Rastersuche, die ebenso einfach zu programmieren und noch dazu praktischer in der Anwendung ist und schneller gegen gute Werte der Hyperparameter konvergiert: die Zufallssuche (engl. *random search*) (Bergstra und Bengio, 2012).

Eine Zufallssuche läuft wie folgt ab: Zunächst definieren wir eine Randverteilung für jeden Hyperparameter, zum Beispiel eine Bernoulli- oder Multinoulli-Verteilung für binäre oder diskrete Hyperparameter oder eine Gleichverteilung auf einer log-Skala für positiv reellwertige Hyperparameter. Ein Beispiel:

$$\log\text{_Lernrate} \sim u(-1, -5), \quad (11.2)$$

$$\text{Lernrate} = 10^{\log\text{_Lernrate}}, \quad (11.3)$$

wobei $u(a, b)$ für eine Stichprobe der Gleichverteilung im Intervall (a, b) steht. Ebenso kann `log_Aanzahl_der_verdeckten_Einheiten` aus $u(\log(50), \log(2000))$ gezogen werden.

Anders als in der Rastersuche dürfen wir die Werte der Hyperparameter *nicht diskretisieren* oder gruppieren, damit wir eine größere Wertemenge untersuchen und zusätzlichen Berechnungsaufwand vermeiden können. Wie Abbildung 11.2 zeigt, kann eine Zufallssuche tatsächlich exponentiell effizienter als eine Rastersuche sein, wenn es mehrere Hyperparameter gibt, die keinen starken Einfluss auf die Leistungsbewertung haben. Eine ausführliche Betrachtung hierzu enthält *Bergstra und Bengio* (2012); sie haben festgestellt, dass die Zufallssuche den Validierungsdatenfehler sehr viel schneller als die Rastersuche reduziert – was die Anzahl der Versuche betrifft, die von den einzelnen Verfahren durchgeführt werden.

Wie bei der Rastersuche bietet es sich an, die Zufallssuche wiederholt durchzuführen und anhand der Ergebnisse der ersten Suche anzupassen.

Der Hauptgrund dafür, dass die Zufallssuche schneller als die Rastersuche zu guten Lösungen findet, besteht darin, dass dabei keine experimentellen Durchläufe verschwendet werden – das ist bei der Rastersuche immer dann der Fall, wenn die beiden Werte eines Hyperparameters (hinsichtlich der Werte der anderen Hyperparameter) zum selben Ergebnis führen. Bei der Rastersuche weisen die anderen Hyperparameter in beiden Fällen dieselben Werte auf, in der Zufallssuche dagegen normalerweise unterschiedliche Werte. Wenn die Änderung zwischen den beiden Werten also kaum einen Unterschied beim Validierungsdatenfehler bedeutet, wiederholt die Rastersuche unnötigerweise zwei gleichwertige Experimente, während die Zufallssuche trotzdem zwei unabhängige Exploration der anderen Hyperparameter liefert.

11.4.5 Hyperparameter-Optimierung auf Modellbasis

Die Suche nach guten Hyperparametern lässt sich als Optimierungsproblem darstellen. Die Entscheidungsvariablen sind die Hyperparameter. Der zu

optimierende Aufwand ist der Validierungsdatenfehler, der aus dem Trainieren mithilfe dieser Hyperparameter resultiert. In vereinfachten Fällen, in denen das Berechnen des Gradienten einer differenzierbaren Fehlergröße für die Validierungsdatenmenge hinsichtlich der Hyperparameter möglich ist, können wir einfach diesem Gradienten folgen (*Bengio et al.*, 1999; *Bengio*, 2000; *Maclaurin et al.*, 2015). Leider steht der Gradient in der Praxis meist nicht zur Verfügung – sei es aufgrund des für seine Bestimmung erforderlichen hohen Berechnungsaufwands und Speicherbedarfs oder weil die Hyperparameter intrinsisch nicht differenzierbare Interaktionen mit dem Validierungsdatenfehler aufweisen, beispielsweise im Fall von diskreten Hyperparametern.

Um dieses Fehlen eines Gradienten auszugleichen, können wir ein Modell des Validierungsdatenfehlers erstellen und anschließend eine Optimierung in diesem Modell durchführen, um neue Hyperparameter zu raten. Die meisten Algorithmen auf Modellbasis für die Hyperparameter-Suche verwenden das bayessche Regressionsmodell zur Schätzung sowohl des Erwartungswerts des Validierungsdatenfehlers für jeden Hyperparameter als auch der Unsicherheit um diesen Erwartungswert. Die Optimierung bedeutet somit einen Kompromiss zwischen Exploration (auch: Erkundung, das Vorschlagen von Hyperparametern mit einer hohen Unsicherheit, die möglicherweise zu einer starken Verbesserung führen, andererseits aber auch sehr schlecht abschneiden könnten) und Exploitation (auch: Ausnutzung, das Vorschlagen von Hyperparametern, bei denen das Modell sicher ist, dass sie ebenso gut wie andere, bisher untersuchte Hyperparameter funktionieren – also gemeinhin Hyperparameter, die den bisherigen sehr ähnlich sind). Aktuelle Ansätze zur Hyperparameter-Optimierung sind zum Beispiel Spearmint (*Snoek et al.*, 2012), TPE (*Bergstra et al.*, 2011) und SMAC (*Hutter et al.*, 2011).

Derzeit können wir keine ausdrückliche Empfehlung für die bayessche Hyperparameter-Optimierung als bewährtes Werkzeug für bessere Lernergebnisse oder ein müheloseres Erreichen dieser Ergebnisse aussprechen. Die bayessche Hyperparameter-Optimierung schneidet manchmal ähnlich ab wie menschliche Experten, manchmal sogar besser – und in anderen Fällen kommt es damit zum Totalversagen. Für bestimmte Probleme ist sie einen Versuch wert, aber das Verfahren ist noch lange nicht ausgereift oder zuverlässig. Dennoch ist die Hyperparameter-Optimierung – obwohl in erster Linie eine Bestrebung im Deep-Learning-Kontext – ein wichtiges Forschungsfeld mit dem Potenzial, sich sowohl für das Machine Learning in seiner Gesamtheit als auch die Ingenieurwissenschaften insgesamt vorteilhaft zu erweisen.

Ein Nachteil, der den meisten Algorithmen zur Hyperparameter-Optimierung, die raffinierter als die Zufallssuche sind, gemein ist, ist das Erfordernis eines vollständigen bis zum Ende durchgeföhrten Trainingsexperiments, bevor Schlüsse aus diesem Experiment gezogen werden können. Das ist hinsichtlich der Menge der zu Beginn eines Experiments ablesbaren Daten deutlich weniger effizient als die manuelle Suche durch menschliche Experten, denn wir können meist schon frühzeitig sagen, ob eine bestimmte Zusammenstellung von Hyperparametern vollkommen unbrauchbar ist. *Swersky et al.* (2014) haben eine frühe Version eines Algorithmus vorgestellt, der eine Menge mit mehreren Experimenten nutzt. Zu verschiedenen Zeitpunkten kann der Algorithmus zur Hyperparameter-Optimierung sich dafür entscheiden, ein neues Experiment zu starten, ein laufendes Experiment, das wenig vielversprechend erscheint, zu pausieren oder ein zuvor pausiertes Experiment, das aufgrund neuerer Erkenntnisse nun doch einen Nutzen haben könnte, wieder aufzunehmen und fortzusetzen.

11.5 Debugging-Verfahren

Wenn ein Machine-Learning-System schlecht abschneidet, lässt sich nicht immer schnell erkennen, ob die Ursache dafür im Algorithmus selbst oder in der Implementierung des Algorithmus liegt. Die Fehlersuche und -behebung in Machine-Learning-Systemen ist aus mehreren Gründen schwierig.

In den meisten Fällen wissen wir nicht im Voraus, wie sich ein Algorithmus eigentlich verhalten sollte. Tatsächlich liegt ja der Sinn des Machine Learnings gerade darin, dass es ein nützliches Verhalten findet, das wir selbst nicht klar beschreiben können. Wenn wir ein neuronales Netz für eine *neue Klassifizierungsaufgabe* trainieren und einen Testfehler von 5 Prozent erreichen, können wir nicht auf die Schnelle sagen, ob dies das erwartete Verhalten oder eher ein suboptimales Abschneiden darstellt.

Ein weiteres Problem ist, dass die meisten Machine-Learning-Modelle mehrere Teile aufweisen, die jeweils adaptiv sind. Ist ein Teil defekt, können die anderen Teile entsprechend reagieren und so noch immer eine im Großen und Ganzen annehmbare Leistung erbringen. Ein Beispiel: Angenommen, wir trainieren ein neuronales Netz mit mehreren Schichten, die mittels der Gewichte \mathbf{W} und der Verzerrungen \mathbf{b} parametrisiert wurden. Weiterhin angenommen, wir haben manuell eine Gradientenabstiegsregel separat für jeden Parameter eingeführt und dabei einen Fehler in das Update der Verzerrungen eingebaut:

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha, \quad (11.4)$$

mit α als Lernrate. Dieses fehlerbehaftete Update verwendet den Gradienten gar nicht. Es führt dazu, dass die Verzerrungen während des Lernprozesses ständig negativ werden – ganz klar keine korrekte Implementierung eines angemessenen Lernalgorithmus. Bei einer Untersuchung der Ausgabe des Modells ist der Fehler aber vielleicht nicht offensichtlich. Je nach Verteilung der Eingabe können die Gewichte die negativen Verzerrungen durch Anpassung kompensieren.

Die meisten Debugging-Verfahren für neuronale Netze sind darauf ausgelegt, eine oder beide dieser Schwierigkeiten zu umgehen. Entweder konzipieren wir einen Fall, der so einfach ist, dass sich das korrekte Verhalten vorhersagen lässt, oder wir konzipieren einen Test, der einen Teil der Implementierung des neuronalen Netzes isoliert überprüft.

Zu den wichtigen Debugging-Tests gehören die folgenden:

Visualisieren des Modells in Aktion: Stellen Sie beim Trainieren eines Modells zur Objekterkennung in Bildern einige Bilder mit vom Modell vorgeschlagenen Erkennungen als Überlagerung auf dem Bild dar. Hören Sie sich beim Trainieren eines generativen Modells für Sprache einige der damit erzeugten Sprachmuster an. Diese Punkte sind so offensichtlich, dass sie in Anbetracht der messbaren Leistungskriterien wie Korrektklassifikationsrate oder Log-Likelihood gern übersehen werden. Eine direkte Beobachtung des Machine-Learning-Modells, während es seine Aufgabe erledigt, hilft dabei zu entscheiden, ob die erzielten Leistungskennzahlen logisch und angemessen sind. Fehler bei der Bewertung können zu den verheerendsten Fehlern zählen – denn diese Fehler können dazu verleiten, ein gutes Abschneiden des Systems anzunehmen, obwohl dies nicht der Fall ist.

Visualisieren der größten Fehler: Die meisten Modelle können eine Art Konfidenzmaß für die jeweilige Aufgabe ausgeben. Zum Beispiel weisen Klassifikatoren auf Basis einer softmax-Ausgabeschicht jeder Klasse eine Wahrscheinlichkeit zu. Die der wahrscheinlichsten Klasse zugewiesene Wahrscheinlichkeit ist somit ein Schätzwert für die Konfidenz, die das Modell in die getroffene Klassifizierungsscheidung hat. Meist führt ein Maximum-Likelihood-Training dazu, dass diese Werte Überschätzungen anstelle exakter Wahrscheinlichkeiten der korrekten Vorhersage sind, aber sie sind dennoch insofern nützlich, als Testdaten, bei denen es weniger wahrscheinlich ist, dass sie mit korrekten Labels gekennzeichnet werden, bei diesem Modell kleinere Wahrscheinlichkeiten erhalten. Durch Betrachten der Beispiele aus der Trainingsdatenmenge, die am schwierigsten zu modellieren sind, lassen sich oft Probleme in der Vorverarbeitung oder Kennzeichnung der Daten mit Labels erkennen. So kam es beim Transkriptionssystem für Street

View anfangs dazu, dass die Bilder für das Erkennen der Hausnummern zu stark beschnitten wurden, wodurch einige Ziffern nicht im Bild enthalten waren. Das Transkriptionsnetz wies der korrekten Antwort für diese Bilder aus diesem Grund eine sehr geringe Wahrscheinlichkeit zu. Nachdem die Bilder sortiert wurden, um die am wahrscheinlichsten auftretenden Fehler (engl. *most confident mistakes*) auszumachen, zeigte sich ein systematisches Problem beim Zuschnitt. Das Erkennungssystem wurde so modifiziert, dass der Beschnitt nicht so eng erfolgte, was zu einer sehr viel besseren Gesamtleistung führte – obwohl das Transkriptionsnetz nun mehr Optionen bezüglich Lage und Größe der Hausnummern verarbeiten musste.

Prüfen der Software anhand von Trainings- und Testfehler: Es ist oft nicht einfach zu entscheiden, ob die zugrunde liegende Software korrekt implementiert wurde. Einige Hinweise finden sich in den Trainings- und Testfehlern. Wenn der Trainingsfehler niedrig ist, der Testfehler aber hoch, dann funktioniert das Trainingsverfahren vermutlich korrekt und das Modell neigt aus grundlegenden algorithmischen Gründen zur Überanpassung. Eventuell wird der Testfehler auch falsch gemessen, weil ein Problem mit dem Speichern des Modells nach dem Training und dem anschließenden Laden zur Bewertung der Testdatenmenge vorliegt – oder weil die Testdaten auf andere Weise als die Trainingsdaten vorbereitet wurden. Wenn Trainings- und Testfehler hoch sind, lässt sich nur schwer entscheiden, ob ein Softwareproblem vorliegt oder ob das Modell aus fundamentalen algorithmischen Gründen zur Unteranpassung neigt. In diesem Fall sind weitere Tests erforderlich, die im Folgenden beschrieben werden.

Anpassen eines sehr kleinen Datensatzes: Kommt es für die Trainingsdatenmenge zu einem hohen Fehler, müssen Sie herausfinden, ob eine echte Unteranpassung oder aber ein Softwarefehler vorliegt. Meistens gewähleisten sogar kleine Modelle das Anpassen eines ausreichend kleinen Datensatzes. Ein Beispiel: Für einen Klassifizierungsdatensatz mit nur einem Beispiel ist die Anpassung durch korrektes Einstellen der Verzerrungen der Ausgabeschicht möglich. Wenn Sie einen Klassifikator nicht darauf trainieren können, ein einzelnes Beispiel mit einem korrekten Label zu kennzeichnen, ein Autoencoder beim Reproduzieren eines einzigen Beispiels mit hoher Konfidenz versagt oder ein generatives Modell immer wieder Muster ausgibt, die einem einzelnen Beispiel ähneln, dann liegt im Normalfall ein Softwarefehler vor, der eine erfolgreiche Optimierung anhand der Trainingsdatenmenge verhindert. Dieser Test kann auf kleine Datensätze mit wenigen Beispielen erweitert werden.

Vergleichen der backpropagierten Ableitungen mit numerischen Ableitungen: Wenn Sie ein Software-Framework verwenden, in dem Sie eigene

Gradientenberechnungen implementieren müssen, oder wenn Sie eine neue Operation zu einer Softwarebibliothek für Differentiationsalgorithmen hinzufügen und deren `bprop`-Methode definieren müssen, kommt es häufig zu Fehlern bei der korrekten Implementierung des Gradientenausdrucks. Ob diese Ableitungen korrekt sind, können Sie durch einen Vergleich der Ableitungen aus Ihrer Implementierung einer automatischen Differentiation mit den durch **finite Differenzen** bestimmten Ableitungen prüfen. Da

$$f'(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x)}{\epsilon} \quad (11.5)$$

gilt, können wir die Ableitung mittels eines kleinen, endlichen ϵ approximieren:

$$f'(x) \approx \frac{f(x + \epsilon) - f(x)}{\epsilon}. \quad (11.6)$$

Wir können die Korrektklassifikationsrate der Approximation mit der **zentrierten Differenz** verbessern:

$$f'(x) \approx \frac{f(x + \frac{1}{2}\epsilon) - f(x - \frac{1}{2}\epsilon)}{\epsilon}. \quad (11.7)$$

Die Störgröße ϵ muss groß genug sein, um sicherzustellen, dass die Störung durch endlich genaue numerische Berechnungen nicht zu stark abgerundet wird.

Meist möchten wir den Gradienten oder die Jacobi-Matrix einer vektorwertigen Funktion $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$ testen. Leider lässt die finite Differentiation nur die Betrachtung einer einzelnen Ableitung gleichzeitig zu. Wir können nun die finite Differentiation entweder mn Mal ausführen, um alle partiellen Ableitungen von g zu bestimmen, oder den Test auf eine neue Funktion anwenden, die zufällig Projektionen für die Eingabe und die Ausgabe von g vornimmt. Zum Beispiel können wir unseren Test für die Implementierung der Ableitungen auf $f(x)$ anwenden, mit $f(x) = \mathbf{u}^T g(\mathbf{v}x)$ und \mathbf{u} und \mathbf{v} als zufällig gewählten Vektoren. Für die korrekte Berechnung von $f'(x)$ müssen wir die Backpropagation korrekt durch g vornehmen; das ist mit finiten Differenzen effizient, da f nur eine Eingabe und eine Ausgabe aufweist. Es ist oft eine gute Idee, diesen Test für mehrere Werte von \mathbf{u} und \mathbf{v} durchzuführen, damit keine Fehler übersehen werden, die orthogonal zu den zufälligen Projektionen liegen.

Wenn die numerische Berechnung komplexer Zahlen möglich ist, gibt es eine sehr effiziente Methode zur numerischen Schätzung des Gradienten mittels komplexer Zahlen als Eingabe für die Funktion (*Squire und Trapp, 1998*). Die Methode beruht auf der Feststellung, dass

$$f(x + i\epsilon) = f(x) + i\epsilon f'(x) + O(\epsilon^2), \quad (11.8)$$

$$\text{real}(f(x + i\epsilon)) = f(x) + O(\epsilon^2), \quad \text{imag}\left(\frac{f(x + i\epsilon)}{\epsilon}\right) = f'(x) + O(\epsilon^2), \quad (11.9)$$

mit $i = \sqrt{-1}$ ist. Anders als im obigen reellwertigen Fall gibt es keinen Aufhebungseffekt, da wir die Differenz zwischen dem Wert von f an verschiedenen Punkten bilden. Damit können winzige Werte für ϵ genutzt werden, zum Beispiel $\epsilon = 10^{-150}$, wodurch der $O(\epsilon^2)$ -Fehler für alle praktischen Zwecke vernachlässigbar wird.

Überwachen von Histogrammen der Aktivierungen und des Gradienten: Es ist oft nützlich, Statistiken neuronaler Netzaktivierungen und Gradienten zu visualisieren, die über eine große Anzahl von Trainingsiterationen (vielleicht eine Epoche) erfasst wurden. Der Prä-Aktivierungswert verdeckter Einheiten gibt an, ob die Einheiten sättigen oder wie oft dies geschieht. Wie häufig sind zum Beispiel Rectifier ausgeschaltet? Gibt es Einheiten, die stets ausgeschaltet sind? Bei tanh-Einheiten gibt der Durchschnittswert des Absolutbetrags der Prä-Aktivierungen an, wie gesättigt die Einheit ist. In einem tiefen Netz, in dem die propagierten Gradienten schnell wachsen oder verschwinden, kann die Optimierung behindert werden. Und schließlich ist es hilfreich, die Größe der Parametergradienten mit der Größe der Parameter selbst zu vergleichen. Wie von Bottou (2015) empfohlen, sollte die Größe der Parameteranpassungen über einen Mini-Batch in etwa einem Prozent des Betrags der Parameter entsprechen – und nicht 50 Prozent oder 0,001 Prozent (wodurch sich die Parameter zu langsam bewegen würden). Vielleicht bewegen sich einige Parametergruppen im angemessenen Tempo, während andere auf der Stelle treten. Wenn die Daten dünnbesetzt sind (wie in der natürlichen Sprache), werden einige Parameter vielleicht sehr selten aktualisiert – das müssen Sie natürlich bei der Beobachtung ihrer Entwicklung bedenken.

Viele Deep-Learning-Algorithmen geben auch in jedem Schritt eine Art Garantie hinsichtlich der erzeugten Ergebnisse. Zum Beispiel zeigen wir in Teil III einige Algorithmen zur approximativen Inferenz, die algebraische Lösungen für Optimierungsprobleme nutzen. Diese lassen sich meist durch Testen der einzelnen Garantien debuggen. Einige Garantien in bestimmten Optimierungsalgorithmen besagen, dass die Zielfunktion niemals bereits nach einem Schritt des Algorithmus zunimmt, dass der Gradient einer Teilmenge der Variablen nach jedem Schritt des Algorithmus Null ist oder dass der Gradient alle Variablen bei Konvergenz Null ist. Aufgrund von Rundungsfehlern werden diese Bedingungen mit digitalen Computern normalerweise nicht exakt erreicht, sodass beim Debugging eine gewisse Toleranz berücksichtigt werden muss.

11.6 Beispiel: Erkennen mehrstelliger Zahlen

Als vollständige Beschreibung für die Anwendung unserer Methodologie in der Praxis geben wir hier einen kurzen Bericht über das Transkriptionssystems für Street View hinsichtlich des Designs der Deep-Learning-Komponenten. Natürlich waren auch viele weitere Komponenten des Gesamtsystems enorm wichtig, darunter die Fahrzeuge, die Datenbankinfrastruktur und mehr.

Aus Sicht der Machine-Learning-Aufgabe beginnt der Prozess mit der Datenerfassung. Die Rohdaten wurden von Fahrzeugen erfasst und dann handisch mit Labeln gekennzeichnet. Vor der Transkriptionsaufgabe erfolgte eine umfangreiche Pflege des Datensatzes, bei der auch weitere Machine-Learning-Verfahren zum Einsatz kamen, um die Hausnummern vor der Transkription zu erkennen.

Am Anfang des Transkriptionsprojekts stand die Entscheidung für die Performance-Kriterien und deren Sollwerte. Es ist allgemein sehr wichtig, die Kennzahlen an die Ziele des Projekts anzupassen. Da Karten nur hilfreich sind, wenn sie genau sind, wurde für das Projekt eine hohe Korrektklassifikationsrate gefordert, nämlich ein Erreichen des mit 98 Prozent Korrektklassifikationsrate hohen menschlichen Niveaus. Dieser Wert kann nicht immer erzielt werden. Im Gegenzug wurde beim Transkriptionssystem für Street View ein gewisses Maß an Deckung geopfert. Die Deckung wurde somit zu dem Kriterium oder Leistungsmaß, dessen Optimierung im Projektverlauf den höchsten Stellenwert hatte – natürlich bei einer gleich bleibenden Korrektklassifikationsrate von 98 Prozent. Mit zunehmender Verbesserung des CNNs war es möglich, die Konfidenzschwelle, unter der das Netz ein Transkribieren der Eingabe verweigerte, zu senken, bis schließlich sogar der Zielwert von 95 Prozent für die Deckung übertroffen wurde.

Nachdem die quantitativen Ziele gesetzt sind, ist der nächste Schritt in unserer Methodologie das schnelle Etablieren eines sinnvollen Baseline-Systems. Für Computer-Vision-Aufgaben läuft das auf ein CNN mit ReLUs hinaus. Das Transkriptionsprojekt begann mit einem solchen Modell. Damals war es für ein CNN unüblich, eine Sequenz von Vorhersagen auszugeben. Als einfachste mögliche Baseline bestand die erste Implementierung der Ausgabeschicht des Modells aus n verschiedenen softmax-Einheiten für die Vorhersage einer Sequenz von n Zeichen. Diese softmax-Einheiten wurden exakt so trainiert, als ginge es um eine Klassifizierungsaufgabe – jede softmax-Einheit wurde unabhängig von den anderen trainiert.

Unsere empfohlene Methodologie ist es, die Baseline iterativ zu verbessern und zu testen, welche Änderung zu einer Verbesserung führt. Die erste Änderung beim Transkriptionssystem für Street View beruhte auf einem theoretischen Verständnis des Deckungskriteriums und der Datenstruktur. Insbesondere verweigerte es das Netz, die Eingabe \mathbf{x} zu klassifizieren, sobald die Wahrscheinlichkeit der Ausgabesequenz $p(\mathbf{y} | \mathbf{x})$ für einen Schwellenwert t kleiner t war. Zu Beginn wurde $p(\mathbf{y} | \mathbf{x})$ ad hoc definiert, durch einfaches Multiplizieren aller softmax-Ausgaben. Das führte zur Entwicklung einer spezialisierten Ausgabeschicht und Kostenfunktion, die tatsächlich eine grundlegende Log-Likelihood berechnete. Dieser Ansatz führte zu einer sehr viel effizienteren Funktion des Mechanismus zum Verwerfen von Beispielen.

Die Deckung betrug noch immer weniger als 90 Prozent, aber es gab keine offensichtlichen theoretischen Probleme mit dem Ansatz mehr. Unsere Methodologie empfiehlt an dieser Stelle die Konfiguration der Leistung für die Trainings- und Testdatenmenge, um herauszufinden, ob eine Unteranpassung oder eine Überanpassung das Problem ist. In diesem Fall waren der Fehler auf den Trainings- und der auf den Testdaten nahezu gleich. Tatsächlich war der Hauptgrund für den recht reibungslosen Projektverlauf die Verfügbarkeit eines Datensatzes mit zig Millionen mit Labels gekennzeichneten Beispielen. Da der Fehler auf den Trainingsdaten und der Fehler auf den Testdaten so nahe beieinander lagen, war das Problem entweder eine Unteranpassung oder aber in den Trainingsdaten zu finden. Eins der von uns empfohlenen Debugging-Verfahren ist das Visualisieren der größten Fehler des Modells. In diesem Fall wurden die falschen Transkriptionen für die Trainingsdatenmenge dargestellt, denen das Modell die höchste Konfidenz gab. Das Ergebnis waren hauptsächlich Beispiele, in denen das Eingabebild zu stark beschnitten worden war, sodass einige Ziffern der Hausnummer nicht mehr zu sehen waren. Beispielsweise zeigte ein Foto anstelle der Hausnummer 1849 durch falsches Beschneiden die Hausnummer 849. Man hätte nun viele Wochen darauf verwenden können, die Korrektklassifikationsrate des Systems zum Erkennen von Hausnummern zu verbessern, das für den Beschnitt verantwortlich war. Stattdessen wählte das Team einen sehr viel praktischeren Ansatz und vergrößerte einfach die Breite des Ausschnitts, sodass sie systematisch größer als vom System vorhergesagt gewählt wurde. Diese einzelne Änderung verbesserte die Deckung des Transkriptionssystems um zehn Prozentpunkte.

Die letzten paar Prozentpunkte an Leistung waren der Anpassung der Hyperparameter geschuldet, und zwar in erster Linie einer Vergrößerung des Modells unter Beachtung gewisser Einschränkungen bezüglich des Berechnungsaufwands. Da Trainings- und Testfehler nach wie vor in etwa

gleich blieben, war stets klar, dass Leistungseinbußen einer Unteranpassung geschuldet sind – und ein paar verbleibenden Problemen im Datensatz selbst.

Insgesamt war das Transkriptionsprojekt ein großer Erfolg und ermöglichte es, Hunderte Millionen von Adressen schneller und preiswerter zu transkribieren, als dies für Menschen möglich gewesen wäre.

Wir hoffen, dass die in diesem Kapitel vorgestellten Designprinzipien auch Ihnen zu ähnlichem Erfolg verhelfen!

12

Anwendungen

In diesem Kapitel zeigen wir, wie Deep Learning in der Computer Vision, bei der Spracherkennung, bei der Verarbeitung natürlicher Sprache (engl. *natural language processing*) und in anderen kommerziellen Bereichen eingesetzt wird. Wir beginnen mit einer Erläuterung umfassender Implementierungen neuronaler Netze, die für die meisten ernstzunehmenden Anwendungen in der Künstlichen Intelligenz benötigt werden. Anschließend widmen wir uns einzelnen Anwendungsbereichen, in denen sich Deep Learning als nützlich erwiesen hat. Zwar ist eines der Ziele von Deep Learning die Entwicklung von Algorithmen, die die unterschiedlichsten Aufgaben zu lösen vermögen, doch bislang ist stets ein gewisses Maß an Spezialisierung gefordert. So müssen für Aufgaben rund um Computer Vision sehr viele Eingabemerkmale (Pixel) für jedes Beispiel verarbeitet werden. Im Sprachkontext wiederum muss eine Vielzahl möglicher Werte (Wörter aus dem Wortschatz) für jedes Eingabemerkmal modelliert werden.

12.1 Deep Learning im großen Maßstab

Deep Learning beruht auf der Philosophie des Konnektionismus: Zwar ist ein einzelnes biologisches Neuron oder ein einzelnes Merkmal in einem Machine-Learning-Modell nicht intelligent, doch wenn eine große Menge dieser Neuronen oder Merkmale zusammenarbeiten, kann diese Gesamtheit intelligentes Verhalten an den Tag legen. Dabei kommt es vor allem auf das Wort *groß* an. Einer der entscheidenden Faktoren, der sich in der Zeit zwischen den 1980er Jahren und heute verändert hat und verantwortlich ist für die Verbesserung der Genauigkeit neuronaler Netze und die Verbesserung ihrer Fähigkeit, komplexe Aufgaben zu lösen, besteht darin, dass die neuronalen

Netze, die wir heute verwenden, deutlich größer geworden sind. Wie wir in Abschnitt 1.2.3 gesehen haben, ist die Größe der Netze über die letzten drei Jahrzehnte exponentiell angewachsen. Trotz alledem sind künstliche neuronale Netze nicht größer als das Nervensystem von Insekten.

Da die Größe neuronaler Netze so bedeutend ist, setzt Deep Learning leistungsfähige Hardware und Software voraus.

12.1.1 Implementierungen auf Basis schneller Prozessoren

In der Vergangenheit wurden neuronale Netze meist auf der CPU einer einzelnen Maschine trainiert. Diese Herangehensweise gilt heutzutage im Allgemeinen als unzureichend. Stand der Technik sind GPUs oder ein Verbund aus den CPUs mehrerer Maschinen. Vor der Nutzung dieser teuren Systeme arbeiteten Forscher hart daran, zu zeigen, dass CPUs die hohe Rechenlast, die für neuronale Netze erforderlich ist, nicht zu stemmen vermochten.

Eine Beschreibung, wie man effizienten numerischen CPU-Code implementiert, geht über den Umfang dieses Buchs hinaus. Nichtsdestotrotz möchten wir betonen, dass eine sorgfältige Implementierung für einzelne CPU-Familien zu erheblichen Verbesserungen führen kann. Zum Beispiel konnten die besten 2011 verfügbaren CPUs neuronale Netze schneller ausführen, wenn Festkomma- anstelle von Gleitkommaarithmetik genutzt wurde. Mittels sorgsam abgestimmter Festkommaimplementierungen erzielten *Vanhoucke et al.* (2011) eine drei Mal höhere Geschwindigkeit als ein starkes Gleitkommasytem. Jedes neue CPU-Modell zeichnet sich durch andere Leistungskennzahlen aus – es mag also durchaus Gleitkommaimplementierungen geben, die schneller sind. Der wichtige Grundsatz ist, dass eine sorgfältige Spezialisierung numerischer Berechnungsroutinen erhebliche Vorteile bieten kann. Neben der Wahl zwischen Fest- und Gleitkommarechnungen gibt es weitere Verfahren, darunter die Optimierung von Datenstrukturen, zur Vermeidung von Cache-Fehlgriffen, oder die Nutzung von Vektoranweisungen. Viele Machine-Learning-Forscher vernachlässigen diese Details bei der Implementierung, aber wenn die Leistung einer bestimmten Implementierung die Modellgröße einschränkt, dann leidet die Modellgenauigkeit darunter.

12.1.2 Implementierungen auf Basis von GPUs

Die meisten modernen Implementierungen neuronaler Netze basieren auf Grafikprozessoren. Grafikprozessoren (kurz GPUs) sind spezielle Hardwarekomponenten, die ursprünglich für Grafikanwendungen entwickelt wurden.

Der Endkundenmarkt für Videospiele und die zugehörigen Systeme hat die Entwicklung immer leistungsfähigerer Hardware für die Grafikverarbeitung beschleunigt. Dabei hat sich gezeigt, dass die Kriterien, die für Videospiele wünschenswert sind, auch für neuronale Netze von Vorteil sind.

Für die Grafik in Videospiele müssen viele Operationen sehr schnell gleichzeitig ausgeführt werden. Charaktermodelle und Umgebungen werden als Listen mit 3-D-Koordinaten einzelner Vertices definiert. Um diese 3-D-Koordinaten in 2-D-Koordinaten der Bildschirmebene umzuwandeln, müssen Grafikkarten Matrizenmultiplikationen und -divisionen für viele Vertices parallel ausführen. Anschließend muss die Grafikkarte für jedes Pixel parallel viele Berechnungen durchführen, um die Farben der einzelnen Pixel zu bestimmen. In beiden Fällen handelt es sich um relativ einfache Berechnungen ohne viele Verzweigungen (die für die Rechenlast einer CPU typisch sind). So wird zum Beispiel jeder Vertex desselben starren Objekts mit derselben Matrix multipliziert; es besteht keine Notwendigkeit, für jeden Vertex mittels einer `if`-Anweisung herauszufinden, mit welcher Matrix er multipliziert werden muss. Die Berechnungen sind auch vollkommen unabhängig voneinander und können daher problemlos parallelisiert werden. Außerdem werden dabei große Pufferspeicher verarbeitet, die Bitmaps der Texturen (Farbmuster) der einzelnen, darzustellenden Objekte enthalten. Das hat dazu geführt, dass Grafikkarten zulasten des Prozessortakts und der Verzweigungsfähigkeiten, die klassischen CPUs aufweisen, auf Parallelität und eine große Speicherbandbreite hin getrimmt wurden.

Algorithmen in neuronalen Netzen erfordern dieselben Leistungskriterien wie die oben beschriebenen Algorithmen für Echtzeitgrafiken. In neuronalen Netzen sind große und viele Puffer mit Parametern, Aktivierungswerten und Gradientenwerten gängig – sie alle müssen in jedem Trainingsschritt vollständig aktualisiert werden. Diese Puffer sind groß genug, um den Speicherplatz im Cache eines klassischen Desktop-Computers zu überschreiten, sodass die Speicherbandbreite des Systems häufig ein limitierender Faktor für die Geschwindigkeit ist. GPUs bieten aufgrund ihrer hohen Speicherbandbreite gegenüber CPUs einen großen Vorteil. Trainingsalgorithmen für neuronale Netze weisen meist kaum Verzweigungen oder ausgefeilte Steuerungsfunktionen auf, sind also für die Verarbeitung mit GPUs geradezu prädestiniert. Da neuronale Netze in mehrere einzelne »Neuronen« aufgeteilt werden können, die unabhängig von den anderen Neuronen derselben Schicht verarbeitet werden können, profitieren sie auch von der Parallelität der GPU-Berechnungen.

Anfangs war GPU-Hardware so auf die angedachte Anwendung maßgeschneidert, dass sie sich tatsächlich nur für Grafikaufgaben eigneten. Im

Laufe der Zeit wurde die GPU-Hardware flexibler, sodass auch individuelle Subroutinen zum Transformieren der Koordinaten der Vertices oder zum Zuweisen von Farben zu Pixeln eingesetzt werden konnten. Im Prinzip gab es keine Bedingung, nach der diese Pixelwerte auf einer echten Rendering-Aufgabe basieren mussten. Man konnte diese GPUs für wissenschaftliche Berechnungen einsetzen, indem die Ausgabe einer Berechnung in einen Puffer mit Pixelwerten geschrieben wurde. *Steinkrau et al.* (2005) setzten ein vollständig verbundenes neuronales Netz mit zwei Schichten auf einer GPU um, bei dem sie ein drei Mal höheres Tempo gegenüber demselben System auf CPU-Basis feststellten. Kurz danach zeigten *Chellapilla et al.* (2006), dass dieselbe Methode auch zur Beschleunigung überwachter CNNs dienen konnte.

Die Beliebtheit von Grafikkarten für das Training neuronaler Netze explodierte nach dem Aufkommen von **GPGPUs**. Diese GPGPUs (engl. *general-purpose computation on graphics processing units*) waren in der Lage, beliebigen Code auszuführen, also nicht mehr nur Rendering-Subroutinen. Die CUDA-Programmiersprache von NVIDIA bot eine Möglichkeit, diesen Code ähnlich wie C abzufassen. Dank der relativ praktischen Programmierbarkeit, der gewaltigen Parallelität und der großen Speicherbandbreite stellen GPGPUs heute eine ideale Plattform für die Programmierung neuronaler Netze dar. Entsprechend rasant verbreitete sich diese Plattform in der Deep-Learning-Forschung (*Raina et al.*, 2009; *Ciresan et al.*, 2010).

Dennoch bleibt das Schreiben von effizientem Code für GPGPUs eine komplexe Aufgabe, die man am besten Spezialisten überlässt. Eine gute Leistung auf einer GPU setzt ganz andere Verfahren als bei einer CPU voraus. Guter CPU-Code zeichnet sich beispielsweise dadurch aus, dass Daten möglichst häufig aus dem Cache gelesen werden. Auf einer GPU werden die meisten beschreibbaren Speicherstellen nicht in den Cache gelegt, damit derselbe Wert schneller zweimal berechnet werden kann – die Alternative wäre eine Berechnung gefolgt von einem weiteren Auslesen des Speichers. GPU-Code zeichnet sich auch generell durch Multithreading aus, wobei die unterschiedlichen Threads sorgfältig aufeinander abgestimmt werden müssen. So sind Speichervorgänge schneller, wenn sie quasi koalesziert werden können. Bei einem koaleszierten Lese- oder Schreibzugriff können mehrere Threads jeweils einen gleichzeitig benötigten Wert im Rahmen eines einzelnen Speicherzugriffs lesen oder schreiben. Unterschiedliche GPU-Modelle können unterschiedliche Arten von Lese- und Schreibmustern koaleszieren. Die meisten Speicherzugriffe lassen sich einfacher koaleszieren, wenn Thread i von n Threads auf das Speicherbyte $i + j$ zugreift und j ein Vielfaches einer Potenz von 2 ist. Die exakten Angaben sind abhängig vom GPU-Modell.

Bei GPUs gilt es auch zu beachten, dass jeder Thread in einer Gruppe zum selben Zeitpunkt dieselbe Anweisung ausführt. Verzweigungen können sich daher auf GPUs als schwierig erweisen. Threads werden in kleine Gruppen, die sogenannten **Warps**, aufgeteilt. Jeder Thread in einem Warp führt in jedem Takt dieselbe Anweisung aus. Müssen unterschiedliche Threads im selben Warp unterschiedliche Code-Pfade ausführen, muss jeder dieser Code-Pfade nacheinander (sequenziell) verfolgt werden, nicht parallel.

Da hochwertiger GPU-Code so schwer zu schreiben ist, sollten Forscher ihre Abläufe so planen, dass möglichst kein neuer GPU-Code zum Testen neuer Modelle oder Algorithmen benötigt wird. Das funktioniert üblicherweise, indem eine Softwarebibliothek mit leistungsstarken Operationen wie Faltung und Matrizenmultiplikation zusammengestellt wird, um dann Modelle zu spezifizieren, die auf diese Bibliothek zugreifen. Ein Beispiel: Die Machine-Learning-Bibliothek Pylearn2 (*Goodfellow et al.*, 2013c) spezifiziert all ihre Machine-Learning-Algorithmen durch Aufrufe von Theano (*Bergstra et al.*, 2010; *Bastien et al.*, 2012) und cuda-convnet (*Krizhevsky*, 2010), die solche leistungsstarken Operationen enthalten. Dieser faktorisierte Ansatz erleichtert auch die Unterstützung mehrerer Hardwaretypen. So lässt sich dasselbe Theano-Programm auf einer CPU oder einer GPU ausführen, ohne dass die Theano-Aufrufe selbst geändert werden müssten. Andere Bibliotheken wie TensorFlow (*Abadi et al.*, 2015) und Torch (*Collobert et al.*, 2011b) bieten ähnliche Funktionen.

12.1.3 Verteilte Implementierungen im großen Maßstab

Häufig reichen die Berechnungsressourcen einer Maschine nicht aus. Wir möchten oder müssen daher die Last von Training und Inferenz auf mehrere Maschinen aufteilen.

Das Verteilen der Inferenz ist einfach, da jedes zu verarbeitende Eingabebeispiel auf einer anderen Maschine ausgeführt werden kann. Man nennt dies **Datenparallelität**.

Auch eine **Modellparallelität**, bei der mehrere Maschinen einen einzelnen Datenpunkt gemeinsam bearbeiten, ist denkbar; in diesem Fall führt jede Maschine einen anderen Teil des Modells aus. Das lässt sich sowohl für Inferenz als auch Training nutzen.

Datenparallelität im Training stellt sich etwas schwieriger dar. Wir können die Größe des Mini-Batches, das für einen einzelnen Schritt im stochastischen Gradientenabstiegsverfahren genutzt wird, erhöhen, im Vergleich zum linearen Verfahren erhalten wir aber meist schlechtere Ergebnisse

hinsichtlich der Optimierungsleistung. Es wäre besser, wenn mehrere Maschinen parallel mehrere Schritte im Gradientenabstiegsverfahren berechnen dürften. Leider ist die Standarddefinition des Gradientenabstiegsverfahrens ein ganz und gar sequenzieller Algorithmus: Der Gradient im Schritt t ist eine Funktion der Parameter, die sich aus Schritt $t - 1$ ergeben.

Als Lösung bietet sich das **asynchrone stochastische Gradientenabstiegsverfahren** an (*Bengio et al., 2001; Recht et al., 2011*). Dabei greifen mehrere Prozessorkerne auf einen gemeinsamen Speicherbereich mit den Parametern zu. Jeder Kern liest Parameter ohne eine Sperre (engl. *lock*), berechnet einen Gradienten und erhöht dann die Parameter ohne eine Sperre. Das verringert die durchschnittliche Höhe der Verbesserung der einzelnen Gradientenschritte, da einige der Kerne die Fortschritte der anderen überschreiben, aber die höhere Frequenz der Generierung der Schritte führt dazu, dass der Lernprozess insgesamt schneller voranschreitet. *Dean et al. (2012)* waren Wegbereiter der Multi-Maschinen-Implementierung für dieses Gradientenabstiegsverfahren ohne Sperren, bei dem die Parameter mittels **Parameterserver** anstelle gemeinsamen Speichers verwaltet werden. Das verteilte asynchrone Gradientenabstiegsverfahren ist nach wie vor die wichtigste Strategie beim Trainieren großer tiefer Netze; es wird von den meisten großen Deep-Learning-Gruppierungen in der Branche eingesetzt (*Chilimbi et al., 2014; Wu et al., 2015*). In der wissenschaftlichen Deep-Learning-Forschung sind verteilte Lernsysteme dieser Größe nur selten erschwinglich, aber ein Teil der Forschung fokussiert verteilte Netze auf Basis relativ günstiger Hardware im Universitätsbereich (*Coates et al., 2013*).

12.1.4 Modellkomprimierung

In vielen kommerziellen Anwendungen ist es sehr viel wichtiger, dass in einem Machine-Learning-Modell der Zeit- und Speicheraufwand beim Durchführen der Inferenz niedrig ist; für das Training ist dieser Aufwand weniger relevant. Für Anwendungen, die keine Personalisierung erfordern, kann ein Modell einmalig trainiert und dann von Milliarden von Anwendern eingesetzt werden. Häufig stehen dem Endanwender weniger Ressourcen als dem Entwickler zur Verfügung. So könnte man ein Spracherkennungsnetz auf einem leistungsstarken Computer-Cluster trainieren, um es dann auf Mobiltelefonen zu nutzen.

Eine wesentliches Verfahren zur Reduzierung des Aufwands für die Inferenz ist die **Modellkomprimierung** (engl. *model compression*) (*Buciluă et al., 2006*). Dabei wird das ursprüngliche aufwendige Modell durch ein

kleineres Modell ersetzt, für dessen Speicherung und Berechnung weniger Speicherplatz und weniger Zeit benötigt werden.

Die Modellkomprimierung ist dann anwendbar, wenn die Größe des ursprünglichen Modells unbedingt erforderlich ist, eine Überanpassung zu verhindern. In den meisten Fällen ist das Modell mit dem niedrigsten Generalisierungsfehler ein Ensemble mehrerer unabhängig voneinander trainierter Modelle. Das Evaluieren aller n Ensembleelemente ist aufwendig. Manchmal generalisiert sogar ein einzelnes Modell besser, wenn es groß ist (beispielsweise wenn es mit Dropout regularisiert wird).

Diese großen Modelle erlernen eine Funktion $f(\mathbf{x})$ unter Zuhilfenahme von sehr viel mehr Parametern, als für die Aufgabe eigentlich erforderlich sind. Ihre Größe ist allein eine Folge der eingeschränkten Anzahl an Trainingsbeispielen. Sobald wir diese Funktion $f(\mathbf{x})$ angepasst haben, können wir eine Trainingsdatenmenge mit unendlich vielen Beispielen erzeugen, indem wir f ganz einfach auf zufällig ausgewählte Punkte \mathbf{x} anwenden. Anschließend trainieren wir das neue kleinere Modell für eine Anpassung von $f(\mathbf{x})$ auf diese Punkte. Für die effizienteste Nutzung der Kapazität des neuen kleinen Modells sollten neue \mathbf{x} -Punkte aus einer Verteilung gezogen werden, die den tatsächlichen Testeingaben ähnelt, die dem Modell später vorgelegt werden. Zu diesem Zweck können Trainingsbeispiele verzerrt oder Punkte aus einem generativen Modell, das mithilfe der ursprünglichen Trainingsdatenmenge trainiert wurde, als Stichproben gezogen werden.

Alternativ kann man ein kleineres Modell nur mithilfe der ursprünglichen Trainingspunkte trainieren, aber das Training so gestalten, dass es andere Merkmale des Modells kopiert, zum Beispiel die A-posteriori-Verteilung über die inkorrekt Klassen (*Hinton et al.*, 2014, 2015).

12.1.5 Dynamische Struktur

Ein Verfahren zur Beschleunigung von Datenverarbeitungssystemen insgesamt besteht darin, Systeme zu erstellen, die eine **dynamische Struktur** im Graphen aufweisen, der die zur Verarbeitung einer Eingabe notwendige Berechnung beschreibt. Datenverarbeitungssysteme können dynamisch entscheiden, welche Teilmenge vieler neuronaler Netze für eine bestimmte Eingabe ausgeführt wird. Einzelne neuronale Netze können intern ebenfalls dynamische Strukturen nutzen, indem sie entscheiden, welche Teilmenge der Merkmale (verdeckte Einheiten) für bestimmte Daten der Eingabe berechnet wird. Diese Form der dynamischen Struktur in neuronalen Netzen wird auch als **bedingte Berechnung** (engl. *conditional computation*) bezeichnet (*Bengio*, 2013; *Bengio et al.*, 2013b). Da viele Komponenten der

Architektur möglicherweise nur für eine kleine Anzahl möglicher Eingaben relevant sind, kann das System schneller arbeiten, indem diese Merkmale nur dann berechnet werden, wenn sie auch benötigt werden.

Die dynamische Struktur von Berechnungen ist ein grundlegendes Prinzip der Informatik, das allgemein in der Softwareentwicklung genutzt wird. Die einfachsten Versionen einer dynamischen Struktur in neuronalen Netzen beruhen auf der Auswahl der Teilmenge einer Gruppe neuronaler Netze (oder anderer Machine-Learning-Modelle), die für eine bestimmte Eingabe zum Einsatz kommen.

Ein beliebtes Verfahren für eine schnellere Inferenz in einem Klassifikator besteht in einer **Kaskade** von Klassifikatoren. Das Kaskadenverfahren kann genutzt werden, wenn das Vorhandensein eines seltenen Objekts (oder Ereignisses) erkannt werden soll. Um sicher festzustellen, dass das Objekt vorhanden ist, müssen wir einen ausgefeilten Klassifikator mit hoher Kapazität einsetzen – und das ist aufwendig. Da das Objekt selten ist, lassen sich Eingaben, die das Objekt nicht enthalten, normalerweise mit sehr viel weniger Berechnungsaufwand aussortieren. Für solche Fälle können wir eine Sequenz mit mehreren Klassifikatoren trainieren. Die ersten Klassifikatoren in der Sequenz weisen eine geringe Kapazität auf und sind auf eine hohe Trefferquote trainiert. Sie sollen also sicherstellen, dass wir nicht versehentlich eine Eingabe abweisen, die das Objekt enthält. Der letzte Klassifikator ist auf eine hohe Genauigkeit trainiert. Zum Testzeitpunkt führen wir die Inferenz durch Ausführen der Klassifikatoren nacheinander durch, wobei jedes Beispiel verworfen wird, sobald ein Element der Kaskade es abweist. So können wir unterm Strich die Präsenz von Objekten mit hoher Konfidenz prüfen (unter Verwendung eines Modells mit hoher Kapazität), ohne für jedes Beispiel den Aufwand der vollständigen Inferenz auf uns nehmen zu müssen. Es gibt zwei unterschiedliche Wege, mit denen eine Kaskade eine hohe Kapazität erzielen kann: Der erste stattet jedes einzelne der später auftretenden Kaskadenelemente mit einer höheren Kapazität aus. Das hat offenkundig eine hohe Kapazität des Gesamtsystems zur Folge, da einige seiner Elemente eine hohe Kapazität aufweisen. Man kann auch eine Kaskade aufbauen, in der jedes einzelne Modell eine geringe Kapazität aufweist, das System insgesamt durch Kombination vieler kleiner Modelle aber eine hohe Kapazität besitzt. *Viola und Jones* (2001) verwendeten eine Kaskade verstärkter Entscheidungsbäume für die Implementierung einer schnellen und robusten Gesichtserkennung auf kleinen Digitalkameras. Ihr Klassifikator setzt im Wesentlichen einen verschobenen Ausschnitt ein; viele dieser Ausschnitte werden untersucht und verworfen, wenn sie keine Gesichter enthalten. Eine andere Kaskadenversion nutzt die frühen Modelle, um

eine Form von hartem Aufmerksamkeitsmechanismus (engl. *hard attention mechanism*) zu implementieren: Die frühen Elemente der Kaskade spüren ein Objekt auf, die späteren übernehmen die weitergehende Verarbeitung, sobald die Position des Objekts bekannt ist. Zum Beispiel transkribiert Google Hausnummern aus Street-View-Aufnahmen in einer zweistufigen Kaskade, die zunächst mit einem Machine-Learning-Modell nach der Hausnummer sucht, um sie dann mit einem weiteren Modell zu transkribieren (Goodfellow et al., 2014d).

Entscheidungsbäume sind ein Beispiel für eine dynamische Struktur, da jeder Knoten im Baum darüber entscheidet, welcher Ast für die jeweilige Eingabe verfolgt wird. Eine einfache Möglichkeit, Deep Learning und dynamische Strukturen zusammenzubringen, ist das Trainieren eines Entscheidungsbaums, bei dem jeder Knoten ein neuronales Netz zur Pfadauswahl nutzt (Guo und Gelfand, 1992); allerdings ist dies kein Standardverfahren, wenn es in erster Linie um schnellere Inferenzberechnungen geht.

Ebenso kann man ein neuronales Netz, das als **Gater** bezeichnet wird, zum Auswählen eines von mehreren **Expertennetzen** zum Berechnen der Ausgabe der aktiven Eingabe nutzen. Die erste Ausprägung dieser Idee ist als **Mixture of Experts** bekannt (Nowlan, 1990; Jacobs et al., 1991). Darin gibt der Gater eine Menge von Wahrscheinlichkeiten oder Gewichten aus (die mittels softmax-Nichtlinearität ermittelt wurden), und zwar ein Element pro Experte; das Endergebnis wird aus der gewichteten Kombination der Ausgabe der Experten ermittelt. In diesem Fall führt der Einsatz des Gaters nicht zu einem geringeren Berechnungsaufwand. Wenn allerdings für jedes Beispiel ein einzelner Experte vom Gater ausgewählt wird, erhalten wir eine **harte Mixture of Experts** (Collobert et al., 2001, 2002), die Trainings- und Inferenzdauer stark beschleunigen kann. Dieses Verfahren funktioniert gut, wenn die Anzahl der Gating-Entscheidungen klein ist, denn sie ist nicht kombinatorisch. Wenn wir aber unterschiedliche Teilmengen von Einheiten oder Parametern auswählen möchten, steht ein »soft switch« (dt. *sanftes Umschalten*) nicht zur Debatte, da für diesen sämtliche Gater-Konfigurationen aufgezählt (und Ausgaben dafür berechnet) werden müssen. Aufgrund dieser Schwierigkeit wurden mehrere Ansätze zum Trainieren kombinatorischer Gater untersucht. Bengio et al. (2013b) haben mehrere Schätzer für den Gradienten der Gating-Wahrscheinlichkeiten auf den Prüfstand gestellt; Bacon et al. (2015) und Bengio et al. (2015a) nutzen Verfahren des Reinforcement Learnings (Policy-Gradient), um eine Form des bedingten Dropouts für Blöcke verdeckter Einheiten zu erlernen und so den Berechnungsaufwand tatsächlich zu reduzieren, ohne dass es zu negativen Auswirkungen auf die Qualität der Approximation kommt.

Eine weitere Art dynamischer Strukturen ist ein Schalter, bei dem eine verdeckte Einheit abhängig vom Kontext Eingaben unterschiedlicher Einheiten erhält. Dieses dynamische Routing lässt sich als Aufmerksamkeitsmechanismus betrachten (Olshausen et al., 1993). Bisher hat sich der Einsatz von hard switch (dt. *hartes Umschalten*) für Anwendungen im großen Maßstab nicht als wirkungsvoll erwiesen. Aktuelle Ansätze nutzen stattdessen ein gewichtetes Mittel über möglichst viele Eingaben, sodass sie nicht von allen möglichen Rechenvorteilen einer dynamischen Struktur profitieren. Abschnitt 12.4.5.1 behandelt moderne Aufmerksamkeitsmechanismen.

Ein großes Hindernis beim Einsatz dynamisch strukturierter Systeme ist das geringere Maß an Parallelität, das Systeme mit unterschiedlichen Code-Verzweigungen bei unterschiedlichen Eingaben auszeichnet. Das bedeutet, dass wenige Operationen im Netz als Matrizenmultiplikation oder Batch-Faltung für einen Mini-Batch mit Beispielen beschrieben werden können. Wir können zwar spezielle Subroutinen schreiben, die jedes Beispiel mit einem anderen Kernel falten oder jede Zeile einer Entwurfsmatrix mit einer anderen Menge von Gewichtungsspalten multiplizieren, doch leider ist es schwer, diese Subroutinen effizient zu implementieren. Implementierungen auf CPU-Basis sind langsam, da die Cache-Kohärenz fehlt. Implementierungen auf GPU-Basis sind langsam, da die koalescierten Speicherzugriffe fehlen und Warps serialisiert werden müssen, wenn die Elemente eines Warps unterschiedlichen Verzweigungen folgen. In einigen Fällen lässt sich dies durch eine Aufteilung der Beispiele in Gruppen, die denselben Zweig folgen, und die anschließende gleichzeitige Verarbeitung dieser Gruppen mit Beispielen abschwächen. Bei einer festen Anzahl von Beispielen in einem Offline-Szenario kann dieses Verfahren die Verarbeitungsdauer reduzieren. In einem Echtzeit-Szenario, in dem kontinuierlich Beispiele ausgewertet werden müssen, kann eine Aufteilung dagegen zu Problemen in der Lastverteilung (engl. *load-balancing*) führen. Ein Beispiel: Wenn wir eine Maschine für die Verarbeitung des ersten Schritts einer Kaskade verwenden und eine weitere Maschine für den letzten Schritt in der Kaskade, besteht für die erste die Gefahr einer Überlastung, während die letzte eher Gefahr läuft, nicht ausgelastet zu sein. Ähnliche Probleme entstehen, wenn jede Maschine für die Implementierung unterschiedlicher Knoten eines neuronalen Entscheidungsbaums verwendet wird.

12.1.6 Spezialisierte Hardwareimplementierungen tiefer Netze

Seit den Anfangstagen der Forschung zu neuronalen Netzen haben Hardwareentwickler daran gearbeitet, spezialisierte Hardwareimplementierungen zu schaffen, die Training und/oder Inferenz der Algorithmen in neuronalen Netzen beschleunigen. Informationen hierzu finden Sie in älteren und jüngeren Abhandlungen über spezielle Hardware für tiefe Netze (*Lindsey und Lindblad, 1994; Beiu et al., 2003; Misra und Saha, 2010*).

In den letzten Jahrzehnten wurden unterschiedliche Formen spezieller Hardware mit ASICs (engl. *application-specific integrated circuits*, dt. *anwendungsspezifische integrierte Schaltungen*) entwickelt (*Graf und Jackel, 1989; Mead und Ismail, 2012; Kim et al., 2009; Pham et al., 2012; Chen et al., 2014a,b*), entweder als digitale (binäre Darstellung von Zahlen), analoge (physische Implementierung stetiger Werte in Form von Spannungen oder Strömen) (*Graf und Jackel, 1989; Mead und Ismail, 2012*) oder hybride Implementierungen (Kombination digitaler und analoger Komponenten). In der jüngeren Vergangenheit wurden flexible FPGA-Implementierungen entwickelt (**FPGA = Field Programmable Gate Array**, also *im Feld programmierbare (Logik-)Gatter-Anordnung*). Dabei können die Einzelheiten des Schaltkreises nach der Fertigung auf den Chip geschrieben werden.

Obwohl Softwareimplementierungen auf Mehrzweckprozessoren (CPUs und GPUs) meist eine Genauigkeit von 32 oder 64 Bit für Gleitkommazahlen verwenden, ist schon lange bekannt, dass auch eine geringe Genauigkeit genutzt werden kann – zumindest für die Inferenz (*Holt und Baker, 1991; Holi und Hwang, 1993; Presley und Haggard, 1994; Simard und Graf, 1994; Wawrynek et al., 1996; Savich et al., 2007*). Das Thema ist in den letzten Jahren ins Zentrum der Aufmerksamkeit gerückt, da Deep Learning in industriellen Produkten immer beliebter wurde und sich bei den GPUs die Vorteile schnellerer Hardware gezeigt haben. Ein weiterer Faktor, der bei der aktuellen Forschung im Bereich spezieller Hardware für tiefe Netze eine Rolle spielt, ist, dass der Fortschritt bei einzelnen CPU- oder GPU-Kernen sich verlangsamt hat. Die meisten aktuellen Verbesserungen in der Rechengeschwindigkeit sind einer Parallelisierung der Kerne geschuldet (bei CPUs und GPUs). Damit unterscheidet sich die Lage deutlich von der in den 1990ern (dem vorherigen Zeitalter der neuronalen Netze); damals konnten die Hardwareimplementierungen neuronaler Netze (von der Idee bis zur Umsetzung auf einem Chip können gut und gerne zwei Jahre vergehen) nicht mit dem rasanten Fortschritt und den geringen Preisen von Mehrzweck-CPUs mithalten. Der Bau spezieller Hardware ermöglicht es somit, die

Grenzen weiter zu verschieben, während neue Hardwaredesigns für Geräte mit geringerer Performance (wie Smartphones) entwickelt werden, die auf verbreitete öffentliche Deep-Learning-Anwendungen abzielen, zum Beispiel Sprache, Computer Vision oder natürliche Sprache.

Jüngere Arbeiten zu Implementierungen von neuronalen Netzen auf Backpropagation-Basis mit geringer Genauigkeit (*Vanhoucke et al.*, 2011; *Courbariaux et al.*, 2015; *Gupta et al.*, 2015) deuten darauf hin, dass eine Genauigkeit zwischen 8 und 16 Bit für das Training neuronaler Netze mit Backpropagation ausreichen kann. Es ist klar, dass während des Trainings eine höhere Genauigkeit als während der Inferenz benötigt wird und dass einige Formen der dynamischen Festkommadarstellung von Zahlen die Anzahl der Bits pro Zahl reduzieren können. Klassische Festkommazahlen sind auf einen festen Bereich eingeschränkt (der einem bestimmten Exponenten in einer Gleitkommadarstellung entspricht). Dynamische Festkommadarstellungen nutzen diesen Bereich über eine Reihe von Zahlen hinweg (zum Beispiel alle Gewichte in einer Schicht). Der Einsatz von Festkomma- anstelle von Gleitkommadarstellungen und der Einsatz von weniger Bits pro Zahl reduziert den Hardwareflächeninhalt, den Leistungsbedarf und die Rechendauer für das Ausführen von Multiplikationen; und Multiplikationen sind die anspruchsvollsten Operationen beim Verwenden oder Trainieren eines modernen tiefen Netzes mit Backpropagation.

12.2 Computer Vision

Computer Vision war schon immer eines der aktivsten Forschungsfelder für Deep-Learning-Anwendungen, da das Sehen eine Aufgabe ist, die Menschen und Tiere quasi nebenbei erledigen, während Computer damit große Probleme haben (*Ballard et al.*, 1983). Viele der wichtigsten Standard-Benchmark-Aufgaben für Deep-Learning-Algorithmen erfordern eine Form der Objekterkennung oder optischen Zeichenerkennung.

Computer Vision ist ein sehr breites Anwendungsfeld mit einer Vielzahl von Möglichkeiten zur Bildverarbeitung und einer erstaunlichen Menge an Anwendungen. Dazu gehören zum Beispiel Anwendungen, die bestimmte Aspekte des menschlichen Sehens nachbilden sollen, darunter die Gesichtserkennung, oder auch die Entwicklung ganz neuer optischer Fähigkeiten wie die Erkennung von Schallwellen aus den Schwingungen von Objekten in einem Video (*Davis et al.*, 2014). Der Großteil der Deep-Learning-Forschung zur Computer Vision konzentriert sich allerdings weniger auf neue und exotische Fähigkeiten, sondern vielmehr auf die Nachbildung menschlicher Fähigkeiten

mithilfe der Künstlichen Intelligenz. Deep Learning für Computer Vision wird meist zur Objekterkennung eingesetzt. Dabei wird angegeben, ob ein Objekt in einem Bild vorhanden ist, es werden Rahmen um die einzelnen Objekte in einem Bild gezeichnet, es wird eine Reihe von Symbolen zu einem Bild transkribiert oder jedes Pixel in einem Bild wird mit einem Label gekennzeichnet, das die Information enthält, zu welchem Objekt es gehört. Da die generative Modellierung ein Leitprinzip der Deep-Learning-Forschung ist, gibt es auch viele Arbeiten zur Bildsynthese mittels tiefer Modelle. Obwohl die Bildsynthese *für sich genommen* nicht zur Computer Vision zählt, sind Modelle, die eine Bildsynthese ermöglichen, meist für die Bildwiederherstellung hilfreich – eine Computer-Vision-Aufgabe, bei der Defekte in Bildern behoben oder störende Objekte daraus entfernt werden.

12.2.1 Vorverarbeitung

Viele Anwendungsbereiche erfordern eine umfassende Vorverarbeitung, da die ursprünglichen Eingangsdaten in einer Form vorliegen, die für viele Deep-Learning-Architekturen nur schwer darzustellen ist. In der Computer Vision ist meist nur eine geringe Vorverarbeitung notwendig. Die Bilder sollten standardisiert werden, damit ihre Pixel alle im selben angemessenen Bereich liegen, zum Beispiel [0,1] oder [-1, 1]. Werden Bilder aus [0,1] mit solchen aus [0, 255] vermischt, führt dies meistens zu einem Fehler. Der einzige unverzichtbare Schritt in der Vorverarbeitung ist also die Formatierung der Bilder, sodass sie dieselbe Skalierung aufweisen. Außerdem setzen viele Computer-Vision-Architekturen Standardabmessungen voraus. Um diese zu erreichen, müssen Bilder beschnitten oder gestreckt werden. Allerdings ist diese Skalierung nicht immer erforderlich. Einige Modelle mit Faltung können mit unterschiedlich großen Eingaben arbeiten und passen die Größe ihrer Pooling-Bereiche dynamisch an, um eine konstante Ausgabegröße zu erreichen (Waibel *et al.*, 1989). Andere Modelle mit Faltung verfügen über variable Ausgabegrößen, die automatisch abhängig von der Größe der Eingabe skaliert werden. Das ist zum Beispiel bei Modellen zum Denoising (Entrauschen) oder zur Kennzeichnung jedes einzelnen Pixels mit einem dazugehörigen Label der Fall (Hadsell *et al.*, 2007).

Das Erweitern des Datensatzes kann als Möglichkeit zur Vorverarbeitung der Trainingsdatenmenge betrachtet werden. Es handelt sich um eine hervorragende Möglichkeit, um den Generalisierungsfehler der meisten Computer-Vision-Modelle zu reduzieren. Eine ähnliche Idee, die zum Testzeitpunkt greift, besteht darin, dem Modell viele verschiedene Versionen derselben Eingabe vorzulegen (zum Beispiel dasselbe Bild mit leicht unterschiedli-

chem Beschnitt) und den verschiedenen Instanziierungen des Modells die Entscheidung über die Ausgabe zu überlassen. Das kann man auch als Ensembleansatz betrachten; in jedem Fall hilft es dabei, den Generalisierungsfehler zu reduzieren.

Weitere Arten der Vorverarbeitung werden auf die Trainings- und die Testdatenmenge angewandt, um jedes Beispiel in eine kanonischere Form zu bringen und so die Variationen zu reduzieren, die das Modell berücksichtigen muss. Weniger Variation in den Daten kann den Generalisierungsfehler und die Modellgröße, die zur Anpassung der Trainingsdatenmenge benötigt wird, reduzieren. Einfachere Aufgaben lassen sich mit kleineren Modellen lösen – und für einfache Lösungen ist eine gute Fähigkeit zur Generalisierung wahrscheinlicher. Eine solche Vorverarbeitung ist normalerweise dazu gedacht, eine gewisse Variabilität aus den Eingangsdaten zu entfernen, die für einen menschlichen Designer leicht zu beschreiben ist; außerdem muss der Designer sicher sein, dass diese Variabilität für die Aufgabe nicht von Bedeutung ist. Wenn große Datensätze und große Modelle für das Training verwendet werden, ist diese Art Vorverarbeitung oft unnötig. Es ist in diesem Fall besser, das Modell selbst lernen zu lassen, gegenüber welchen Variabilitäten es invariant werden sollte. Ein Beispiel: Das AlexNet-System zur Klassifizierung von ImageNet kommt mit einem vorbereitenden Schritt aus, nämlich der Subtraktion des Mittelwerts aller Trainingsbeispiele für jedes Pixel (Krizhevsky et al., 2012).

12.2.1.1 Kontrastnormalisierung

Eine der offensichtlichsten Quellen der Variation, die für viele Aufgaben sicher entfernt werden kann, ist der Kontrast im Bild. Kontrast bezeichnet ganz einfach den Unterschied zwischen den hellen und den dunklen Pixeln eines Bildes. Es gibt viele Möglichkeiten, den Bildkontrast zu quantifizieren. Im Deep-Learning-Kontext bezieht sich Kontrast meist auf die Standardabweichung der Pixel in einem Bild oder einem Bildbereich. Angenommen, wir haben ein Bild, das durch einen Tensor $\mathbf{X} \in \mathbb{R}^{r \times c \times 3}$ dargestellt wird (wobei $X_{i,j,1}$ die Rot-Intensität in Zeile i und Spalte j ist, $X_{i,j,2}$ die Grün-Intensität und $X_{i,j,3}$ die Blau-Intensität). Der Kontrast des gesamten Bildes ergibt sich dann aus

$$\sqrt{\frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^3 (X_{i,j,k} - \bar{\mathbf{X}})^2}, \quad (12.1)$$

wobei $\bar{\mathbf{X}}$ die mittlere Intensität des gesamten Bildes ist:

$$\bar{\mathbf{X}} = \frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^3 X_{i,j,k}. \quad (12.2)$$

Die **globale Kontrastnormalisierung** (engl. *global contrast normalization*, GCN) soll einen variablen Kontrast zwischen Bildern verhindern; hierzu wird der Mittelwert von jedem Bild subtrahiert, woraufhin eine erneute Skalierung erfolgt, die dafür sorgt, dass die Standardabweichung über alle Pixel hinweg gleich einer Konstanten s ist. Dieser Ansatz wird dadurch kompliziert, dass es keinen Skalierungsfaktor zum Ändern des Kontrasts eines Bildes ohne Kontrast gibt (das ist ein Bild, dessen Pixel alle dieselbe Intensität aufweisen). Bilder mit einem sehr niedrigen, aber von Null verschiedenen Kontrast haben oft nur einen geringen Informationsgehalt. Eine Division durch die wahre Standardabweichung führt in solchen Fällen normalerweise nur dazu, dass das Sensorrauschen oder die Kompressionsartefakte verstärkt werden. Darum wird ein kleiner positiver Regularisierungsparameter λ eingeführt, der den Schätzwert der Standardabweichung verzerrt. Alternativ kann auch der Nenner auf einen Mindestwert ϵ eingeschränkt werden. Für ein Eingabebild \mathbf{X} erzeugt die GCN ein Ausgabebild \mathbf{X}' gemäß der Definition

$$X'_{i,j,k} = s \frac{X_{i,j,k} - \bar{X}}{\max \left\{ \epsilon, \sqrt{\lambda + \frac{1}{3rc} \sum_{i=1}^r \sum_{j=1}^c \sum_{k=1}^3 (X_{i,j,k} - \bar{X})^2} \right\}}. \quad (12.3)$$

Datensätze mit sehr großen Bildern, aus denen nur die relevanten Objekte herausgenommen werden, enthalten nur selten Bilder mit nahezu konstanter Intensität. In diesen Fällen kann das Problem des kleinen Nenners (engl. *small denominator problem*) quasi vernachlässigt werden, indem $\lambda = 0$ gesetzt und die Division durch 0 in extrem seltenen Fällen durch Setzen von ϵ auf einen extrem niedrigen Wert wie 10^{-8} verhindert wird. Diesen Ansatz nutzen Goodfellow et al. (2013a) für den CIFAR-10-Datensatz. Kleine Bilder, die zufällig beschnitten werden, können eher eine nahezu konstante Intensität aufweisen, sodass eine aggressive Regularisierung Vorteile bietet. Coates et al. (2011) verwendeten $\epsilon = 0$ und $\lambda = 10$ für kleine zufällig ausgewählte Bereiche aus CIFAR-10.

Der Maßstabsparameter s kann meist auf 1 gesetzt (siehe Coates et al., 2011) oder so gewählt werden, dass jedes einzelne Pixel über alle Beispiele eine Standardabweichung nahe 1 aufweist (siehe Goodfellow et al., 2013a).

Die Standardabweichung in Gleichung 12.3 stellt lediglich eine erneute Skalierung der L^2 -Norm des Bildes dar (unter der Voraussetzung, dass der Mittelwert des Bildes bereits entfernt wurde). Für die GCN sollte vorzugsweise die Standardabweichung und nicht die L^2 -Norm verwendet werden, da die Standardabweichung eine Division durch die Anzahl der Pixel enthält; daher kann für eine GCN auf Basis der Standardabweichung derselbe Wert s ungeachtet der Bildgröße verwendet werden. Allerdings kann die Feststellung, dass die L^2 -Norm proportional zur Standardabweichung ist, dabei helfen, ein nützliches Gespür zu entwickeln. Die GCN kann als Mapping von Beispielen zu einer Kugelschale betrachtet werden. Abbildung 12.1 verdeutlicht dies. Das ist eine nützliche Eigenschaft, denn neuronale Netze reagierten oft besser auf Richtungen im Raum als auf exakte Positionen. Die Reaktion auf mehrere Abstände in derselben Richtung setzt verdeckte Einheiten mit kollinearen Gewichtungsvektoren, aber unterschiedlichen Verzerrungen voraus. Eine solche Koordininierung kann für den Lernalgorithmus schwer erkennbar sein. Außerdem haben viele flache graphische Modelle Probleme mit der Darstellung mehrerer separater Modi auf derselben Linie. Die GCN umgeht diese Probleme, indem jedes Beispiel auf eine Richtung (anstelle einer Richtung und einer Strecke) reduziert wird.

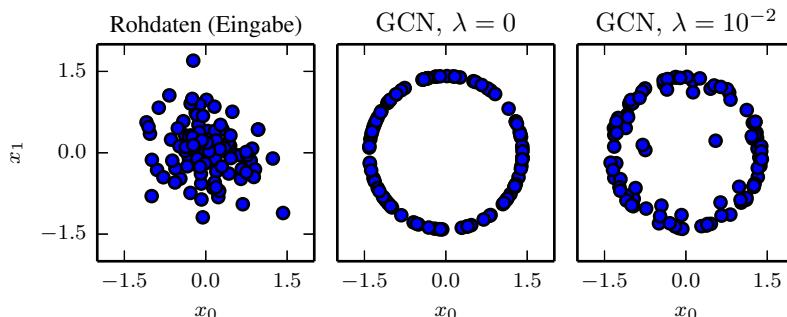


Abbildung 12.1: GCN ordnet Beispiele auf einer Kugel an. (*Links*) Rohdaten auf der Eingabeseite können eine beliebige Norm aufweisen. (*Mitte*) Eine GCN mit $\lambda = 0$ ordnet alle von Null verschiedenen Beispiele einer perfekten Kugel zu. Hier ist $s = 1$ und $\epsilon = 10^{-8}$. Da wir die GCN auf Basis einer Normalisierung der Standardabweichung anstelle der L^2 -Norm verwenden, ist die resultierende Kugel keine Einheitskugel. (*Rechts*) Eine regularisierte GCN mit $\lambda > 0$ zieht Beispiele in Richtung der Kugel, ohne jegliche Variation in der Norm wegfallen zu lassen. s und ϵ sind unverändert.

Es gibt eine Vorverarbeitung, die als **Sphering** bezeichnet wird – dabei handelt es sich aber nicht um die GCN. Sphering führt nicht dazu, dass die Daten auf einer Kugelschale liegen; es geht dabei vielmehr darum, die Hauptkomponenten so zu skalieren, dass sie die gleiche Varianz aufweisen,

wodurch die mehrdimensionale Normalverteilung in der Hauptkomponentenanalyse (engl. *principal components analysis*, PCA) kugelförmige (engl. *spherical*) Konturen aufweist. Sphering wird häufig auch als **Whitening** bezeichnet.

Die GCN scheitert häufig daran, Bildmerkmale hervorzuheben, die wir gerne betonen möchten, zum Beispiel Kanten und Ecken. Bei einer Aufnahme mit einem großen dunklen und einem großen hellen Bereich (zum Beispiel ein Platz, der zur Hälfte im Schatten eines Gebäudes liegt) sorgt die GCN dafür, dass es einen großen Unterschied zwischen der Helligkeit im dunklen Bereich und der Helligkeit im hellen Bereich gibt. Allerdings wird dabei nicht sichergestellt, dass die Kanten im dunklen Bereich herausstechen.

Damit kommen wir zur **lokalen Kontrastnormalisierung** (engl. *local contrast normalization*, LCN). Die LCN sorgt dafür, dass der Kontrast in kleinen Ausschnitten normalisiert wird, nicht im gesamten Bild. Abbildung 12.2 vergleicht die GCN mit der LCN.

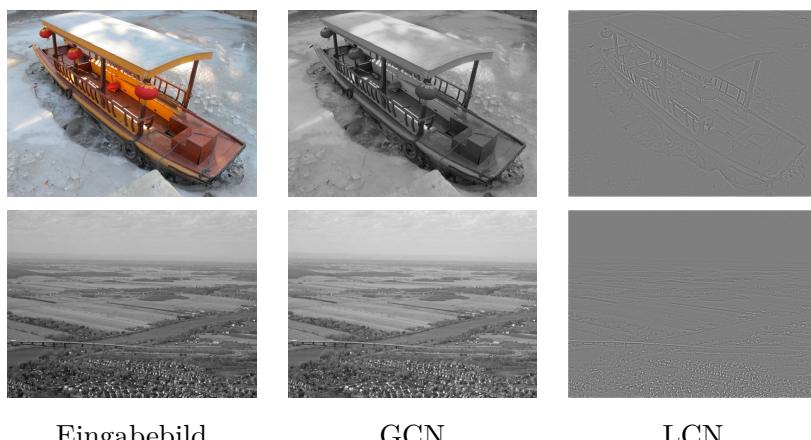


Abbildung 12.2: Vergleich von GCN und LCN. Die optischen Auswirkungen der GCN sind eher subtil. Der Maßstab sämtlicher Bilder wird in etwa vereinheitlicht, sodass der Lernalgorithmus nicht durch viele Maßstäbe belastet wird. Die LCN verändert das Bild stärker, da sie alle Bereiche konstanter Intensität verwirft. So kann sich das Modell ganz auf die Kanten konzentrieren. In Bereichen mit einer zu feinen Textur, zum Beispiel bei Häusern in der zweiten Reihe, gehen möglicherweise einige Details aufgrund einer zu hohen Bandbreite im Normalisierungskernell verloren.

Die LCN kann auf mehrere Arten definiert werden. In allen Fällen wird jedes Pixel durch Subtrahieren eines Mittelwerts der nahegelegenen Pixel und Division durch eine Standardabweichung der nahegelegenen Pixel verändert.

In einigen Fällen handelt es sich dabei tatsächlich um den Mittelwert und die Standardabweichung aller Pixel in einem rechteckigen Ausschnitt, in dessen Zentrum das zu ändernde Pixel liegt (*Pinto et al.*, 2008). In anderen Fällen werden ein gewichtetes Mittel und eine gewichtete Standardabweichung mithilfe normalverteilter Gewichte bestimmt, die auf dem zu ändernden Pixel zentriert sind. Bei Farbaufnahmen behandeln einige Herangehensweisen jeden Farbkanal unabhängig voneinander, während andere die Daten aus den verschiedenen Kanälen zur Normalisierung der einzelnen Pixel verwenden (*Sermanet et al.*, 2012).

Die LCN lässt sich meist effektiv implementieren, indem separierbare Faltung (engl. *separable convolution*) (vgl. Abschnitt 9.8) eingesetzt wird, um Merkmalskarten lokaler Mittelwerte und lokaler Standards zu berechnen; anschließend können die elementweise Subtraktion und die elementweise Division auf unterschiedliche Merkmalskarten angewandt werden.

Die LCN ist eine differenzierbare Operation; sie kann auch als Nichtlinearität für verdeckte Schichten eines Netzes oder als Vorverarbeitung für die Eingabe verwendet werden.

Wie die GCN muss auch die LCN im Normalfall regularisiert werden, um eine Division durch Null zu verhindern. Tatsächlich ist dies bei der LCN noch wichtiger, da sie meist auf kleinere Ausschnitte wirkt. Für kleinere Ausschnitte ist die Wahrscheinlichkeit, dass sie durchgehend nahezu identische Werte enthalten, höher, sodass eine Standardabweichung von Null wahrscheinlicher wird.

12.2.1.2 Erweitern des Datensatzes

Wie in Abschnitt 7.4 beschrieben, ist es einfach, die Generalisierung eines Klassifikators zu steigern, indem die Größe der Trainingsdatenmenge erhöht wird, und zwar anhand zusätzlicher Kopien der Trainingsbeispiele, die klassenerhaltende Transformationen aufweisen. Die Objekterkennung ist eine Klassifizierungsaufgabe, die für diese Art der Datenerweiterung besonders geeignet ist, da die Klasse invariant gegenüber so vielen Transformationen ist und die Eingabe problemlos mittels vieler geometrischer Operationen transformiert werden kann. Wie wir bereits gezeigt haben, können Klassifikatoren von zufälligen Verschiebungen, Drehungen und manchmal Spiegelungen der Eingabe zum Erweitern des Datensatzes profitieren. In speziellen Computer-Vision-Anwendungen werden meist komplexere Transformationen zur Erweiterung verwendet. Dazu gehören zufällige Störungen der Bildfarben (*Krizhevsky et al.*, 2012) und nichtlineare geometrische Verzerrungen der Eingabe (*LeCun et al.*, 1998b).

12.3 Spracherkennung

Die Aufgabe der Spracherkennung besteht darin, ein akustisches Signal, das eine Äußerung in einer gesprochenen natürlichen Sprache enthält, einer passenden Sequenz von Wörtern, die der Absicht des Sprechers entspricht, zuzuordnen. Sei $\mathbf{X} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)})$ die Sequenz der akustischen Eingabeketten (üblicherweise wird die Aufnahme in 20 ms lange Frames zerlegt). Die meisten Spracherkennungssysteme führen eine Vorverarbeitung der Eingabe mithilfe sorgfältig ausgearbeiteter Merkmale durch, aber einige tiefe Lernsysteme erlernen die Merkmale auch anhand der unbearbeiteten Eingabe (engl. *raw input*) (*Jaitly und Hinton*, 2011). Sei $\mathbf{y} = (y_1, y_2, \dots, y_N)$ der Zielwert der Ausgabesequenz (üblicherweise eine Sequenz von Wörtern oder Zeichen). Die Aufgabe der **automatischen Spracherkennung** (engl. *automatic speech recognition*, ASR) besteht darin, eine Funktion f_{ASR}^* zu erzeugen, die die wahrscheinlichste linguistische Sequenz \mathbf{y} aus der akustischen Sequenz \mathbf{X} berechnet:

$$f_{\text{ASR}}^*(\mathbf{X}) = \arg \max_{\mathbf{y}} P^*(\mathbf{y} \mid \mathbf{X} = \mathbf{X}), \quad (12.4)$$

wobei P^* die wahre bedingte Verteilung der Eingaben \mathbf{X} für die Zielwerte \mathbf{y} ist.

Von den 1980ern bis etwa 2009 - 2012 kombinierten moderne Spracherkennungssysteme in erster Linie verdeckte Markow-Modelle (kurz HMM von *Hidden Markov Model*) und gaußsche Mischmodelle (kurz GMM von *Gaussian Mixture Model*). GMMs modellierten die Verknüpfung zwischen akustischen Merkmalen und Phonemen (*Bahl et al.*, 1987), wohingegen HMMs die Sequenz der Phoneme modellierten. Die GMM-HMM-Modelfamilie behandelt akustische Wellenformen als Ergebnis des folgenden Prozesses: Zunächst erzeugt ein HMM eine Sequenz aus Phonemen und diskreten subphonemischen Zuständen (beispielsweise Anfang, Mittelteil und Ende eines jeden Phonems). Anschließend transformiert ein GMM jedes diskrete Symbol in ein kurzes Segment einer Audio-Wellenform. Obwohl GMM-HMM-Systeme die ASR bis vor Kurzem dominiert haben, gehörte die Spracherkennung zu den ersten Gebieten, in denen neuronale Netze genutzt wurden; viele Systeme der automatischen Spracherkennung von Ende der 1980er- bis Anfang der 1990er-Jahre setzten neuronale Netze ein (*Bourlard und Wellekens*, 1989; *Waibel et al.*, 1989; *Robinson und Fallside*, 1991; *Bengio et al.*, 1991, 1992; *Konig et al.*, 1996). Damals lag automatische Spracherkennung auf Basis von neuronalen Netzen in etwa auf demselben Leistungsniveau wie GMM-HMM-Systeme. Zum Beispiel erzielten *Robinson und Fallside* (1991) eine Phonem-Fehlerquote von 26 Prozent für den TIMIT-Korpus (*Garofolo*

et al., 1993) (mit 39 zu unterscheidenden Phonemen) – ein besserer oder ähnlicher Wert wie bei Systemen auf HMM-Basis. Seitdem dient TIMIT als Benchmark für die Phonem-Erkennung und spielt somit dort eine ähnliche Rolle wie MNIST bei der Objekterkennung. Aufgrund der komplexen Entwicklungsarbeit bei Softwaresystemen für die Spracherkennung und der bereits in den Aufbau von GMM-HMM-Systemen geflossenen Bemühungen sah die Industrie keinen triftigen Grund für einen Wechsel zu neuronalen Netzen. Im Folgenden konzentrierten sich bis in die späten 2000er-Jahre sowohl akademische als auch kommerzielle Forschung bei neuronalen Netzen für die Spracherkennung in erster Linie darauf, mit diesen neuronalen Netzen zusätzliche Merkmale für GMM-HMM-Systeme zu erlernen.

Später, mit der Verfügbarkeit *viel größerer und tieferer Modelle* und viel größerer Datensätze, wurde die Erkennungsgenauigkeit durch Einsatz neuronaler Netze anstelle von GMMs für die Zuordnung der akustischen Merkmale zu Phonemen (oder subphonemischen Zuständen) enorm gesteigert. Ab 2009 nutzten Sprachforscher eine Form des Deep Learnings auf Basis des unüberwachten Lernens für die Spracherkennung. Dieser Ansatz für das Deep Learning beruhte auf einem Trainieren ungerichteter probabilistischer Modelle zum Modellieren der Eingabedaten. Diese sogenannten Restricted Boltzmann Machines (RBMs) werden in Teil III beschrieben. Zum Lösen von Aufgaben zur Spracherkennung wurden mittels unüberwachtem Pretraining tiefe Feedforward-Netze erstellt, deren Schichten jeweils durch Trainieren einer RBM initialisiert wurden. Diese Netze nehmen spektrale akustische Repräsentationen in einem Eingabefenster mit fester Größe (um einen zentralen Frame herum) entgegen und sagen die bedingten Wahrscheinlichkeiten der HMM-Zustände für den zentralen Frame voraus. Das Training derartiger tiefer Netze half dabei, die Erkennungsrate für den TIMIT-Datensatz (*Mohamed et al.*, 2009, 2012a) deutlich zu erhöhen, sodass die Phonem-Fehlerquote von etwa 26 Prozent auf etwa 20,7 Prozent sank. Die Gründe für den Erfolg dieser Modelle werden in *Mohamed et al.* (2012b) untersucht. Erweiterungen der grundlegenden Phonem-Erkennungs-Pipeline bestanden unter anderem in zusätzlichen Sprecher-adaptiven Merkmalen (*Mohamed et al.*, 2011), die die Fehlerquote nochmals reduzierten. Schon bald wurde an der Erweiterung der Architektur gearbeitet: Neben der Phonem-Erkennung (der Fokus von TIMIT) sollten auch große Wortschätzte erkannt werden (*Dahl et al.*, 2012); hierfür müssen neben Phonemen auch Sequenzen von Wörtern aus einer großen Wortbasis erkannt werden. Bei tiefen Netzen für die Spracherkennung fand schließlich ein Wechsel von Pretraining und Boltzmann-Maschinen zu Techniken wie ReLUs und Dropout statt (*Zeiler et al.*, 2013; *Dahl et al.*, 2013). Zu diesem Zeitpunkt hatten mehrere große Gruppierungen, die sich

mit Spracherkennung beschäftigen, begonnen, gemeinsam mit akademischen Forschern die Möglichkeiten des Deep Learnings zu erkunden. *Hinton et al.* (2012a) beschreiben die bahnbrechenden Ergebnisse dieser Zusammenarbeit, die heutzutage in Produkten wie Smartphones allgegenwärtig sind.

Als diese Gruppierungen im Laufe der Zeit immer größere mit Labels gekennzeichnete Datensätze untersuchten und einige Verfahren für Initialisierung, Training und Einrichten der Architektur tiefer Netze miteinbezogen, stellten sie fest, dass die Phase des unüberwachten Pretrainings entweder unnötig war oder keine wesentlichen Verbesserungen bot.

Diese Durchbrüche bei der Erkennungsleistung für die Wort-Fehlerquote in der Spracherkennung waren ohnegleichen (etwa 30 Prozent Steigerung); es folgte eine Periode von etwa zehn Jahren, in der die Fehlerquoten für die klassische GMM-HMM-Technologie trotz stetig wachsender Trainingsdatenmengen kaum verbessert wurden (vgl. Abbildung 2.4 in *Deng und Yu*, 2014). Dies führte dazu, dass sich die Spracherkennungs rasch dem Deep Learning zuwandte. Schon nach etwa zwei Jahren nutzten die meisten kommerziellen Produkte zur Spracherkennung auch tiefe neuronale Netze. Dieser Erfolg leitete eine neue Forschungswelle bei den Deep-Learning-Algorithmen und -Architekturen für die automatische Spracherkennung ein, die bis heute anhält.

Eine dieser Innovationen war der Einsatz von CNNs (*Sainath et al.*, 2013), die Gewichte über Zeit und Häufigkeit replizieren und so besser als die bisherigen zeitverzögerten neuronalen Netze (TDNN) abschnitten, in denen Gewichte nur über die Zeit repliziert wurden. Die neuen zweidimensionalen Modelle mit Faltung betrachten das Eingabe-Spekrogramm nicht mehr als einen langen Vektor, sondern als Bild, dessen eine Achse der Zeit und dessen andere Achse der Häufigkeit der spektralen Komponenten entspricht.

Ein weiterer bis heute andauernder Impuls betraf die Ende-zu-Ende-Spracherkennungssysteme mit Deep-Learning-Architekturen, die ganz auf das HMM verzichten. Der erste große Durchbruch in dieser Richtung stammt von *Graves et al.* (2013), die ein tiefes LSTM-RNN (siehe Abschnitt 10.10) mittels MAP-Inferenz über die Frame-zu-Phonem-Ausrichtung trainierten, wie in *LeCun et al.* (1998b) und im CTC-Framework (*Graves et al.*, 2006; *Graves*, 2012). Ein tiefes RNN (*Graves et al.*, 2013) enthält Zustandsvariablen aus mehreren Schichten für jeden Zeitschritt, sodass der aufgefaltete Graph zwei Arten von Tiefe aufweist: die gewöhnliche Tiefe aufgrund der Stapel von Schichten und eine Tiefe aufgrund des zeitlichen Auffaltens (engl. *time unfolding*). Diese Arbeit senkte die Phonem-Fehlerquote für TIMIT auf ein

Rekordtief von 17,7 Prozent. *Pascanu et al.* (2014a) und *Chung et al.* (2014) beschreiben weitere Varianten von tiefen RNNs für andere Fälle.

Ein weiterer zeitgemäßer Schritt hin zu Ende-zu-Ende-Spracherkennungssystemen mit Deep-Learning-Architekturen besteht darin, das System erlernen zu lassen, wie die akustischen Daten an die phonetischen Daten angepasst werden können (*Chorowski et al.*, 2014; *Lu et al.*, 2015).

12.4 Verarbeitung natürlicher Sprache

Verarbeitung natürlicher Sprache (engl. *natural language processing*, NLP) ist der Gebrauch menschlicher Sprachen – wie Deutsch, Englisch oder Französisch – durch einen Computer. Computerprogramme verwenden zum Einlesen und Ausgeben normalerweise spezielle Sprachen, die eine effiziente und eindeutige Auswertung mittels einfacher Programme ermöglichen. Natürliche Sprachen sind oft vieldeutig und widersetzen sich einer formalen Beschreibung. Die Verarbeitung natürlicher Sprache umfasst Anwendungen wie die maschinelle Übersetzung, bei der der Klassifikator einen Satz in einer menschlichen Sprache einlesen und einen entsprechenden Satz in einer anderen menschlichen Sprache ausgeben muss. Viele Anwendungen für die Verarbeitung natürlicher Sprache beruhen auf Sprachmodellen, die eine Wahrscheinlichkeitsverteilung über Sequenzen von Wörtern, Zeichen oder Bytes in einer natürlichen Sprache definieren.

Wie bei anderen in diesem Kapitel vorgestellten Anwendungen können sehr allgemeine Verfahren für neuronale Netze mit Erfolg auf die Verarbeitung natürlicher Sprache angewandt werden. Für eine hervorragende Leistung und die Umsetzbarkeit in sehr großen Anwendungen werden jedoch einige auf den jeweiligen Bereich zugeschnittene Verfahren wichtig. Um ein effizientes Modell der natürlichen Sprache zu erstellen, müssen wir Verfahren einsetzen, die speziell der Verarbeitung sequenzieller Daten dienen. In vielen Fällen betrachten wir natürliche Sprache als eine Sequenz von Wörtern und nicht als eine Sequenz von einzelnen Zeichen oder Bytes. Da die Gesamtzahl möglicher Wörter so groß ist, müssen Sprachmodelle auf Wortbasis in einem extrem hochdimensionalen und dünnbesetzten diskreten Raum arbeiten. Es wurden mehrere Verfahren entwickelt, um Modelle eines solchen Raums effizient zu gestalten – sowohl rechnerisch als auch statistisch.

12.4.1 N-Gramme

Ein **Sprachmodell** definiert eine Wahrscheinlichkeitsverteilung über eine Sequenz von Tokens in einer natürlichen Sprache. Je nach Konzipierung des Modells kann ein solches Token für ein Wort, ein Zeichen oder sogar ein Byte stehen. Tokens sind stets diskrete Elemente. Die ersten erfolgreichen Sprachmodelle basierten auf Modellen mit Sequenzen fester Länge, die *N*-Gramme heißen. Ein *N*-Gramm ist eine Sequenz aus n Tokens.

Modelle auf Basis von *N*-Grammen definieren die bedingte Wahrscheinlichkeit für das n -te Token anhand der vorhergehenden $n - 1$ Tokens. Das Modell nutzt Produkte dieser bedingten Verteilungen, um die Wahrscheinlichkeitsverteilung über längere Sequenzen zu definieren:

$$P(x_1, \dots, x_\tau) = P(x_1, \dots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t | x_{t-n+1}, \dots, x_{t-1}). \quad (12.5)$$

Diese Zerlegung lässt sich durch die Produktregel für Wahrscheinlichkeiten begründen. Die Wahrscheinlichkeitsverteilung über die anfängliche Sequenz $P(x_1, \dots, x_{n-1})$ kann durch ein anderes Modell mit einem kleineren Wert für n modelliert werden.

Das Trainieren von *N*-Gramm-Modellen ist recht einfach, da der Maximum-Likelihood-Schätzwert berechnet werden kann, indem gezählt wird, wie oft jedes mögliche *N*-Gramm in der Trainingsdatenmenge vorhanden ist. Modelle auf Basis von *N*-Grammen waren viele Jahrzehnte lang der Grundstein für die statistische Sprachmodellierung (*Jelinek und Mercer*, 1980; *Katz*, 1987; *Chen und Goodman*, 1999).

Für kleine n tragen die Modelle eigene Namen: **Unigramm** für $n = 1$, **Bigramm** für $n = 2$ und **Trigramm** für $n = 3$. Diese Namen setzen sich aus den lateinischen Vorsilben für die entsprechenden Zahlen und dem griechischen Suffix »-gram« für etwas, das geschrieben ist, zusammen.

Normalerweise werden gleichzeitig ein *N*-Gramm-Modell und ein $N - 1$ -Gramm-Modell trainiert. So kann der Term

$$P(x_t | x_{t-n+1}, \dots, x_{t-1}) = \frac{P_n(x_{t-n+1}, \dots, x_t)}{P_{n-1}(x_{t-n+1}, \dots, x_{t-1})} \quad (12.6)$$

einfach durch Nachschlagen von zwei gespeicherten Wahrscheinlichkeiten berechnet werden. Damit die Inferenz in P_n hierdurch exakt reproduziert wird, müssen wir das letzte Zeichen jeder Sequenz beim Trainieren von P_{n-1} weglassen.

Als Beispiel zeigen wir, wie ein Trigramm-Modell die Wahrscheinlichkeit für den englischen Satz »THE DOG RAN AWAY« (Der Hund ist wegelaufen)

berechnet. Für das erste Wort im Satz kann die Standardformel auf Basis der bedingten Wahrscheinlichkeit nicht verwendet werden, da es am Satzanfang noch keinen Kontext gibt. Stattdessen müssen wir die Randwahrscheinlichkeit über Wörter am Satzanfang verwenden. Wir berechnen also $P_3(\text{THE DOG RAN})$. Das letzte Wort kann anhand des typischen Falls mittels der bedingten Verteilung $P(\text{AWAY} | \text{DOG RAN})$ vorhergesagt werden. In Verbindung mit Gleichung 12.6 erhalten wir:

$$P(\text{THE DOG RAN AWAY}) = P_3(\text{THE DOG RAN})P_3(\text{DOG RAN AWAY})/P_2(\text{DOG RAN}). \quad (12.7)$$

Eine grundlegende Einschränkung der Maximum Likelihood für N -Gramm-Modelle ist, dass das anhand der Anzahl aus der Trainingsdatenmenge geschätzte P_n in vielen Fällen höchstwahrscheinlich Null ist, obwohl das Tupel (x_{t-n+1}, \dots, x_t) in der Testdatenmenge enthalten sein kann. Dies kann zwei unterschiedliche katastrophale Folgen haben: Ist P_{n-1} Null, dann ist das Verhältnis undefiniert und das Modell erzeugt keinerlei sinnvolle Ausgabe. Ist P_{n-1} von Null verschieden, aber P_n Null, beträgt die Log-Likelihood für den Test $-\infty$. Um solche katastrophalen Folgen zu vermeiden, nutzen die meisten N -Gramm-Modelle eine Art **Glättung** (engl. *smoothing*). Mit der Glättung wird die Wahrscheinlichkeit(smasse) von den beobachteten Tupeln auf ähnliche, nicht beobachtete verschoben. *Chen und Goodman* (1999) enthält eine Abhandlung samt empirischer Vergleiche. Ein grundlegendes Verfahren fügt von Null verschiedene Wahrscheinlichkeit(smassen) zu allen möglichen folgenden Symbolwerten hinzu. Dieses Verfahren ähnelt einer bayesschen Inferenz mit einer A-priori-Gleichverteilung oder einer A-priori-Dirichlet-Verteilung über die Zählerparameter. Eine weitere beliebte Variante ist das Bilden eines Mischmodells mit N -Gramm-Modellen höherer und niedriger Ordnung, wobei die Modelle höherer Ordnung mehr Kapazität bieten, während die niedrigeren Ordnungen eher Nullzähler vermeiden. **Backoff-Verfahren** nutzen die N -Gramme niedrigerer Ordnung, wenn die Häufigkeit des Kontextes $x_{t-1}, \dots, x_{t-n+1}$ zu klein für das Modell höherer Ordnung ist. Formal ausgedrückt schätzen sie die Verteilung über x_t anhand der Kontexte $x_{t-n+k}, \dots, x_{t-1}$ zum Erhöhen von k , bis ein hinreichend zuverlässiger Schätzwert gefunden ist.

Klassische N -Gramm-Modelle sind besonders anfällig für den Fluch der Dimensionalität. Es gibt $|\mathbb{V}|^n$ mögliche N -Gramme und $|\mathbb{V}|$ ist oft sehr groß. Selbst mit einer riesigen Trainingsdatenmenge und kleinem n werden die meisten N -Gramme nicht in der Trainingsdatenmenge auftreten. Man könnte sagen, ein klassisches N -Gramm-Modell führt eine Nearest-Neighbor-Prüfung durch. Es handelt sich quasi um einen lokalen nichtparametrischen Prädiktor, ähnlich k -Nearest-Neighbor. Die statistischen Probleme dieser ex-

trem lokalen Prädiktoren werden in Abschnitt 5.11.2 beschrieben. Bei einem Sprachmodell tritt das Problem noch deutlicher zutage, da zwei beliebige Wörter im One-hot-Vektorraum denselben Abstand zueinander haben. Es ist daher schwierig, viele Informationen von den »Nachbarn« zu gewinnen – nur Trainingsbeispiele, die buchstäblich denselben Kontext wiederholen, sind für die lokale Generalisierung von Nutzen. Als Gegenmaßnahme muss ein Sprachmodell in der Lage sein, Wissen zwischen einem Wort und semantisch ähnlichen Wörtern zu teilen.

Um die statistische Effizienz von N -Gramm-Modellen zu steigern, führen **Sprachmodelle auf Klassenbasis** (*Brown et al., 1992; Ney und Kneser, 1993; Niesler et al., 1998*) das Konzept das Konzept der Wortkategorien ein und geben dann die statistische Aussagekraft zwischen Wörtern weiter, die sich in derselben Kategorie befinden. Die Idee dahinter ist ein Clustering-Algorithmus, der die Wortmengen aufgrund ihrer Kookkurrenzhäufigkeiten zu anderen Wörtern in Cluster oder Klassen aufteilt. Das Modell kann dann die Kennzahlen der Wortklassen anstelle der Kennzahlen einzelner Wörter verwenden, um den Kontext auf der rechten Seite des Bedingungszeichens darzustellen. Verbundmodelle, die Modelle auf Wort- und auf Klassenbasis durch Mischen oder Backoff kombinieren, sind ebenfalls denkbar. Zwar bieten Wortklassen eine Möglichkeit zur Generalisierung zwischen Sequenzen, in denen ein Wort durch ein anderes derselben Klasse ersetzt wird, aber in dieser Repräsentation gehen viele Informationen verloren.

12.4.2 Neuronale Sprachmodelle

Neuronale Sprachmodelle (engl. *neural language models, NLMs*) sind eine Klasse von Sprachmodellen, die angesichts des Fluchs der Dimensionalität beim Modellieren von Sequenzen natürlicher Sprache entwickelt wurden; sie nutzen eine verteilte Repräsentation der Wörter (*Bengio et al., 2001*). Anders als N -Gramm-Modelle auf Klassenbasis können neuronale Sprachmodelle erkennen, dass zwei Wörter einander ähnlich sind, ohne die Fähigkeit zu verlieren, diese Wörter als unterschiedlich zu codieren. Neuronale Sprachmodelle teilen die statistische Aussagekraft zwischen einem Wort (und seinem Kontext) und anderen ähnlichen Wörtern und Kontexten. Die verteilte Repräsentation, die das Modell für jedes Wort erlernt, ermöglicht dieses Teilen, indem sie dem Modell ermöglicht, Wörter, die gemeinsame Merkmale aufweisen, ähnlich zu behandeln. Wenn zum Beispiel das Wort **Hund** und das Wort **Katze** Repräsentationen mit vielen gemeinsamen Attributen zugeordnet werden, dann können Sätze, die das Wort **Katze** enthalten, die Vorhersagen des Modells für Sätze, die das Wort **Hund** enthalten, beein-

flussen und umgekehrt. Da es viele solcher Attribute gibt, gibt es auch viele Möglichkeiten für die Generalisierung, bei der Informationen der einzelnen Trainingssätze auf eine exponentiell große Anzahl semantisch verwandter Sätze übertragen werden. Der Fluch der Dimensionalität veranlasst das Modell, auf eine Anzahl von Sätzen zu generalisieren, die in der Satzlänge exponentiell ist. Das Modell tritt diesem Fluch entgegen, indem jeder Trainingssatz mit einer exponentiellen Anzahl ähnlicher Sätze in Beziehung gesetzt wird.

Wir bezeichnen diese Wortrepräsentationen manchmal als **Wort-Embedding** (dt. *Worteinbettung*). In dieser Interpretation betrachten wir die Rohsymbole als Punkte in einem Raum, dessen Dimension der Größe des Wortschatzes entspricht. Die Wortrepräsentationen betten diese Punkte in einen Merkmalsraum mit niedrigerer Dimension ein. Im ursprünglichen Raum wird jedes Wort als One-hot-Vektor dargestellt, sodass jedes Wortpaar einen euklidischen Abstand $\sqrt{2}$ zueinander aufweist. Im Embedding-Raum sind Wörter, die häufig in ähnlichen Kontexten erscheinen (oder beliebige Wortpaare, die vom Modell erlernte »Merkmale« teilen) einander nah. Das führt häufig dazu, dass Wörter mit ähnlicher Bedeutung Nachbarn sind. Abbildung 12.3 greift bestimmte Bereiche eines erlernten Wort-Embedding-Raums heraus, um zu zeigen, wie semantisch ähnliche Wörter auf Repräsentationen abgebildet werden, die einander nahe sind.

Auch für neuronale Netze in anderen Domänen werden Embeddings definiert. Zum Beispiel stellt eine verdeckte Schicht in einem CNN ein »Bild-Embedding« dar. Besonders für die Verarbeitung natürlicher Sprache sind solche Embeddings interessant, da die natürliche Sprache ursprünglich nicht in einem reellwertigen Vektorraum liegt. Die verdeckte Schicht hat eine gravierende qualitative Veränderung eingeführt – bezogen auf die Art und Weise, wie Daten dargestellt werden.

Verteilte Repräsentation zur Verbesserungen von Modellen für die Verarbeitung natürlicher Sprache zu verwenden, ist nicht auf neuronale Netze beschränkt. Das Konzept kann auch für graphische Modelle verwendet werden, die verteilte Repräsentationen in Form mehrerer latenter Variablen aufweisen (*Mnih und Hinton, 2007*).

12.4.3 Hochdimensionale Ausgaben

In vielen Anwendungen im Bereich der natürlichen Sprache möchten wir, dass die Modelle Wörter (anstelle von Zeichen) als Basiseinheit der Ausgabe erzeugen. Bei großen Wortschätzten kann es sehr rechenaufwendig sein, eine Ausgabeverteilung bezüglich der Auswahl eines Worts darzustellen, da das

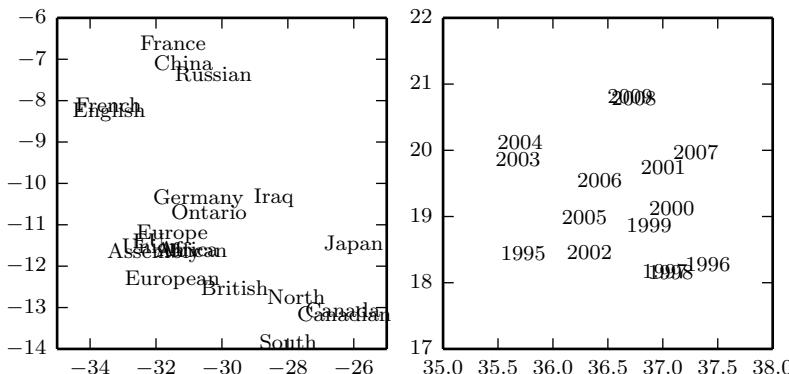


Abbildung 12.3: Zweidimensionale Visualisierungen von Wort-Embeddings, erhalten aus einem neuronalen Modell zur maschinellen Übersetzung (*Bahdanau et al., 2015*). Dargestellt sind bestimmte vergrößerte Ausschnitte, in denen semantisch verwandte Wörter Embedding-Vektoren haben, die nahe beieinander liegen. Länder werden links angezeigt, Zahlen rechts. Bedenken Sie, dass diese Embeddings aus Darstellungsgründen zweidimensional sind. In echten Anwendungen haben Embeddings üblicherweise eine höhere Dimensionalität und können viele Arten von Ähnlichkeiten zwischen Wörtern parallel erfassen.

Vokabular umfangreich ist. In vielen Anwendungen enthält \mathbb{V} Hunderttausende Wörter. Der naive Ansatz zur Repräsentation einer solchen Verteilung wendet nacheinander eine affine Transformation aus einer verdeckten Repräsentation auf den Ausgaberaum und die softmax-Funktion an. Gegeben sei ein Wortschatz (Vokabular) \mathbb{V} der Größe $|\mathbb{V}|$. Die Gewichtungsmatrix zur Beschreibung der linearen Komponente dieser affinen Transformation ist sehr groß, da ihre Ausgabedimension $|\mathbb{V}|$ ist. Das zieht einen hohen Speicherbedarf zur Darstellung der Matrix nach sich sowie einen hohen Berechnungsaufwand für die Multiplikation damit. Da die softmax über alle $|\mathbb{V}|$ -Ausgaben normalisiert ist, muss sowohl zum Trainingszeitpunkt als auch zum Testzeitpunkt eine vollständige Matrizenmultiplikation erfolgen – wir können nicht nur das Skalarprodukt mit dem Gewichtungsvektor für die korrekte Ausgabe berechnen. Der hohe Berechnungsaufwand für die Ausgabeschicht entsteht somit sowohl zum Trainingszeitpunkt (für die Berechnung der Likelihood und des zugehörigen Gradienten) als auch zum Testzeitpunkt (für die Berechnung der Wahrscheinlichkeiten aller oder der ausgewählten Wörter). Für diese spezialisierten Verlustfunktionen kann der Gradient effizient berechnet werden (*Vincent et al., 2015*), aber der Standard-Kreuzentropieverlust der klassischen softmax-Ausgabeschicht bereitet uns viele Schwierigkeiten.

\mathbf{h} sei die oberste verdeckte Schicht für die Vorhersage der Ausgabewahrscheinlichkeiten $\hat{\mathbf{y}}$. Wenn wir die Transformation von \mathbf{h} zu $\hat{\mathbf{y}}$ mit den erlernten Gewichten \mathbf{W} und den erlernten Verzerrungen \mathbf{b} parametrisieren, dann führt die affin-softmax-Ausgabeschicht die folgenden Berechnungen aus:

$$a_i = b_i + \sum_j W_{ij} h_j \quad \forall i \in \{1, \dots, |\mathbb{V}|\}, \quad (12.8)$$

$$\hat{y}_i = \frac{e^{a_i}}{\sum_{i'=1}^{|\mathbb{V}|} e^{a_{i'}}}. \quad (12.9)$$

Enthält \mathbf{h} nun n_h Elemente, ist die oben gezeigte Operation $O(|\mathbb{V}|n_h)$. Geht n_h in die Tausende und $|\mathbb{V}|$ in die Hunderttausende, dominiert diese Operation die Berechnung der meisten neuronalen Sprachmodelle.

12.4.3.1 Verwenden einer Shortlist

Die ersten neuronalen Sprachmodelle (*Bengio et al., 2001, 2003*) gingen das Problem des hohen Aufwands, der durch die Verwendung der softmax über eine große Anzahl von Ausgabewörtern entsteht, durch einen eingeschränkten Wortschatz mit lediglich 10 000 oder 20 000 Wörtern an. *Schwenk und Gauvain (2002)* und *Schwenk (2007)* ergänzten diesen Ansatz, indem sie den Wortschatz \mathbb{V} in eine **Shortlist** \mathbb{L} der häufigsten Wörter (verwaltet durch das neuronale Netz) und eine Nebenliste $\mathbb{T} = \mathbb{V} \setminus \mathbb{L}$ mit selteneren Wörtern (verwaltet durch ein N -Gramm-Modell) aufteilten. Um die beiden Vorhersagen zu kombinieren, muss das neuronale Netz auch die Wahrscheinlichkeit vorhersagen, mit der ein Wort, das nach einem Kontext C auftritt, zur Nebenliste gehört. Hierzu kann eine zusätzliche sigmoide Ausgabeeinheit verwendet werden, um so einen Schätzwert für $P(i \in \mathbb{T} | C)$ anzugeben. Die zusätzliche Ausgabe dient dann zum Erreichen eines Schätzwerts der Wahrscheinlichkeitsverteilung über alle Wörter in \mathbb{V} :

$$\begin{aligned} P(y = i | C) &= 1_{i \in \mathbb{L}} P(y = i | C, i \in \mathbb{L}) (1 - P(i \in \mathbb{T} | C)) \\ &\quad + 1_{i \in \mathbb{T}} P(y = i | C, i \in \mathbb{T}) P(i \in \mathbb{T} | C), \end{aligned} \quad (12.10)$$

wobei $P(y = i | C, i \in \mathbb{L})$ vom neuronalen Sprachmodell stammt und $P(y = i | C, i \in \mathbb{T})$ vom N -Gramm-Modell. Mit einer kleinen Änderung funktioniert dieser Ansatz auch mit einem zusätzlichen Ausgabewert in der softmax-Schicht des neuronalen Sprachmodells anstelle einer separaten sigmoiden Einheit.

Ein offensichtlicher Nachteil der Shortlist ist, dass der mögliche Generalisierungsvorteil der neuronalen Sprachmodelle auf die häufigsten Wörter

eingeschränkt ist – gerade den Teil also, in dem der geringste Nutzen entsteht. Dieser Nachteil hat zur Suche nach anderen Verfahren für die Arbeit mit hochdimensionalen Ausgaben geführt, die unten beschrieben werden.

12.4.3.2 Hierarchische softmax

Ein klassischer Ansatz (*Goodman, 2001*) zur Verringerung der Rechenlast bei hochdimensionalen Ausgabeschichten über große Wortschatzmengen \mathbb{V} besteht im hierarchischen Zerlegen der Wahrscheinlichkeiten. Statt eine Anzahl von Berechnungen vorauszusetzen, die proportional zu $|\mathbb{V}|$ (und auch proportional zur Anzahl der verdeckten Einheiten n_h) ist, kann der Faktor $|\mathbb{V}|$ bis auf $\log |\mathbb{V}|$ reduziert werden. *Bengio (2002)* und *Morin und Bengio (2005)* nutzten diesen faktorisierten Ansatz für den Kontext in neuronalen Sprachmodellen.

Man kann sich diese Hierarchie so vorstellen, dass zunächst Kategorien für Wörter, anschließend Kategorien für die Wortkategorien, dann Kategorien der Kategorien der Wortkategorien usw. gebildet werden. Diese verschachtelten Kategorien bilden einen Baum, an dem die Blätter Wörter sind. Ein ausbalancierter Baum hat die Tiefe $O(\log |\mathbb{V}|)$. Die Wahrscheinlichkeit für die Wahl eines Worts ergibt sich aus dem Produkt der Wahrscheinlichkeiten für die Wahl des Zweigs, der an jedem einzelnen Knoten auf dem Weg von der Wurzel zu dem Blatt mit dem Wort zu eben diesem Wort führt. Abbildung 12.4 stellt das an einem einfachen Beispiel dar. *Mnih und Hinton (2009)* beschreiben auch, wie mehrere Pfade verwendet werden können, um ein einzelnes Wort mit dem Ziel zu identifizieren, Wörter mit mehreren Bedeutungen besser zu modellieren. Die Berechnung der Wahrscheinlichkeit eines Worts umfasst dann die Aufsummierung über alle Pfade, die zu diesem Wort führen.

Um die in jedem Knoten des Baums benötigten bedingten Wahrscheinlichkeiten vorherzusagen, verwenden wir üblicherweise in jedem Knoten ein logistisches Regressionsmodell und verwenden denselben Kontext C als Eingabe für all diese Modelle. Da die korrekte Ausgabe in der Trainingsdatenmenge codiert ist, können wir die logistischen Regressionsmodelle mittels überwachtem Lernen trainieren. Dazu verwenden wir meist einen normalen Kreuzentropieverlust, was einem Maximieren der Log-Likelihood der korrekten Entscheidungsfolge entspricht.

Da die Ausgabe-Log-Likelihood effizient bestimmt werden kann (bis hinunter zu $\log |\mathbb{V}|$ statt $|\mathbb{V}|$), können auch ihre Gradienten effizient berechnet werden. Das umfasst nicht nur den Gradienten bezüglich der Ausgabepa-

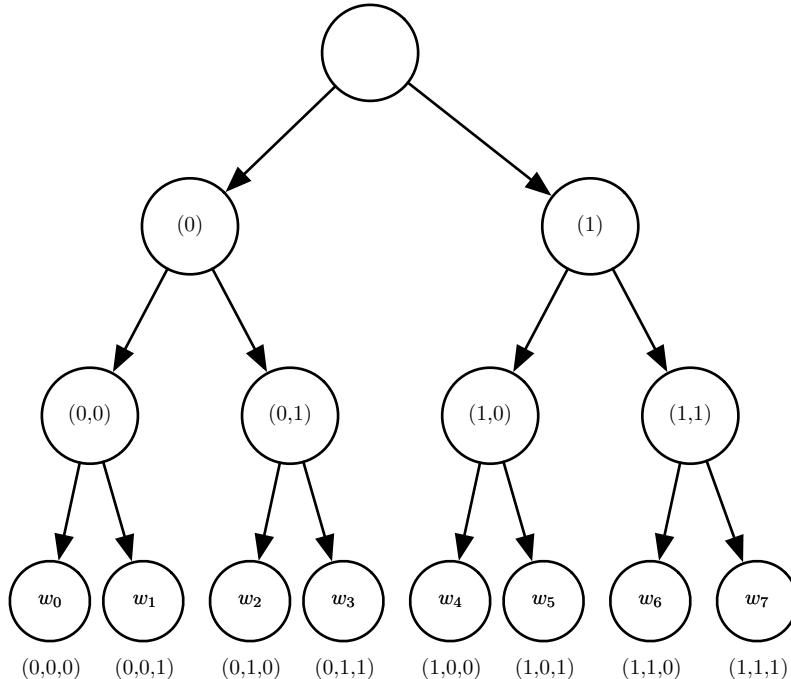


Abbildung 12.4: Darstellung einer einfachen Worthierarchie mit Wortkategorien für 8+ Wörter w_0, \dots, w_7 , die in einer Hierarchie mit drei Ebenen angeordnet sind. Die Blätter des Baums stehen für die einzelnen Wörter. Innenknoten stehen für Wortgruppen. Jeder Knoten kann durch die Abfolge binärer Entscheidungen (0 = links, 1 = rechts) indiziert werden, durch die er von der Wurzel aus erreicht wird. Die Superklasse (0) enthält die Klassen (0, 0) und (0, 1), die wiederum die Mengen der Wörter $\{w_0, w_1\}$ und $\{w_2, w_3\}$ enthalten. Ebenso enthält die Superklasse (1) die Klassen (1, 0) und (1, 1) mit den Wörtern (w_4, w_5) und (w_6, w_7) . Wenn der Baum hinreichend ausbalanciert ist, dann ist die maximale Tiefe (Anzahl der binären Entscheidungen) von der Ordnung des Logarithmus der Anzahl an Wörtern $|\mathbb{V}|$: Die Entscheidung für eines der Wörter aus $|\mathbb{V}|$ ergibt sich aus $O(\log |\mathbb{V}|)$ Operationen (ein Schritt für jeden der Knoten auf dem Weg von der Wurzel). In diesem Beispiel kann die Wahrscheinlichkeit für ein Wort y durch Multiplikation von drei Wahrscheinlichkeiten ermittelt werden. Die Wahrscheinlichkeiten entsprechen den binären Entscheidungen (links/rechts) in jedem Knoten auf dem Weg von der Wurzel zu einem Knoten y . Sei $b_i(y)$ die i -te binäre Entscheidung auf dem Pfad zum Wert y . Die Wahrscheinlichkeit, eine Stichprobe einer Ausgabe y zu ziehen, wird unter Verwendung der Produktregel für bedingte Wahrscheinlichkeiten in ein Produkt der bedingten Wahrscheinlichkeiten zerlegt, wobei jeder Knoten durch das Präfix dieser Bits indiziert ist. Ein Beispiel: Der Knoten (1, 0) entspricht dem Präfix $(b_0(w_4) = 1, b_1(w_4) = 0)$; die Wahrscheinlichkeit für w_4 lässt sich wie folgt zerlegen:

$$P(y = w_4) = P(b_0 = 1, b_1 = 0, b_2 = 0) \quad (12.11)$$

$$= P(b_0 = 1)P(b_1 = 0 | b_0 = 1)P(b_2 = 0 | b_0 = 1, b_1 = 0). \quad (12.12)$$

rameter, sondern auch die Gradienten bezüglich der Aktivierungen der verdeckten Schicht.

Es ist möglich – aber meist nicht praktikabel –, die Baumstruktur so zu optimieren, dass die erwartete Anzahl von Berechnungen minimiert wird. Tools aus der Informationstheorie geben an, wie der optimale binäre Code auf Grundlage der relativen Worthäufigkeiten aussehen müsste. Hierzu strukturieren wir den Baum so, dass die Anzahl der Bits, die einem Wort zugerechnet werden, in etwa gleich dem Logarithmus der Häufigkeit dieses Worts ist. In der Praxis sind die Einsparungen bei der Berechnung diesen Aufwand allerdings nur selten wert, da die Berechnung der Ausgabewahrscheinlichkeiten nur ein Teil aller Berechnungen im neuronalen Sprachmodell ist. Nehmen wir zum Beispiel an, dass es l vollständig verbundene verdeckte Schichten der Breite n_h gibt. Sei n_b das gewichtete Mittel der Anzahl der Bits, die zum Bestimmen eines Worts benötigt werden, wobei sich das Gewicht aus der Häufigkeit dieser Wörter ergibt. In diesem Beispiel nimmt die Anzahl der Operationen zum Berechnen der verdeckten Aktivierungen als $O(\ln_h^2)$ zu, während die Ausgabeberechnungen als $O(n_h n_b)$ zunehmen. Sofern $n_b \leq \ln_h$ können wir die Berechnung durch Verkleinern von n_h stärker reduzieren als durch Verkleinern von n_b . Tatsächlich ist n_b häufig klein. Da die Größe des Wortschatzes nur selten eine Million Wörter übersteigt und $\log_2(10^6) \approx 20$, ist es möglich, n_b auf etwa 20 zu reduzieren – aber n_h ist oft viel größer, etwa 10^3 oder mehr. Statt einen Baum mit einem Verzweigungsfaktor von 2 mühevoll zu optimieren, können wir einfach einen Baum mit der Tiefe 2 und einem Verzweigungsfaktor $\sqrt{|\mathbb{V}|}$ definieren. Ein solcher Baum entspricht der Definition einer Menge von einander ausschließenden Wortklassen. Der einfache Ansatz auf einem Baum der Tiefe 2 nutzt die meisten Rechenvorteile des hierarchischen Verfahrens.

Eine Frage, die nicht vollständig beantwortet wird, bleibt: Wie werden die Wortklassen am besten definiert? Oder anders: Wie wird die Worthierarchie grundsätzlich definiert? Frühe Arbeiten nutzten vorhandene Hierarchien (*Morin und Bengio, 2005*), aber die Hierarchie kann auch erlernt werden – idealerweise gemeinsam mit dem neuronalen Sprachmodell. Das Erlernen der Hierarchie ist schwierig. Eine exakte Optimierung der Log-Likelihood erscheint nicht effizient durchführbar, da die Auswahl einer Worthierarchie diskret erfolgt und für die Optimierung auf Gradientenbasis nicht zugänglich ist. Allerdings könnten wir eine diskrete Optimierung verwenden, um die Aufteilung der Wörter in Wortklassen näherungsweise zu optimieren.

Ein großer Vorteil einer hierarchischen softmax ist, dass es sowohl zum Trainings- als auch zum Testzeitpunkt rechnerische Vorteile gibt, wenn wir

zum Testzeitpunkt die Wahrscheinlichkeit bestimmter Wörter ermitteln möchten.

Natürlich ist die Berechnung der Wahrscheinlichkeiten aller $|\mathbb{V}|$ Wörter selbst mit einer hierarchischen softmax aufwendig. Eine weitere wichtige Operation ist die Auswahl des wahrscheinlichsten Worts für einen bestimmten Kontext. Leider hält die Baumstruktur keine effiziente und exakte Lösung für dieses Problem bereit.

Ein Nachteil besteht darin, dass die hierarchische softmax in der Praxis dazu neigt, schlechtere Testergebnisse als stichprobenbasierende Verfahren (diese beschreiben wir im Folgenden) zu liefern. Das kann an einer schlechten Wahl der Wortklassen liegen.

12.4.3.3 Importance Sampling

Eine Möglichkeit, das Trainings neuronaler Sprachmodelle zu beschleunigen, besteht darin, die explizite Berechnung des Beitrags des Gradienten aller Wörter, die nicht an der nächsten Stelle erscheinen, zu vermeiden. Jedes inkorrekte Wort müsste im Rahmen des Modells eine niedrige Wahrscheinlichkeit haben. Es kann rechenaufwendig sein, all diese Wörter aufzulisten. Stattdessen ist es möglich, nur Stichproben aus einer Teilmenge der Wörter ziehen. Anhand der Notation aus Gleichung 12.8 lässt sich der Gradient wie folgt notieren:

$$\frac{\partial \log P(y | C)}{\partial \theta} = \frac{\partial \log \text{softmax}_y(\mathbf{a})}{\partial \theta} \quad (12.13)$$

$$= \frac{\partial}{\partial \theta} \log \frac{e^{a_y}}{\sum_i e^{a_i}} \quad (12.14)$$

$$= \frac{\partial}{\partial \theta} (a_y - \log \sum_i e^{a_i}) \quad (12.15)$$

$$= \frac{\partial a_y}{\partial \theta} - \sum_i P(y = i | C) \frac{\partial a_i}{\partial \theta}, \quad (12.16)$$

wobei \mathbf{a} der Vektor von Aktivierungen vor der softmax (oder Scores) ist, und zwar mit einem Element pro Wort. Der erste Ausdruck ist der **positive Phasenterm**, der a_y erhöht. Der zweite Ausdruck ist der **negative Phasenterm**, der a_i für alle i verringert, mit dem Gewicht $P(i | C)$. Da der negative Phasenterm ein Erwartungswert ist, können wir ihn mit einer Monte-Carlo-Stichprobe schätzen. Allerdings muss dazu die Stichprobenentnahme aus dem Modell selbst erfolgen. Für das Ziehen von Stichproben aus dem Modell müssen wir $P(i | C)$ für alle i im Wortschatz berechnen – und genau das möchten wir ja vermeiden.

Statt also Stichproben aus dem Modell zu ziehen, können wir aus einer anderen Verteilung Stichproben ziehen, Proposal-Verteilung (q) genannt, und angemessene Gewichte zum Korrigieren der Verzerrung nutzen, die durch die Stichprobenentnahme aus der falschen Verteilung entstanden ist (Bengio und Sénécal, 2003; Bengio und Sénécal, 2008). Dies ist eine Anwendung einer allgemeineren Verfahren namens **Importance Sampling** (Stichprobenentnahme nach Wichtigkeit), auf die wir in Abschnitt 17.2 genauer eingehen. Leider ist selbst ein exaktes Importance Sampling nicht effizient, da die Gewichte p_i/q_i für $p_i = P(i | C)$ berechnet werden müssen, und dies nur möglich ist, wenn alle Scores für a_i berechnet werden. Die für diese Anwendung genutzte Lösung trägt die Bezeichnung **Biased Importance Sampling** (verzerrte Stichprobenentnahme nach Wichtigkeit); die Gewichte nach Wichtigkeit sind dabei so normalisiert, dass sie in Summe 1 ergeben. Wenn das Ergebnis der Stichprobe das negative Wort n_i ist, wird der zugehörige Gradient gewichtet mit

$$w_i = \frac{p_{n_i}/q_{n_i}}{\sum_{j=1}^N p_{n_j}/q_{n_j}}. \quad (12.17)$$

Diese Gewichte werden verwendet, um den m negativen Stichproben aus q eine angemessene Wichtigkeit zu geben, da anhand dieser Stichproben der geschätzte negative Phasenbeitrag zum Gradienten gebildet wird:

$$\sum_{i=1}^{|\mathbb{V}|} P(i | C) \frac{\partial a_i}{\partial \theta} \approx \frac{1}{m} \sum_{i=1}^m w_i \frac{\partial a_{n_i}}{\partial \theta}. \quad (12.18)$$

Eine Unigramm- oder Bigramm-Verteilung funktioniert gut als Proposal-Verteilung q . Es ist einfach, die Parameter einer solchen Verteilung anhand der Daten zu schätzen. Nach der Parameterschätzung können außerdem Stichproben sehr effizient aus einer Verteilung gezogen werden.

Das Importance Sampling ist allerdings nicht nur zum Beschleunigen von Modellen mit großen softmax-Ausgaben nützlich. Es kann auch ganz allgemein zum Beschleunigen des Trainings mit großen dünnbesetzten Ausgabeschichten verwendet werden, in denen die Ausgabe ein dünnbesetzter Vektor ist und keine 1-von- n -Auswahl. Ein Beispiel dafür ist ein **Bag-of-Words**. Ein Bag-of-Words ist ein dünnbesetzter Vektor \mathbf{v} , wobei v_i für das Vorhandensein oder Fehlen des Worts i im Wortschatz des Dokuments steht. Alternativ kann v_i die Häufigkeit des Worts i angeben. Machine-Learning-Modelle, die solche dünnbesetzten Vektoren ausgeben, können aus verschiedenen Gründen einen hohen Trainingsaufwand verursachen. Zu Beginn des Lernprozesses kann das Modell sich gegen eine wirklich dünnbesetzte

Ausgabe entscheiden. Außerdem könnte die für das Training verwendete Verlustfunktion am einfachsten als Vergleich der einzelnen Elemente der Ausgabe mit jedem Element des Zielwerts beschrieben werden. Es ist also nicht immer klar, dass dünnbesetzte Ausgaben einen rechnerischen Nutzen bieten, da das Modell sich vielleicht dafür entscheidet, den Großteil der Ausgabe von Null verschieden festzulegen; all diese von Null verschiedenen Werte müssen mit dem entsprechenden Trainingsziel verglichen werden, selbst wenn das Trainingsziel Null ist. *Dauphin et al.* (2011) haben gezeigt, dass solche Modelle mithilfe des Importance Samplings beschleunigt werden können. Der effiziente Algorithmus minimiert die Verlust-Wiederherstellung für die »positiven Wörter« (die im Ziel von Null verschieden sind) und einer gleich großen Anzahl von »negativen Wörtern«. Die negativen Wörter werden zufällig ausgewählt, und zwar anhand einer Heuristik zum Auswählen von Wörtern, die am ehesten verwechselt werden. Die durch dieses heuristische Oversampling eingeführte Verzerrung kann anschließend mithilfe der Gewichte nach Wichtigkeit korrigiert werden.

In all diesen Fällen wird die rechnerische Komplexität der Gradientenschätzung für die Ausgabeschicht auf einen Wert reduziert, der proportional zur Anzahl der negativen Stichproben ist, und nicht proportional zur Größe des Ausgabevektors.

12.4.3.4 Noise-Contrastive Estimation und Rankingverlust

Andere vorgeschlagene Ansätze, die auf Stichprobenentnahme basieren, sollen den Berechnungsaufwand für das Training neuronaler Sprachmodelle mit großen Wortschätzungen reduzieren. Ein frühes Beispiel ist der von *Collobert und Weston* (2008a) vorgeschlagene Rankingverlust, der die Ausgabe des neuronalen Sprachmodells für jedes Wort als Score betrachtet und versucht, den Score des korrekten Worts a_y relativ zu den anderen Scores a_i hoch einzustufen. Der vorgeschlagene Rankingverlust ist dann

$$L = \sum_i \max(0, 1 - a_y + a_i). \quad (12.19)$$

Der Gradient ist Null für den i -ten Term, wenn der Score des beobachteten Worts, a_y , um 1 größer als der Score des negativen Worts a_i ist. Ein Problem mit diesem Kriterium ist, dass es keine geschätzten bedingten Wahrscheinlichkeiten liefert, die in einigen Anwendungen nützlich sind, darunter Spracherkennung und Texterzeugung (einschließlich der Erzeugung bedingter Texte, beispielsweise Übersetzungen).

Ein jüngeres Trainingsziel für neuronale Sprachmodelle ist die Noise-Contrastive Estimation, die in Abschnitt 18.6 vorgestellt wird. Dieser Ansatz

wurde erfolgreich auf neuronale Sprachmodelle angewandt (*Mnih und Teh*, 2012; *Mnih und Kavukcuoglu*, 2013).

12.4.4 Kombinieren neuronaler Sprachmodelle mit *N*-Grammen

Ein großer Vorteil der *N*-Gramm-Modelle gegenüber neuronalen Netzen ist, dass sie bereits bei wenigen Berechnungen für die Verarbeitung eines Beispiels (durch Nachschlagen weniger Tupels, die zum aktuellen Kontext passen) eine hohe Modellkapazität erreichen (durch Speichern der Häufigkeiten sehr vieler Tupel). Wenn wir Hash-Tabellen oder -Bäume zum Zugreifen auf die Zähler verwenden, ist die für die *N*-Gramme aufgewendete Berechnung beinahe unabhängig von der Kapazität. Dagegen führt eine Verdopplung der Anzahl der Parameter in einem neuronalen Netz auch zu einer annähernden Verdopplung der Rechendauer. Ausnahmen sind unter anderem Modelle, die es vermeiden, in jedem Durchgang alle Parameter zu verwenden. Embedding-Schichten indizieren nur ein einzelnes Embedding pro Durchgang, sodass wir die Wortschatzgröße erhöhen können, ohne dabei auch die Berechnungsdauer pro Beispiel zu erhöhen. Einige andere Modelle, beispielsweise Tiled Convolutional Networks (TCNNs), können Parameter hinzufügen, während der Grad des Parameter Sharings so reduziert wird, dass der Berechnungsaufwand gleich bleibt. Typische Schichten neuronaler Netze auf Basis der Matrizenmultiplikation verwenden dagegen einen Rechenaufwand, der proportional zur Anzahl der Parameter ist.

Eine einfache Möglichkeit, für weitere Kapazität zu sorgen, besteht daher darin, beide Ansätze in einem Ensemble bestehend aus einem neuronalen Sprachmodell und einem *N*-Gramm-Sprachmodell zu kombinieren (*Bengio et al.*, 2001, 2003). Wie bei jedem anderen Ensemble kann diese Methode den Testfehler reduzieren, wenn die Ensembleelemente unabhängige Fehler machen. Im Bereich des Ensemble Learnings finden sich viele Möglichkeiten zum Kombinieren der Vorhersagen von Ensembleelementen, darunter das gleichförmige Gewicht und die für eine Validierungsdatenmenge gewählten Gewichte. *Mikolov et al.* (2011a) haben das Ensemble so erweitert, dass es nicht nur zwei Modelle, sondern eine große Anzahl von Modellen enthält. Es ist auch möglich, ein neuronales Netz mit einem Modell für maximale Entropie zu koppeln und die beiden gemeinsam zu trainieren (*Mikolov et al.*, 2011b). Dieser Ansatz kann als Training eines neuronalen Netzes mit einer zusätzlichen Menge von Eingaben, die direkt mit der Ausgabe – aber nicht mit anderen Teilen des Modells – verbunden sind, betrachtet werden. Die zusätzlichen Eingaben sind Indikatoren für das Vorhandensein bestimmter

N -Gramme im Eingabekontext, sodass diese Variablen sehr hochdimensional und sehr dünnbesetzt sind. Der Anstieg der Modellkapazität ist gewaltig – der neue Teil der Architektur enthält bis zu $|sV|^n$ Parameter –, aber die Menge der für die Verarbeitung einer Eingabe zusätzlich benötigten Berechnungen ist minimal, da die zusätzlichen Eingaben sehr dünnbesetzt sind.

12.4.5 Neuronale maschinelle Übersetzung

Maschinelle Übersetzung bezeichnet die Aufgabe, bei der ein Satz in einer natürlichen Sprache eingelesen und ein Satz mit gleichwertiger Bedeutung in einer anderen Sprache ausgegeben wird. Systeme für maschinelle Übersetzungen umfassen häufig viele Komponenten. Auf hohem Niveau gibt es häufig eine Komponente, die viele mögliche Übersetzungen vorschlägt. Viele dieser Übersetzungen sind aufgrund der Sprachunterschiede grammatisch nicht korrekt. Zum Beispiel stehen in vielen Sprachen Adjektive nach Nomen, sodass bei einer direkten Übersetzung ins Deutsche Formulierungen wie »Apfel rot« entstehen. Der Vorschlagsmechanismus empfiehlt Varianten der empfohlenen Übersetzungen, von denen eine idealerweise »roter Apfel« lautet. Eine zweite Komponente des Übersetzungssystems – ein Sprachmodell – bewertet die Übersetzungsvorschläge und kann »roter Apfel« mit einem besseren Score als »Apfel rot« bewerten.

Die früheste Erforschung neuronaler Netze für die maschinelle Übersetzung setzte bereits auf Encoder und Decoder (*Allen* 1987; *Chrisman* 1991; *Forcada und Ñeco* 1997). Der erste groß angelegte kompetitive Einsatz neuronaler Netze zur Übersetzung bestand in einer Aktualisierung des Sprachmodells eines Übersetzungssystems anhand eines neuronalen Sprachmodells (*Schwenk et al.*, 2006; *Schwenk*, 2010). Bis dahin hatten die meisten maschinellen Übersetzungssysteme für diese Komponente ein N -Gramm-Modell genutzt. Zu den für die maschinelle Übersetzung eingesetzten Modellen auf N -Gramm-Basis gehören nicht nur traditionelle Backoff-Modelle auf N -Gramm-Basis (*Jelinek und Mercer*, 1980; *Katz*, 1987; *Chen und Goodman*, 1999), sondern auch **Sprachmodelle mit maximaler Entropie** (*Berger et al.*, 1996), in denen eine affin-softmax-Schicht das nächste Wort abhängig vom Vorhandensein häufiger N -Gramme im Kontext vorhersagt.

Klassische Sprachmodelle geben einfach die Wahrscheinlichkeit eines Satzes in natürlicher Sprache aus. Da die maschinelle Übersetzung das Erzeugen eines Ausgabesatzes auf Grundlage des Eingabesatzes beinhaltet, ist es sinnvoll, das natürliche Sprachmodell als bedingtes Modell zu gestalten. Wie in Abschnitt 6.2.1.1 beschrieben, ist es recht einfach, ein

Modell, das eine Randverteilung über eine Variable definiert, so zu erweitern, dass eine bedingte Verteilung über diese Variable für einen Kontext C definiert wird, wobei C eine einzelne Variable oder eine Variablenliste sein kann. *Devlin et al.* (2014) übertrafen den Stand der Technik in einigen statistischen Benchmarks für die maschinelle Übersetzung mit einem mehrschichtigen Perzepton (MLP), das eine Phrase t_1, t_2, \dots, t_k in der Zielsprache anhand einer Phrase s_1, s_2, \dots, s_n in der Ausgangssprache bewertete. Das mehrschichtige Perzepton schätzt $P(t_1, t_2, \dots, t_k | s_1, s_2, \dots, s_n)$. Der von diesem mehrschichtigen Perzepton gebildete Schätzwert ersetzt den Schätzwert der bedingten N -Gramm-Modelle.

Ein Nachteil des MLP-Ansatzes ist, dass die Sequenzen in einem vorbereitenden Schritt auf eine feste Länge vereinheitlicht werden müssen. Für flexiblere Übersetzungen sollte das Modell jedoch mit Ein- und Ausgaben unterschiedlicher Länge zurechtkommen. Ein RNN ist dazu in der Lage. Abschnitt 10.2.4 beschreibt mehrere Möglichkeiten zum Konstruieren eines RNNs, das eine bedingte Verteilung über eine Sequenz für eine Eingabe darstellt. Abschnitt 10.4 beschreibt, wie diese Bedingung erfolgen kann, wenn die Eingabe eine Sequenz ist. In allen Fällen liest das Modell zunächst die Eingabesequenz ein und gibt dann eine Datenstruktur aus, die diese Eingabesequenz zusammenfasst. Wir nennen diese Zusammenfassung den »Kontext« C . Beim Kontext C kann es sich um eine Liste mit Vektoren, einen Vektor oder einen Tensor handeln. Beim Modell, das die Eingabe zum Erzeugen von C einliest, kann es sich um ein RNN (*Cho et al.*, 2014a; *Sutskever et al.*, 2014; *Jean et al.*, 2014) oder ein CNN handeln (*Kalchbrenner und Blunsom*, 2013). Ein zweites Modell – meist ein RNN – liest dann den Kontext C ein und erzeugt den Satz in der Zielsprache. Diese allgemeine Idee eines Encoder–Decoder-Frameworks für die maschinelle Übersetzung ist in Abbildung 12.5 dargestellt.

Um einen vollständigen Satz zu erzeugen, der vom Ausgangssatz abhängt, muss das Modell den gesamten Ausgangssatz repräsentieren können. Frühere Modelle konnten nur einzelne Wörter oder Phrasen repräsentieren. Vom Standpunkt des Representation Learnings kann es nützlich sein, eine Repräsentation zu erlernen, in der Sätze mit derselben Bedeutung ähnliche Repräsentationen aufweisen – und zwar ungeachtet der Tatsache, ob sie in der Ausgangssprache oder der Zielsprache verfasst wurden. Dieses Verfahren wurde zunächst mit einer Kombination aus Faltungen und RNNs untersucht (*Kalchbrenner und Blunsom*, 2013). Spätere Arbeiten verwendeten ein RNN zur Bewertung von Übersetzungsvorschlägen (*Cho et al.*, 2014a) und zur Erzeugung übersetzter Sätze (*Sutskever et al.*, 2014). *Jean et al.* (2014) haben diese Modelle auf größere Wortschätzte skaliert.

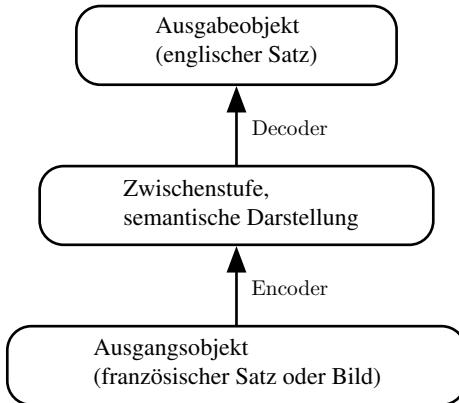


Abbildung 12.5: Die Encoder–Decoder–Architektur für die wechselseitige Abbildung einer Oberflächenrepräsentation (beispielsweise einer Sequenz von Wörtern oder einem Bild) und einer semantischen Repräsentation. Durch Verwendung der Ausgabe eines Encoders für Daten aus einer Modalität (zum Beispiel der Encoder-Zuordnung der französischen Sätze zu verdeckten Repräsentationen, die die Bedeutung der Sätze erfassen) als Eingabe für einen Decoder einer anderen Modalität (zum Beispiel der Decoder-Zuordnung von verdeckten Repräsentationen zum Erfassen der Bedeutung der Sätze ins Englische) können wir Systeme trainieren, die von einer Modalität in eine andere übersetzen. Diese Idee wurde nicht nur erfolgreich für die maschinelle Übersetzung genutzt, sondern auch zum Erzeugen von Bildunterschriften.

12.4.5.1 Verwenden eines Aufmerksamkeitsmechanismus und Ausrichten von Datenelementen

Es ist sehr schwierig, eine Repräsentation fester Größe zur Erfassung aller semantischen Details eines sehr langen Satzes mit beispielsweise 60 Wörtern zu nutzen. Das ist nur durch gutes und langes Trainieren eines hinreichend großen RNNs möglich, wie *Cho et al.* (2014a) und *Sutskever et al.* (2014) gezeigt haben. Effizienter ist es jedoch, den gesamten Satz oder Absatz (um den Kontext und die Kernaussage zu erfassen) einzulesen und anschließend die übersetzten Wörter eines nach dem anderen zu erzeugen, wobei der Fokus jeweils auf einem anderen Teil des Eingabesatzes liegt, um die semantischen Einzelheiten für das Erzeugen des nächsten Ausgabeworts zu erfassen. Genau diese Idee brachten *Bahdanau et al.* (2015) als Erste auf. Der Aufmerksamkeitsmechanismus, der eingesetzt wird, um in jedem Zeitschritt bestimmte Teile der Eingabesequenz zu fokussieren, ist in Abbildung 12.6 dargestellt.

Ein auf Aufmerksamkeit basierendes System kann aus drei Komponenten bestehen:

1. Ein Prozess, der Rohdaten (zum Beispiel Wörter eines Ausgangssatzes) *einliest* und in verteilte Repräsentationen umwandelt, wobei jeder Wortposition ein Merkmalsvektor zugeordnet wird.
2. Eine Liste mit Merkmalsvektoren, die die Ausgabe der lesenden Komponente speichert. Sie entspricht einem *Speicher*, der eine Sequenz von Fakten enthält, die später (nicht unbedingt in derselben Reihenfolge) abgerufen werden können, ohne auf alle davon zugreifen zu müssen.
3. Ein Prozess, der den Inhalt des Speichers *auswertet*, um eine Aufgabe sequenziell zu erledigen, wobei in jedem Zeitschritt die Aufmerksamkeit dem Inhalt eines Speicherelements (oder mehrerer Elemente, mit einem unterschiedlichen Gewicht) gewidmet werden kann.

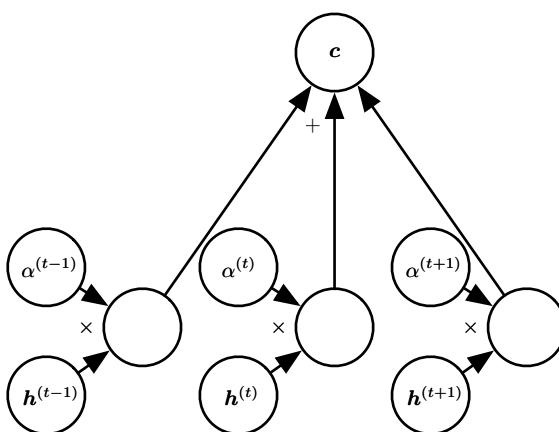


Abbildung 12.6: Ein moderner Aufmerksamkeitsmechanismus, wie er von *Bahdanau et al.* (2015) vorgestellt wurde, ist letztlich ein gewichtetes Mittel. Ein Kontextvektor c wird durch Bilden eines gewichteten Mittels der Merkmalsvektoren $h^{(t)}$ mit den Gewichten $\alpha^{(t)}$ geformt. In einigen Anwendungen sind die Merkmalsvektoren h verdeckte Einheiten eines neuronalen Netzes, aber es kann sich auch um unbearbeitete Eingaben des Modells handeln. Die Gewichte $\alpha^{(t)}$ werden vom Modell selbst erzeugt. Es handelt sich normalerweise um Werte im Bereich $[0, 1]$, die sich auf nur ein $h^{(t)}$ konzentrieren sollen, damit das gewichtete Mittel die Ausgabewerte exakt dieses spezifischen Zeitschritts approximiert. Die Gewichte $\alpha^{(t)}$ werden meist durch Anwenden einer softmax-Funktion auf Relevanzbewertungen, die von einem anderen Teil des Modells stammen, erzeugt. Der Aufmerksamkeitsmechanismus ist rechenaufwendiger als die direkte Indizierung der gewünschten $h^{(t)}$, aber die direkte Indizierung kann nicht im Gradientenabstiegsverfahren trainiert werden. Der Aufmerksamkeitsmechanismus auf Basis der gewichteten Mittel ist eine glatte differenzierbare Approximation, die sich mit bestehenden Optimierungsalgorithmen trainieren lässt.

Die dritte Komponente erzeugt den übersetzten Satz.

Wenn Wörter eines Satzes in einer Sprache mit den entsprechenden Wörtern in einem übersetzten Satz in einer anderen Sprache in Übereinstimmung gebracht werden, wird es möglich, die entsprechenden Wort-Embeddings zueinander in Beziehung zu setzen. Frühere Arbeiten zeigten, dass man eine Art Translationsmatrix erlernen könnte, die Wort-Embeddings der einen den Embeddings in der anderen Sprache zuordnet (*Kočiský et al.*, 2014), was zu niedrigeren Fehlerquoten führt als klassische Ansätze, die auf Häufigkeitszählungen in der Phrasentabelle basieren. Es gibt sogar noch frühere Arbeiten zum Erlernen von sprachübergreifenden Wortvektoren (*Klementiev et al.*, 2012). Viele Erweiterungen dieses Ansatzes sind möglich. Zum Beispiel ermöglichen effizientere sprachübergreifende Ausrichtungen das Trainieren mit größeren Datensätzen (*Gouws et al.*, 2014).

12.4.6 Historische Einblicke

Das Konzept verteilter Repräsentationen für Symbole wurde von *Rumelhart et al.* (1986a) in einer der ersten Untersuchungen über die Backpropagation vorgestellt. Dabei entsprechen Symbole der Identität von Familienmitgliedern und das neuronale Netz erfasst die Beziehungen zwischen Familienmitgliedern anhand von Trainingsbeispielen, die Triplets wie (Colin, Mutter, Victoria) bilden. Die erste Schicht des neuronalen Netzes erlernte eine Repräsentation für jedes Familienmitglied. Zum Beispiel können die Merkmale für Colin anzeigen, zu welchem Familienbaum Colin gehört, in welchem Zweig des Baums er sich befindet, aus welcher Generation er stammt und so weiter. Man kann sich vorstellen, dass das neuronale Netz erlernte Regeln, die diese Attribute miteinander in Beziehung setzen, berechnet, um die gewünschten Vorhersagen zu erzielen. Das Modell kann dann Vorhersagen treffen und zum Beispiel schließen, wer Colins Mutter ist.

Das Konzept des Embeddings für ein Symbol wurde von *Deerwester et al.* (1990) zum Konzept des Embeddings für ein Wort erweitert. Diese Embeddings wurden mittels Singulärwertzerlegung (engl. *singular value decomposition*, SVD) erlernt. Später wurden die Embeddings von neuronalen Netzen erlernt.

Die Geschichte der Verarbeitung natürlicher Sprache zeichnet sich durch wechselnde Popularität unterschiedlicher Arten der Darstellung der Eingabe für das Modell aus. Nach dieser frühen Arbeit zu Symbolen und Wörtern stellten einige der ersten Anwendungen neuronaler Netze für die Verarbeitung natürlicher Sprache (*Miikkulainen und Dyer*, 1991; *Schmidhuber*, 1996) die Eingabe als Sequenz von Zeichen dar.

Bengio et al. (2001) rückten das Modellieren von Wörtern wieder in den Fokus und stellten neuronale Sprachmodelle vor, die interpretierbare Wort-Embeddings erzeugen. Diese neuronalen Modelle wurden vom Definieren von Repräsentationen für eine kleine Symbolmenge in den 1980ern auf Millionen von Wörtern (einschließlich Eigennamen und falscher Schreibweisen) in modernen Anwendungen skaliert. Diese rechnerische Skalierung führte zur Entwicklung der in Abschnitt 12.4.3 beschriebenen Verfahren.

Zu Beginn führte die Verwendung von Wörtern als Grundeinheiten in Sprachmodellen zu einer höheren Leistung bei der Sprachmodellierung (*Bengio et al.*, 2001). Bis heute treiben immer neue Verfahren die Entwicklung von Modellen auf Zeichenbasis (*Sutskever et al.*, 2011) und Modellen auf Wortbasis kontinuierlich voran. Jüngere Arbeiten (*Gillick et al.*, 2015) modellieren sogar einzelne Bytes von Unicode-Zeichen.

Die Konzepte hinter neuronalen Sprachmodellen wurden in mehrere Anwendungen für die Verarbeitung natürlicher Sprache übernommen, zum Beispiel für das Parsing (*Henderson*, 2003, 2004; *Collobert*, 2011), die Kennzeichnung von Wortarten (Part-of-speech Tagging), das Labeln semantischer Rollen (semantic Role Labeling), Chunking usw. Zum Teil wird eine einzelne Multitask-Learning-Architektur (*Collobert und Weston*, 2008a; *Collobert et al.*, 2011a) verwendet, in der die Wort-Embeddings über mehrere Aufgaben geteilt werden.

Zweidimensionale Visualisierungen von Embeddings wurden zu einem beliebten Hilfsmittel bei der Analyse von Sprachmodellen, nachdem 2009 der t-SNE-Algorithmus zur Reduzierung der Dimensionalität (*van der Maaten und Hinton*, 2008) und seine berühmte Anwendung zur Visualisierung von Wort-Embeddings von Joseph Turian entwickelt wurde.

12.5 Weitere Anwendungen

In diesem Abschnitt behandeln wir einige weitere Deep-Learning-Anwendungen, die sich von den bereits erwähnten üblichen Aufgaben zur Objekterkennung, Spracherkennung und Verarbeitung natürlicher Sprache unterscheiden. Teil III dieses Buchs enthält weitere Aufgaben, die derzeit hauptsächlich in den Bereich der Forschung fallen.

12.5.1 Empfehlungsdienste

Eine wesentliche Anwendungsfamilie in der IT ist das Aussprechen von Empfehlungen für bestimmte Artikel oder Themen gegenüber Interessenten oder

Kunden. Für derartige Anwendungen gibt es zwei Haupteinsatzbereiche, nämlich die Onlinewerbung und Artikelempfehlungen (die ebenfalls oft den Verkauf von Produkten zum Ziel haben). Beide beruhen auf einer Vorhersage von Verbindungen zwischen einem Benutzer und einem Artikel, um so die Wahrscheinlichkeit einer Aktion (der Benutzer kauft das Produkt oder führt eine gleichwertige Aktion aus) oder den erwarteten Gewinn bzw. Nutzen (die vom Produktwert abhängen kann) beim Ausspielen einer Werbung oder Empfehlung für ein Produkt an einen Benutzer vorherzusagen. Das Internet wird derzeit größtenteils durch verschiedene Formen der Onlinewerbung finanziert. Große Teile der Wirtschaft beruhen auf Onlineshopping. Unternehmen wie Amazon und eBay nutzen Machine Learning und dabei auch Deep Learning für ihre Produktempfehlungen. Es muss sich bei den Artikeln aber nicht unbedingt um zum Kauf stehende Produkte handeln. Es kann dabei auch um die in einem sozialen Netzwerk im Newsfeed anzuseigenden Beiträge gehen, Film- und Serienempfehlungen, Witze, Expertentipps, geeignete Spielpartner für Videospiele oder potenzielle Partner in einer Dating-App.

Häufig wird diese Zuordnung wie eine Aufgabenstellung für überwachtes Lernen behandelt: Anhand bestimmter Informationen über den Artikel (oder Service usw.) und den Nutzer soll die gewünschte Aktion (Anklicken einer Anzeige, Eingeben einer Bewertung, Anklicken einer »Gefällt mir«-Schaltfläche, Kauf eines Produkts, Zahlen eines bestimmten Geldbetrags für das Produkt, Verweildauer auf einer Produktseite usw.) vorhergesagt werden. Am Ende steht oft eine Aufgabenstellung einer Regression (Vorhersage eines bedingten Erwartungswerts) oder einer probabilistischen Klassifizierung (Vorhersage der bedingten Wahrscheinlichkeit eines diskreten Ereignisses).

Frühe Arbeiten zu Empfehlungsdiensten (auch Empfehlungssysteme oder *Recommender Systems* genannt) verließen sich auf wenige Informationen als Eingaben für diese Vorhersagen: Benutzererkennung und Artikelnummer. In diesem Zusammenhang lässt sich nur aufgrund der Ähnlichkeit zwischen den Mustern von Werten der Zielvariable für unterschiedliche Benutzer oder unterschiedliche Artikel generalisieren. Wenn sowohl Benutzer 1 als auch Benutzer 2 die Artikel A, B und C mögen, können wir schließen, dass beide Benutzer einen ähnlichen Geschmack haben. Wenn Benutzer 1 Artikel D mag, ist das möglicherweise ein starker Hinweis darauf, dass auch Benutzer 2 den Artikel D mag. Algorithmen, die auf dieser Basis arbeiten, werden als **kollaborative Filter** bezeichnet. Es sind sowohl nichtparametrische Ansätze (wie Nearest-Neighbor-Verfahren auf Basis der geschätzten Ähnlichkeit zwischen Mustern oder Präferenzen) als auch parametrische Verfahren möglich. parametrische Verfahren verlassen sich oft auf das Erlernen einer

verteilten Repräsentation (auch Embedding genannt, dt. *Einbettung*) für jeden Benutzer und jeden Artikel. Die bilineare Vorhersage der Zielvariable (zum Beispiel einer Bewertung) ist ein einfaches parametrisches Verfahren, das extrem erfolgreich und häufig ein Teil moderner Systeme ist. Die Vorhersage wird aus dem Skalarprodukt zwischen dem Benutzer-Embedding (also der Repräsentation des Benutzers) und dem Artikel-Embedding (also der Repräsentation des Artikels) gewonnen – möglicherweise mit Korrekturkonstanten, die nur von der Benutzererkennung oder nur der Artikelnummer abhängig sind. $\hat{\mathbf{R}}$ sei die Matrix unserer Vorhersagen, \mathbf{A} eine Matrix mit den Repräsentationen der Benutzer in den Zeilen und \mathbf{B} eine Matrix mit den Repräsentationen der Artikel in den Spalten. \mathbf{b} und \mathbf{c} seien Vektoren, die jeweils eine Art Verzerrung (engl. *bias*) für jeden Benutzer (ein Maß für seine grundsätzliche negative oder positive Haltung) und jeden Artikel (ein Maß für dessen allgemeine Beliebtheit) enthalten. Die bilineare Vorhersage ergibt sich somit wie folgt:

$$\hat{R}_{u,i} = b_u + c_i + \sum_j A_{u,j} B_{j,i}. \quad (12.20)$$

Üblicherweise möchten wir den quadratischen Fehler zwischen den vorhergesagten Bewertungen $\hat{R}_{u,i}$ und den tatsächlichen Bewertungen $R_{u,i}$ minimieren. Repräsentationen der Benutzer und der Artikel können ganz einfach visualisiert werden, wenn sie zunächst auf eine niedrige Dimension (zwei oder drei) reduziert werden. Sie können auch, wie Wort-Embeddings, zum Vergleichen von Benutzern oder Artikeln miteinander dienen. Eine Möglichkeit zum Erhalten dieses Embeddings ist die Singulärwertzerlegung der Matrix \mathbf{R} mit den tatsächlichen Zielen (zum Beispiel Bewertungen). Dies entspricht der Faktorisierung $\mathbf{R} = \mathbf{UDV}'$ (oder einer normalisierten Variante) in das Produkt der beiden Faktoren, der niederrangigen Matrizen $\mathbf{A} = \mathbf{UD}$ und $\mathbf{B} = \mathbf{V}'$. Ein Problem bei der Singulärwertzerlegung ist, dass sie die fehlenden Einträge auf willkürliche Art behandelt, als würden sie einem Zielwert von 0 entsprechen. Wir möchten stattdessen den Aufwand für Vorhersagen zu fehlenden Einträgen vermeiden. Zum Glück kann die Summe der quadratischen Fehler der beobachteten Bewertungen ebenfalls einfach durch eine Optimierung auf Gradientenbasis minimiert werden. Die Singulärwertzerlegung und die bilineare Vorhersage aus Gleichung 12.20 haben im Rennen um den Netflix-Preis (*Bennett und Lanning*, 2007) beide gut abgeschnitten. Bei diesem Wettbewerb müssen die Bewertungen für Filme aufgrund bisheriger Bewertungen einer großen Anzahl anonymer Nutzer vorhergesagt werden. Viele Machine-Learning-Experten haben an diesem Wettbewerb, der von 2006 bis 2009 stattfand, teilgenommen. Er führte zu verstärkter Forschung an Empfehlungsdiensten mit modernen Machine-

Learning-Verfahren und führte zu Verbesserungen bei den Empfehlungsdiesten. Obwohl die einfache bilineare Vorhersage oder die Singulärwertzerlegung für sich genommen nicht gewannen, waren sie eine Komponente der von den meisten Teilnehmern vorgestellten Ensemblemodellen, darunter auch die Gewinner (*Töscher et al., 2009; Koren, 2009*).

Neben diesen bilinearen Modellen mit verteilten Repräsentationen beruht eine der ersten Anwendungen neuronaler Netze für kollaboratives Filtern auf dem ungerichteten probabilistischen RBM-Modell (*Salakhutdinov et al., 2007*). RBMs (engl. *restricted Boltzmann machines*) waren ein wichtiges Element des Ensembles von Verfahren, das den Netflix-Wettbewerb gewann (*Töscher et al., 2009; Koren, 2009*). Modernere Varianten der Faktorisierung der Bewertungsmatrix wurden von der Forschungsgemeinde rund um neuronale Netze ebenfalls untersucht (*Salakhutdinov und Mnih, 2008*).

Systeme mit kollaborativen Filtern haben allerdings eine grundlegende Einschränkung: Wenn ein Artikel oder ein Benutzer hinzukommt, kann dieser aufgrund einer fehlenden Bewertungshistorie nicht hinsichtlich seiner Ähnlichkeit zu anderen Artikeln oder Benutzern bewertet werden. Es kann auch keine Art der Verbindung zwischen dem neuen Benutzer und vorhandenen Artikeln bestimmt werden. Dies wird als Cold-Start-Problem bezeichnet. Eine allgemeine Möglichkeit zum Lösen des Cold-Start-Problems besteht im Hinzufügen zusätzlicher Informationen über die einzelnen Benutzer und Artikel. Dabei kann es sich zum Beispiel um Angaben aus dem Benutzerprofil oder Merkmale für jeden Artikel handeln. Systeme, die auf solche Daten zugreifen, werden als **inhaltsabhängige Empfehlungsdienste** bezeichnet. Die Zuordnung aus einer umfassenden Menge von Benutzer- oder Artikelmerkmalen zu einem Embedding kann mithilfe einer Deep-Learning-Architektur erlernt werden (*Huang et al., 2013; Elkahky et al., 2015*).

Spezielle Deep-Learning-Architekturen wie CNNs wurden auch zum Erlernen der Merkmalsextraktion aus umfangreichen Inhalten eingesetzt, beispielsweise für Musikempfehlungen anhand von Musiktiteln (*van den Oörd et al., 2013*). Dabei zieht das CNN die akustischen Merkmale als Eingabe heran und berechnet ein Embedding für den zugehörigen Titel. Das Skalarprodukt von diesem Embedding für einen Titels und dem Embedding für einen Benutzer wird anschließend verwendet, um vorherzusagen, ob ein Benutzer sich den Titel anhört.

12.5.1.1 Exploration versus Exploitation

Wenn Empfehlungen für Benutzer ausgesprochen werden, kommt es zu einem Problem, das über das gewöhnliche überwachte Lernen hinausgeht

und sich in den Bereich des Reinforcement Learnings (dt. *bestärkendes* oder *verstärkendes Lernen*) erstreckt. Viele Empfehlungsprobleme lassen sich theoretisch sehr treffend als **kontextabhängige Banditen** (engl. *contextual bandits*) beschreiben (*Langford und Zhang*, 2008; *Lu et al.*, 2010). Beim Verwenden eines Empfehlungsdienstes zum Sammeln von Daten erhalten wir ein verzerrtes und unvollständiges Bild der Präferenzen des Benutzers: Wir sehen nur seine Reaktionen auf die empfohlenen Artikel, wissen aber nicht, was er von anderen Artikeln hält. Außerdem erhalten wir manchmal gar keine Informationen zu Benutzern, denen keine Empfehlungen unterbreitet wurden (zum Beispiel kann es bei Anzeigenauktionen vorkommen, dass der für die Anzeige gebotene Preis unter einer Mindestpreisgrenze lag oder die Auktion nicht gewonnen wurde, sodass die Anzeige überhaupt nicht ausgespielt wurde). Viel wichtiger aber: Wir erhalten keine Informationen darüber, welches Ergebnis wir bei einer anderen Artikelempfehlung erzielt hätten. Das wäre so, als würden wir einen Klassifikator trainieren, indem wir eine Klasse \hat{y} für jedes Trainingsbeispiel \mathbf{x} auswählen (meist die Klasse mit der höchsten Wahrscheinlichkeit laut Modell) und dann nur die Angaben sehen würden, ob es sich um die richtige Klasse handelte (oder nicht). Offensichtlich vermittelt jedes Beispiel weniger Informationen als im überwachten Fall, in dem das korrekte Label y direkt verfügbar ist; es sind also mehr Beispiele erforderlich. Noch schlimmer: Wenn wir nicht vorsichtig sind, könnten wir am Ende ein System haben, das immer wieder die falschen Entscheidungen trifft, obwohl mehr und mehr Daten gesammelt werden, weil die korrekte Entscheidung zu Beginn eine sehr geringe Wahrscheinlichkeit aufwies: Nur, wenn der Klassifikator diese korrekte Entscheidung trifft, lernt er etwas über die korrekte Entscheidung. Das ähnelt dem Reinforcement Learning, wo nur die Belohnung für die ausgewählte Aktion beobachtet wird. Generell kann Reinforcement Learning eine Abfolge vieler Aktionen und vieler Belohnungen umfassen. Das Banditen-Szenario ist ein Sonderfall des Reinforcement Learnings, bei dem der Klassifikator nur eine Aktion vornimmt und eine einzelne Belohnung erhält. Das Banditenproblem ist insofern einfacher, als der Klassifikator weiß, welche Belohnung mit welcher Aktion verbunden ist. Im allgemeinen Fall des Reinforcement Learnings könnte eine kürzliche Aktion oder eine lang zurückliegende Aktion der Grund für eine hohe oder eine niedrige Belohnung sein. Der Begriff **kontextabhängige Banditen** bezeichnet den Fall, in dem die Aktion im Kontext einer Eingangsvariable erfolgt, die die Entscheidung begründen kann. Ein Beispiel: Wir kennen in jedem Fall die Benutzeridentität und möchten einen Artikel auswählen. Die Zuordnung von Kontext zu Aktion wird auch **Policy** genannt. Die Feedbackschleife zwischen dem Klassifikator und der Datenverteilung (die nun von

den Aktionen des Klassifikators abhängt) ist ein zentrales Forschungsthema in der Literatur zu Reinforcement Learning und Banditen.

Reinforcement Learning erfordert einen Kompromiss zwischen **Exploration** (Erkundung) und **Exploitation** (Ausnutzung). Exploitation bezeichnet die Aktionen, die sich aus der aktuellen besten Version der erlernten Policy ergeben – Aktionen, von denen wir wissen, dass sie zu einer hohen Belohnung führen. Exploration bezeichnet die Aktionen, die speziell zum Sammeln weiterer Trainingsdaten ausgeführt werden. Wenn wir wissen, dass im Kontext x die Aktion a zu einer Belohnung von 1 führt, wissen wir noch nicht, ob dies die bestmögliche Belohnung darstellt. Wir können uns entscheiden, die momentane Policy auszunutzen (exploit), und uns weiterhin für Aktion a entscheiden, damit wir relativ sicher eine Belohnung von 1 erhalten. Wir können uns allerdings auch für Exploration entscheiden und es mit Aktion a' probieren. Wir wissen nicht, was geschieht, wenn wir uns für Aktion a' entscheiden. Wir hoffen auf eine Belohnung von 2, aber es besteht die Gefahr, mit einer Belohnung von 0 zu enden. In jedem Fall gewinnen wir jedoch an Kenntnis.

Exploration lässt sich auf vielerlei Weise umsetzen – von gelegentlichen zufälligen Aktionen, mit denen der gesamte Raum möglicher Aktionen abgedeckt werden soll, bis hin zu Ansätzen auf Modellbasis, die eine Auswahl der Aktion aufgrund der erwarteten Belohnung und der Unsicherheit des Modells bezüglich dieser Belohnung berechnen.

Viele Faktoren bestimmen das Ausmaß unserer Vorliebe für Exploration oder Exploitation. Zu den wichtigsten gehört wohl die Zeitskala, die uns wichtig ist. Wenn der Agent nur wenig Zeit zum Ansammeln der Belohnung hat, tendieren wir eher zur Exploitation. Steht dagegen viel Zeit zum Ansammeln der Belohnung zur Verfügung, widmen wir uns verstärkt der Exploration, damit künftige Aktionen effektiver auf Basis eines größeren Wissens geplant werden können. Mit fortschreitender Zeit und verbesserter erlernter Policy neigt sich die Waage in Richtung Exploitation.

Überwachtes Lernen kennt keinen Kompromiss zwischen Exploration und Exploitation, da das Überwachungssignal stets vorgibt, welche Ausgabe für welche Eingabe korrekt ist. Es ist nicht notwendig, unterschiedliche Ausgaben auszuprobieren, um herauszufinden, ob eine davon besser als die aktuelle Ausgabe des Modells ist – wir wissen bereits, dass das Label die beste Ausgabe ist.

Eine weitere Schwierigkeit im Rahmen des Reinforcement Learnings neben dem Kompromiss zwischen Exploration und Exploitation liegt in der Bewertung und dem Vergleich unterschiedlicher Policies. Reinforcement

Learning beinhaltet eine Interaktion zwischen dem Klassifikator und der Umgebung. Diese Feedbackschleife bedeutet, dass es keinen geradlinigen Pfad zur Bewertung der Leistung des Klassifikators anhand einer festen Menge von Eingabewerten der Testdatenmenge gibt. Die Policy selbst bestimmt, welche Eingaben wir zu Gesicht bekommen. *Dudik et al.* (2011) stellen Verfahren zur Bewertung kontextabhängiger Banditen vor.

12.5.2 Wissensrepräsentation, Schlussfolgern und Beantworten von Fragen

Deep-Learning-Ansätze haben sich dank der Embeddings von Symbolen (*Rumelhart et al.*, 1986a) und Wörtern (*Deerwester et al.*, 1990; *Bengio et al.*, 2001) als sehr erfolgreich für Sprachmodellierung, maschinelle Übersetzung und die Verarbeitung natürlicher Sprache erwiesen. Diese Embeddings stellen semantisches Wissen über einzelne Wörter und Konzepte dar. Ein unüberwindbares Problem in der Forschung ist die Entwicklung von Embedding für Phrasen und für Beziehungen zwischen Wörtern und Fakten. Suchmaschinen setzen hierfür bereits Machine Learning ein, aber es gibt noch viel zu tun, um derart fortschrittliche Repräsentationen zu verbessern.

12.5.2.1 Wissen, Beziehungen und Beantworten von Fragen

Eine interessante Forschungsrichtung beschäftigt sich mit der Frage, wie verteilte Repräsentationen so trainiert werden können, dass die **Beziehungen** zwischen zwei Elementen erfasst werden. Diese Beziehungen ermöglichen es, Fakten über Objekte und deren Interaktion miteinander zu formalisieren.

In der Mathematik ist eine **binäre Relation** eine Menge sortierter Objektpaare. Man sagt, dass die Paare der Menge in Relation zueinander stehen, während Paare, die nicht in der Menge enthalten sind, dies nicht tun. Zum Beispiel können wir die Beziehung »ist kleiner als« für die Menge der Elemente $\{1, 2, 3\}$ durch Definieren einer Menge sortierter Paare $\mathbb{S} = \{(1, 2), (1, 3), (2, 3)\}$ angeben. Sobald diese Beziehung definiert ist, können wir sie wie ein Verb benutzen. Weil $(1, 2) \in \mathbb{S}$, können wir sagen, dass 1 kleiner ist als 2. Weil $(2, 1) \notin \mathbb{S}$, können wir nicht sagen, dass 2 kleiner ist als 1. Natürlich muss es sich bei den in Beziehung zueinander stehenden Elementen nicht um Zahlen handeln. Wir könnten auch eine Beziehung **ist_eine_Art_von** definieren, die Tupel wie (Hund, Säugetier) enthält.

Im Rahmen der Künstlichen Intelligenz stellen wir uns eine Beziehung als Satz in einer syntaktisch einfachen und stark strukturierten Sprache vor. Die Beziehung übernimmt die Aufgabe des Verbs, die beiden Argumente der

Beziehung entsprechen dem Subjekt und dem Objekt. Diese Sätze bilden ein Triplet der Tokens

$$(\text{Subjekt}, \text{Verb}, \text{Objekt}) \quad (12.21)$$

mit den Werten

$$(\text{Element}_i, \text{Beziehung}_j, \text{Element}_k). \quad (12.22)$$

Wir können außerdem ein **Attribut** definieren – ein Konzept, das einer Beziehung ähnelt, aber nur ein Argument aufweist:

$$(\text{Element}_i, \text{Attribut}_j). \quad (12.23)$$

Denkbar wäre das Attribut hat_Fell für Elemente wie Hund.

Viele Anwendungen erfordern das Repräsentieren von Beziehungen und Schlussfolgerungen daraus. Wie lässt sich das für neuronale Netze am besten erreichen?

Machine-Learning-Modelle erfordern natürlich Trainingsdaten. Wir können anhand der Trainingsdatensätze, die aus unstrukturierter natürlicher Sprache bestehen, auf Beziehungen zwischen Elementen schließen. Es gibt auch strukturierte Datenbanken, die Beziehungen explizit angeben. Eine häufige Struktur für solche Datenbanken ist die **relationale Datenbank**, in der dieselben Daten abgelegt werden – allerdings nicht in Form von Sätzen mit drei Tokens. Wenn eine Datenbank Allgemeinwissen über den Alltag oder Expertenwissen über einen Anwendungsbereich für ein KI-System vermitteln soll, nennen wir diese Datenbank **Wissensbasis**. Eine Wissensbasis kann sehr allgemein sein wie **Freebase**, **OpenCyc**, **WordNet**, **Wikibase**,¹ und so weiter. Es gibt aber auch spezialisierte Varianten wie **GeneOntology**². Repräsentationen für Elemente und Beziehungen können durch Betrachten der einzelnen Triplets in einer Wissensbasis als Trainingsbeispiele erlernt werden; dazu muss das Trainingsziel, das ihre multivariate Verteilung erfasst, maximiert werden (*Bordes et al., 2013a*).

Neben den Trainingsdaten müssen wir außerdem die zu trainierende Modellfamilie definieren. Ein gängiger Ansatz besteht in der Anwendung neuronaler Sprachmodelle auf Modellelemente und Beziehungen. Neuronale Sprachmodelle erlernen einen Vektor, der die verteilte Repräsentation für jedes Wort angibt. Sie lernen auch bezüglich der Interaktionen zwischen

¹ Auf den jeweiligen Websites erhältlich: freebase.com, cyc.com/opencyc, wordnet.princeton.edu, wikiba.se

² geneontology.org

Wörtern, zum Beispiel welches Wort vermutlich auf eine Reihe anderer Wörter folgt; hierzu werden Funktionen dieser Vektoren erlernt. Wir können diesen Ansatz auf Elemente und Beziehungen ausweiten, indem wir für jede Beziehung einen Embedding-Vektor erlernen. Tatsächlich sind das Modellieren von Sprache und das Modellieren von als Beziehungen codiertem Wissen einander so ähnlich, dass Forscher Repräsentationen solcher Elemente *sowohl* mit Wissensbasen *als auch* Sätzen in natürlicher Sprache (*Bordes et al.*, 2011, 2012; *Wang et al.*, 2014a) oder durch Kombinieren der Daten aus mehreren relationalen Datenbanken (*Bordes et al.*, 2013b) trainiert haben. Es gibt viele Möglichkeiten für die jeweilige Parametrisierung eines solchen Modells. Frühe Arbeiten über das Erlernen von Beziehungen zwischen Elementen (*Paccanaro und Hinton*, 2000) postulierten parametrische Formen unter starken Nebenbedingungen (»linear relationale Embeddings«) und nutzten häufig verschiedene Formen der Repräsentation für die Beziehung und die Elemente. Zum Beispiel nutzten *Paccanaro und Hinton* (2000) und *Bordes et al.* (2011) ausgehend von der Annahme, dass eine Beziehung wie ein Operator auf Elemente wirkt, Vektoren für die Elemente und Matrizen für die Beziehungen. Alternativ können Beziehungen als ein weiteres Element betrachtet werden (*Bordes et al.*, 2012), sodass wir Aussagen über Beziehungen machen können, während das System, das diese kombiniert, mit Sicht auf das Modellieren der multivariaten Verteilungen flexibler wird.

Eine praktische Anwendung solcher Modelle ist die **Link Prediction**. Dabei werden fehlende Bögen in der Wissenskurve vorhergesagt. Es handelt sich um eine Art Generalisierung auf neue Fakten anhand alter Fakten. Die meisten aktuellen Wissensbasen wurden mit großem händischen Aufwand erstellt, sodass viele, wenn nicht gar die meisten echten Beziehungen darin fehlen. *Wang et al.* (2014b), *Lin et al.* (2015) und *Garcia-Duran et al.* (2015) zeigen Beispiele für eine solche Anwendung.

Die Leistungsbewertung eines Modells für die Link Prediction ist schwierig, da wir nur über einen Datensatz mit positiven Beispielen verfügen (Fakten, von denen wir wissen, dass sie wahr sind). Wenn das Modell ein Faktum vorschlägt, das nicht im Datensatz enthalten ist, sind wir nicht sicher, ob es einen Fehler gemacht oder ein neues, bisher unbekanntes Faktum gefunden hat. Die Metriken sind damit in einer gewissen Weise ungenau und basieren darauf, zu überprüfen, wie das Modell eine Menge bekannter positiver Fakten im Vergleich zu anderen Fakten, deren Korrektheit weniger wahrscheinlich ist, bewertet. Zum Konstruieren interessanter Beispiele, die wahrscheinlich negativ sind (Fakten, die wahrscheinlich falsch sind), können wir mit einem wahren Faktum beginnen und es beschädigen, zum Beispiel durch Austauschen eines Elements in der Beziehung gegen ein anderes

zufällig ausgewähltes Element. Das verbreitete 10-Prozent-Kriterium der Genauigkeit gibt an, wie oft das Modell ein »korrektes« Faktum zu den obersten 10 Prozent aller beschädigten Versionen des Faktums zählt.

Eine weitere Anwendung für Wissensbasen und ihre verteilten Repräsentationen ist die **Disambiguierung** (engl. *word-sense disambiguation*) (*Navigli und Velardi, 2005; Bordes et al., 2012*), bei der entschieden wird, welche Bedeutung eines Worts in einem bestimmten Kontext die passende ist.

Letztendlich könnte uns das Wissen über Beziehungen in Kombinationen mit Schlussfolgerungen und einem Verständnis der natürlichen Sprache dabei helfen, ein allgemeines Frage-Antwort-System zu entwickeln. Ein allgemeines Frage-Antwort-System muss in der Lage sein, Eingaben zu verarbeiten und sich wichtige Fakten zu merken. Dabei muss es so aufgebaut sein, dass diese Daten zu einem späteren Zeitpunkt abgerufen werden können, um Schlüsse daraus zu ziehen. Das ist nach wie vor ein komplexes Problem, das nur in sehr kontrollierten Umgebungen »gelöst« werden kann. Derzeit ist der beste Ansatz zum Erinnern und Abrufen einzelner deklarativer Fakten ein explizites Gedächtnis, das in Abschnitt 10.12 beschrieben wird. Memory-Netze wurden zunächst zum Lösen spielerischer Frage-Antwort-Aufgaben vorgeschlagen (*Weston et al., 2014*). *Kumar et al. (2015)* haben eine Erweiterung mittels rekurrenter GRU-Netze angeregt, die Eingaben in den Speicher lesen und Antworten auf Grundlage des Speicherinhalts geben.

Deep Learning ist in vielen anderen Bereichen neben den hier beschriebenen eingesetzt worden. Gewiss werden nach dem Schreiben dieser Zeilen weitere dazukommen. Es ist praktisch unmöglich, ein solches Thema umfassend zu behandeln. Diese Übersicht stellt daher lediglich eine repräsentatives Auswahl der aktuellen Möglichkeiten dar.

Damit ist Teil II abgeschlossen. Sie haben darin den Einsatz tiefer Netze in der modernen Praxis sowie die meisten erfolgreichen Verfahren kennengelernt. Ganz allgemein werden bei diesen Verfahren Gradienten als Kostenfunktion eingesetzt, um die Parameter eines Modells zu suchen, das eine gewünschte Funktion approximiert. Mit einer ausreichenden Menge an Trainingsdaten ist dieser Ansatz extrem leistungsstark. Wir wenden uns nun Teil III zu, in dem wir das Gebiet der Forschung betreten – Verfahren, die mit weniger Trainingsdaten auskommen oder für eine größere Menge von Aufgaben geeignet sein sollen. Hier stellen wir uns komplexeren Herausforderungen und Problemen, von deren Lösung wir noch weiter entfernt sind, als dies bei den bisher beschriebenen Anwendungen der Fall war.

III

Deep-Learning-Forschung

Dieser Teil des Buchs beschreibt die ambitionierteren und fortschrittlicheren Deep-Learning-Ansätze, denen sich die Forschungsgemeinde derzeit widmet.

In den bisherigen Teilen des Buchs haben wir gezeigt, wie überwachte Lernprobleme gelöst werden – wie das Mapping eines Vektors zu einem anderen erlernt wird, sofern genügend Beispiele für das Mapping vorhanden sind.

Allerdings fallen nicht alle Probleme, die wir lösen wollen, in diese Kategorie. Vielleicht möchten wir neue Beispiele erzeugen, die Wahrscheinlichkeit eines Punktes bestimmen oder mit fehlenden Werten umgehen und mit einem großen Datensatz arbeiten, der Beispiele ohne Label oder Beispiele aus ähnlichen Aufgaben enthält. Ein Nachteil aktueller industrieller Anwendungen ist, dass unsere Lernalgorithmen eine umfangreiche Menge überwachter Daten benötigen, um eine gute Genauigkeit zu erzielen. In diesem Teil des Buchs geht es um einige spekulative Ansätze, mit denen sich die Menge der Datensätze mit Labels, die für eine gute Funktion vorhandener Modelle notwendig sind, reduzieren lässt, sodass diese Modelle zugleich für weitere Aufgaben taugen. Um diese Ziele zu erreichen, wird im Normalfall eine Art des unüberwachten oder halb-überwachten Lernens benötigt.

Viele Deep-Learning-Algorithmen wurden entwickelt, um unüberwachte Lernprobleme anzugehen – aber keiner davon hat das Problem wirklich auf dieselbe Weise gelöst, wie das Deep Learning es größtenteils für das Problem des überwachten Lernens bei einer Vielzahl von Aufgaben getan hat. In diesem Teil beschreiben wir bestehende Ansätze für unüberwachtes Lernen und wie in diesem Bereich Fortschritte erzielt werden können.

Eine zentrale Ursache für die Schwierigkeiten beim unüberwachten Lernen ist die hohe Dimensionalität der Zufallsvariablen, die modelliert werden. Dadurch kommt es zu zwei wesentlichen Herausforderungen, und zwar zu einer statistischen und einer rechnerischen. Die *statistische Herausforderung* betrifft die Generalisierung: Die Anzahl der Konfigurationen, die wir unterscheiden möchten, kann exponentiell mit der Anzahl der betrachteten Dimensionen wachsen, die schnell größer als die Anzahl der Beispiele sein kann, die wir zusammenträgen (oder mit begrenzten Berechnungsressourcen verwenden) können. Die *rechnerische Herausforderung* im Zusammenhang mit hochdimensionalen Verteilungen entsteht, da viele Algorithmen für das Erlernen oder Verwenden eines trainierten Modells (insbesondere solche, die eine explizite Wahrscheinlichkeitsfunktion abschätzen) nicht effizient lösbar (engl. *intractable*) Berechnungen enthalten, die mit der Anzahl der Dimensionen exponentiell zunehmen.

Bei probabilistischen Modellen entsteht diese rechnerische Herausforderung aus der Notwendigkeit, eine nicht effizient berechenbare Inferenz durchzuführen oder die Verteilung zu normalisieren.

- *Nicht effizient durchführbare Inferenz*: Das Thema Inferenz wird hauptsächlich in Kapitel 19 behandelt. Es geht dabei um das Erraten der wahrscheinlichen Werte bestimmter Variablen a (aufgrund von anderen Variablen b) für ein Modell, das die multivariate Verteilung über a , b und c erfasst. Um solche bedingten Wahrscheinlichkeiten auch zu berechnen, muss man sowohl die Summe über die Werte der Variablen c bilden als auch eine Normalisierungskonstante berechnen, die die Summe über die Werte von a und c bildet.
- *Nicht effizient berechenbare Normalisierungskonstanten (die Partitionsfunktion)*: Die Partitionsfunktion wird hauptsächlich in Kapitel 18 behandelt. Das Normalisieren von Konstanten von Wahrscheinlichkeitsfunktionen ist in der Inferenz (oben) und beim Lernen von Bedeutung. Viele probabilistische Modelle enthalten eine solche normalisierende Konstante. Leider erfordert das Erlernen eines solchen Modells häufig das Berechnen des Gradienten des Logarithmus der Partitionsfunktion bezüglich der Modellparameter. Diese Berechnung ist allgemein ebenso wenig effizient durchführbar wie das Berechnen der Partitionsfunktion selbst. MCMC-Verfahren (Monte-Carlo-Markow-Ketten-Verfahren, Kapitel 17) werden häufig verwendet, mit der Partitionsfunktion umgehen zu können (Berechnen der Funktion oder ihres Gradienten). Leider lässt der Nutzen von MCMC-Verfahren nach, wenn die Modi der Modellverteilung vielfältig und gut separiert sind, insbesondere in hochdimensionalen Räumen (Abschnitt 17.5).

Eine Möglichkeit zum Umgang mit diesen schwer lösbareren Berechnungen besteht darin, sie zu approximieren; in diesem dritten Teil besprechen wir viele dafür vorgeschlagene Ansätze. Ein weiterer hier dargestellter Weg würde darin bestehen, diese nicht effizient durchführbaren Berechnungen von Anfang an zu vermeiden, sodass man gerne Verfahren verwendet, die ohne derartige Berechnungen auskommen. In den letzten Jahren wurden dafür mehrere generative Modelle vorgestellt. Eine Vielzahl aktueller Ansätze zur generativen Modellierung wird in Kapitel 20 behandelt.

Teil III dürfte dann besonders wichtig für Sie sein, wenn Sie in der Forschung tätig sind und die vielen Möglichkeiten, die sich im Deep Learning bieten, und das Fachgebiet für eine effektive Künstliche Intelligenz weiter voranbringen möchten.

13

Lineare Faktorenmodelle

Viele der Grenzen bei der Forschung im Deep Learning bestehen darin, ein probabilistisches Modell der Eingabe, $p_{\text{model}}(\mathbf{x})$ zu erstellen. Ein solches Modell kann prinzipiell probabilistische Inferenz nutzen, um beliebige Variablen in seiner Umgebung anhand beliebiger anderer Variablen vorherzusagen. Viele dieser Modelle enthalten auch latente Variablen \mathbf{h} , für die gilt $p_{\text{model}}(\mathbf{x}) = \mathbb{E}_{\mathbf{h}} p_{\text{model}}(\mathbf{x} | \mathbf{h})$. Diese latenten Variablen sind eine weitere Möglichkeit zur Repräsentation der Daten. Verteilte Repräsentationen auf Basis latenter Variablen können alle Vorteile des Representation Learnings aufweisen, die wir in Verbindung mit tiefen Feedforward-Netzen und RNNs kennengelernt haben.

In diesem Kapitel beschreiben wir einige sehr einfache probabilistische Modelle mit latenten Variablen, die linearen Faktorenmodelle. Diese Modelle werden manchmal als Bausteine von Mischmodellen (*Hinton et al.*, 1995a; *Ghahramani und Hinton*, 1996; *Roweis et al.*, 2002) oder größeren tiefen probabilistischen Modellen (*Tang et al.*, 2012) bezeichnet. Sie weisen auch viele der grundlegenden Ansätze auf, die zum Erstellen generativer Modelle erforderlich sind, die durch fortschrittlichere tiefe Modelle noch erweitert werden.

Ein lineares Faktorenmodell ist gekennzeichnet durch den Einsatz einer stochastischen linearen Decoder-Funktion, die \mathbf{x} durch Hinzufügen von Rauschen zu einer linearen Transformation von \mathbf{h} erzeugt.

Diese Modelle sind deshalb interessant, weil sie uns die Möglichkeit geben, erklärende Faktoren zu entdecken, die eine einfache multivariate Verteilung aufweisen. Die Einfachheit eines linearen Decoders machte diese Modelle zu den ersten Modellen mit latenten Variablen, die umfassend untersucht wurden.

Ein lineares Faktorenmodell beschreibt den datengenerierenden Prozess wie folgt: Zunächst ziehen wir Stichproben der erklärenden Faktoren \mathbf{h} aus einer Verteilung

$$\mathbf{h} \sim p(\mathbf{h}), \quad (13.1)$$

wobei $p(\mathbf{h})$ eine faktorielle Verteilung mit $p(\mathbf{h}) = \prod_i p(h_i)$ ist, sodass Stichproben leicht gezogen werden können. Anschließend ziehen wir die reellwertigen, beobachtbaren Variablen als Stichproben mit den Faktoren

$$\mathbf{x} = \mathbf{W}\mathbf{h} + \mathbf{b} + \text{Rauschen}, \quad (13.2)$$

wobei das Rauschen meist normalverteilt und diagonal (also dimensionsübergreifend unabhängig) ist. Das wird in Abbildung 13.1 dargestellt.

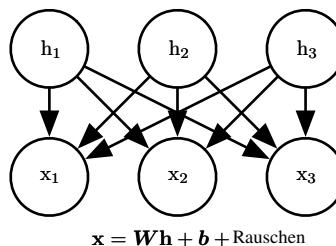


Abbildung 13.1: Das gerichtete graphische Modell, das die Familie von linearen Faktorenmodellen beschreibt. Wir nehmen an, dass ein beobachteter Datenvektor \mathbf{x} mithilfe einer Linearkombination der unabhängigen latenten Faktoren \mathbf{h} plus etwas Rauschen ermittelt wird. Unterschiedliche Modelle wie die probabilistische PCA, die Faktorenanalyse oder die Unabhängigkeitsanalyse treffen verschiedene Annahmen über die Form des Rauschens und die A-priori-Wahrscheinlichkeit $p(\mathbf{h})$.

13.1 Probabilistische PCA und Faktorenanalyse

Die probabilistische Hauptkomponentenanalyse (engl. *principal components analysis*, PCA), die Faktorenanalyse und andere lineare Faktorenmodelle sind Sonderfälle der obigen Gleichungen (13.1 und 13.2) und unterscheiden sich nur hinsichtlich der Auswahl der Rauschverteilung und der A-priori-Wahrscheinlichkeit des Modells über die latenten Variablen \mathbf{h} vor der Beobachtung von \mathbf{x} .

In der **Faktorenanalyse** (Bartholomew, 1987; Basilevsky, 1994) ist die Wahrscheinlichkeitsverteilung der latenten Variable gerade die Normalverteilung mit der Einheitsmatrix als Varianz

$$\mathbf{h} \sim \mathcal{N}(\mathbf{h}; \mathbf{0}, \mathbf{I}), \quad (13.3)$$

wobei für die beobachteten Variablen x_i eine **bedingte Unabhängigkeit** angenommen wird (aus \mathbf{h}). Insbesondere wird angenommen, dass das Rauschen aus einer Normalverteilung mit diagonaler Kovarianzmatrix mit der Kovarianzmatrix $\boldsymbol{\psi} = \text{diag}(\boldsymbol{\sigma}^2)$ stammt, mit $\boldsymbol{\sigma}^2 = [\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2]^\top$ als Vektor der Varianzen vor der Variable.

Die Rolle der latenten Variablen ist somit das *Erfassen der Abhängigkeiten* zwischen den unterschiedlichen beobachteten Variablen x_i . Tatsächlich lässt sich problemlos zeigen, dass \mathbf{x} lediglich eine mehrdimensionale normale Zufallsvariable ist, mit

$$\mathbf{x} \sim \mathcal{N}(\mathbf{x}; \mathbf{b}, \mathbf{W}\mathbf{W}^\top + \boldsymbol{\psi}). \quad (13.4)$$

Um die PCA in einen probabilistischen Rahmen zu fassen, können wir das Modell der Faktorenanalyse geringfügig ändern und die bedingten Varianzen σ_i^2 gleichsetzen. In diesem Fall ist die Kovarianz von \mathbf{x} einfach $\mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I}$, wobei σ^2 nun ein Skalar ist. Das führt zur bedingten Verteilung

$$\mathbf{x} \sim \mathcal{N}(\mathbf{x}; \mathbf{b}, \mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I}), \quad (13.5)$$

oder gleichwertig

$$\mathbf{x} = \mathbf{W}\mathbf{h} + \mathbf{b} + \sigma\mathbf{z}, \quad (13.6)$$

mit $\mathbf{z} \sim \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$ als normalverteiltem Rauschen. Wie *Tipping und Bishop* (1999) zeigen, können wir anschließend einen iterativen EM-Algorithmus verwenden, um die Parameter \mathbf{W} und σ^2 zu schätzen.

Dieses **probabilistische** PCA-Modell nutzt die Beobachtung, dass die meisten Variationen in den Daten von den latenten Variablen \mathbf{h} erfasst werden können, bis auf einen kleinen Rest-**Rekonstruktionsfehler** σ^2 . Wie *Tipping und Bishop* (1999) zeigen, wird die probabilistische PCA für $\sigma \rightarrow 0$ zur PCA. In diesem Fall wird der bedingte Erwartungswert für \mathbf{h} auf Basis von \mathbf{x} zu einer orthogonalen Projektion von $\mathbf{x} - \mathbf{b}$ auf dem Raum, der von den d -Spalten von \mathbf{W} aufgespannt wird, wie in der PCA.

Da $\sigma \rightarrow 0$, wird das durch die probabilistische PCA definierte Dichtemodell im Bereich dieser von den Spalten von \mathbf{W} aufgespannten d -Dimensionen sehr scharf. Das kann dazu führen, dass das Modell den Daten eine sehr geringe Likelihood zuweist, wenn diese sich nicht in der Nähe einer Hyperebene versammeln.

13.2 Unabhängigkeitsanalyse

Die Unabhängigkeitsanalyse (engl. *independent component analysis*, Die Unabhängigkeitsanalyse) gehört zu den ältesten Representation-Learning-

Algorithmen (*Herault und Ans*, 1984; *Jutten und Herault*, 1991; *Comon*, 1994; *Hyvärinen*, 1999; *Hyvärinen et al.*, 2001a; *Hinton et al.*, 2001; *Teh et al.*, 2003). Sie dient dem Modellieren von Linearfaktoren und versucht, ein beobachtetes Signal in viele zugrunde liegende Signale aufzuteilen, die dann skaliert und addiert werden, um die beobachteten Daten zu bilden. Diese Signale sollen nicht nur voneinander dekorreliert, sondern vollständig unabhängig sein.¹

Viele unterschiedliche Einzelmethoden werden als Unabhängigkeitsanalyse bezeichnet. Die Variante, die den anderen hier beschriebenen generativen Modellen am ähnlichsten ist, trainiert ein vollständig parametrisches generatives Modell (*Pham et al.*, 1992). Die A-priori-Verteilung über die zugrunde liegenden Faktoren, $p(\mathbf{h})$, muss im Voraus durch den Anwender festgelegt werden. Das Modell erzeugt dann deterministisch $\mathbf{x} = \mathbf{W}\mathbf{h}$. Wir können eine nichtlineare Änderung der Variablen vornehmen (mittels Gleichung 3.47), um $p(\mathbf{x})$ zu bestimmen. Das Erlernen des Modells erfolgt dann wie gewohnt mithilfe der Maximum Likelihood.

Die Idee dahinter ist, dass wir dadurch, dass wir $p(\mathbf{h})$ als unabhängig wählen, zugrunde liegende Faktoren wiederherstellen können, die so weit wie möglich unabhängig sind. Dieser Ansatz wird häufig verwendet – nicht zum Erfassen hochrangiger abstrakter kausaler Faktoren, sondern zum Wiederherstellen niederrangiger Signale, die miteinander vermischt wurden. In diesem Fall stellt jedes Trainingsbeispiel einen bestimmten Zeitpunkt dar, jedes x_i die Beobachtung der gemischten Signale durch einen bestimmten Sensor und jedes h_i ein Schätzwert eines der ursprünglichen Signale. Zum Beispiel könnten n Personen gleichzeitig reden. Wenn wir n verschiedene Mikrofone an unterschiedlichen Stellen platzieren, kann die Unabhängigkeitsanalyse die Änderungen in der Lautstärke zwischen den einzelnen Sprechern in jedem der Mikrofone erkennen und die Signale voneinander trennen, sodass jedes h_i nur die klare Stimme einer Person enthält. Dieses Konzept ist in der Neurowissenschaft für die Elektroenzephalografie (EEG) üblich – die Technologie zeichnet elektrische Signale aus dem Gehirn auf. Mehrere am Kopf der Versuchsperson angebrachte Elektroden messen alle elektrischen Signale des Körpers. Der Versuchsleiter interessiert sich meist nur für die Signale, die vom Gehirn stammen. Allerdings sind die Signale von Herz und Augen der Versuchsperson stark genug, um sich in den Messungen am Schädel niederzuschlagen. Die an den Elektroden eingehenden Signale sind also miteinander vermischt und müssen per Unabhängigkeitsanalyse getrennt werden, und zwar zunächst in

¹ Siehe Abschnitt 3.8 für den Unterschied zwischen nicht korrelierten und unabhängigen Variablen.

Herz- und Hirnsignale und anschließend in Signale aus den unterschiedlichen Hirnregionen.

Wie bereits erwähnt, sind viele Varianten der Unabhängigkeitsanalyse denkbar. Einige verwenden keinen deterministischen Decoder, sondern fügen Rauschen zur Erzeugung von \mathbf{x} hinzu. Die meisten verwenden das Kriterium der Maximum Likelihood nicht, sondern versuchen, die Elemente von $\mathbf{h} = \mathbf{W}^{-1}\mathbf{x}$ unabhängig voneinander zu machen. Es gibt viele Kriterien, die dieses Ziel erreichen. Gleichung 3.47 arbeitet mit der Determinante von \mathbf{W} , eine möglicherweise aufwendige und rechnerisch instabile Operation. Einige Varianten der Unabhängigkeitsanalyse vermeiden diese problembehaftete Operation, indem sie \mathbf{W} auf Orthogonalität beschränken.

Für alle Varianten der Unabhängigkeitsanalyse muss $p(\mathbf{h})$ nicht normalverteilt sein. Das liegt daran, dass \mathbf{W} nicht identifizierbar ist, wenn $p(\mathbf{h})$ eine unabhängige A-priori-Wahrscheinlichkeit mit normalverteilten Komponenten ist. Wir können dieselbe Verteilung über $p(\mathbf{x})$ für viele Werte von \mathbf{W} erhalten. Das ist ein großer Unterschied zu anderen linearen Faktorenmodellen wie der probabilistischen PCA und der Faktorenanalyse, in denen $p(\mathbf{h})$ häufig normalverteilt sein muss, damit viele Operationen am Modell geschlossene Lösungen (engl. *closed form solutions*) aufweisen. Beim Maximum-Likelihood-Ansatz, für den der Anwender die Verteilung explizit vorgibt, ist $p(h_i) = \frac{d}{dh_i}\sigma(h_i)$ eine typische Wahl. Übliche Vertreter dieser Nicht-Normalverteilungen haben größere Gipfel in der Nähe von 0 als Normalverteilungen, sodass die meisten Implementierungen der Unabhängigkeitsanalyse auch dünnbesetzte Merkmale erlernen können.

Viele Varianten der Unabhängigkeitsanalyse sind keine generativen Modelle im üblichen Sinne des Begriffs. In diesem Buch repräsentiert ein generatives Modell entweder $p(\mathbf{x})$ oder kann Stichproben daraus ziehen. Viele Varianten der Unabhängigkeitsanalyse können nur zwischen \mathbf{x} und \mathbf{h} transformieren, haben aber keine Möglichkeit, $p(\mathbf{h})$ zu repräsentieren, sodass sie keine Verteilung über $p(\mathbf{x})$ festlegen. Zum Beispiel haben viele Varianten der Unabhängigkeitsanalyse das Ziel, die Kurtosis einer Stichprobe $\mathbf{h} = \mathbf{W}^{-1}\mathbf{x}$ zu erhöhen, da eine hohe Kurtosis angibt, dass $p(\mathbf{h})$ nicht normalverteilt ist; allerdings wird dies ohne explizite Repräsentation von $p(\mathbf{h})$ erreicht. Das liegt daran, dass die Unabhängigkeitsanalyse häufiger als Analysetool zum Trennen von Signalen verwendet wird, und nicht zum Erzeugen von Daten oder Schätzen ihrer Dichte.

Wie die PCA für nichtlineare Autoencoder generalisiert werden kann (siehe Kapitel 14), kann die Unabhängigkeitsanalyse für ein nichtlineares generatives Modell generalisiert werden, in dem wir eine nichtlineare

Funktion f zum Erzeugen der beobachteten Daten verwenden. *Hyvärinen und Pajunen* (1999) enthält die erste Arbeit zur nichtlinearen Unabhängigkeitsanalyse und einem erfolgreichen Einsatz beim Ensemble Learning durch *Roberts und Everson* (2001) und *Lappalainen et al.* (2000). Eine andere nichtlineare Erweiterung der Unabhängigkeitsanalyse ist der Ansatz der **nichtlinearen Unabhängigkeitsschätzung** (engl. *nonlinear independent components estimation*, NICE) (*Dinh et al.*, 2014), bei der eine Reihe umkehrbarer Transformationen (Encoder-Stufen) gestapelt werden, die die Eigenschaft aufweisen, dass die Determinante der Jacobi-Matrix jeder Transformation effizient berechnet werden kann. Das ermöglicht die exakte Berechnung der Likelihood. Wie die Unabhängigkeitsanalyse zielt auch die nichtlineare Unabhängigkeitsschätzung darauf ab, die Daten in einen Raum zu transformieren, in dem eine faktorierte Randverteilung dafür existiert. Allerdings ist dabei die Erfolgswahrscheinlichkeit aufgrund des nichtlinearen Encoders höher. Da der Encoder mit einem vollständig inversen Decoder verknüpft ist, gelingt die Stichprobengenerierung aus dem Modell ganz einfach (durch Stichprobenentnahme aus $p(\mathbf{h})$ mit nachfolgender Anwendung des Decoders).

Eine weitere Generalisierung der Unabhängigkeitsanalyse ist das Erlernen von Merkmalsgruppen. Dabei ist eine statistische Abhängigkeit innerhalb einer Gruppe erlaubt, allerdings nicht zwischen den Gruppen (*Hyvärinen und Hoyer*, 1999; *Hyvärinen et al.*, 2001b). Wenn die Gruppen verwandter Einheiten ohne Überschneidung gewählt werden, trägt dieses Verfahren den Namen **Independent Subspace Analysis** (dt. *unabhängige Unterraumanalyse*). Es ist auch möglich, jeder verdeckten Einheit räumliche Koordinaten zuzuweisen und einander überschneidende Gruppen räumlich benachbarter Einheiten zu bilden. Das animiert benachbarte Einheiten dazu, ähnliche Merkmale zu erlernen. Wird diese **topografische Unabhängigkeitsanalyse** auf natürliche Bilder angewandt, lernt sie Gaborfilter, wie zum Beispiel dass benachbarte Merkmale eine ähnliche Ausrichtung, Position oder Häufigkeit aufweisen. Viele unterschiedliche Phasenversätze ähnlicher Gaborfunktionen tauchen in jedem Bereich auf, sodass ein Pooling über kleine Bereiche zu einer Invarianz gegenüber Verschiebungen führt.

13.3 Slow Feature Analysis

Die **Slow Feature Analysis** (SFA) ist ein lineares Faktorenmodell, das Informationen aus Zeitsignalen zum Erlernen invarianter Merkmale nutzt (*Wiskott und Sejnowski, 2002*).

Sie beruht auf einem allgemeinen Prinzip, dem Prinzip der Langsamkeit (engl. *slowness principle*). Die Idee dahinter ist, dass die wichtigen Eigenschaften der Szenen sich vergleichen zu den einzelnen Messungen, die die Beschreibung einer Szene bilden, nur sehr langsam verändern. Ein Beispiel: In der Computer Vision können sich die Werte einzelner Pixel sehr schnell ändern. Wenn ein Zebra von links nach rechts durch das Bild läuft, wechselt ein einzelnes Pixel schnell von Schwarz zu Weiß und wieder zurück, da die Zebrastreifen abwechselnd über dem Pixel liegen. Dagegen ändert sich das Merkmal, das anzeigt, ob sich ein Zebra im Bild befindet, überhaupt nicht, und das Merkmal, das die Position des Zebras im Bild angibt, ändert sich nur sehr langsam. Wir möchten unser Modell also so regularisieren, dass es Merkmale erlernt, die sich im Laufe der Zeit langsam verändern.

Das Prinzip der Langsamkeit ist älter als die Slow Feature Analysis und wurde auf eine Vielzahl von Modellen angewandt (*Hinton, 1989; Földiák, 1989; Mobahi et al., 2009; Bergstra und Bengio, 2009*). Generell können wir das Prinzip der Langsamkeit auf jedes differenzierbare Modell anwenden, das mit dem Gradientenabstiegsverfahren trainiert wird. Dazu können wir die Kostenfunktion um einen Term der Form

$$\lambda \sum_t L(f(\mathbf{x}^{(t+1)}), f(\mathbf{x}^{(t)})) \quad (13.7)$$

ergänzen, wobei λ ein Hyperparameter ist, der die Stärke des Langsamkeits-Regularisierungsterms (engl. *slowness regularization term*) bestimmt, t der Index einer zeitlichen Abfolge von Beispielen, f der Merkmalsextraktor, der regularisiert werden soll, und L eine Verlustfunktion, die den Abstand zwischen $f(\mathbf{x}^{(t)})$ und $f(\mathbf{x}^{(t+1)})$ misst. Häufig wird für L das mittlere Quadrat der Differenz gewählt.

Die Slow Feature Analysis ist eine besonders effiziente Umsetzung des Prinzips der Langsamkeit. Sie ist effizient, weil sie auf einen linearen Merkmalsextraktor angewandt wird und daher in geschlossener Form trainiert werden kann. Wie einige Varianten der Unabhängigkeitsanalyse entspricht die SFA per se nicht vollständig einem generativen Modell, da sie eine lineare Zuordnung zwischen Eingaberaum und Merkmalsraum definiert, aber keine A-priori-Wahrscheinlichkeitsverteilung über den Merkmalsraum und somit keine Verteilung $p(\mathbf{x})$ im Eingaberaum erzwingt.

Der SFA-Algorithmus (*Wiskott und Sejnowski, 2002*) besteht aus der Definition von $f(\mathbf{x}; \theta)$ als eine lineare Transformation und anschließendem Lösen des Optimierungsproblems

$$\min_{\theta} \mathbb{E}_t (f(\mathbf{x}^{(t+1)})_i - f(\mathbf{x}^{(t)})_i)^2 \quad (13.8)$$

unter Berücksichtigung der Bedingungen

$$\mathbb{E}_t f(\mathbf{x}^{(t)})_i = 0 \quad (13.9)$$

und

$$\mathbb{E}_t [f(\mathbf{x}^{(t)})_i^2] = 1. \quad (13.10)$$

Die Bedingung, dass der Erwartungswert des erlernten Merkmals gleich Null sein muss, ist erforderlich, damit das Problem eine eindeutige Lösung aufweist – ansonsten könnten wir eine Konstante zu allen Merkmalswerten addieren und unterschiedliche Lösungen für gleiche Werte der Zielfunktion der Langsamkeit (engl. *slowness objective*) erhalten. Die Bedingung, dass die Merkmale eine Einheitsvarianz aufweisen müssen, ist erforderlich, um die pathologische Lösung zu verhindern, bei der alle Merkmale 0 werden. Wie bei der PCA sind auch die Merkmale der SFA geordnet, wobei das erste Merkmal das langsamste ist. Um mehrere Merkmale zu erlernen, müssen wir auch die folgende Bedingung verwenden:

$$\forall i < j, \mathbb{E}_t [f(\mathbf{x}^{(t)})_i f(\mathbf{x}^{(t)})_j] = 0. \quad (13.11)$$

Sie gibt an, dass die erlernten Merkmale zueinander linear dekorreliert sein müssen. Ohne diese Bedingung würden alle erlernten Merkmale einfach das eine langsamste Signal erfassen. Auch andere Mechanismen sind vorstellbar, zum Beispiel das Minimieren des Rekonstruktionsfehlers zum Erzwingen diversifizierter Merkmale, aber dieser Dekorrelationsmechanismus bietet aufgrund der Linearität der SFA-Merkmale eine einfache Lösung. Das SFA-Problem kann mit linearer Algebra in geschlossener Form gelöst werden.

Die SFA wird gern zum Erlernen nichtlinearer Merkmale verwendet, indem eine nichtlineare Basisexpansion (auch Basiserweiterung genannt) zu \mathbf{x} vor der SFA erfolgt. Es ist beispielsweise üblich, \mathbf{x} durch die quadratische Basisexpansion zu ersetzen, einen Vektor, der die Elemente $x_i x_j$ für alle i und j enthält. Anschließend können lineare SFA-Module zusammengesetzt werden, um tiefe nichtlineare Slow-Feature-Extraktoren zu erlernen; dabei wird wiederholt ein linearer SFA-Merkalsextraktor erlernt, eine nichtlineare Basisexpansion auf dessen Ausgabe angewandt und anschließend ein weiterer linearer SFA-Merkalsextraktor für diese Expansion erlernt.

Wird die SFA mit quadratischen Basisexpansionen anhand kleiner räumlicher Bereiche von Videos mit natürlichen Szenen trainiert, erlernt sie Merkmale, die viele Eigenschaften mit den komplexen Zellen im V1-Kortex teilen (*Berkes und Wiskott, 2005*). Wird eine tiefe SFA anhand von Videos mit zufälligen Bewegungen in computergerenderten 3-D-Umgebungen trainiert, erlernt sie Merkmale, die viele Eigenschaften mit den Merkmalen teilen, die durch Neuronen in Rattengehirnen repräsentiert werden, die für die Navigation verwendet werden (*Franzius et al., 2007*). Somit scheint die SFA ein biologisch angemessen plausibles Modell zu sein.

Ein großer Vorteil der SFA ist, dass theoretisch vorhergesagt werden kann, welche Merkmale sie erlernen wird – auch in tiefen nichtlinearen Fällen. Um derart theoretische Vorhersagen zu treffen, muss man die Dynamik der Umgebung bezüglich des Konfigurationsraums kennen (z. B. basiert die theoretische Analyse im Fall der zufälligen Bewegungen in einer gerenderten 3-D-Umgebung auf der Kenntnis der Wahrscheinlichkeitsverteilung über Position und Geschwindigkeit der Kamera). Wenn wir wissen, wie sich die zugrunde liegenden Faktoren tatsächlich ändern, können wir die optimalen Funktionen, die diese Faktoren ausdrücken, analytisch bestimmen. In der Praxis scheinen Experimente mit tiefer SFA für simulierte Daten die theoretisch vorhergesagten Funktionen wiederherzustellen. Bei anderen Lernalgorithmen, deren Kostenfunktion stark von einzelnen Pixelwerten abhängt, ist es viel schwieriger zu bestimmen, welche Merkmale das Modell erlernen wird.

Die tiefe SFA wurde auch zum Erlernen von Merkmalen für die Objekterkennung und Pose Estimation (Ermittlung der Pose) verwendet (*Franzius et al., 2008*). Bisher dient das Prinzip der Langsamkeit noch nicht als Basis für eine moderne Anwendung. Es ist nicht sicher, welcher Faktor seine Leistungsfähigkeit eingeschränkt hat. Wir vermuten, dass die A-priori-Wahrscheinlichkeit der Langsamkeit (engl. *slowness prior*) möglicherweise zu stark ist und dass es besser wäre, auf die Annahme, dass die Merkmale näherungsweise konstant sein sollten, zu verzichten, sondern stattdessen davon auszugehen, dass die Merkmale von einem Zeitschritt zum nächsten leicht vorherzusagen sein sollten. Die Position eines Objekts ist ein nützliches Merkmal – ob die Objektgeschwindigkeit nun hoch oder gering ist –, aber das Prinzip der Langsamkeit führt dazu, dass das Modell die Position von Objekten mit hoher Geschwindigkeit eher ignoriert.

13.4 Sparse Coding

Sparse Coding (*Olshausen und Field*, 1996) ist ein lineares Faktorenmodell, das als Mechanismus für das unüberwachte Lernen von Merkmalen (engl. *unsupervised feature learning*) und die Merkmalsextraktion gründlich untersucht wurde. Streng genommen bezieht sich der Begriff »Sparse Coding« auf den Vorgang, bei dem auf den Wert von \mathbf{h} in diesem Modell geschlossen wird, während »Sparse Modeling« sich auf Entwurf und Erlernen des Modells bezieht – dennoch wird »Sparse Coding« häufig für beides verwendet.

Wie bei den meisten linearen Faktorenmodellen kommt auch hier ein linearer Decoder mit Rauschen zum Einsatz, um Rekonstruktionen von \mathbf{x} zu erhalten (Gleichung 13.2). Genauer ausgedrückt gehen Modelle mit Sparse Coding häufig davon aus, dass die linearen Faktoren normalverteiltes Rauschen mit isotroper Präzision β aufweisen:

$$p(\mathbf{x} | \mathbf{h}) = \mathcal{N}(\mathbf{x}; \mathbf{W}\mathbf{h} + \mathbf{b}, \frac{1}{\beta}\mathbf{I}). \quad (13.12)$$

Die Verteilung $p(\mathbf{h})$ wird so gewählt, dass sie nahe 0 mehrfach steil ansteigt und wieder abfällt (*Olshausen und Field*, 1996). Üblich sind faktorierte Laplace-, Cauchy- oder faktorierte studentsche t -Verteilungen. Ein Beispiel: Die Laplace-a-priori-Wahrscheinlichkeit für den Strafkoeffizienten λ für die dünne Besetzung ergibt sich aus

$$p(h_i) = \text{Laplace}(h_i; 0, \frac{2}{\lambda}) = \frac{\lambda}{4} e^{-\frac{1}{2}\lambda|h_i|}, \quad (13.13)$$

und die studentsche t -a-priori-Wahrscheinlichkeit aus

$$p(h_i) \propto \frac{1}{(1 + \frac{h_i^2}{\nu})^{\frac{\nu+1}{2}}}. \quad (13.14)$$

Das Trainieren von Sparse Coding mit Maximum Likelihood ist nicht effizient durchführbar. Stattdessen wechselt das Training zwischen dem Codieren der Daten und dem Trainieren des Decoders, um die Daten auf Basis des Codierens besser rekonstruieren zu können. Dieser Ansatz wird in Abschnitt 19.3 als grundlegende Approximation der Maximum Likelihood näher begründet.

Bei Modellen wie der PCA haben Sie den Einsatz einer parametrischen Encoder-Funktion kennengelernt, die \mathbf{h} vorhersagt und nur die Multiplikation mit einer Gewichtungsmatrix beinhaltet. Der Encoder, den wir für

Sparse Coding nutzen, ist kein parametrischer Encoder. Vielmehr handelt es sich um einen Optimierungsalgorithmus, der ein Optimierungsproblem löst, indem wir den einen wahrscheinlichsten Code-Wert suchen:

$$\mathbf{h}^* = f(\mathbf{x}) = \arg \max_{\mathbf{h}} p(\mathbf{h} | \mathbf{x}). \quad (13.15)$$

In Kombination mit Gleichung 13.13 und Gleichung 13.12 führt dies zu folgendem Optimierungsproblem:

$$\arg \max_{\mathbf{h}} p(\mathbf{h} | \mathbf{x}) \quad (13.16)$$

$$= \arg \max_{\mathbf{h}} \log p(\mathbf{h} | \mathbf{x}) \quad (13.17)$$

$$= \arg \min_{\mathbf{h}} \lambda \|\mathbf{h}\|_1 + \beta \|\mathbf{x} - \mathbf{W}\mathbf{h}\|_2^2, \quad (13.18)$$

wobei wir nicht von \mathbf{h} abhängige Terme entfernt und eine Division durch positive Skalierungsfaktoren vorgenommen haben, um die Gleichung zu vereinfachen.

Aufgrund der Auferlegung einer L^1 -Norm auf \mathbf{h} führt dieses Verfahren zu einem dünnbesetzten \mathbf{h}^* (siehe Abschnitt 7.1.2).

Um das Modell zu trainieren und nicht nur Inferenz durchzuführen, wechseln wir zwischen einer Minimierung bezüglich \mathbf{h} und einer Minimierung bezüglich \mathbf{W} hin und her. In dieser Darstellung behandeln wir β als Hyperparameter. Üblicherweise wird er auf 1 gesetzt, da seine Rolle in diesem Optimierungsproblem von λ geteilt wird und es keinen Grund dafür gibt, beide Hyperparameter zu verwenden. Prinzipiell könnten wir auch β als Parameter des Modells behandeln und erlernen. Unsere Darstellung hier hat einige Terme verworfen, die nicht von \mathbf{h} , sondern von β abhängen. Um β zu erlernen, müssen diese Terme eingeschlossen werden – ansonsten kollabiert β zu 0.

Nicht alle Ansätze für Sparse Coding erstellen explizit $p(\mathbf{h})$ und $p(\mathbf{x} | \mathbf{h})$. Häufig möchten wir lediglich ein Dictionary von Merkmalen mit Aktivierungswerten erlernen, die beim Extrahieren mithilfe dieses Inferenzverfahrens oft Null sind.

Wenn wir \mathbf{h} als Stichprobe aus einer Laplace-a-priori-Wahrscheinlichkeit ziehen, ist die Wahrscheinlichkeit, dass ein Element aus \mathbf{h} wirklich Null ist, tatsächlich ein Ereignis der Wahrscheinlichkeit Null. Das generative Modell selbst ist nicht sonderlich dünnbesetzt – nur der Merkmalsextraktor ist dünnbesetzt. *Goodfellow et al.* (2013d) beschreiben die approximative Inferenz für eine andere Modellfamilie, nämlich das Spike-and-Slab-Sparse-

Coding-Modell, dessen Stichproben aus der A-priori-Wahrscheinlichkeit meist echte Nullen enthalten.

Sparse Coding kann in Verbindung mit einem nichtparametrischen Encoder prinzipiell die Kombination aus Rekonstruktionsfehler und Log-a-priori-Wahrscheinlichkeit besser minimieren als jeder spezifische parametrische Encoder. Ein weiterer Vorteil ist das Fehlen eines Generalisierungsfehlers für den Encoder. Ein parametrischer Encoder muss lernen, wie \mathbf{x} auf generalisierungsfähige Weise \mathbf{h} zugeordnet wird. Für unübliche \mathbf{x} , die den Trainingsdaten nicht ähneln, findet ein erlernerter parametrischer Encoder möglicherweise kein \mathbf{h} , das zu einer exakten Rekonstruktion oder Sparse Coding führt. Bei den meisten Modellen mit Sparse Coding, in denen das Inferenzproblem konvex ist, findet das Optimierungsverfahren stets den optimalen Code (es sei denn, es kommt zu einem degenerierten Fall, wie replizierten Gewichtungsvektoren). Offensichtlich können die dünne Besetzung und der Aufwand für die Rekonstruktion an unbekannten Punkten noch immer steigen, was aber dem Generalisierungsfehler in den Decoder-Gewichten geschuldet ist, nicht dem Generalisierungsfehler im Encoder. Das Fehlen eines Generalisierungsfehlers bei der Codierung auf Optimierungsbasis im Rahmen von Sparse Coding kann zu einer besseren Generalisierung führen, wenn Sparse Coding als Merkmalsextraktor für einen Klassifikator eingesetzt und auf eine parametrische Funktion zum Vorhersagen des Codes verzichtet wird. *Coates und Ng* (2011) haben gezeigt, dass Merkmale mit Sparse Coding bei Objekterkennungsaufgaben besser generalisieren als Merkmale verwandter Modelle, die auf einem parametrischen Encoder basieren, den linear-sigmoiden Autoencoder. Von dieser Arbeit inspiriert haben *Goodfellow et al.* (2013d) gezeigt, dass eine Variante des Sparse Codings besser als andere Merkmalsextraktoren generalisiert, wenn extrem wenige Labels verfügbar sind (20 oder weniger Labels pro Klasse).

Der wesentliche Nachteil des nichtparametrischen Encoders besteht darin, dass die Berechnung von \mathbf{h} aus \mathbf{x} damit länger dauert, weil der nichtparametrische Ansatz einen iterativen Algorithmus ausführen muss. Der parametrische Autoencoder, der in Kapitel 14 entwickelt wird, verwendet nur eine feste Anzahl von Schichten – häufig sogar nur eine. Ein weiterer Nachteil ist die komplizierte Backpropagation durch den nichtparametrischen Encoder, wodurch ein Modell mit Sparse Coding nur schwer mit einem unüberwachten Kriterium trainiert und anschließend mit einem überwachten Kriterium feinabgestimmt werden kann. Es gibt zwar veränderte Versionen des Sparse Codings, die approximative Ableitungen erlauben, aber sie sind nicht sonderlich verbreitet (*Bagnell und Bradley*, 2009).

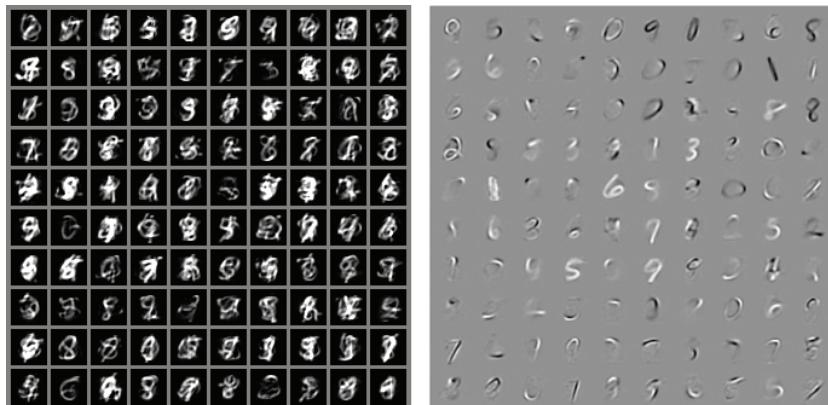


Abbildung 13.2: Beispielhafte Stichproben und Gewichte aus einem Modell mit Spike and Slab Sparse Coding, das anhand des MNIST-Datensatzes trainiert wurde. (*Links*) Die Stichproben aus dem Modell ähneln den Trainingsbeispielen nicht. Auf den ersten Blick könnte man annehmen, dass das Modell schlecht angepasst ist. (*Rechts*) Die Gewichtungsvektoren des Modells haben gelernt, die Stiftbewegungen und manchmal ganze Ziffern darzustellen. Das Modell hat also nützliche Merkmale erlernt. Das Problem liegt in der faktoriellen A-priori-Wahrscheinlichkeitsverteilung über die Merkmale, die dazu führt, dass zufällige Teilmengen der Merkmale miteinander kombiniert werden. Nur wenige dieser Teilmengen ermöglichen das Bilden einer erkennbaren MNIST-Ziffer. Das führt zur Entwicklung generativer Modelle, die leistungsstärkere Verteilungen über die latenten Codes aufweisen. (Abbildung mit freundlicher Genehmigung von *Goodfellow et al. (2013d)* wiedergegeben)

Sparse Coding führt, wie andere lineare Faktorenmödelle, häufig zu schlechten Stichproben (vgl. Abbildung 13.2). Das geschieht auch dann, wenn das Modell die Daten gut wiederherstellen kann und nützliche Merkmale für einen Klassifikator liefert. Das liegt daran, dass zwar jedes einzelne Merkmal gut erlernt werden kann, aber die faktorielle A-priori-Wahrscheinlichkeit für die verdeckte Repräsentation zu einem Modell führt, das zufällige Teilmengen aller Merkmale in jeder generierten Stichprobe enthält. Daher werden tiefere Modelle entwickelt, die eine nichtfaktorielle Verteilung für die tiefste Code-Schicht vorgeben können sowie zur Entwicklung ausgereifterer flacher Modelle.

13.5 Interpretation der Mannigfaltigkeit der PCA

Lineare Faktorenmodelle mit PCA und Faktorenanalyse erlernen quasi eine Mannigfaltigkeit (engl. *manifold*) (Hinton et al., 1997). Sie können sich die probabilistische PCA so vorstellen, als würde ein dünner Bereich – ähnlich einem Pfannkuchen – hoher Wahrscheinlichkeit definiert, eine Normalverteilung, die entlang einiger Achsen sehr schmal ist (wie ein Pfannkuchen in der Vertikalen sehr flach ist), in den anderen Achsrichtungen dagegen von länglicher Ausdehnung (wie ein Pfannkuchen in der Horizontalen). Abbildung 13.3 zeigt dies. Die PCA richtet nun diesen Pfannkuchen an einer linearen Mannigfaltigkeit in einem höherdimensionalen Raum aus. Diese Interpretation gilt nicht nur für die klassische PCA, sondern auch für jeden anderen linearen Autoencoder, der die Matrizen \mathbf{W} und \mathbf{V} lernt, damit die Rekonstruktion von \mathbf{x} möglichst dicht an \mathbf{x} liegt.

Der Encoder sei

$$\mathbf{h} = f(\mathbf{x}) = \mathbf{W}^\top(\mathbf{x} - \boldsymbol{\mu}). \quad (13.19)$$

Der Encoder berechnet eine geringdimensionale Repräsentation von \mathbf{h} . Aus Sicht des Autoencoders verfügen wir über einen Decoder, der die Rekonstruktion berechnet:

$$\hat{\mathbf{x}} = g(\mathbf{h}) = \mathbf{b} + \mathbf{V}\mathbf{h}. \quad (13.20)$$

Die Wahl des linearen Encoders und Decoders, die den Rekonstruktionsfehler

$$\mathbb{E}[||\mathbf{x} - \hat{\mathbf{x}}||^2] \quad (13.21)$$

minimiert, entspricht $\mathbf{V} = \mathbf{W}$, $\boldsymbol{\mu} = \mathbf{b} = \mathbb{E}[\mathbf{x}]$ und die Spalten von \mathbf{W} bilden eine orthonormale Basis, die denselben Unterraum aufspannt wie die Hauptvektoren der Kovarianzmatrix

$$\mathbf{C} = \mathbb{E}[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top]. \quad (13.22)$$

Im Falle der PCA sind die Spalten von \mathbf{W} diese Eigenvektoren, sortiert nach der Größe der zugehörigen Eigenwerte (die alle reell und nicht-negativ sind).

Es lässt sich auch zeigen, dass der Eigenwert λ_i von \mathbf{C} der Varianz von \mathbf{x} in Richtung des Eigenvektors $\mathbf{v}^{(i)}$ entspricht. Für $\mathbf{x} \in \mathbb{R}^D$ und $\mathbf{h} \in \mathbb{R}^d$ mit $d < D$ beträgt der optimale Rekonstruktionsfehler (bei Wahl von $\boldsymbol{\mu}$, \mathbf{b} , \mathbf{V} und \mathbf{W} wie oben)

$$\min \mathbb{E}[||\mathbf{x} - \hat{\mathbf{x}}||^2] = \sum_{i=d+1}^D \lambda_i. \quad (13.23)$$

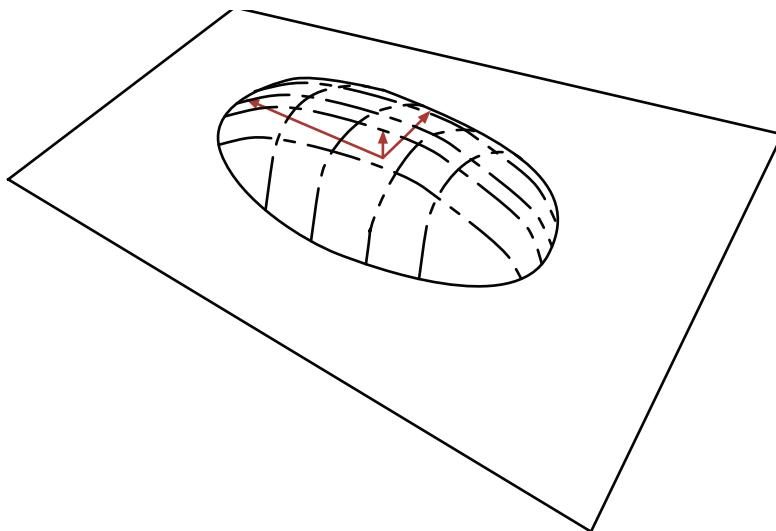


Abbildung 13.3: Flache Normalverteilung, die eine Wahrscheinlichkeitskonzentration in der Nähe einer geringdimensionalen Mannigfaltigkeit erfasst. Die Abbildung zeigt die obere Hälfte des »Pfannkuchens« über der »Mannigfaltigkeitsebene«, die durch die Mitte verläuft. Die Varianz in der Richtung orthogonal zur Mannigfaltigkeit ist sehr gering (Pfeil, der aus der Ebene herauszeigt) und kann als »Rauschen« betrachtet werden. Die anderen Varianzen sind dagegen hoch (Pfeile in der Ebene) und entsprechen dem »Signal« und einem Koordinatensystem für Daten mit verringelter Dimension.

Wenn also die Kovarianz vom Rang d ist, sind die Eigenwerte λ_{d+1} bis λ_D 0 und der Rekonstruktionsfehler beträgt 0.

Außerdem lässt sich zeigen, dass die obige Lösung durch Maximieren der Varianzen der Elemente von \mathbf{h} für ein orthogonales \mathbf{W} anstelle durch Minimieren des Rekonstruktionsfehlers erreicht werden kann.

Lineare Faktorenmödelle gehören zu den einfachsten generativen Modellen und den einfachsten Modellen zum Erlernen einer Repräsentation von Daten. Ähnlich wie sich lineare Klassifikatoren und lineare Regressionsmodelle auf tiefe Feedforward-Netze ausweiten lassen, können auch diese linearen Faktorenmödelle auf Autoencoder-Netze und tiefe probabilistische Modelle ausgeweitet werden, die dieselben Aufgaben mit einer sehr viel leistungsstärkeren und flexibleren Modellfamilie durchführen.

14

Autoencoder

Ein **Autoencoder** ist ein neuronales Netz, das darauf trainiert ist, zu versuchen, seine Eingabe in seine Ausgabe zu kopieren. Intern verfügt dieser über eine verdeckte Schicht \mathbf{h} , die einen **Code** beschreibt, der zur Repräsentation der Eingabe verwendet wird. Das Netz besteht aus zwei Teilen: einer Encoder-Funktion $\mathbf{h} = f(\mathbf{x})$ und einem Decoder, der eine Rekonstruktion $\mathbf{r} = g(\mathbf{h})$ erzeugt. Diese Architektur ist in Abbildung 14.1 zu sehen. Wenn ein Autoencoder lediglich lernt, $g(f(\mathbf{x}))$ überall mit \mathbf{x} gleichzusetzen, ist er nicht besonders nützlich. Vielmehr werden Autoencoder so konzipiert, dass sie nicht in der Lage sind, perfektes Kopieren zu erlernen. Meist werden sie so beschränkt, dass nur approximative Kopien möglich sind – und dass nur Eingaben kopiert werden, die den Trainingsdaten ähneln. Da das Modell gezwungen wird, die zu kopierenden Aspekte der Eingabe zu priorisieren, erlernt es häufig nützliche Eigenschaften der Daten.

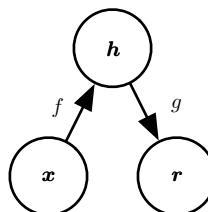


Abbildung 14.1: Der allgemeine Aufbau eines Autoencoders, bei dem eine Eingabe \mathbf{x} über eine interne Repräsentation oder einen Code \mathbf{h} einer Ausgabe \mathbf{r} (der sogenannten Rekonstruktion) zugeordnet wird. Der Autoencoder weist zwei Komponenten auf, nämlich den Encoder f (der die Zuordnung von \mathbf{x} zu \mathbf{h} vornimmt) und den Decoder g (der die Zuordnung von \mathbf{h} zu \mathbf{r} vornimmt).

Moderne Autoencoder haben das Konzept von Encoder und Decoder über die deterministischen Funktionen hinaus auf stochastische Zuordnungen $p_{\text{encoder}}(\mathbf{h} | \mathbf{x})$ und $p_{\text{decoder}}(\mathbf{x} | \mathbf{h})$ generalisiert.

Die Idee eines Autoencoders ist bereits seit Jahrzehnten Teil der Geschichte neuronaler Netze (*LeCun, 1987; Bourlard und Kamp, 1988; Hinton und Zemel, 1994*). Bisher wurden Autoencoder zur Dimensionsreduktion oder zum Lernen von Merkmalen (engl. *feature learning*) eingesetzt. In letzter Zeit haben theoretische Verbindungen zwischen Autoencodern und Modellen mit latenten Variablen die Autoencoder in den Mittelpunkt der generativen Modellierung gerückt – wir gehen in Kapitel 20 darauf ein. Autoencoder stellen eine Art Sonderfall der Feedforward-Netze dar und können auf dieselbe Weise trainiert werden, also meist mittels Mini-Batch-Gradientenabstieg für durch Backpropagation berechnete Gradienten. Anders als allgemeine Feedforward-Netze können Autoencoder aber auch durch **Rezirkulation** (*Hinton und McClelland, 1988*) trainiert werden, einem Lernalgorithmus, der die Aktivierungen des Netzes mit der ursprünglichen Eingabe in die Aktivierungen der wiederhergestellten Eingabe vergleicht. Rezirkulation wird als biologisch plausibler als die Backpropagation betrachtet, aber nur selten für Machine-Learning-Anwendungen genutzt.

14.1 Untervollständige Autoencoder

Das Kopieren der Eingabe in die Ausgabe hört sich zunächst sinnlos an – allerdings sind wir normalerweise gar nicht an der Ausgabe des Decoders interessiert. Stattdessen hoffen wir, dass ein Training des Autoencoders für das Kopieren der Eingabe zu einem \mathbf{h} mit nützlichen Eigenschaften führt.

Um nützliche Merkmale aus dem Autoencoder zu gewinnen, können wir zum Beispiel \mathbf{h} auf kleinere Dimensionen als \mathbf{x} einschränken. Ein Autoencoder, dessen Code-Dimension kleiner als die Eingabedimension ist, wird **untervollständig** (engl. *undercomplete*) genannt. Das Erlernen einer unterniedrigen Repräsentation zwingt den Autoencoder, die charakteristischsten Merkmale der Trainingsdaten zu erfassen.

Der Lernprozess lässt sich einfach beschreiben als Minimieren einer Verlustfunktion

$$L(\mathbf{x}, g(f(\mathbf{x}))), \quad (14.1)$$

wobei L eine Verlustfunktion ist, die $g(f(\mathbf{x}))$ mit einem Strafterm für die Unähnlichkeit mit \mathbf{x} belegt, zum Beispiel mit dem mittleren quadratischen Fehler.

Wenn der Decoder linear ist und L der mittlere quadratische Fehler, erlernt ein untvollständiger Autoencoder, sich über denselben Unterraum wie die Hauptkomponentenanalyse (engl. *principal components analysis*, PCA) aufzuspannen. In diesem Fall hat ein Autoencoder, der auf das Kopieren trainiert wurde, den wesentlichen Unterraum der Trainingsdaten quasi nebenbei erlernt.

Autoencoder mit nichtlinearen Encoder-Funktionen f und nichtlinearen Decoder-Funktionen g können somit eine leistungsstärkere nichtlineare Generalisierung der PCA erlernen. Leider kann der Autoencoder – wenn Encoder und Decoder eine zu hohe Kapazität zugestanden wird – lernen, den Kopiervorgang auszuführen, ohne nützliche Informationen über die Verteilung der Daten zu extrahieren. Theoretisch kann man sich einen Autoencoder mit einem eindimensionalen Code und sehr leistungsstarkem nichtlinearem Encoder vorstellen, der lernen kann, jedes Trainingsbeispiel $x^{(i)}$ mit dem Code i zu repräsentieren. Der Decoder könnte lernen, diese ganzzahligen Indizes wieder den Werten bestimmter Trainingsbeispiele zuzuordnen. Dieses spezielle Szenario tritt in der Praxis nicht auf, zeigt aber deutlich, dass ein für das Kopieren trainierter Autoencoder eventuell keinerlei nützliche Informationen über den Datensatz erlernt, wenn seine Kapazität zu groß werden darf.

14.2 Regularisierte Autoencoder

Untervollständige Autoencoder, deren Code-Dimension kleiner als die Eingabedimension ist, können die charakteristischsten Merkmale der Datenverteilung erlernen. Wir haben gesehen, dass diese Autoencoder keine nützlichen Informationen erlernen, wenn die Kapazität für Encoder und Decoder zu hoch ist.

Ein ähnliches Problem tritt auf, wenn die Dimension der verdeckten Repräsentation gleich der Eingabedimension sein darf bzw. im **übergvollständigen** (engl. *overcomplete*) Fall, in dem der verdeckte Code eine größere Dimension als die Eingabe aufweist. In diesen Fällen können sogar ein linearer Encoder und ein linearer Decoder lernen, die Eingabe in die Ausgabe zu kopieren, ohne dabei nützliche Informationen über die Datenverteilung zu erlernen.

Idealerweise könnte man jede beliebige Architektur eines Autoencoders erfolgreich trainieren, indem die Code-Dimension und die Kapazität von Encoder und Decoder passend zur Komplexität der zu modellierenden Verteilung gewählt werden. Regularisierte Autoencoder bieten diese Möglichkeit.

Statt die Modellkapazität mittels flacher Encoder und Decoder sowie einer geringen Code-Größe einzuschränken, nutzen regularisierte Autoencoder eine Verlustfunktion, die das Modell dazu anregt, neben der Fähigkeit, die eigene Eingabe in die Ausgabe zu kopieren, weitere Eigenschaften zu haben. Diese anderen Eigenschaften umfassen eine dünne Besetzung der Repräsentation, eine geringe Größe der Ableitung der Repräsentation und eine Robustheit gegenüber Rauschen oder fehlenden Eingaben. Ein regularisierter Autoencoder kann nichtlinear sowie übervollständig sein und dennoch nützliche Informationen über die Datenverteilung erlernen – sogar wenn die Modellkapazität groß genug ist, um eine triviale identische Abbildung zu erlernen.

Neben den hier beschriebenen Verfahren, die naturgemäß als regularisierte Autoencoder betrachtet werden, lässt sich nahezu jedes generative Modell mit latenten Variablen und einem Inferenzverfahren (zum Berechnen latenter Repräsentationen aus der Eingabe) als besondere Art eines Autoencoders betrachten. Zwei generative Modellierungsansätze, die diese Verbindung zu Autoencodern betonen, sind Nachfahren der Helmholtz-Maschine (*Hinton et al.*, 1995b), zum Beispiel der VAE (Variational Autoencoder) (Abschnitt 20.10.3) und die generativen stochastischen Netze (Abschnitt 20.12). Diese Modelle erlernen übervollständige Codierungen mit hoher Kapazität für die Eingabe und benötigen keinerlei Regularisierung, damit diese Codierungen nützlich sind. Ihre Codierungen sind nützlich, da die Modelle darauf trainiert wurden, die Wahrscheinlichkeit der Trainingsdaten näherungsweise zu maximieren, und nicht darauf, die Eingabe in die Ausgabe zu kopieren.

14.2.1 Sparse Autoencoder

Ein Sparse Autoencoder ist ein Autoencoder, dessen Trainingskriterium einen Strafterm $\Omega(\mathbf{h})$, der die Güte der dünnen Besetzung angibt, für die Code-Schicht \mathbf{h} zusätzlich zum Rekonstruktionsfehler aufweist:

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}), \quad (14.2)$$

wobei $g(\mathbf{h})$ die Decoder-Ausgabe ist; meist gilt $\mathbf{h} = f(\mathbf{x})$, die Encoder-Ausgabe.

Sparse Autoencoder werden üblicherweise zum Erlernen von Merkmalen für andere Aufgaben, beispielsweise eine Klassifizierung, verwendet. Ein Autoencoder, der regularisiert wurde, sodass er dünnbesetzt ist, muss auf eindeutige statistische Merkmale der Datenmenge, mit der er trainiert wurde, reagieren und darf nicht einfach als identische Abbildung agieren. Auf diese

Weise kann das Trainieren für das Kopieren mit einem Strafterm, der die Güte der dünnen Besetzung angibt, zu einem Modell führen, das nützliche Merkmale als Nebenprodukt erlernt hat.

Wir können uns den Strafterm $\Omega(\mathbf{h})$ einfach als Regularisierungsterm vorstellen, der einem Feedforward-Netz hinzugefügt wird, dessen Hauptaufgabe das Kopieren der Eingabe in die Ausgabe (unüberwachtes Lernziel) und möglicherweise auch das Ausführen einer überwachten Aufgabe (mit einem überwachten Lernziel) ist, die von diesen dünnbesetzten Merkmalen abhängt. Anders als bei anderen Regularisierern wie dem Weight Decay gibt es keine einfache bayessche Interpretation dieses Regularisierers. Wie in Abschnitt 5.6.1 beschrieben, lässt sich das Trainieren mit Weight Decay und anderen Regularisierungsstraftermen als MAP-Approximation der bayesschen Inferenz mit einem zusätzlichen regularisierenden Strafterm betrachten, der einer A-priori-Wahrscheinlichkeitsverteilung über die Modellparameter entspricht. Bei dieser Sichtweise entspricht die regularisierte Maximum Likelihood dem Maximieren von $p(\boldsymbol{\theta} | \mathbf{x})$, was gleichwertig mit dem Maximieren von $\log p(\mathbf{x} | \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$ ist. Der Term $\log p(\mathbf{x} | \boldsymbol{\theta})$ ist der übliche Term für die Log-Likelihood der Daten. Der Term $\log p(\boldsymbol{\theta})$ (die Log-a-priori-Verteilung über die Parameter) enthält die Präferenz über bestimmten Werten von $\boldsymbol{\theta}$. Diese Sichtweise wird in Abschnitt 5.6 beschrieben. Regularisierte Autoencoder trotzen einer solchen Interpretation, da der Regularisierer von den Daten abhängig ist und somit per definitionem keine A-priori-Verteilung im Sinne des Wortes sein kann. Wir können uns nach wie vor vorstellen, dass diese Regularisierungsterme implizit eine Präferenz über Funktionen ausdrücken.

Statt sich den Strafterm, der die Güte der dünnen Besetzung angibt, als Regularisierer für das Kopieren vorzustellen, können wir das gesamte Sparse Autoencoder-Framework als Approximation des Maximum-Likelihood-Trainings eines generativen Modells mit latenten Variablen ansehen. Gegeben sei ein Modell mit sichtbaren Variablen \mathbf{x} und latenten Variablen \mathbf{h} , mit einer expliziten multivariaten Verteilung $p_{\text{model}}(\mathbf{x}, \mathbf{h}) = p_{\text{model}}(\mathbf{h})p_{\text{model}}(\mathbf{x} | \mathbf{h})$. Wir bezeichnen $p_{\text{model}}(\mathbf{h})$ als A-priori-Verteilung des Modells über die latenten Variablen; es handelt sich also um die Annahmen des Modells, bevor \mathbf{x} bekannt ist. Bisher haben wir hierbei oft die Verteilung $p(\boldsymbol{\theta})$ als Codierung unserer Annahmen über die Modellparameter vor dem Bekanntwerden der Trainingsdaten verwendet. Die Log-Likelihood lässt sich wie folgt zerlegen:

$$\log p_{\text{model}}(\mathbf{x}) = \log \sum_{\mathbf{h}} p_{\text{model}}(\mathbf{h}, \mathbf{x}). \quad (14.3)$$

Wir können uns den Autoencoder als Approximation dieser Summe mit einem Punktschätzwert für nur einen höchst wahrscheinlichen Wert für

\mathbf{h} vorstellen. Das ähnelt dem generativen Modell mit Sparse Coding (Abschnitt 13.4), wobei \mathbf{h} jedoch die Ausgabe des parametrischen Encoders ist und nicht das Ergebnis einer Optimierung, die auf das wahrscheinlichste \mathbf{h} schließt. So gesehen und mit diesem ausgewählten \mathbf{h} maximieren wir

$$\log p_{\text{model}}(\mathbf{h}, \mathbf{x}) = \log p_{\text{model}}(\mathbf{h}) + \log p_{\text{model}}(\mathbf{x} \mid \mathbf{h}). \quad (14.4)$$

Der Term $\log p_{\text{model}}(\mathbf{h})$ kann dünne Besetzung herbeiführen. Zum Beispiel entspricht die Laplace-a-priori-Verteilung

$$p_{\text{model}}(h_i) = \frac{\lambda}{2} e^{-\lambda|h_i|} \quad (14.5)$$

einem Absolutbetrag eines Strafterms, der die Güte der dünnen Besetzung angibt. Wird die Log-a-priori-Verteilung als Absolutbetrag eines Strafterms ausgedrückt, erhalten wir

$$\Omega(\mathbf{h}) = \lambda \sum_i |h_i|, \quad (14.6)$$

$$-\log p_{\text{model}}(\mathbf{h}) = \sum_i \left(\lambda |h_i| - \log \frac{\lambda}{2} \right) = \Omega(\mathbf{h}) + \text{const}, \quad (14.7)$$

wobei der konstante Term nur von λ abhängig ist, nicht von \mathbf{h} . Wir behandeln λ üblicherweise als Hyperparameter und verwerfen den konstanten Term, da er das Parameterlernen nicht beeinflusst. Andere A-priori-Verteilungen wie die studentsche t -Verteilung können ebenfalls eine dünne Besetzung verursachen. Betrachtet man die dünne Besetzung als Ergebnis der Auswirkung von $p_{\text{model}}(\mathbf{h})$ auf das approximative Maximum Likelihood Learning, ist der Strafterm, der die Güte der dünnen Besetzung angibt, überhaupt kein Regularisierungsterm. Er resultiert lediglich aus der Modellverteilung über seine latenten Variablen. Diese Sichtweise bietet einen anderen Anreiz für das Trainieren eines Autoencoders: Auf diese Weise lässt sich ein generatives Modell näherungsweise trainieren. Sie liefert auch einen anderen Grund, wieso die vom Autoencoder erlernten Merkmale nützlich sind: Sie beschreiben die latenten Variablen, die die Eingabe erklären.

Frühe Arbeiten zu Sparse Autoencodern (*Ranzato et al., 2007a, 2008*) untersuchten diverse Formen der dünnen Besetzung und schlügen eine Verbindung zwischen dem Strafterm, der die Güte der dünnen Besetzung angibt, und dem Term $\log Z$ vor, der entsteht, wenn Maximum Likelihood für ein ungerichtetes probabilistisches Modell $p(\mathbf{x}) = \frac{1}{Z} \tilde{p}(\mathbf{x})$ verwendet wird. Die Idee dahinter ist, dass durch Minimierung von $\log Z$ verhindert wird, dass ein probabilistisches Modell überall eine hohe Wahrscheinlichkeit hat und

dass das Aufzwingen der dünnen Besetzung für einen Autoencoder diesen daran hindert, überall einen niedrigen Rekonstruktionsfehler aufzuweisen. In diesem Fall ist die oben genannte Verbindung eher auf dem Level eines intuitiven Verständnisses eines allgemeinen Mechanismus und ist eher keine mathematische Zuordnung. Die Interpretation, dass der Strafterm, der die Güte der dünnen Besetzung angibt, dem $\log p_{\text{model}}(\mathbf{h})$ in einem gerichteten Modell $p_{\text{model}}(\mathbf{h})p_{\text{model}}(\mathbf{x} \mid \mathbf{h})$ entspricht, ist mathematisch einfacher.

Eine Möglichkeit zum Erzielen *tatsächlicher Nullen* in \mathbf{h} für Sparse Autoencoder (und Denoising Autoencoder) wurde in Glorot *et al.* (2011b) vorgestellt. Dabei werden ReLUs zum Erzeugen der Codeschicht verwendet. Mit einer A-priori-Wahrscheinlichkeitsverteilung, die die Repräsentationen tatsächlich in Richtung Null drückt (ähnlich dem Absolutbetrag des Strafterms) kann man somit indirekt die durchschnittliche Anzahl von Nullen in der Repräsentation beeinflussen.

14.2.2 Denoising Autoencoder

Statt einen Strafterm Ω zur Kostenfunktion hinzuzufügen, können wir einen Autoencoder erhalten, der etwas Nützliches durch Verändern des Rekonstruktionsfehlerterms der Kostenfunktion erlernt.

Traditionell minimiert der Autoencoder eine Funktion

$$L(\mathbf{x}, g(f(\mathbf{x}))), \quad (14.8)$$

wobei L eine Verlustfunktion ist, die $g(f(\mathbf{x}))$ für die Unähnlichkeit zu \mathbf{x} bestraft, zum Beispiel die L^2 -Norm der Differenz. So tendiert $g \circ f$ dazu, zu lernen, lediglich eine identische Abbildung zu sein, wenn die entsprechende Kapazität vorliegt.

Ein **Denoising Autoencoder** (DAE) minimiert dagegen

$$L(\mathbf{x}, g(f(\tilde{\mathbf{x}}))), \quad (14.9)$$

wobei $\tilde{\mathbf{x}}$ eine Kopie von \mathbf{x} ist, die durch eine Form von Rauschen beschädigt wurde. Denoising Autoencoder müssen diese Beschädigung daher zurücknehmen und nicht nur lediglich die Eingabe kopieren.

Ein Denoising Training zwingt f und g dazu, die Struktur von $p_{\text{data}}(\mathbf{x})$ implizit zu erlernen, wie Alain und Bengio (2013) und Bengio *et al.* (2013c) gezeigt haben. Denoising Autoencoder sind daher ein weiteres Beispiel dafür, wie nützliche Eigenschaften als Nebenprodukt bei der Minimierung des Rekonstruktionsfehlers entstehen können. Sie sind auch ein Beispiel dafür, wie übervollständige Modelle mit hoher Kapazität als Autoencoder

genutzt werden können, sofern verhindert wird, dass sie die identische Abbildung erlernen. Denoising Autoencoder werden in Abschnitt 14.5 genauer behandelt.

14.2.3 Regularisierung durch Bestrafung von Ableitungen

Eine weitere Strategie zur Regularisierung von Autoencodern ist die Verwendung eines Strafterms Ω , wie in Sparse Autoencodern,

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}, \mathbf{x}), \quad (14.10)$$

aber mit einer anderen Form von Ω :

$$\Omega(\mathbf{h}, \mathbf{x}) = \lambda \sum_i \|\nabla_{\mathbf{x}} h_i\|^2. \quad (14.11)$$

Dadurch wird das Modell gezwungen, eine Funktion zu erlernen, die sich bei einer geringfügigen Änderung von \mathbf{x} kaum ändert. Da dieser Strafterm nur für die Trainingsbeispiele verwendet wird, muss der Autoencoder Merkmale erlernen, die Informationen über die Verteilung der Trainingsdaten erfassen.

Ein derart regularisierter Autoencoder wird als **Contractive Autoencoder** (CAE, dt. *kontrahierender Autoencoder*) bezeichnet. Dieser Ansatz weist theoretische Verbindungen zu Denoising Autoencodern, zum Manifold Learning und zur probabilistischen Modellierung auf. Der CAE wird in Abschnitt 14.7 genauer beschrieben.

14.3 Repräsentationsleistung, Schichtgröße und Tiefe

Autoencoder werden häufig lediglich mit einem einschichtigen Encoder und einem einschichtigen Decoder trainiert. Das muss aber nicht so sein. Tatsächlich bietet die Nutzung tiefer Encoder und Decoder viele Vorteile.

In Abschnitt 6.4.1 haben Sie erfahren, dass ein Feedforward-Netz von seiner Tiefe profitiert. Da Autoencoder ebenfalls Feedforward-Netze sind, gelten diese Vorteile auch für sie. Außerdem ist der Encoder selbst ein Feedforward-Netz und ebenso der Decoder. Somit kann jede dieser Komponenten des Autoencoders für sich genommen von einer größeren Tiefe profitieren.

Ein großer Vorteil nichttrivialer Tiefe ist, dass das Theorem der universellen Approximation garantiert, dass ein neuronales Feedforward-Netz mit mindestens einer verdeckten Schicht eine Approximation einer beliebigen

Funktion (in einer breiten Klasse) mit einem beliebigen Maß an Genauigkeit repräsentieren kann, sofern es über genügend verdeckte Einheiten verfügt. Somit kann ein Autoencoder mit einer verdeckten Schicht die identische Abbildung entlang des Eingabebereichs der Daten beliebig gut repräsentieren. Allerdings ist die Zuordnung von der Eingabe zum Code flach. Wir können somit keine beliebigen Bedingungen erzwingen, wie zum Beispiel, dass der Code dünnbesetzt sein sollte. Ein tiefer Autoencoder mit mindestens einer zusätzlichen verdeckten Schicht im Encoder selbst kann jede beliebige Zuordnung von der Eingabe zum Code beliebig gut approximieren, sofern genügend verdeckte Einheiten vorhanden sind.

Die Tiefe kann den Berechnungsaufwand für die Repräsentation einiger Funktionen exponentiell reduzieren. Sie kann auch die Menge der für das Erlernen einiger Funktionen erforderlichen Trainingsdaten exponentiell verringern. In Abschnitt 6.4.1 werden die Vorteile der Tiefe in Feedforward-Netzen vorgestellt.

Experimentell bieten tiefe Autoencoder eine deutlich bessere Komprimierung als entsprechende flache oder lineare Autoencoder (*Hinton und Salakhutdinov, 2006*).

Ein übliches Verfahren beim Trainieren tiefer Autoencoder besteht darin, die tiefe Architektur einem Pretraining mit Greedy-Algorithmen zu unterziehen, bei dem ein Stapel flacher Autoencoder trainiert wird. Wir begegnen also häufig flachen Autoencodern – sogar dann, wenn das eigentliche Ziel im Training eines tiefen Autoencoders besteht.

14.4 Stochastische Encoder und Decoder

Autoencoder sind letztendlich nichts anderes als Feedforward-Netze. Dieselben Arten von Verlustfunktionen und Ausgabeeinheiten, die für klassische Feedforward-Netze verwendet werden, kommen auch bei Autoencodern zum Einsatz.

Wie in Abschnitt 6.2.2.4 beschrieben, besteht ein allgemeines Verfahren zum Entwerfen der Ausgabeeinheiten und der Verlustfunktion eines Feedforward-Netzes darin, eine Ausgabeverteilung $p(\mathbf{y} | \mathbf{x})$ zu definieren und die negative Log-Likelihood $-\log p(\mathbf{y} | \mathbf{x})$ zu minimieren. In diesem Fall ist \mathbf{y} ein Vektor mit Zielwerten, zum Beispiel Klassen-Labels.

In einem Autoencoder ist \mathbf{x} nun gleichzeitig Zielwert und Eingabe. Allerdings können wir noch immer dieselben Abläufe wie zuvor nutzen. Für eine verdeckte Repräsentation \mathbf{h} können wir uns vorstellen, dass der Decoder

eine bedingte Verteilung $p_{\text{decoder}}(\mathbf{x} | \mathbf{h})$ liefert. Dann können wir den Autoencoder durch Minimieren von $-\log p_{\text{decoder}}(\mathbf{x} | \mathbf{h})$ trainieren. Die exakte Form dieser Verlustfunktion ändert sich abhängig von der Form von p_{decoder} . Wie bei klassischen Feedforward-Netzen verwenden wir meist lineare Ausgabeeinheiten zum Parametrisieren des Mittelwerts einer Normalverteilung, wenn \mathbf{x} reellwertig ist. In diesem Fall ergibt die negative Log-Likelihood ein mittleres quadratisches Fehlerkriterium. Ebenso entsprechen binäre \mathbf{x} -Werte einer Bernoulli-Verteilung, deren Parameter sich aus einer sigmoiden Ausgabeinheit ergeben. Diskrete \mathbf{x} -Werte entsprechen einer softmax-Verteilung usw. Normalerweise werden die AusgabevARIABLEN als bedingt unabhängig für \mathbf{h} betrachtet, sodass diese Wahrscheinlichkeitsverteilung ohne großen Aufwand berechnet werden kann. Aber einige Methoden wie Mixture-Density-Ausgaben ermöglichen eine effizient durchführbare Modellierung der Ausgaben mit Korrelationen.

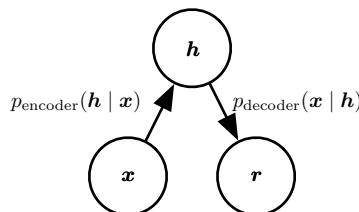


Abbildung 14.2: Der Aufbau eines stochastischen Autoencoders, in dem sowohl Encoder als auch Decoder keine einfachen Funktionen sind. Vielmehr wird ihnen etwas Rauschen hinzugefügt, sodass ihre Ausgabe als Stichprobe aus einer Verteilung betrachtet werden kann: $p_{\text{encoder}}(\mathbf{h} | \mathbf{x})$ für den Encoder und $p_{\text{decoder}}(\mathbf{x} | \mathbf{h})$ für den Decoder.

Um uns stärker von den bisher betrachteten Feedforward-Netzen abzuwenden, können wir den Begriff einer **Encoder-Funktion** $f(\mathbf{x})$ auf eine **Encoder-Verteilung** $p_{\text{encoder}}(\mathbf{h} | \mathbf{x})$ generalisieren (vgl. Abbildung 14.2).

Jedes Modell $p_{\text{model}}(\mathbf{h}, \mathbf{x})$ mit latenten Variablen definiert einen stochastischen Encoder

$$p_{\text{encoder}}(\mathbf{h} | \mathbf{x}) = p_{\text{model}}(\mathbf{h} | \mathbf{x}) \quad (14.12)$$

und einen stochastischen Decoder

$$p_{\text{decoder}}(\mathbf{x} | \mathbf{h}) = p_{\text{model}}(\mathbf{x} | \mathbf{h}). \quad (14.13)$$

Grundsätzlich müssen die Encoder- und Decoder-Verteilungen nicht notwendigerweise bedingte Verteilungen sein, die mit einer eindeutigen multivariaten Verteilung $p_{\text{model}}(\mathbf{x}, \mathbf{h})$ kompatibel sind. Alain et al. (2015) haben

gezeigt, dass das Trainieren von Encoder und Decoder als Denoising Autoencoder dazu führt, dass diese asymptotisch kompatibel sind (ausreichend Kapazität und Beispiele vorausgesetzt).

14.5 Denoising Autoencoder

Der **Denoising Autoencoder** (DAE) ist ein Autoencoder, der einen beschädigten Datenpunkt als Eingabe erhält und darauf trainiert wird, den ursprünglichen, nicht beschädigten Datenpunkt als Ausgabe vorherzusagen.

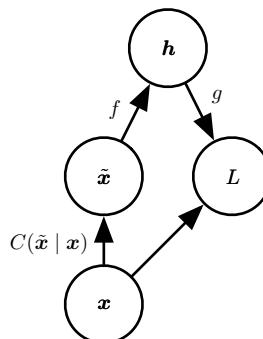


Abbildung 14.3: Der Berechnungsgraph der Kostenfunktion eines Denoising Autoencoders, der auf die Rekonstruktion des einwandfreien Datenpunkts \mathbf{x} für die beschädigte Version $\tilde{\mathbf{x}}$ trainiert wird. Dazu wird der Verlust $L = -\log p_{\text{decoder}}(\mathbf{x} | \mathbf{h} = f(\tilde{\mathbf{x}}))$ minimiert, wobei $\tilde{\mathbf{x}}$ eine beschädigte Version des Datenbeispiels \mathbf{x} ist, das in einem bestimmten Beschädigungsprozess $C(\tilde{\mathbf{x}} | \mathbf{x})$ entsteht. Üblicherweise handelt es sich bei der Verteilung p_{decoder} um eine faktorielle Verteilung, deren mittlere Parameter aus einem Feedforward-Netz g stammen.

Das DAE-Trainingsverfahren ist in Abbildung 14.3 dargestellt. Wir führen einen Beschädigungsprozess $C(\tilde{\mathbf{x}} | \mathbf{x})$ ein, der eine bedingte Verteilung über beschädigte Stichproben $\tilde{\mathbf{x}}$ für eine Stichprobe \mathbf{x} repräsentiert. Der Autoencoder erlernt anschließend eine **Rekonstruktionsverteilung** $p_{\text{reconstruct}}(\mathbf{x} | \tilde{\mathbf{x}})$, die wie folgt aus Trainingspaaren $(\mathbf{x}, \tilde{\mathbf{x}})$ geschätzt wird:

1. Ziehe ein Trainingsbeispiel \mathbf{x} aus den Trainingsdaten.
2. Ziehe eine beschädigte Version $\tilde{\mathbf{x}}$ aus $C(\tilde{\mathbf{x}} | \mathbf{x} = \mathbf{x})$.
3. Verwende $(\mathbf{x}, \tilde{\mathbf{x}})$ als Trainingsbeispiel zum Schätzen der Autoencoder-Rekonstruktionsverteilung $p_{\text{reconstruct}}(\mathbf{x} | \tilde{\mathbf{x}}) = p_{\text{decoder}}(\mathbf{x} | \mathbf{h})$ mit \mathbf{h} als Ausgabe des Encoders $f(\tilde{\mathbf{x}})$ und p_{decoder} , die üblicherweise mittels Decoder $g(\mathbf{h})$ definiert wird.

Normalerweise können wir einfach eine approximative Minimierung auf Gradientenbasis (zum Beispiel der Mini-Batch-Gradientenabstieg) für die negative Log-Likelihood $-\log p_{\text{decoder}}(\mathbf{x} \mid \mathbf{h})$ durchführen. Sofern der Encoder deterministisch ist, handelt es sich beim Denoising Autoencoder um ein Feedforward-Netz, das mit denselben Verfahren wie alle anderen Feed-forward-Netze trainiert werden kann.

Wir können sagen, der DAE führt ein stochastisches Gradientenabstiegsverfahren für den folgenden Erwartungswert durch:

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}(\mathbf{x})} \mathbb{E}_{\tilde{\mathbf{x}} \sim C(\tilde{\mathbf{x}} \mid \mathbf{x})} \log p_{\text{decoder}}(\mathbf{x} \mid \mathbf{h} = f(\tilde{\mathbf{x}})), \quad (14.14)$$

wobei $\hat{p}_{\text{data}}(\mathbf{x})$ die Verteilung der Trainingsdaten ist.

14.5.1 Schätzen des Scores

Score Matching (*Hyvärinen*, 2005) ist eine Alternative zur Maximum Likelihood. Es bietet einen stetigen Schätzer für Wahrscheinlichkeitsverteilungen, wobei das Modell dazu angeregt wird, in jedem Trainingspunkt \mathbf{x} denselben **Score** (dieselbe Bewertung oder Punktzahl) wie die Datenverteilung aufzuweisen. In diesem Zusammenhang ist der Score ein bestimmtes Gradientenfeld:

$$\nabla_{\mathbf{x}} \log p(\mathbf{x}). \quad (14.15)$$

In Abschnitt 18.4 befassen wir uns näher mit dem Score Matching. Für den Moment reicht es aus, zu wissen, dass das Erlernen des Gradientenfeldes für $\log p_{\text{data}}$ eine Möglichkeit zum Erlernen der Struktur von p_{data} selbst darstellt.

Eine sehr wichtige Eigenschaft von DAEs ist, dass ihr Trainingskriterium (mit der bedingten Normalverteilung $p(\mathbf{x} \mid \mathbf{h})$) dazu führt, dass der Autoencoder ein Vektorfeld $(g(f(\mathbf{x})) - \mathbf{x})$ erlernt, das den Score der Datenverteilung schätzt. Abbildung 14.4 verdeutlicht dies.

Das Denoising Training ist ein spezieller Autoencoder (sigmoide verdeckte Einheiten, lineare Rekonstruktionseinheiten), der normalverteiltes Rauschen und den mittleren quadratischen Fehler bei gleichwertigem Rekonstruktionsaufwand (*Vincent*, 2011) zum Trainieren eines bestimmten ungerichteten probabilistischen Modells namens Restricted Boltzmann Machine (RBM) mit normalverteilten sichtbaren Einheiten verwendet. Diese Art von Modell wird detailliert in Abschnitt 20.5.1 beschrieben. Für den Moment reicht es aus, zu wissen, dass dieses Modell eine explizite $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ liefert. Wenn die RBM mit **Denoising Score Matching** (*Kingma und LeCun*, 2010) trainiert wird, entspricht ihr Lernalgorithmus dem Denoising

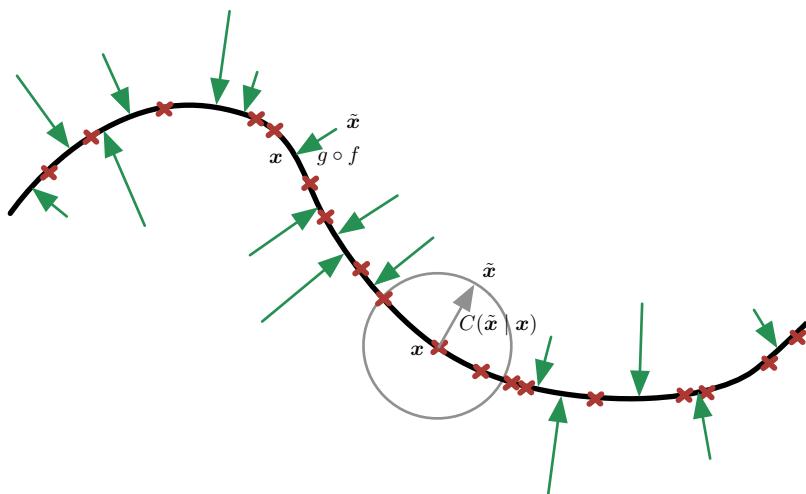


Abbildung 14.4: Ein Denoising Autoencoder wird darauf trainiert, einen beschädigten Datenpunkt \tilde{x} wieder dem ursprünglichen Datenpunkt x zuzuordnen. Wir stellen die Trainingsbeispiele x als Kreuze in der Nähe einer geringdimensionalen Mannigfaltigkeit (die dicke schwarze Linie) dar. Der graue Kreis ist der Beschädigungsprozess $C(\tilde{x} | x)$ mit gleichwahrscheinlichen Beschädigungen. Ein hellgrauer Pfeil innerhalb des Kreises zeigt, wie ein Trainingsbeispiel zu einer Stichprobe aus dem Beschädigungsprozess wird. Wenn der Denoising Autoencoder darauf trainiert wird, den Durchschnittswert der quadratischen Fehler $\|g(f(\tilde{x})) - x\|^2$ zu minimieren, schätzt die Rekonstruktionsfunktion $g(f(\tilde{x}))$ also $\mathbb{E}_{x, \tilde{x} \sim p_{\text{data}}(x)C(\tilde{x}|x)}[x | \tilde{x}]$. Der Vektor $g(f(\tilde{x})) - x$ weist näherungsweise in Richtung des nächstgelegenen Punktes auf der Mannigfaltigkeit, da $g(f(\tilde{x}))$ das Massenzentrum der einwandfreien Punkte x schätzt, die einem Anstieg von \tilde{x} möglicherweise dienlich waren. Der Autoencoder lernt somit ein Vektorfeld $g(f(x)) - x$ (mit dunkelgrauen Pfeilen, die in Richtung der Linie weisen, angedeutet). Dieses Vektorfeld schätzt den Score $\nabla_x \log p_{\text{data}}(x)$ bis zu einem multiplikativen Faktor, der den Durchschnittswert des mittleren quadratischen Rekonstruktionsfehlers darstellt.

Training des entsprechenden Autoencoders. Bei einem unveränderlichen Rauschpegel ist das regularisierte Score Matching kein stetiger Schätzer, sondern stellt eine verzerrte Version der Verteilung wieder her. Wird der Rauschpegel so gewählt, dass er sich 0 annähert, wenn die Anzahl der Beispiele gegen unendlich geht, wird die Konsistenz jedoch wiederhergestellt. Denoising Score Matching wird in Abschnitt 18.5 genauer behandelt.

Es gibt noch weitere Verbindungen zwischen Autoencodern und RBMs. Score Matching führt bei RBMs zu einer Kostenfunktion, die gleich dem Rekonstruktionsfehler ist, der mit einem Regularisierungsterm kombiniert wurde, der dem kontrahierenden Strafterm (engl. *contractive penalty*) des

CAE ähnelt (*Swersky et al.*, 2011). *Bengio und Delalleau* (2009) haben gezeigt, dass ein Gradient des Autoencoders eine Approximation an das Trainings mit kontrastiver Divergenz für RBMs liefert.

Für stetigwertige \mathbf{x} führt das Denoising-Kriterium mit Beschädigungs- und Rekonstruktionsverteilung zu einem Schätzer für den Score, der für die allgemeinen Parametrisierungen von Encoder und Decoder genutzt werden kann (*Alain und Bengio*, 2013). Das bedeutet, dass eine generische Encoder-Decoder-Architektur dazu gebracht werden kann, den Score durch Trainieren mit dem Kriterium des mittleren quadratischen Fehlers

$$\|g(f(\tilde{\mathbf{x}})) - \mathbf{x}\|^2 \quad (14.16)$$

und der Beschädigung

$$C(\tilde{\mathbf{x}} = \tilde{\mathbf{x}}|\mathbf{x}) = \mathcal{N}(\tilde{\mathbf{x}}; \mu = \mathbf{x}, \Sigma = \sigma^2 I) \quad (14.17)$$

mit der Rauschvarianz σ^2 zu schätzen. Abbildung 14.5 stellt diesen Vorgang dar.

Grundsätzlich gibt es keine Garantie dafür, dass die Rekonstruktion $g(f(\mathbf{x}))$ abzüglich der Eingabe \mathbf{x} dem Gradienten irgendeiner Funktion oder gar dem Score entspricht. Darum gelten die frühen Ergebnisse (*Vincent*, 2011) für ganz bestimmte Parametrisierungen, in denen $g(f(\mathbf{x})) - \mathbf{x}$ durch Verwenden der Ableitung einer anderen Funktion ermittelt werden kann. *Kamyshanska und Memisevic* (2015) haben die Ergebnisse von *Vincent* (2011) durch Bestimmen einer Familie flacher Autoencoder, für die $g(f(\mathbf{x})) - \mathbf{x}$ einem Score für alle Elemente der Familie entspricht, generalisiert.

Bisher haben wir nur beschrieben, wie der Denoising Autoencoder das Repräsentieren einer Wahrscheinlichkeitsverteilung erlernt. Allgemein betrachtet möchten wir vielleicht den Autoencoder als generatives Modell verwenden und Stichproben aus dieser Verteilung ziehen. Das wird in Abschnitt 20.11 beschrieben.

14.5.2 Historische Einblicke

Die Idee, mehrschichtige Perzeptren für das Denoising (Entrauschen) zu nutzen, geht auf die Arbeit von *LeCun* (1987) und *Gallinari et al.* (1987) zurück. Auch *Behnke* (2001) nutzte RNNs für das Denoising von Bildern. Denoising Autoencoder sind in gewisser Weise nichts anderes als mehrschichtige Perzeptren, die für das Denoising trainiert wurden. Der Begriff »Denoising Autoencoder« bezeichnet allerdings ein Modell, das nicht nur das Denoising seiner Eingabe erlernen soll, sondern in diesem Zuge auch eine gute interne

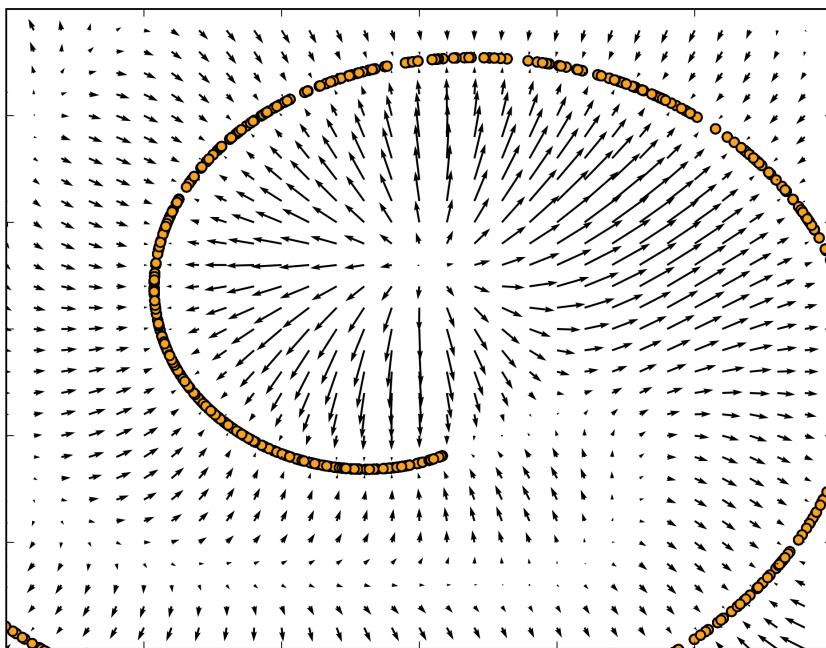


Abbildung 14.5: Von einem Denoising Autoencoder erlerntes Vektorfeld um eine gekrümmte 1-D-Mannigfaltigkeit, in deren Nähe die Daten sich in einem 2-D-Raum konzentrieren. Jeder Pfeil ist proportional zur Rekonstruktion abzüglich des Eingabevektors für den Autoencoder. Die Pfeile weisen jeweils zur höheren Wahrscheinlichkeit laut implizit geschätzter Wahrscheinlichkeitsverteilung. Das Vektorfeld weist in beiden Maxima der geschätzten Dichtefunktion (auf den Daten-Mannigfaltigkeiten) und in den Minima dieser Dichtefunktion Nullen auf. Zum Beispiel bildet der Spiralarm eine 1-D-Mannigfaltigkeit der lokalen Maxima, die miteinander verbunden sind. Lokale Minima erscheinen in der Nähe der Mitte der Lücke zwischen den beiden Armen. Wenn die Norm des Rekonstruktionsfehlers (entspricht der Länge der Pfeile) groß ist, kann die Wahrscheinlichkeit durch Bewegung in Pfeilrichtung erheblich gesteigert werden; dies ist hauptsächlich an Stellen mit geringer Wahrscheinlichkeit der Fall. Der Autoencoder ordnet diese Punkte geringerer Wahrscheinlichkeit den Rekonstruktionen höherer Wahrscheinlichkeit zu. An den Stellen, wo die Wahrscheinlichkeit maximal ist, werden die Pfeil kürzer, da die Rekonstruktion exakter wird. (Abbildung mit freundlicher Genehmigung von Alain und Bengio (2013))

Repräsentation. Diese Idee kam erst viel später auf (*Vincent et al.*, 2008, 2010). Die erlernte Repräsentation kann dann für das Pretraining eines tiefen unüberwachten Netzes oder eines überwachten Netzes verwendet werden. Ähnlich wie bei Sparse Autoencodern, dem Sparse Coding, kontrahierenden Autoencodern und anderen regularisierten Autoencodern war der Grund für DAEs das Lernen eines Encoders mit sehr hoher Kapazität zu ermöglichen und dabei zu verhindern, dass Encoder und Decoder eine nicht brauchbare identische Abbildung erlernen.

Vor der Einführung des modernen DAE haben *Inayoshi und Kurita* (2005) einige dieser Zielsetzungen mit einigen dieser Verfahren untersucht. Ihr Ansatz minimiert den Rekonstruktionsfehler zusätzlich zu einer überwachten Zielvorgabe, während das Rauschen in der verdeckten Schicht eines überwachten mehrschichtigen Perzeptrons (MLP) hinzugefügt wird, um die Generalisierung durch Einführung des Rekonstruktionsfehlers und des hinzugefügten Rauschens zu verbessern. Ihr Verfahren basierte allerdings auf einem linearen Encoder und konnte nicht so leistungsfähige Funktionsfamilien erlernen wie der moderne DAE.

14.6 Erlernen von Mannigfaltigkeiten mit Autoencodern

Wie viele andere Machine-Learning-Algorithmen nutzen Autoencoder das Konzept, dass sich Daten um eine geringdimensionale Mannigfaltigkeit (engl. *manifold*) oder eine kleine Menge solcher Mannigfaltigkeiten (vgl. Abschnitt 5.11.3) sammeln. Einige Machine-Learning-Algorithmen nutzen dieses Konzept nur insofern, als sie eine Funktion erlernen, die sich auf der Mannigfaltigkeit korrekt verhält, aber für Eingaben abseits der Mannigfaltigkeit ein ungewöhnliches Verhalten an den Tag legen können. Autoencoder bauen das Konzept weiter aus und versuchen, die Struktur der Mannigfaltigkeit zu erlernen.

Um zu verstehen, wie Autoencoder dies tun, müssen wir uns mit einigen wichtigen Eigenschaften von Mannigfaltigkeiten befassen.

Eine wichtige Charakterisierung einer Mannigfaltigkeit ist die Menge ihrer **Tangentialebenen**. In einem Punkt x auf einer d -dimensionalen Mannigfaltigkeit ergibt sich die Tangentialebene aus d Basisvektoren, die die auf der Mannigfaltigkeit zulässigen lokalen Richtungen der Variationen aufspannen. Wie Abbildung 14.6 zeigt, geben diese lokalen Richtungen an, wie x infinitesimal geändert werden kann, ohne die Mannigfaltigkeit zu verlassen.

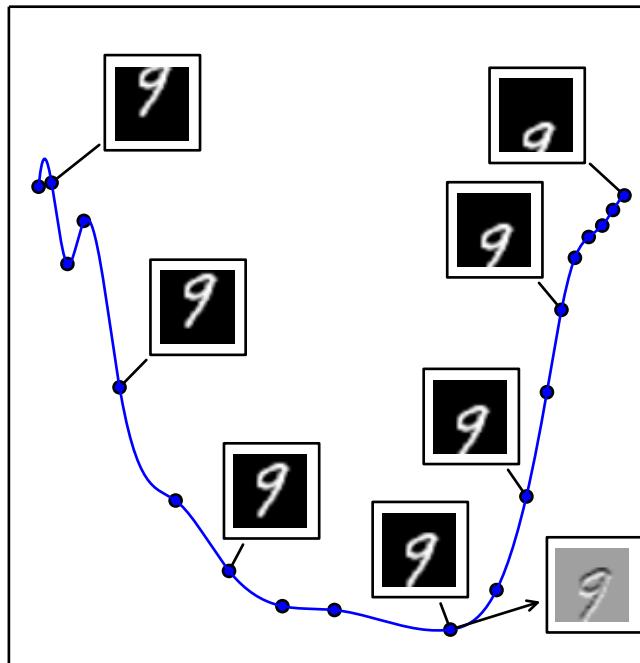


Abbildung 14.6: Eine Darstellung des Konzepts einer tangentialen Hyperebene. Hier erstellen wir eine 1-D-Mannigfaltigkeit im 784-D-Raum. Wir transformieren ein MNIST-Bild mit 784 Pixeln durch vertikale Verschiebung. Der Betrag der vertikalen Verschiebung definiert eine Koordinate entlang einer 1-D-Mannigfaltigkeit, die einen gekrümmten Pfad durch den Bildraum beschreibt. Dieser Plot zeigt einige Punkte entlang der Mannigfaltigkeit. Zur Visualisierung haben wir die Mannigfaltigkeit mittels PCA in den 2-D-Raum projiziert. Eine n -dimensionale Mannigfaltigkeit weist in jedem Punkt eine n -dimensionale Tangentialebene auf. Diese Tangentialebene berührt die Mannigfaltigkeit exakt in diesem Punkt und ist parallel zur Oberfläche an diesem Punkt ausgerichtet. Sie definiert den Raum der Richtungen, in dem Bewegungen ohne Verlassen der Mannigfaltigkeit möglich sind. Diese 1-D-Mannigfaltigkeit hat eine einzelne Tangente. Wir kennzeichnen eine Beispieldtangente in einem Punkt mit einem Bild, das angibt, wie diese Tangentenrichtung im Bildraum erscheint. Grau dargestellte Pixel ändern sich bei einer Bewegung entlang der Tangente nicht, weiße Pixel werden heller und schwarze Pixel werden dunkler.

Alle Trainingsverfahren für Autoencoder gehen einen Kompromiss zwischen zwei Kräften ein:

1. Erlernen einer Repräsentation \mathbf{h} eines Trainingsbeispiels \mathbf{x} in der Art, dass \mathbf{x} mit einem Decoder aus \mathbf{h} näherungsweise wiederhergestellt werden kann. Die Tatsache, dass \mathbf{x} aus den Trainingsdaten gezogen wird,

ist entscheidend, da dies bedeutet, dass der Autoencoder im Rahmen der datengenerierenden Verteilung unwahrscheinliche Eingaben nicht erfolgreich wiederherstellen muss.

2. Erfüllen der Bedingung oder des Strafterms für die Regularisierung. Es kann sich hierbei um eine architektonische Bedingung handeln, die die Kapazität des Autoencoders einschränkt, oder um einen Regularisierungsterm, der zum Rekonstruktionsaufwand addiert wird. Diese Verfahren bevorzugen insgesamt Lösungen, die weniger stark von der Eingabe abhängig sind.

Offensichtlich wäre keine der Kräfte für sich genommen nützlich – das Kopieren der Eingabe in die Ausgabe bietet für sich keinen Nutzen, ebenso wenig wie das Ignorieren der Eingabe. Die beiden Kräfte werden erst nützlich, weil sie gemeinsam die verdeckte Repräsentation dazu zwingen, Informationen über die Struktur der datengenerierenden Verteilung zu erfassen. Das wichtige Prinzip ist, dass der Autoencoder es sich leisten kann, *nur die Variationen zu repräsentieren, die zur Rekonstruktion von Trainingsbeispielen benötigt werden*. Wenn sich die datengenerierende Verteilung in der Nähe einer geringdimensionalen Mannigfaltigkeit konzentriert, führt dies zu Repräsentationen, die implizit ein lokales Koordinatensystem dieser Mannigfaltigkeit erfassen: Nur die tangentiale Variationen der Mannigfaltigkeit um \mathbf{x} müssen auf Änderungen in $\mathbf{h} = f(\mathbf{x})$ reagieren. Somit erlernt der Encoder eine Zuordnung vom Eingaberaum \mathbf{x} zu einem Darstellungsraum, die empfindlich gegenüber Änderungen in den Richtungen entlang der Mannigfaltigkeit reagiert, aber nicht gegenüber Änderungen, die orthogonal zur Mannigfaltigkeit erfolgen.

In Abbildung 14.7 ist ein eindimensionales Beispiel dargestellt. Es zeigt, dass eine Rekonstruktionsfunktion, die unempfindlich gegenüber Störungen der Eingabe in der Nähe der Datenpunkte ist, dazu führt, dass der Autoencoder die Mannigfaltigkeitsstruktur wiederherstellt.

Um zu verstehen, warum Autoencoder für das Manifold Learning nützlich sind, hilft ein Vergleich mit anderen Ansätzen. Eine **Repräsentation** der Datenpunkte auf (oder in der Nähe) der Mannigfaltigkeit ist die üblicherweise erlernte Eigenschaft einer Mannigfaltigkeit. Eine solche Repräsentation für ein bestimmtes Beispiel wird auch als dessen Embedding (dt. *Einbettung*) bezeichnet. Diese ergibt sich üblicherweise aus einem geringdimensionalen Vektor, der weniger Dimensionen als der »Umgebungsraum« aufweist, deren geringdimensionale Teilmenge die Mannigfaltigkeit ist. Einige Algorithmen (nichtparametrische Manifold-Learning-Algorithmen, siehe unten) erlernen ein Embedding für jedes Trainingsbeispiel direkt, andere dagegen erlernen

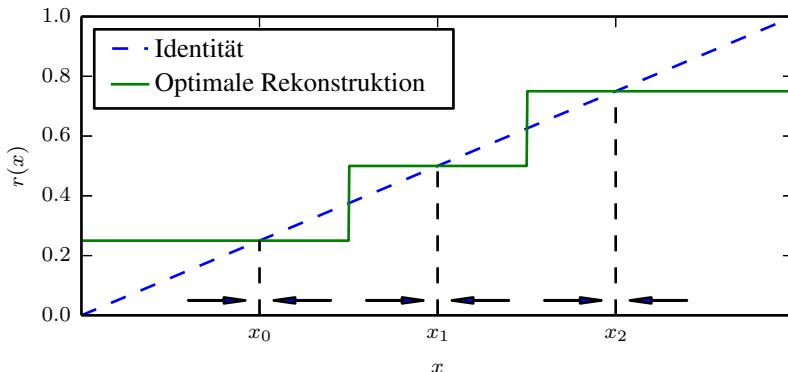


Abbildung 14.7: Wenn der Autoencoder eine Rekonstruktionsfunktion erlernt, die invariant gegenüber kleinen Störungen in der Nähe der Datenpunkte ist, erfasst er die Mannigfaltigkeitsstruktur der Daten. Hier ist die Mannigfaltigkeitsstruktur eine Sammlung 0-dimensionaler Mannigfaltigkeiten. Die gestrichelte Diagonale markiert das Ziel der identischen Abbildung für die Rekonstruktion. Die optimale Rekonstruktionsfunktion schneidet die identische Abbildung in jedem Datenpunkt. Die horizontalen Pfeile unten in der Grafik geben den Rekonstruktionsrichtungsvektor $r(\mathbf{x}) - \mathbf{x}$ an der Basis des Pfeils an (im Eingaberaum) und weisen stets in Richtung der nächstgelegenen »Mannigfaltigkeit« (einem einzelnen Datenpunkt im 1-D-Fall). Der Denoising Autoencoder versucht explizit, die Ableitung der Rekonstruktionsfunktion $r(\mathbf{x})$ in der Nähe der Datenpunkte klein zu machen. Der Contractive Autoencoder erledigt dasselbe für den Encoder. Obwohl die Ableitung von $r(\mathbf{x})$ in der Nähe der Datenpunkte klein sein soll, kann sie zwischen den Datenpunkten groß sein. Der Raum zwischen den Datenpunkten entspricht dem Bereich zwischen den Mannigfaltigkeiten, in dem die Rekonstruktionsfunktion eine große Ableitung aufweisen muss, um beschädigte Punkte wieder auf der Mannigfaltigkeit zuzuordnen.

eine allgemeinere Zuordnung, die manchmal als Encoder- oder Repräsentationsfunktion bezeichnet wird, mit der jeder Punkt im Umgebungsraum (dem Eingaberaum) seinem Embedding zugewiesen wird.

Beim Manifold Learning geht es in erster Linie um Verfahren für unüberwachtes Lernen, mit denen diese Mannigfaltigkeiten erfasst werden sollen. Der Großteil der ursprünglichen Machine-Learning-Forschung hinsichtlich des Erlernens nichtlinearer Mannigfaltigkeiten hat sich auf **nichtparametrische** Verfahren mittels **Nearest-Neighbor-Graphen** konzentriert. Ein solcher Graph hat einen Knoten für jedes Trainingsbeispiel und Kanten, die die nächsten Nachbarn miteinander verbinden. Diese Verfahren (*Schölkopf et al., 1998; Roweis und Saul, 2000; Tenenbaum et al., 2000; Brand, 2003; Belkin und Niyogi, 2003; Donoho und Grimes, 2003; Weinberger und Saul, 2004; Hinton und Roweis, 2003; van der Maaten und Hinton, 2008*)

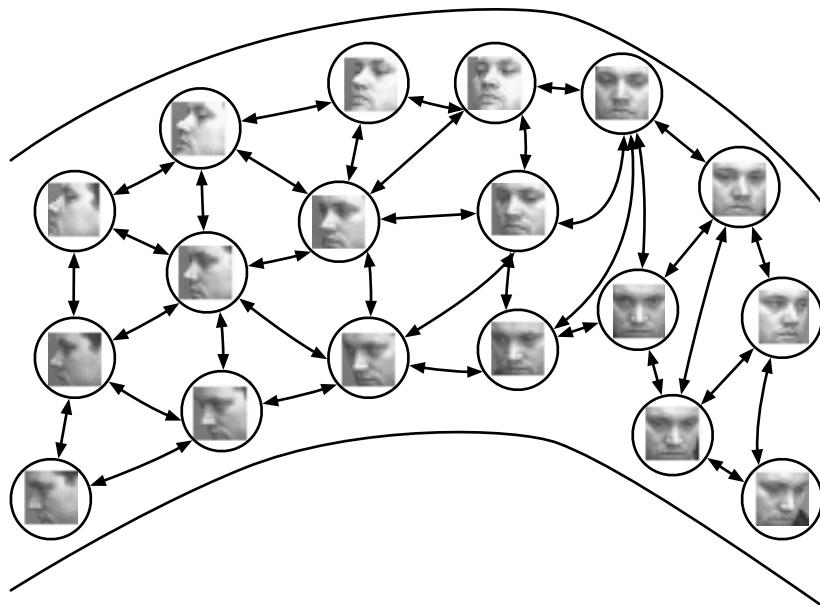


Abbildung 14.8: Nichtparametrische Manifold-Learning-Verfahren erstellen einen Nearest-Neighbor-Graphen, in dem Knoten die Trainingsbeispiele und gerichtete Kanten die Nearest-Neighbor-Beziehungen darstellen. Unterschiedliche Verfahren können somit die Tangentialebene einer Umgebung des Graphen sowie ein Koordinatensystem ermitteln, das jedes Trainingsbeispiel mit einer reellwertigen Vektorposition oder einem **Embedding** verknüpft. Es ist möglich, eine solche Repräsentation für neue Beispiele zu generalisieren, und zwar mittels Interpolation. Sofern die Anzahl der Beispiele groß genug ist, um die Krümmung und die Windungen der Mannigfaltigkeit abzudecken, funktionieren diese Ansätze gut. Bilder aus dem QMUL-Multiview-Face-Datensatz (Gong et al., 2000).

verknüpfen jeden Knoten mit einer Tangentialebene, die die Richtungen der Variationen aufspannt, die mit den Differenzvektoren zwischen dem Beispiel und seinen Nachbarn verknüpft sind (vgl. Abbildung 14.8).

Anschließend lässt sich ein globales Koordinatensystem mittels Optimierung oder durch Lösen eines linearen Systems ermitteln. Abbildung 14.9 zeigt, wie eine Mannigfaltigkeit aus einer Vielzahl lokaler linearer nahezu normalverteilter Bereiche (oder »Pfannkuchen«, da die Normalverteilungen in den Tangentenrichtungen flach sind) zusammengesetzt werden kann.

Ein fundamentales Problem bei diesen lokalen nichtparametrischen Herangehensweisen an das Manifold Learning wird in *Bengio und Monperrus* (2005) angesprochen: Wenn die Mannigfaltigkeiten nicht sehr glatt sind (also viele Höhen, Tiefen und Wendungen aufweisen), werden eventuell sehr

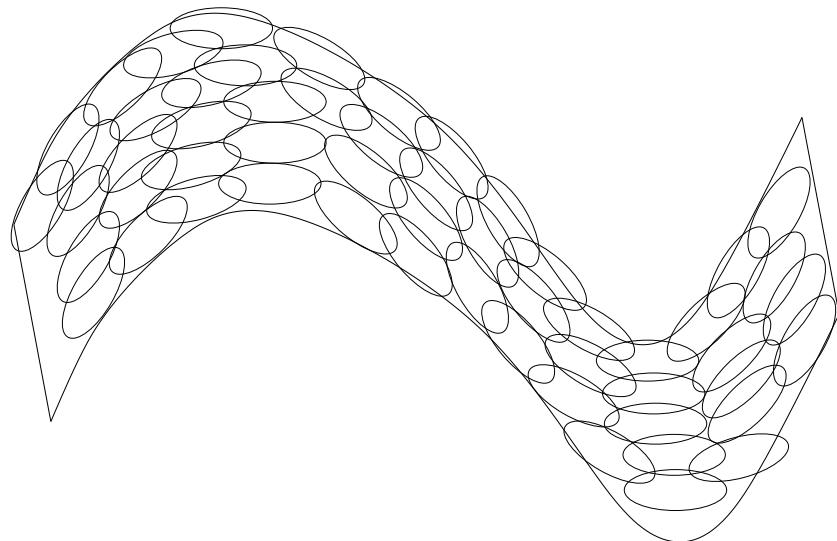


Abbildung 14.9: Wenn die Tangentialebenen (vgl. Abbildung 14.6) an jeder Stelle bekannt sind, können sie zu einem globalen Koordinatensystem oder einer Dichtefunktion verbunden werden. Jeder lokale Bereich ist eine Art lokales euklidisches Koordinatensystem oder eine lokale flache Normalverteilung oder ein »Pfannkuchen« mit einer sehr geringen Varianz in den Richtungen orthogonal zum Pfannkuchen und einer sehr großen Varianz in den Richtungen, die das Koordinaten- system auf dem Pfannkuchen definieren. Eine Mischung dieser Normalverteilungen ergibt eine Dichteschätzung(sfunktion) wie im Parzen-Fenster-Algorithmus für Mannigfaltigkeiten (engl. *manifold Parzen window algorithm*) (Vincent und Bengio, 2003) oder der nichtlokalen Variante auf Basis eines neuronalen Netzes (Bengio et al., 2006c).

viele Trainingsbeispiele benötigt, um jede dieser Variationen abzudecken, ohne dass die Chance besteht, verborgene Variationen zu generalisieren. Tatsächlich können diese Verfahren die Form der Mannigfaltigkeit nur durch Interpolieren zwischen benachbarten Beispielen generalisieren. Leider können Mannigfaltigkeiten im KI-Kontext sehr komplexe Strukturen aufweisen, die sich alleine durch lokale Interpolation kaum erfassen lassen. Betrachten Sie das Beispiel der Mannigfaltigkeit, die aus der Verschiebung in Abbildung 14.6 entsteht. Wenn wir nur eine Koordinate im Eingabevektor, x_i , betrachten, während das Bild verschoben wird, stellen wir fest, dass eine Koordinate für jedes Hoch oder jedes Tief der Helligkeit im Bild ein Hoch oder eine Tief in ihrem eigenen Wert aufweist. Anders ausgedrückt: Die Komplexität der Helligkeitsmuster in einer zugrunde liegenden Bildvorlage steuert die Komplexität der Mannigfaltigkeiten, die aus einfachen Bildtransformationen

entstehen. Das fördert den Einsatz von verteilten Repräsentationen und Deep Learning zum Erfassen der Mannigfaltigkeitsstruktur.

14.7 Contractive Autoencoder

Der Contractive Autoencoder (dt. *kontrahierender Autoencoder*, CAE) (*Rifai et al.*, 2011a,b) führt einen expliziten Regularisator für den Code $\mathbf{h} = f(\mathbf{x})$ ein, der auf möglichst kleine Ableitungen von f abzielt:

$$\Omega(\mathbf{h}) = \lambda \left\| \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right\|_F^2. \quad (14.18)$$

Der Strafterm $\Omega(\mathbf{h})$ ist das Quadrat der Frobenius-Norm (Summe der Quadrate ihrer Elemente) der Jacobi-Matrix der partiellen Ableitungen der Encoder-Funktion.

Es gibt eine Verbindung zwischen dem Denoising Autoencoder und dem Contractive Autoencoder: *Alain und Bengio* (2013) haben gezeigt, dass im Grenzbereich des kleinen normalverteilten Eingaberauschens der Rekonstruktionsfehler beim Denoising einem kontrahierenden Strafterm für die Rekonstruktionsfunktion gleichwertig ist, die \mathbf{x} und $\mathbf{r} = g(f(\mathbf{x}))$ einander zuordnet. Mit anderen Worten: Denoising Autoencoder führen dazu, dass die Rekonstruktionsfunktion gegenüber kleinen aber betragsmäßig endlichen Störungen der Eingabe robust ist, wohingegen Contractive Autoencoder dazu führen, dass die Funktion zur Merkmalsextraktion gegenüber infinitesimalen Störungen der Eingabe robust ist. Wird der kontrahierende Strafterm auf Basis der Jacobi-Matrix für das Pretraining der Merkmale $f(\mathbf{x})$ für die Verwendung mit einem Klassifikator eingesetzt, resultiert die beste Korrektklassifikationsrate normalerweise aus der Anwendung des kontrahierenden Strafterms auf $f(\mathbf{x})$, nicht auf $g(f(\mathbf{x}))$. Ein kontrahierender Strafterm auf $f(\mathbf{x})$ weist auch enge Verbindungen zum Score Matching auf (siehe Abschnitt 14.5.1).

Die Bezeichnung **kontrahierend** (engl. *contractive*) erinnert an die Art, auf die der CAE den Raum krümmt. Da der CAE so trainiert wird, dass er robust gegenüber Störungen seiner Eingabe ist, wird er dazu animiert, eine Umgebung der Eingabepunkte auf eine kleinere Umgebung der Ausgabepunkte abzubilden. Sie können sich dies als Kontrahieren der Eingabe-Umgebung zu einer kleineren Ausgabe-Umgebung vorstellen.

Genauer gesagt, ist der CAE lediglich lokal kontrahierend – sämtliche Störungen eines Trainingspunkts \mathbf{x} werden in der Nähe von $f(\mathbf{x})$ abgebildet. Global werden die zwei unterschiedlichen Punkte \mathbf{x} und \mathbf{x}' den Punkten $f(\mathbf{x})$

und $f(\mathbf{x}')$ zugeordnet, die einen größeren Abstand zueinander aufweisen als die ursprünglichen Punkte. Es ist plausibel, dass f zwischen den oder weit abseits der Datenmannigfaltigkeiten expandieren kann (vgl. das simple 1-D-Beispiel in Abbildung 14.7). Wenn der Strafterm $\Omega(\mathbf{h})$ auf sigmoide Einheiten angewandt wird, lässt sich die Jacobi-Matrix leicht verkleinern, wenn die sigmoiden Einheiten auf 0 oder 1 sättigen. Das animiert den CAE dazu, Eingabepunkte mit extremen Werten der Sigmoidfunktion, die als Binärcode interpretiert werden kann, zu codieren. Es sorgt auch dafür, dass der CAE seine Code-Werte im Großteil des Hyperwürfels verteilt, den seine sigmoiden verdeckten Einheiten aufspannen können.

Sie können sich vorstellen, dass die Jacobi-Matrix \mathbf{J} in einem Punkt \mathbf{x} den nichtlinearen Encoder $f(\mathbf{x})$ als linearen Operator approximiert. So können wir den Begriff »kontrahierend« formaler nutzen. In der Theorie linearer Operatoren gilt ein linearer Operator dann als kontrahierend, wenn die Norm von $\mathbf{J}\mathbf{x}$ kleiner als oder gleich 1 bleibt für alle \mathbf{x} mit Norm gleich 1. Anders ausgedrückt: \mathbf{J} ist kontrahierend, wenn es die Einheitskugel verkleinert. Sie können sich vorstellen, dass der CAE die Frobenius-Norm der lokalen linearen Approximation von $f(\mathbf{x})$ in jedem Trainingspunkt \mathbf{x} bestraft, damit jeder dieser lokal linearen Operatoren zu einer Kontraktion wird.

Wie in Abschnitt 14.6 beschrieben, erlernen regularisierte Autoencoder Mannigfaltigkeiten durch das Abwägen einander widerstrebender Kräfte. Im Falle des CAEs handelt es sich bei diesen Kräften um den Rekonstruktionsfehler und den kontrahierenden Strafterm $\Omega(\mathbf{h})$. Der Rekonstruktionsfehler allein würde dazu anregen, dass der CAE eine identische Abbildung erlernt. Der kontrahierende Strafterm allein würde dazu animieren, dass der CAE Merkmale erlernt, die bezüglich \mathbf{x} konstant sind. Der Kompromiss zwischen diesen beiden Kräften resultiert in einem Autoencoder, dessen Ableitungen $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ größtenteils winzig sind. Nur eine geringe Anzahl verdeckter Einheiten – was einer geringen Anzahl von Richtungen in der Einheit entspricht – dürfen bedeutende Ableitungen aufweisen.

Das Ziel des CAEs ist es, die Mannigfaltigkeitsstruktur der Daten zu erlernen. Richtungen \mathbf{x} mit großem $\mathbf{J}\mathbf{x}$ verursachen schnelle Änderungen von \mathbf{h} , sodass es sich dabei wahrscheinlich um Richtungen handelt, die die Tangentialebenen der Mannigfaltigkeit approximieren. Experimente von Rifai *et al.* (2011a,b) zeigen, dass ein Training des CAEs zu hauptsächlich singulären Werten für \mathbf{J} führt, deren Größe unter 1 fällt, sodass sie kontrahierend werden. Einige Singulärwerte bleiben jedoch größer als 1, da der Strafterm für den Rekonstruktionsfehler den CAE dazu animiert, die Richtungen mit der größten lokalen Varianz zu codieren. Die Richtungen,

die den größten Singulärwerten entsprechen, werden als Tangentenrichtungen interpretiert, die vom kontrahierenden Autoencoder erlernt wurden. Idealerweise sollten diese Tangentenrichtungen tatsächlichen Variationen in den Daten entsprechen. Ein Beispiel: Ein für Bilder eingesetzter CAE sollte Tangentenvektoren erlernen, die zeigen, wie sich das Bild ändert, wenn Objekte darin nach und nach ihre Position verändern (vgl. Abbildung 14.6). Visualisierungen der experimentell ermittelten Singulärvektoren scheinen mit aussagekräftigen Transformationen des Eingabebildes übereinzustimmen (vgl. Abbildung 14.10).

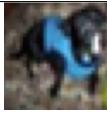
Eingabe	Tangentenvektoren
	
	Lokale PCA (kein Parameter Sharing über Bereiche hinweg)
	
	Contractive Autoencoder (CAE)

Abbildung 14.10: Darstellung der Tangentenvektoren der mittels lokaler PCA und Contractive Autoencoder geschätzten Mannigfaltigkeit. Die Lage der Mannigfaltigkeit wird über das Eingabebild eines Hundes (aus dem CIFAR-10-Datensatz) definiert. Die Tangentenvektoren werden anhand der führenden Singulärvektoren der Jacobi-Matrix $\frac{\partial h}{\partial x}$ der Eingabe-zu-Code-Zuordnung geschätzt. Obwohl sowohl lokale PCA als auch CAE lokale Tangenten erfassen können, kann der CAE exaktere Schätzungen aus eingeschränkten Trainingsdaten bilden, da er ein Parameter Sharing über unterschiedliche Positionen nutzt, die eine gemeinsame Teilmenge aktiver verdeckter Einheiten aufweisen. Die CAE-Tangentenrichtungen entsprechen üblicherweise sich bewegenden oder sich ändernden Teilen des Objekts (wie dem Kopf oder den Beinen). (Abbildung mit freundlicher Genehmigung von *Rifai et al.* (2011c) wiedergegeben)

Ein Problem in der praktischen Umsetzung beim CAE-Regularisierungskriterium ist, dass es zwar für einen Autoencoder mit einer einzigen verdeckten Schicht ohne großen Aufwand berechnet werden kann, dies aber bei tieferen Autoencodern nicht mehr der Fall ist. Die von *Rifai et al.* (2011a) verfolgte Strategie besteht darin, eine Reihe einschichtiger Autoencoder getrennt voneinander so zu trainieren, dass jeder davon die verdeckte Schicht des vorhergehenden Autoencoders rekonstruiert. Die Zusammen-

stellung dieser Autoencoder bildet dann einen tiefen Autoencoder. Da jede Schicht separat als lokal kontrahierend trainiert wurde, ist auch der tiefe Autoencoder kontrahierend. Das Ergebnis unterscheidet sich von dem eines gemeinsamen Trainings der gesamten Architektur mit einem Strafterm für die Jacobi-Matrix des tiefen Modells, aber es erfasst viele der gewünschten qualitativen Eigenschaften.

Ein weiteres Problem in der praktischen Umsetzung besteht darin, dass der kontrahierende Strafterm unbrauchbare Ergebnisse erzielen könnte, wenn wir den Decoder ohne irgendeinen Maßstab arbeiten lassen. Der Encoder könnte zum Beispiel die Eingabe mit einer kleinen Konstanten ϵ multiplizieren, während der Decoder den Code durch ϵ teilt. Wenn ϵ gegen 0 geht, drängt der Encoder den kontrahierenden Strafterm $\Omega(\mathbf{h})$ in Richtung 0, ohne irgendetwas über die Verteilung gelernt zu haben. In der Zwischenzeit erreicht der Decoder eine perfekte Rekonstruktion. In *Rifai et al.* (2011a) wird dies durch Verknüpfen der Gewichte von f und g verhindert. Sowohl f als auch g sind standardmäßige Schichten des neuronalen Netzes, die aus affinen Transformationen gefolgt von einer elementweisen Nichtlinearität bestehen; somit ist es ganz normal, die Gewichtungsmatrix von g zur Transponierten der Gewichtungsmatrix von f zu machen.

14.8 Prädiktive dünnbesetzte Zerlegung

Prädiktive dünnbesetzte Zerlegung (engl. *predictive sparse decomposition*, PSD) ist ein Modell, in dem Sparse Coding und parametrische Autoencoder kombiniert werden (*Kavukcuoglu et al.*, 2008). Ein parametrischer Encoder wird darauf trainiert, die Ausgabe der iterativen Inferenz vorherzusagen. PSD wurde für unüberwachtes Lernen von Merkmalen (engl. *unsupervised feature learning*) im Rahmen der Objekterkennung in Bildern und Videos (*Kavukcuoglu et al.*, 2009, 2010; *Jarrett et al.*, 2009; *Farabet et al.*, 2011) sowie für Audiodaten (*Henaff et al.*, 2011) verwendet. Das Modell besteht aus einem Encoder $f(\mathbf{x})$ und einem Decoder $g(\mathbf{h})$ – beide sind parametrisiert. Während des Trainings wird \mathbf{h} über den Optimierungsalgorithmus gesteuert. Das Training läuft weiter durch Minimierung von

$$\|\mathbf{x} - g(\mathbf{h})\|^2 + \lambda|\mathbf{h}|_1 + \gamma\|\mathbf{h} - f(\mathbf{x})\|^2. \quad (14.19)$$

Wie beim Sparse Coding wechselt der Trainingsalgorithmus zwischen der Minimierung bezüglich \mathbf{h} und bezüglich der Modellparameter. Die Minimierung bezüglich \mathbf{h} ist schnell, da $f(\mathbf{x})$ einen guten Startwert für \mathbf{h} liefert und die Kostenfunktion \mathbf{h} ohnehin auf das Umfeld von $f(\mathbf{x})$ einschränkt. Das

einfache Gradientenabstiegsverfahren kann sinnvolle Werte für \mathbf{h} bereits nach nur zehn Schritten ermitteln.

Das Trainingsverfahren der PSD unterscheidet sich vom Training eines Modells mit Sparse Coding und anschließendem Training von $f(\mathbf{x})$ zur Vorhersage der Werte für Merkmale mit Sparse Coding. Das PSD-Trainingsverfahren regularisiert den Decoder auf die Nutzung der Parameter, für die $f(\mathbf{x})$ auf gute Code-Werte schließen kann.

Das prädiktive Sparse Coding ist ein Beispiel für eine **erlernte approximative Inferenz**. In Abschnitt 19.5 wird dieses Thema vertieft. Die in Kapitel 19 vorgestellten Werkzeuge zeigen deutlich, dass die PSD betrachtet werden kann als Trainieren eines gerichteten probabilistischen Modells mit Sparse Coding durch Maximieren einer unteren Schranke für die Log-Likelihood des Modells.

In praktischen Anwendungen der PSD wird die iterative Optimierung nur während des Trainings verwendet. Der parametrische Encoder f wird zum Berechnen der erlernten Merkmale genutzt, wenn das Modell eingesetzt wird. Das Berechnen von f erfordert wenig Aufwand im Vergleich zum Schließen auf \mathbf{h} mittels Gradientenabstiegsverfahren. Da f eine differenzierbare parametrische Funktion ist, können PSD-Modelle gestapelt und zum Initialisieren eines tiefen Netzes verwendet werden, das mit einem anderen Kriterium trainiert wird.

14.9 Anwendungen für Autoencoder

Autoencoder wurden und werden erfolgreich zur Dimensionsreduktion und zur Informationsgewinnung verwendet. Die Dimensionsreduktion war eine der ersten Anwendungen im Representation Learning und Deep Learning. Sie gehörte zu den frühen Motiven für die Beschäftigung mit Autoencodern. So trainierten *Hinton und Salakhutdinov* (2006) einen Stapel von RBMs und nutzten anschließend deren Gewichte zur Initialisierung eines tiefen Autoencoders mit allmählich kleiner werdenden verdeckten Schichten bis hin zu einem Flaschenhals mit 30 Einheiten. Der resultierende Code wies einen geringeren Rekonstruktionsfehler auf als eine PCA in 30 Dimensionen und die erlernte Repräsentation war qualitativ einfacher auszuwerten und in Bezug zu den zugrunde liegenden Kategorien zu setzen, wobei diese Kategorien als gut separierte Cluster vorlagen.

Geringer-dimensionale Repräsentationen können die Leistung bei vielen Aufgaben steigern, unter anderem bei der Klassifizierung. Modelle kleinerer Räume benötigen weniger Speicher und Laufzeit. Viele Formen der Dimen-

sionsreduktion platzieren semantisch verwandte Beispiele nahe beieinander, wie *Salakhutdinov und Hinton* (2007b) und *Torralba et al.* (2008) beobachtet haben. Die aus der Abbildung auf den geringer-dimensionalen Raum gewonnenen Hinweise helfen bei der Generalisierung.

Eine Aufgabe, die stärker als gewöhnlich von der Dimensionsreduktion profitiert, ist die **Informationsgewinnung**, also das Suchen nach Einträgen in einer Datenbank, die einer Suchabfrage ähneln. Diese Aufgabe nutzt die üblichen Vorteile der Dimensionsreduktion, die auch andere Aufgaben nutzen, und profitiert zusätzlich davon, dass die Suche in bestimmten Arten geringdimensionaler Räume besonders effizient werden kann. Insbesondere beim Trainieren des Algorithmus zur Dimensionsreduktion mit dem Ziel eines geringdimensionalen und *binären* Codes können wir alle Datenbankeinträge in einer Hash-Tabelle speichern, die binäre Code-Vektoren auf Einträge abbildet. Diese Hash-Tabelle ermöglicht die Informationsgewinnung durch Ausgeben aller Datenbankeinträge, die denselben Binärkode wie die Abfrage aufweisen. Wir können auch sehr effizient in geringfügig weniger ähnlichen Einträgen suchen, indem wir die einzelnen Bits aus der Abfrage-Codierung flippen. Dieser Ansatz zur Informationsgewinnung mittels Dimensionsreduktion und Binärisierung wird als **semantisches Hashing** (*Salakhutdinov und Hinton*, 2007b, 2009b) bezeichnet und wurde sowohl auf Texteingaben (*Salakhutdinov und Hinton*, 2007b, 2009b) als auch auf Bilder (*Torralba et al.*, 2008; *Weiss et al.*, 2008; *Krizhevsky und Hinton*, 2011) angewandt.

Um Binärcodes für ein semantisches Hashing zu erzeugen, verwenden wir üblicherweise eine Encoder-Funktion mit Sigmoiden in der letzten Schicht. Die sigmoiden Einheiten müssen so trainiert werden, dass sie für alle Eingabewerte auf fast 0 oder fast 1 sättigen. Ein Trick hierfür besteht darin, während des Trainings einfach additives Rauschen direkt vor der sigmoiden Nichtlinearität einzuspeisen. Die Stärke des Rauschens muss im Laufe der Zeit zunehmen. Um das Rauschen zu bekämpfen und möglichst viele Informationen zu erhalten, muss das Netz die Größe der Eingaben für die Sigmoidfunktion erhöhen, bis es zur Sättigung kommt.

Das Konzept des Erlernens einer Hashing-Funktion wurde in mehreren Richtungen weiter erforscht, darunter als Training der Repräsentationen zur Optimierung eines Verlusts, der direkter mit der Aufgabe zur Suche nahegelegener Beispiele in der Hash-Tabelle verknüpft ist (*Norouzi und Fleet*, 2011).

15

Representation Learning

In diesem Kapitel erklären wir zunächst, was es bedeutet, Repräsentationen zu erlernen, und wie der Begriff der Repräsentation beim Design tiefer Architekturen hilft. Wir untersuchen, wie Lernalgorithmen statistisches Wissen weitergeben können, um dieses in verschiedenen Aufgaben zu nutzen, darunter zum Beispiel die Verwendung von Informationen aus unüberwachten Aufgaben in überwachten Aufgaben. Gemeinsam genutzte Repräsentationen sind nützlich beim Umgang mit mehreren Modalitäten oder Domänen sowie zum Übertragen des erlernten Wissens an Aufgaben, für die es zwar wenige oder gar keine Beispiele gibt, dafür aber eine Repräsentation der Aufgaben (engl. *task representation*). Schließlich widmen wir uns den Gründen für den Erfolg des Representation Learnings, angefangen bei den theoretischen Vorteilen verteilter Repräsentationen (*Hinton et al.*, 1986) und tiefer Repräsentationen bis hin zu dem allgemeinen Konzept der zugrunde liegenden Annahmen über den datengenerierenden Prozess, insbesondere die zugrunde liegenden Ursachen der beobachteten Daten.

Viele Aufgaben zur Informationsverarbeitung können sich abhängig von der Repräsentation der Informationen als sehr einfach oder sehr schwierig erweisen. Das gilt grundsätzlich überall – in der Informatik generell und eben auch im Machine Learning. So stellt es für Sie vermutlich kein Problem dar, 210 schriftlich durch 6 zu teilen. Die Aufgabe wird aber beträchtlich komplizierter, wenn sie mit römischen Ziffern gestellt wird. Die meisten von uns würden, wenn sie gebeten werden, CCX durch VI zu teilen, wohl diese Zahlen zunächst in arabische Ziffern umwandeln, um die schriftliche Division im vertrauten Stellenwertsystem durchzuführen. Auf den konkreten Fall bezogen: Wir können die asymptotische Laufzeit unterschiedlicher Operationen anhand angemessener oder nicht angemessener Repräsentationen

beziffern. So ist das Einfügen einer Zahl an der korrekten Stelle in einer sortierten Zahlenliste eine $O(n)$ -Operation, wenn es sich um eine verkettete Liste handelt, aber nur eine $O(\log n)$ -Operation, wenn es sich um einen Rot-Schwarz-Baum handelt.

Im Machine-Learning-Kontext stellt sich also die Frage, wodurch eine Repräsentation besser als eine andere ist. Allgemein ausgedrückt, ist jede Repräsentation gut, die die darauf folgende Lernaufgabe vereinfacht. Die Wahl der Repräsentation richtet sich meist nach der Wahl der darauf folgenden Lernaufgabe.

Wir können uns Feedforward-Netze, die mittels überwachtem Lernen trainiert werden, als eine Art Representation Learning vorstellen. Die letzte Schicht des Netzes ist meist ein linearer Klassifikator, zum Beispiel der softmax-Regressionsklassifikator. Der Rest des Netzes lernt, diesem Klassifikator eine Repräsentation zu liefern. Das Trainieren mit einem überwachten Kriterium führt gewöhnlich dazu, dass die Repräsentation in jeder verdeckten Schicht (aber ganz besonders in der Nähe der obersten verdeckten Schicht) Eigenschaften annimmt, die die Klassifizierungsaufgabe erleichtern. Zum Beispiel können Klassen, die in den Eingabemerkmale nicht linear separierbar waren, in der letzten verdeckten Schicht linear separierbar werden. Prinzipiell könnte es sich bei der letzten Schicht um eine andere Art von Modell handeln, zum Beispiel einen Nearest-Neighbor-Klassifikator (*Salakhutdinov und Hinton, 2007a*). Die Merkmale in der vorletzten Schicht sollten je nach Art der letzten Schicht andere Eigenschaften erlernen.

Das überwachte Training von Feedforward-Netzen umfasst keine explizite Vorgabe bestimmter Bedingungen für die erlernten Zwischenmerkmale. Andere Arten von Representation-Learning-Algorithmen werden häufig explizit für eine bestimmte Form der Repräsentation aufgebaut. Ein Beispiel: Es soll eine Repräsentation erlernt werden, die die Dichteschätzung erleichtert. Verteilungen mit mehr Unabhängigkeiten sind einfacher zu modellieren. Wir könnten also eine Zielfunktion entwerfen, die die Elemente des Repräsentationsvektors \mathbf{h} animiert, unabhängig zu sein. Wie überwachte Netze haben auch unüberwachte Deep-Learning-Algorithmen ein primäres Trainingsziel und erlernen eine Repräsentation quasi nebenbei. Ungeachtet dessen, wie eine Repräsentation ermittelt wurde, lässt sie sich für andere Aufgaben nutzen. Es können auch mehrere Aufgaben (einige überwacht, einige unüberwacht) gleichzeitig mit einer gemeinsamen internen Repräsentation erlernt werden.

Die meisten Probleme des Representation Learnings bedeuten einen Kompromiss zwischen dem Erhalt möglichst vieler Informationen über die

Eingabe und dem Erlangen guter Eigenschaften (zum Beispiel Unabhängigkeit).

Representation Learning ist vor allem deshalb interessant, weil es eine Möglichkeit zum unüberwachten und halb-überwachten Lernen bietet. Wir verfügen oft über sehr große Mengen von Trainingsdaten, die nicht mit einem Label gekennzeichnet sind, und über relativ wenige mit Labels gekennzeichnete Trainingsdaten. Das Training mit überwachten Lernmethoden anhand einer Teilmenge der Datensätze ohne Label führt oft zu einer starken Überanpassung. Halb-überwachtes Lernen bietet die Möglichkeit, dieses Überanpassungsproblem zu lösen, indem auch die Daten ohne Label zum Lernen verwendet werden. Wir können vor allem gute Repräsentationen für die Daten ohne Label erlernen und diese Repräsentationen dann zum Lösen der überwachten Lernaufgabe nutzen.

Menschen und Tiere können schon mit sehr wenigen mit Labels gekennzeichneten Beispielen lernen. Wir wissen noch nicht, wie das möglich ist. Viele Faktoren könnten die gesteigerte menschliche Leistung erklären – vielleicht verwendet das Gehirn sehr große Klassifikator-Ensembles oder bayessche Inferenzverfahren. Eine beliebte Hypothese ist, dass das Gehirn in der Lage ist, unüberwachtes oder halb-überwachtes Lernen zu nutzen. Es gibt viele Möglichkeiten zur Nutzung von Daten ohne Label. In diesem Kapitel konzentrieren wir uns auf die Hypothese, dass die Daten ohne Label zum Erlernen einer guten Repräsentation verwendet werden können.

15.1 Schichtweises unüberwachtes Pretraining mit Greedy-Algorithmen

Das unüberwachte Lernen spielte eine wesentliche historische Rolle für das Wiederaufleben tiefer neuronaler Netze, denn es versetzte die Forscher erstmals in die Lage, ein tiefes überwachtes Netz ohne besondere architektonische Spezialisierungen wie Faltung (engl. *convolution*) oder Rekurrenz zu trainieren. Wir nennen dieses Verfahren **unüberwachtes Pretraining**, genauer **schichtweises unüberwachtes Pretraining mit Greedy-Algorithmen**. Dieses Verfahren ist ein klassisches Beispiel dafür, wie die für eine Aufgabe (unüberwachtes Lernen, mit dem Versuch, die Form der Eingabeverteilung zu erfassen) erlernte Repräsentation manchmal für eine andere Aufgabe (überwachtes Lernen mit demselben Eingabebereich) nützlich sein kann.

Das schichtweise unüberwachte Pretraining mit Greedy-Algorithmen nutzt einen einschichtigen Representation-Learning-Algorithmus wie eine

Restricted Boltzmann Machine (RBM), einen einschichtigen Autoencoder, ein dünnbesetztes Modell oder ein anderes Modell, das latente Repräsentationen erlernt. Jede Schicht erhält ein Pretraining mittels unüberwachtem Lernen, wobei die Ausgabe der vorhergehenden Schicht verwendet wird, um eine neue Repräsentation der Daten auszugeben, deren Verteilung (oder deren Beziehung zu anderen Variablen wie den vorherzusagenden Kategorien) hoffentlich einfacher ist. Algorithmus 15.1 zeigt eine formale Beschreibung.

Algorithmus 15.1 Protokoll für das schichtweise unüberwachte Pretraining mit Greedy-Algorithmen

Gegeben sind: Algorithmus für unüberwachtes Lernen von Merkmalen (engl. *unsupervised feature learning*) \mathcal{L} , der eine Trainingsdatenmenge mit Beispielen verwendet, um eine Encoder- oder Merkmalsfunktion f zurückzugeben. Die Rohdaten der Eingabe sind \mathbf{X} , wobei jedes Beispiel eine Zeile belegt, und $f^{(1)}(\mathbf{X})$ ist die Ausgabe des Encoders der ersten Phase für \mathbf{X} . Wenn eine Feinabstimmung (engl. *fine-tuning*) durchgeführt wird, verwenden wir einen Klassifikator \mathcal{T} , der eine anfängliche Funktion f und Eingabebeispiele \mathbf{X} (sowie bei der überwachten Feinabstimmung auch zugehörige Zielwerte \mathbf{Y}) nutzt und sodann die abgestimmte Funktion ausgibt. Die Anzahl der Phasen ist m .

```
 $f \leftarrow$  identische Abbildung  
 $\tilde{\mathbf{X}} = \mathbf{X}$   
for  $k = 1, \dots, m$  do  
   $f^{(k)} = \mathcal{L}(\tilde{\mathbf{X}})$   
   $f \leftarrow f^{(k)} \circ f$   
   $\tilde{\mathbf{X}} \leftarrow f^{(k)}(\tilde{\mathbf{X}})$   
end for  
if Feinabstimmung then  
   $f \leftarrow \mathcal{T}(f, \mathbf{X}, \mathbf{Y})$   
end if  
Return  $f$ 
```

Schichtweise Trainingsverfahren mit Greedy-Algorithmen auf Basis unüberwachter Kriterien werden schon lange Zeit eingesetzt, um die Schwierigkeit eines gemeinsamen Trainings der Schichten tiefer neuronaler Netze für überwachte Aufgaben zu umgehen. Dieser Ansatz geht auf das Neocognitron zurück (Fukushima, 1975), vielleicht sogar noch weiter. Die Renaissance des Deep Learnings im Jahr 2006 begann mit der Entdeckung, dass dieses Lernverfahren mit Greedy-Algorithmen dazu dienen könnte, einen guten Startpunkt für ein gemeinsames Lernverfahren über alle Schichten zu finden, und dass dieser Ansatz genutzt werden könnte, um sogar vollständig

verbundene Architekturen zu trainieren (*Hinton et al.*, 2006; *Hinton und Salakhutdinov*, 2006; *Hinton*, 2006; *Bengio et al.*, 2007; *Ranzato et al.*, 2007a). Vor dieser Entdeckung nahm man an, dass nur tiefe CNNs oder Netze, deren Tiefe das Ergebnis von Rekurrenz war, trainiert werden könnten. Heute wissen wir, dass das schichtweise Pretraining mit Greedy-Algorithmen nicht zum Trainieren vollständig verbundener tiefer Architekturen benötigt wird, aber der Ansatz zum unüberwachten Pretraining war die erste erfolgreiche Methode.

Das schichtweise Pretraining wird **greedy** (also »gierig«) genannt, weil es sich um einen **Greedy-Algorithmus** handelt, der jeden Teil der Lösung unabhängig optimiert – ein Teil nach dem anderen –, und nicht etwa alle Teile gemeinsam. Die Bezeichnung **schichtweise** (engl. *layer-wise*) gibt an, dass diese unabhängigen Teile die Schichten des Netzes sind. Tatsächlich erfolgt das schichtweise Pretraining mit Greedy-Algorithmen Schicht für Schicht; die k -te Schicht wird also trainiert, ohne die vorherigen Schichten zu verändern. Insbesondere werden die tieferen Schichten (die zuerst trainiert werden) nicht angepasst, nachdem die oberen Schichten eingeführt sind. Die Bezeichnung **unüberwacht** gibt an, dass jede Schicht mit einem unüberwachten Representation-Learning-Algorithmus trainiert wird. Und **Pretraining** gibt an, dass es sich hierbei nur um einen ersten Schritt handelt, der einem gemeinsamen Trainingsalgorithmus zur **Feinabstimmung** aller Schichten gemeinsam vorgeschaltet ist. Im Rahmen einer überwachten Lernaufgabe kann es als Regularisierer (in einigen Experimenten vermindert das Pretraining den Testfehler, ohne den Trainingsfehler zu vermindern) und eine Art der Parameterinitialisierung betrachtet werden.

Das Wort »Pretraining« bezeichnet häufig nicht nur die Phase des Pretrainings selbst, sondern das gesamte Protokoll, das die beiden Phasen »Pretraining« und »überwachtes Lernen« zusammenfasst. Beim überwachten Lernen kann es sich um das Trainieren eines einfachen Klassifikators zusätzlich zu den während des Pretrainings erlernten Merkmalen handeln, es kann aber auch um eine überwachte Feinabstimmung des gesamten im Pretraining erlernten Netzes gehen. Ungeachtet der Art des Algorithmus für unüberwachtes Lernen oder des verwendeten Modelltyps läuft das Training in den meisten Fällen nahezu identisch ab. Obwohl die Wahl des Algorithmus für unüberwachtes Lernen natürlich einige Details beeinflusst, folgt das unüberwachte Pretraining in den meisten Fällen diesem grundlegenden Protokoll.

Das schichtweise unüberwachte Pretraining mit Greedy-Algorithmen kann auch zur Initialisierung für andere Algorithmen für unüberwachtes Lernen verwendet werden, zum Beispiel für tiefe Autoencoder (*Hinton und*

(*Salakhutdinov*, 2006) und probabilistische Modelle mit vielen Schichten latenter Variablen. Dazu gehören Deep-Belief-Netze (*Hinton et al.*, 2006) und DBMs (*Salakhutdinov und Hinton*, 2009a). Diese tiefen generativen Modelle werden in Kapitel 20 beschrieben.

Wie Sie in Abschnitt 8.7.4 gesehen haben, ist auch ein schichtweises, *überwachtes* Pretraining mit Greedy-Algorithmen möglich. Dies beruht auf der Annahme, dass sich ein flaches Netz leichter als ein tiefes Netz trainieren lässt. Diese Annahme scheint für mehrere Zusammenhänge bestätigt (*Erhan et al.*, 2010).

15.1.1 Wann und warum funktioniert ein unüberwachtes Pretraining?

Bei vielen Aufgaben kann das schichtweise unüberwachte Pretraining mit Greedy-Algorithmen zu deutlichen Verbesserungen des Testfehlers für Klassifizierungsaufgaben führen. Diese Beobachtung war für das erneuerte Interesse an tiefen neuronalen Netzen ab 2006 verantwortlich (*Hinton et al.*, 2006; *Bengio et al.*, 2007; *Ranzato et al.*, 2007a). Bei vielen anderen Aufgaben bietet ein unüberwachtes Pretraining jedoch keine Vorteile oder schadet sogar. *Ma et al.* (2015) haben die Auswirkung des Pretrainings auf Machine-Learning-Modelle für die Vorhersage chemischer Aktivität untersucht und festgestellt, dass das Pretraining im Mittel von leichtem Nachteil war, für viele Aufgaben aber bemerkenswert hilfreich. Da ein unüberwachtes Pretraining manchmal hilfreich, doch häufig auch nachteilig ist, müssen Sie unbedingt wissen, wann und warum es funktioniert, um zu entscheiden, ob es für eine vorliegende Aufgabe zum Einsatz kommen sollte.

Zu Beginn müssen wir festhalten, dass die hier vorgebrachten Punkte sich ausschließlich und ganz speziell auf das unüberwachte Pretraining mit Greedy-Algorithmen beziehen. Es gibt weitere gänzlich andere Paradigmen für ein halb-überwachtes Lernen mit neuronalen Netzen, zum Beispiel das in Abschnitt 7.13 beschriebene virtuelle Adversarial Training. Es ist auch möglich, einen Autoencoder oder ein generatives Modell zeitgleich mit dem überwachten Modell zu trainieren. Zu den Beispielen für diesen einphasigen Ansatz gehören die diskriminativen RBMs (*Larochelle und Bengio*, 2008) und das *Ladder Network* (*Rasmus et al.*, 2015), in dem das Gesamtziel eine explizite Summe der beiden Terme ist (einer verwendet die Labels, einer nur die Eingabe).

Das unüberwachte Pretraining kombiniert zwei unterschiedliche Konzepte. Zunächst nutzt es die Vorstellung, dass die Wahl der Ausgangsparameter für ein tiefes neuronales Netz einen starken Regularisierungseffekt auf das

Modell haben kann (und – in geringerem Maß – auch die Optimierung verbessern kann). Zweitens nutzt es das allgemeinere Konzept, dass das Lernen über die Eingabeverteilung beim Erlernen der Zuordnung von der Eingabe zur Ausgabe helfen kann.

Beide Konzepte umfassen viele komplizierte Interaktionen zwischen diversen Teilen des Machine-Learning-Algorithmus, die wir noch nicht gänzlich verstehen.

Gerade hinsichtlich der ersten Idee – dass die Auswahl der Ausgangsparameter für ein tiefes neuronales Netz einen starken Regularisierungseffekt auf dessen Leistung haben kann – mangelt es uns noch an Verständnis. Als das Pretraining beliebt wurde, betrachtete man es als ein Mittel zur Initialisierung des Modells an einer Stelle, die zu einer Annäherung an ein lokales Minimum anstelle eines anderen führen würde. Heute gelten lokale Minima nicht länger als ernsthaftes Problem bei der Optimierung neuronaler Netze. Wir wissen mittlerweile, dass unsere üblichen Trainingsverfahren für neuronale Netze nur selten einen kritischen Punkt jeglicher Art erreichen. Es ist weiterhin möglich, dass das Pretraining das Modell an einer Stelle initialisiert, die anderenfalls unzugänglich wäre – zum Beispiel in einem Bereich, der von Regionen umgeben ist, in denen die Kostenfunktion zwischen einzelnen Beispielen so stark schwankt, dass Mini-Batches nur zu einer stark verrauschten Gradientenschätzung führen, oder in einem Bereich, der von Regionen umgeben ist, in denen die Hesse-Matrix so schlecht konditioniert ist, dass das Gradientenabstiegsverfahren sehr kleine Schritte machen muss. Allerdings ist unsere Fähigkeit, genau zu benennen, welche Aspekte der vorab trainierten Parameter während der überwachten Trainingsphase erhalten bleiben, eingeschränkt. Das ist ein Grund dafür, dass moderne Ansätze meist unüberwachtes und überwachtes Lernen gleichzeitig und nicht in zwei aufeinanderfolgenden Phasen einsetzen. Man kann auch vermeiden, sich mit diesen komplizierten Konzepten abzumühen, bei denen es darum geht, wie die Optimierung in der überwachten Lernphase Informationen aus der unüberwachten Lernphase bewahrt, indem man die Parameter für die Merkmalsextraktoren einfach »einfriert« und das überwachte Lernen nur verwendet, um den erlernten Merkmalen einen Klassifikator hinzuzufügen.

Das andere Konzept – dass ein Lernalgorithmus die in der unüberwachten Phase erlernten Informationen nutzen kann, um in der überwachten Lernphase bessere Ergebnisse zu erzielen – ist besser untersucht. Die Grundidee besagt, dass einige Merkmale, die für die unüberwachte Aufgabe nützlich sind, auch für die überwachte Lernaufgabe nützlich sein könnten. Ein Beispiel: Wenn wir ein generatives Modell mit Bildern von Pkws und Motorrädern trainieren, interessieren wir uns für Räder und die Anzahl der Räder in den

Bildern. Wenn wir Glück haben, ist die Repräsentation der Räder von einer Form, die für den überwachten Klassifikator leicht zugänglich ist. Diesen Punkt verstehen wir noch nicht auf einer mathematischen theoretischen Ebene – somit lässt sich nicht immer vorhersagen, welche Aufgaben von einem solchen unüberwachten Lernen profitieren. Viele Aspekte des Ansatzes sind stark von den verwendeten Modellen abhängig. Wenn wir zum Beispiel die vorab trainierten Merkmale um einen linearen Klassifikator ergänzen möchten, müssen die Merkmale die zugrunde liegenden Klassen linear separierbar machen. Diese Eigenschaften entstehen häufig von selbst, aber eben nicht immer. Das ist ein weiterer Grund dafür, das zeitgleiche überwachte und unüberwachte Lernen vorzuziehen – die von der Ausgabeschicht auferlegten Bedingungen sind von Anfang an inhärent.

Unter dem Gesichtspunkt des unüberwachten Pretrainings zum Erlernen einer Repräsentation können wir davon ausgehen, dass unüberwachtes Pretraining effektiver ist, wenn die anfängliche Repräsentation schlecht ist. Ein anschauliches Beispiel hierfür ist die Verwendung von Wort-Embeddings. Durch One-hot-Vektoren repräsentierte Wörter sind nicht besonders informativ, da zwei eindeutige One-hot-Vektoren denselben Abstand zueinander aufweisen (Quadrat des L^2 -Abstands von 2). Erlernte Wort-Embeddings codieren auf natürliche Weise die Ähnlichkeit zwischen Wörtern über deren Abstand zueinander. Daher ist ein unüberwachtes Pretraining bei der Verarbeitung von Wörtern besonders nützlich. Bei der Bildverarbeitung dagegen ist es weniger nützlich, da Bilder bereits in einem umfassenden Vektorraum liegen, in dem Abstände eine Ähnlichkeitsmetrik niederer Qualität bieten.

Aus Sicht des unüberwachten Pretrainings als Regularisierer können wir davon ausgehen, dass unüberwachtes Pretraining dann am hilfreichsten ist, wenn die Anzahl der mit Labeln gekennzeichneten Beispiele sehr gering ist. Da die dem unüberwachten Pretraining hinzugefügte Informationsquelle aus den Daten ohne Label besteht, können wir auch davon ausgehen, dass unüberwachtes Pretraining am besten funktioniert, wenn die Anzahl der Beispiele ohne Label sehr groß ist. Der Vorteil des halb-überwachten Lernens mittels unüberwachtem Pretraining mit vielen Beispielen ohne Label und wenigen mit Labeln gekennzeichneten Beispielen wurde 2011 besonders deutlich, als das unüberwachte Pretraining zwei internationale Wettbewerbe zum Transfer Learning gewann (*Mesnil et al.*, 2011; *Goodfellow et al.*, 2011), bei denen die Anzahl der mit Labeln gekennzeichneten Beispiele gering war (eine Handvoll bis Dutzende von Beispielen für jede Klasse). Diese Effekte wurden auch in sorgfältig überwachten Experimenten von *Paine et al.* (2014) bestätigt.

Auch andere Faktoren dürften beteiligt sein. Zum Beispiel ist unüberwachtes Pretraining vermutlich dann am nützlichsten, wenn die zu erlernende Funktion besonders kompliziert ist. Unüberwachtes Lernen unterscheidet sich von Regularisierung wie Weight Decay, da es den Klassifikator nicht beeinflusst, eine einfache Funktion zu entdecken, sondern ihn vielmehr dazu führt, Merkmalsfunktionen zu entdecken, die für die unüberwachte Lernaufgabe nützlich sind. Wenn die wahren zugrunde liegenden Funktionen kompliziert und durch Regelmäßigkeiten der Eingabeverteilung geprägt sind, kann unüberwachtes Lernen den passenderen Regularisierer darstellen.

Dies vorausgeschickt können wir nun einige erfolgreiche Fälle betrachten, in denen unüberwachtes Pretraining zu einer Verbesserung geführt hat, und erklären, was über die Gründe dafür bekannt ist. Unüberwachtes Pretraining wurde meist zum Verbessern von Klassifikatoren eingesetzt und ist besonders unter dem Gesichtspunkt der Reduzierung des Fehlers auf den Testdaten interessant. Unüberwachtes Pretraining kann aber auch bei anderen Aufgaben als der Klassifizierung helfen und die Optimierung verbessern, also mehr sein als ein reiner Regularisierer. Es kann zum Beispiel den Trainings- und Test-Rekonstruktionsfehler tiefer Autoencoder verbessern (*Hinton und Salakhutdinov, 2006*).

Erhan et al. (2010) haben viele Experimente durchgeführt, um einige Erfolge des unüberwachten Pretrainings zu erklären. Verbesserungen des Trainingsfehlers und Verbesserungen des Testfehlers können beide eventuell so erklärt werden, dass das unüberwachte Pretraining die Parameter in einen Bereich versetzt, der ansonsten unerreichbar bliebe. Das Training neuronaler Netze ist nichtdeterministisch und konvergiert bei jeder Ausführung gegen eine andere Funktion. Es kann an einem Punkt enden, an dem der Gradient klein wird, oder an einem Punkt, an dem der frühe Abbruch das Training anhält, um Überanpassung zu verhindern, oder an einem Punkt, an dem der Gradient groß ist, aber die Suche nach einem Abstiegsschritt aufgrund von Problemen wie Stochastizität oder schlechter Konditionierung der Hesse-Matrix schwierig wird. Neuronale Netze, die einem stetigen unüberwachten Pretraining unterzogen werden, stoppen im selben Bereich des Funktionsraums, wohingegen neuronale Netze ohne stetiges Pretraining in anderen Bereichen stoppen. Abbildung 15.1 stellt dies dar. Der Bereich, in dem vorab trainierte Netze ankommen, ist kleiner – scheinbar reduziert das Pretraining die Varianz der Schätzung, wodurch die Gefahr einer erheblichen Überanpassung reduziert werden kann. Anders ausgedrückt: Unüberwachtes Pretraining initialisiert die Parameter neuronaler Netze in einem Bereich, den sie nicht verlassen können; die Ergebnisse nach dieser

Initialisierung sind einheitlicher und weniger wahrscheinlich schlecht als ohne diese Initialisierung.

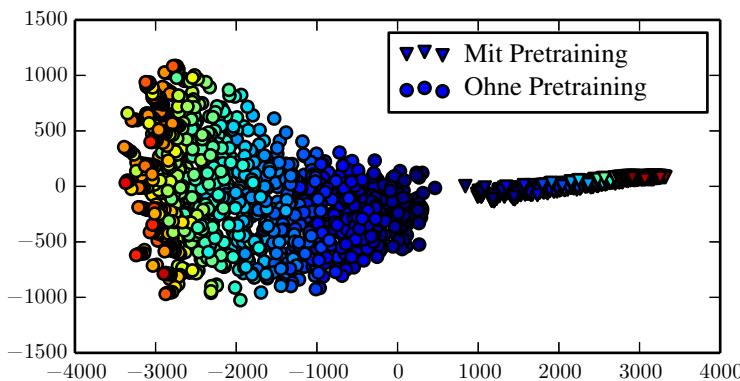


Abbildung 15.1: Visualisierung mittels nichtlinearer Projektion der Lerntrajektorien unterschiedlicher neuronaler Netze im *Funktionsraum* (nicht im Parameterraum, um das Problem der n:1-Zuordnung von Parametervektoren zu Funktionen zu vermeiden), mit unterschiedlichen Zufallsinitialisierungen und mit bzw. ohne unüberwachtes Pretraining. Jeder Punkt entspricht einem anderen neuronalen Netz zu einem bestimmten Zeitpunkt während seines Trainingsverfahrens. (Diese Abbildung wurde mit freundlicher Genehmigung von *Erhan et al. (2010)* angepasst.) Eine Koordinate im Funktionsraum ist ein unendlich-dimensionaler Vektor, der jede Eingabe \mathbf{x} mit einer Ausgabe \mathbf{y} verknüpft. *Erhan et al. (2010)* haben eine lineare Projektion in den hochdimensionalen Raum mittels Verkettung der \mathbf{y} für viele spezifische \mathbf{x} -Punkte vorgenommen. Anschließend haben sie eine weitere nichtlineare Projektion in zwei Dimensionen mittels Isomap durchgeführt (*Tenenbaum et al., 2000*). Alle Netze werden in der Nähe des Zentrums der Grafik initialisiert (dies entspricht dem Bereich der Funktionen, der zu einer approximativen Gleichverteilung über die Klasse y der meisten Eingaben führt). Im Laufe der Zeit führt das Lernen zu einer Außenverschiebung der Funktion hin zu Punkten, die starke Vorhersagen machen. Stetiges Trainieren endet mit Pretraining in einem Bereich und ohne Pretraining in einem anderen Bereich. Diese beiden Bereiche überlappen einander nicht. Isomap versucht, die globalen relativen Abstände (und somit Massen) zu erhalten, sodass der kleine Bereich für Modelle mit Pretraining andeuten könnte, dass ein Schätzer auf Basis des Pretrainings eine geringere Varianz aufweist.

Erhan et al. (2010) liefern auch einige Antworten darauf, wann das Pretraining am besten funktioniert – der Mittelwert und die Varianz des Testfehlers wurden durch Pretraining für tiefere Netze größtenteils reduziert. Bedenken Sie, dass diese Experimente vor der Erfindung und Verbreitung modernerer Verfahren für das Trainieren sehr tiefer Netze durchgeführt wurden (ReLUs, Dropout und Batch-Normalisierung). Es gibt also weniger Erkennt-

nisse über die Auswirkung des unüberwachten Pretrainings in Verbindung mit modernen Ansätzen.

Eine wichtige Frage ist, wie unüberwachtes Pretraining als Regularisierer fungieren kann. Eine Hypothese besagt, dass das Pretraining den Lernalgorithmus dazu animiert, Merkmale zu entdecken, die in einer Beziehung zu den zugrunde liegenden Ursachen stehen, die die beobachteten Daten erzeugen. Das ist eine wichtige Idee, die hinter vielen anderen Algorithmen neben dem unüberwachten Pretraining steckt. Wir gehen in Abschnitt 15.3 näher darauf ein.

Verglichen mit anderen Formen des unüberwachten Lernens bringt unüberwachtes Pretraining einen Nachteil mit sich: Es werden zwei separate Trainingsphasen benötigt. Viele Regularisierungsverfahren bieten den Vorteil, dass der Anwender die Stärke der Regularisierung durch Anpassen des Werts eines einzelnen Hyperparameters steuern kann. Unüberwachtes Pretraining bietet keine offensichtliche Möglichkeit zum Anpassen der Regularisierungsstärke aus der unüberwachten Phase. Stattdessen gibt es sehr viele Hyperparameter, deren Effekte sich im Anschluss messen lassen, die aber im Vorfeld nur schwierig vorhergesagt werden können. Wenn wir unüberwachtes und überwachtes Lernen zeitgleich durchführen, statt ein Pretraining-Verfahren zu nutzen, gibt es einen einzigen Hyperparameter – meist einen Koeffizienten für den unüberwachten Aufwand –, der bestimmt, wie stark das unüberwachte Ziel das überwachte Modell regularisiert. Indem wir den Koeffizienten vermindern, gelangen wir garantiert zu einer geringeren Regularisierung. Beim unüberwachten Pretraining gibt es keine Möglichkeit zur flexiblen Anpassung der Regularisierungsstärke – entweder wird das überwachte Modell mit vorab trainierten Parametern initialisiert oder nicht.

Ein weiterer Nachteil der beiden getrennten Trainingsphasen ist, dass jede Phase eigene Hyperparameter aufweist. Die Leistung der zweiten Phase lässt sich normalerweise nicht während der ersten Phase vorhersagen, sodass es zu einer langen Verzögerung zwischen dem Vorschlagen der Hyperparameter für die erste Phase und der Möglichkeit, diese mit dem Feedback der zweiten Phase zu aktualisieren, kommt. Der grundlegendste Ansatz besteht im Einsatz des Validierungsdatenfehlers in der überwachten Phase zur Auswahl der Hyperparameter für die Pretraining-Phase (vgl. *Larochelle et al.*, 2009). In der Praxis lassen sich einige Hyperparameter wie die Anzahl der Iterationen für das Pretraining während der Pretraining-Phase einfacher einstellen; dazu wird der frühe Abbruch für das unüberwachte Ziel verwendet. Das ist zwar nicht ideal, bedeutet aber einen deutlich geringeren Rechenaufwand als beim überwachten Ziel.

Heute verzichtet man größtenteils auf unüberwachtes Pretraining, ausgenommen im Bereich der Verarbeitung natürlicher Sprache, wo die natürliche Repräsentation der Wörter als One-hot-Vektoren keinerlei Ähnlichkeitsinformationen vermittelt und sehr große Datensätze ohne Label bereitstehen. In diesem Fall bietet das Pretraining den Vorteil, dass man eine gewaltige Menge von Datensätzen ohne Label einmalig vorab trainieren (zum Beispiel mit einem Corpus, der Milliarden von Wörtern enthält), dabei eine gute Repräsentation erlernen (meist in Form von Wörtern, aber auch von Sätzen) und diese dann für eine überwachte Aufgabe, deren Trainingsdatenmenge erheblich weniger Beispiele umfasst, verwenden oder fein abstimmen kann. Dieser Ansatz wurde zuerst von *Collobert und Weston* (2008b), *Turian et al.* (2010) und *Collobert et al.* (2011a) eingesetzt und ist auch heute noch üblich.

Deep-Learning-Verfahren auf Basis des überwachten Lernens mit Regularisierung über Dropout oder Batch-Normalisierung können bei vielen Aufgaben ein menschliches Leistungsniveau erreichen – aber nur mit extrem großen mit Labels gekennzeichneten Datensätzen. Genau diese Verfahren sind dem unüberwachten Pretraining bei mittelgroßen Datensätzen wie CIFAR-10 und MNIST (die etwa 5.000 mit Labels gekennzeichnete Beispiele pro Klasse aufweisen) überlegen. Bei extrem kleinen Datensätzen wie dem Alternative-Splicing-Datensatz schneiden bayessche Verfahren besser als solche mit unüberwachtem Pretraining ab (*Srivastava*, 2013). Aus diesen Gründen ist unüberwachtes Pretraining heute nicht sonderlich beliebt. Dennoch bleibt unüberwachtes Pretraining ein wichtiger Meilenstein in der Geschichte der Deep-Learning-Forschung und wirkt sich auch auf zeitgenössische Ansätze aus. Das Konzept des Pretrainings wurde als **überwachtes Pretraining** generalisiert (vgl. Abschnitt 8.7.4), nämlich als sehr allgemeiner Ansatz für das Transfer Learning. Überwachtes Pretraining für Transfer Learning ist in Verbindung mit CNNs beliebt (*Oquab et al.*, 2014; *Yosinski et al.*, 2014), die mittels ImageNet-Datensatz vorab trainiert werden. In der Fachpresse werden die Parameter der für diesen Zweck trainierten Netze veröffentlicht, ähnlich der Veröffentlichung vorab trainierter Wortvektoren für Aufgaben der natürlichen Sprache (*Collobert et al.*, 2011a; *Mikolov et al.*, 2013a).

15.2 Transfer Learning und Domänenadaption

Transfer Learning und Domänenadaption (engl. *domain adaptation*) bezeichnen die Situation, in der in einer Einstellung (z. B. Verteilung P_1) Erlerntes für eine bessere Generalisierung in einer anderen Einstellung genutzt wird

(z. B. Verteilung P_2). Das stellt eine Generalisierung des im vorhergehenden Abschnitt vorgestellten Konzepts dar – dort wurden Repräsentationen von einer unüberwachten für eine überwachte Lernaufgabe übernommen.

Beim **Transfer Learning** muss der Klassifikator zwei oder mehr unterschiedliche Aufgaben ausführen, aber wir gehen davon aus, dass viele der Faktoren, die die Variationen in P_1 erklären, auch für Variationen relevant sind, die zum Erlernen von P_2 erfasst werden müssen. Das ist generell im Kontext des überwachten Lernens zu verstehen, wo die Eingabe identisch ist, das Ziel aber anderer Art sein kann. Ein Beispiel: Wir können zunächst etwas über eine Menge visueller Kategorien lernen, zum Beispiel Katzen und Hunde, und anschließend etwas über eine andere Menge visueller Kategorien, zum Beispiel Ameisen und Wespen. Wenn im ersten Fall wesentlich mehr Daten vorliegen (die aus P_1 gezogen wurden), dann kann dies helfen, Repräsentationen zu erlernen, die für eine schnelle Generalisierung anhand weniger aus P_2 stammender Beispiele nützlich sind. Viele visuelle Kategorien *teilen* auf einer niedrigen Ebene Begrifflichkeiten über Kanten und visuelle Formen, Auswirkungen geometrischer Änderungen, Änderungen der Beleuchtung usw. Grundsätzlich lassen sich Transfer Learning, Multitask Learning (Abschnitt 7.7) und Domänenadaption mittels Representation Learning umsetzen, wenn es Merkmale gibt, die für unterschiedliche Kontexte oder Aufgaben nützlich sind und die den zugrunde liegenden Faktoren entsprechen, die in mehr als einer Einstellung auftreten. Abbildung 7.2 zeigt dies für geteilte untere Schichten und aufgabenabhängige obere Schichten.

Manchmal wird zwischen den unterschiedlichen Aufgaben jedoch nicht die Semantik der Eingabe, sondern die Semantik der Ausgabe geteilt. So muss ein System zur Spracherkennung gültige Sätze auf der Ausgabeseite erzeugen, wohingegen die früheren Schichten in der Nähe der Eingabe möglicherweise sehr verschiedene Versionen derselben Phoneme oder subphonemischen Aussprachevarianten erkennen müssen, je nach Sprecher. In derartigen Fällen ergibt es mehr Sinn, die oberen Schichten (in der Nähe der Ausgabe) des neuronalen Netzes zu teilen und eine aufgabenspezifische Vorverarbeitung einzusetzen (vgl. Abbildung 15.2).

Im verwandten Fall der **Domänenadaption** bleibt die Aufgabe (und die optimale Eingabe-Ausgabe-Zuordnung) in allen Kontexten gleich, aber die Ausgabeverteilung weist geringfügige Unterschiede auf. Ein Beispiel: Beim Analysieren der Stimmungslage muss ermittelt werden, ob ein Kommentar eine positive oder negative Haltung ausdrückt. Im Internet veröffentlichte Kommentare stammen aus vielen Kategorien. Es kommt zu einer Domänenadaption, wenn ein Prädiktor für die Stimmungslage zunächst anhand von Kundenbewertungen zu Medien wie Büchern, Videos und Musik trainiert

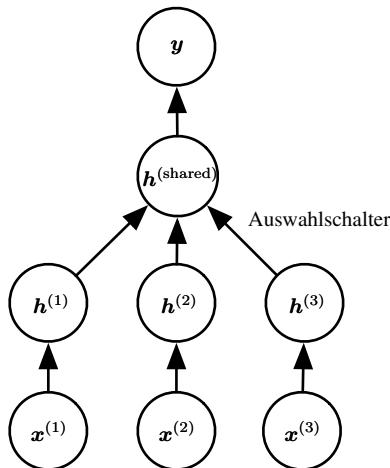


Abbildung 15.2: Beispielarchitektur für Multitask Learning oder Transfer Learning, in der die AusgabevARIABLE y dieselbe Semantik für alle Aufgaben aufweist, die Eingangsvariable x hingegen eine unterschiedliche Bedeutung (und möglicherweise sogar eine unterschiedliche Dimension) für jede Aufgabe (oder beispielsweise jeden Anwender) hat, hier mit $x^{(1)}$, $x^{(2)}$ und $x^{(3)}$ für drei Aufgaben. Die unteren Ebenen (bis zum Auswahlschalter) sind aufgabenspezifisch, die oberen Ebenen werden gemeinsam genutzt. Die unteren Ebenen lernen, ihre aufgabenspezifische Eingabe in eine generische Menge von Merkmalen zu übersetzen.

wird, dann aber auch zur Analyse von Kommentaren zu elektronischen Geräten wie Fernsehern und Smartphones verwendet werden soll. Es ist sicher denkbar, dass es eine zugrunde liegende Funktion gibt, die bestimmt, ob eine Aussage positiv, neutral oder negativ ist. Aber natürlich werden in den einzelnen Domänen (Bereichen) unterschiedliche Begriffe und Stile benutzt, sodass eine Generalisierung über Domänengrenzen schwieriger ist. Es hat sich gezeigt, dass ein einfaches unüberwachtes Pretraining (mit Denoising Autoencodern) für die Stimmungsanalyse mit Domänenadaption sehr erfolgreich ist (*Glorot et al.*, 2011b).

Damit verwandt ist der **Konzept-Drift** (engl. *concept drift*). Wir können dieses Phänomen als eine Art Transfer Learning infolge langsamer Veränderungen der Datenverteilung im Laufe der Zeit beschreiben. Konzept-Drift und Transfer Learning sind quasi spezielle Formen des Multitask Learnings. Der Begriff »Multitask Learning« bezeichnet üblicherweise überwachte Lernaufgaben. Der allgemeinere Begriff Transfer Learning bezieht dagegen auch unüberwachtes Lernen und Reinforcement Learning mit ein.

In all diesen Fällen sollen die Daten aus der ersten Einstellung genutzt werden, um Informationen zu gewinnen, die beim Erlernen oder sogar beim direkten Treffen von Vorhersagen in der zweiten Einstellung hilfreich sein könnten. Die Grundidee des Representation Learnings ist, dass dieselbe Repräsentation in beiden Kontexten nützlich sein kann. Wird dieselbe Repräsentation in beiden Kontexten genutzt, kann die Repräsentation von den Trainingsdaten für beide Aufgaben profitieren.

Wie bereits erwähnt, hat unüberwachtes Deep Learning für Transfer Learning in einigen Machine-Learning-Wettbewerben zum Erfolg geführt (*Mesnil et al.*, 2011; *Goodfellow et al.*, 2011). Im ersten dieser Wettbewerbe sahen die experimentellen Voraussetzungen wie folgt aus: Jeder Teilnehmer erhielt zunächst einen Datensatz aus der ersten Einstellung (aus der Verteilung P_1), der Beispiele einer Menge von Kategorien darstellte. Die Teilnehmer mussten damit einen guten Merkmalsraum erlernen (Zuordnung der unbearbeiteten Eingabe zu einer Repräsentation), sodass bei der Anwendung der erlernten Transformation auf die Eingaben des Transfer-Kontexts (Verteilung P_2) ein linearer Klassifikator trainiert werden konnte, der aus wenigen mit Labels gekennzeichneten Beispielen gut generalisiert. Eine sehr beeindruckende Erkenntnis in diesem Wettbewerb war, dass eine Architektur, die immer tiefere Repräsentationen nutzt (die auf rein unüberwachte Weise aus den in der ersten Einstellung – P_1 – gesammelten Daten erlernt wurden), bei den neuen Kategorien der zweiten Einstellung (Transfer-Einstellung, P_2) eine viel bessere Lernkurve aufweist. Tiefe Repräsentationen benötigen weniger mit Labels gekennzeichnete Beispiele der Transfer-Aufgabe, um die scheinbar asymptotische Generalisierungsleistung zu erzielen.

Zwei extreme Beispiele für Transfer Learning sind **One-Shot Learning** und **Zero-Shot Learning**, manchmal auch **Zero-Data Learning** genannt. Beim One-Shot Learning existiert nur ein mit einem Label gekennzeichnetes Beispiel für die Transfer-Aufgabe, beim Zero-Shot Learning dagegen gar keines.

One-Shot Learning (*Fei-Fei et al.*, 2006) ist möglich, weil die Repräsentation bereits in der ersten Phase lernt, die zugrunde liegenden Klassen sauber zu trennen. Während der Transfer-Learning-Phase wird nur ein Beispiel mit Labeln benötigt, um auf die Labels vieler möglicher Testbeispiele zu schließen, die sich alle um denselben Punkt im Darstellungsraum versammeln. Das funktioniert insofern, als die Faktoren der Variation, die diesen Invarianzen entsprechen, deutlich von den anderen Faktoren im erlernten Darstellungsraum getrennt wurden und wir irgendwie gelernt haben, welche Faktoren bei der Unterscheidung von Objekten bestimmter Kategorien eine Rolle spielen.

Als Beispiel für das Zero-Shot Learning soll ein Klassifikator dienen, der eine große Menge an Texten »liest« und dann Probleme der Objekterkennung löst. Es ist möglich, eine bestimmte Objektklasse zu erkennen, ohne dass jeweils ein Bild dieses Objekts gezeigt wurde, wenn der Text das Objekt hinreichend genau beschreibt. Wenn der Klassifikator beispielsweise gelesen hat, dass Katzen vier Beine und spitze Ohren haben, kann er vielleicht Katzenbilder erkennen, ohne jemals eine Katze gesehen zu haben.

Zero-Data Learning (*Larochelle et al.*, 2008) und Zero-Shot Learning (*Palatucci et al.*, 2009; *Socher et al.*, 2013b) sind nur möglich, weil während des Trainings weitere Informationen zur Verfügung standen. Wir können uns Zero-Data Learning als Szenario mit drei Zufallsvariablen vorstellen: klassische Eingaben \mathbf{x} , klassische Ausgaben oder Zielwerte \mathbf{y} und eine weitere Zufallsvariable, die die Aufgabe T beschreibt. Das Modell wird darauf trainiert, die bedingte Verteilung $p(\mathbf{y} \mid \mathbf{x}, T)$ zu schätzen, wobei T eine Beschreibung der Aufgabe ist, die das Modell ausführen soll. In unserem Beispiel (Erkennen von Katzen nach dem Lesen von Texten über Katzen) ist die Ausgabe eine binäre Variable y mit $y = 1$ für »ja« und $y = 0$ für »nein«. Die Aufgabenvariable T steht dann für die zu beantwortenden Fragen, beispielsweise »Zeigt dieses Bild eine Katze?«. Wenn wir über eine Trainingsdatenmenge verfügen, die unüberwachte Beispiele der Objekte enthält, die im selben Raum wie T vorkommen, können wir eventuell auf die Bedeutung bisher nicht betrachteter Instanzen von T schließen. In unserem Beispiel (Erkennen von Katzen, ohne zuvor ein Bild einer Katze gesehen zu haben) kommt es darauf an, dass Textdaten ohne Label zur Verfügung standen, in denen zum Beispiel folgende Sätze vorkommen: »Katzen haben vier Beine« und »Katzen haben spitze Ohren«.

Für das Zero-Shot Learning ohne Beispiele muss T auf eine Weise repräsentiert werden, die eine Art Generalisierung ermöglicht. T darf also nicht einfach ein One-hot-Code sein, der auf eine Objektkategorie verweist. *Socher et al.* (2013b) liefern stattdessen eine verteilte Repräsentation von Objektkategorien unter Verwendung eines erlernten Wort-Embeddings für das mit der jeweiligen Kategorie verknüpfte Wort.

Ein ähnliches Phänomen tritt in der maschinellen Übersetzung auf (*Klementiev et al.*, 2012; *Mikolov et al.*, 2013b; *Gouws et al.*, 2014): Wir verfügen über Wörter in einer Sprache und die Beziehungen zwischen den Wörtern können mithilfe einsprachiger Corpora erlernt werden. Auf der anderen Seite stehen uns übersetzte Sätze zur Verfügung, die Wörter aus der einen Sprache mit Wörtern aus der anderen in eine Beziehung setzen. Obwohl wir vielleicht keine mit Labels gekennzeichneten Beispiele für die Übersetzung des Worts A aus der Sprache X in das Wort B in der Sprache

Y haben, können wir generalisieren und eine Übersetzung für das Wort A erraten, weil wir eine verteilte Repräsentation der Wörter in der Sprache X und eine verteilte Repräsentation der Wörter in der Sprache Y erlernt und anschließend eine Verbindung (möglicherweise in beiden Richtungen) der beiden Räume anhand von Trainingsbeispielen zugeordneter Satzpaare in beiden Sprachen erstellt haben. Dieser Transfer ist besonders erfolgreich, wenn alle drei Zutaten (die beiden Repräsentationen und die Beziehungen zwischen ihnen) gemeinsam erlernt werden.

Zero-Shot Learning ist eine spezielle Form des Transfer Learnings. Dasselbe Prinzip erklärt das **multimodale Lernen**, bei dem eine Repräsentation in jeder der beiden Modalitäten sowie die Beziehung (im Allgemeinen eine multivariate Verteilung) zwischen den Paaren (\mathbf{x}, \mathbf{y}) , bestehend aus einer Beobachtung \mathbf{x} in der einen und einer weiteren Beobachtung \mathbf{y} in der anderen Modalität, erfasst werden (*Srivastava und Salakhutdinov, 2012*). Durch das Erlernen aller drei Parametermengen (von \mathbf{x} zur zugehörigen Repräsentation, von \mathbf{y} zur zugehörigen Repräsentation und der Beziehung zwischen den beiden Repräsentationen) werden Konzepte aus einer Repräsentation in der anderen verankert (und umgekehrt), sodass eine aussagekräftige Generalisierung für neue Paare möglich wird. Das Verfahren ist in Abbildung 15.3 dargestellt.

15.3 Halb-überwachtes Separieren kausaler Faktoren

Eine wichtige Frage zum Representation Learning ist: Was macht eine Repräsentation besser als eine andere? Eine Hypothese ist, dass eine ideale Repräsentation eine solche ist, in der die Merkmale innerhalb der Repräsentation dem entsprechen, was die beobachteten Daten grundlegend ausmachen, also deren zugrundeliegenden Ursachen darstellen, wobei separate Merkmale oder Richtungen im Merkmalsraum unterschiedlichen Ursachen entsprechen, damit die Repräsentation die einzelnen Ursachen voneinander unterscheiden und separieren kann. Diese Hypothese führt zu Ansätzen, in denen wir zuerst eine gute Repräsentation für $p(\mathbf{x})$ suchen. Eine solche Repräsentation kann auch eine gute Repräsentation zur Berechnung von $p(\mathbf{y} | \mathbf{x})$ sein, wenn \mathbf{y} zu den charakteristischsten Ursachen für \mathbf{x} gehört. Diese Idee ist mindestens seit den 1990er-Jahren bestimmend für einen Großteil der Deep-Learning-Forschung gewesen (*Becker und Hinton, 1992; Hinton und Sejnowski, 1999*). Andere Argumente dazu, wann halb-überwachtes Lernen dem reinen über-

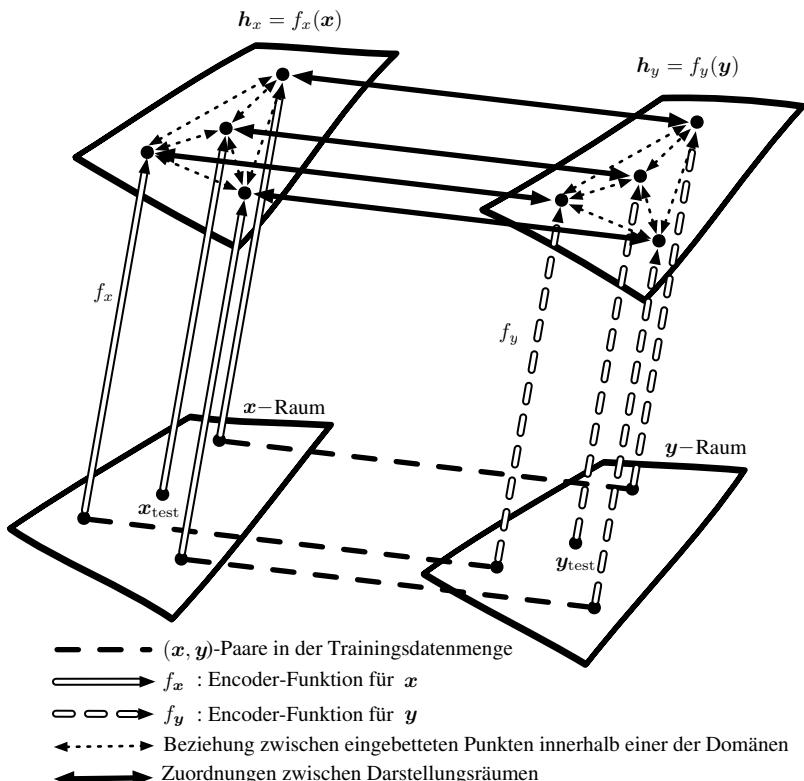


Abbildung 15.3: Transfer Learning zwischen den beiden Definitionsbereichen x und y ermöglicht Zero-Shot Learning. Beispiele mit oder ohne Label für x erlauben das Erlernen einer Repräsentationsfunktion f_x . Ebenso wird anhand von Beispielen für y die Funktion f_y erlernt. Jede Anwendung der Funktionen f_x und f_y wird durch einen Aufwärtspfeil dargestellt; der Pfeiltyp gibt die Funktion an. Der Abstand im h_x -Raum dient als Ähnlichkeitsmetrik zwischen beliebigen Punktpaaren im x -Raum, die möglicherweise aussagekräftiger als der Abstand im x -Raum ist. Ebenso dient der Abstand im h_y -Raum als Ähnlichkeitsmetrik zwischen beliebigen Punktpaaren im y -Raum. Diese beiden Ähnlichkeitsfunktionen werden durch gepunktete Pfeile mit zwei Spitzen dargestellt. Mit Labeln gekennzeichnete Beispiele (gestrichelte horizontale Linien) sind Paare (x, y) , die das Erlernen einer Ein- oder Zwei-Wege-Zuordnung (durchgezogener Pfeil mit zwei Spitzen) zwischen den Repräsentationen $f_x(x)$ und den Repräsentationen $f_y(y)$ erlauben und diese Repräsentationen aneinander verankern. Zero-Data Learning wird dann wie folgt ermöglicht: Ein Bild x_{test} kann sogar dann mit einem Wort y_{test} verknüpft werden, wenn noch nie ein Bild des Worts vorgelegt wurde, und zwar aus dem einfachen Grund, dass die Wortrepräsentationen $f_y(y_{\text{test}})$ und die Bildrepräsentationen $f_x(x_{\text{test}})$ mithilfe der Zuordnungen zwischen den Darstellungsräumen in eine Beziehung zueinander gesetzt werden können. Das funktioniert trotzdem, der niemals erfolgten Paarung von Bild und Wort, weil ihre jeweiligen Merkmalsvektoren $f_x(x_{\text{test}})$ und $f_y(y_{\text{test}})$ miteinander in Beziehung gesetzt wurden. Abbildung nach einer Idee von Hrant Khachatrian.

wachten Lernen überlegen ist, finden Sie in Abschnitt 1.2 von *Chapelle et al.* (2006).

In anderen Ansätzen zum Representation Learning geht es oft um eine einfache zu modellierende Repräsentation, deren Einträge zum Beispiel dünnbesetzt oder unabhängig voneinander sind. Eine Repräsentation, die die zugrunde liegenden ursächlichen Faktoren sauber trennt, ist nicht unbedingt auch einfach zu modellieren. Allerdings besagt ein weiterer Teil der Hypothese, die dem halb-überwachten Lernen den Vorzug gegenüber dem unüberwachten Representation Learning gibt, dass diese Eigenschaften in vielen KI-Aufgaben zusammenfallen: Sobald wir in der Lage sind, die zugrundeliegenden Erklärungen zu für das, was wir beobachten, zu erhalten, wird es meist einfach, die einzelnen Attribute von den anderen zu isolieren. Vor allem, wenn eine Repräsentation \mathbf{h} viele der zugrunde liegenden Ursachen des beobachteten \mathbf{x} darstellt und die Ausgaben \mathbf{y} zu den charakteristischsten Ursachen gehören, ist es einfach, \mathbf{y} aus \mathbf{h} vorherzusagen.

Sehen wir uns zunächst an, wie halb-überwachtes Lernen fehlschlagen kann, wenn das unüberwachte Erlernen von $p(\mathbf{x})$ für das Erlernen von $p(\mathbf{y} \mid \mathbf{x})$ nicht hilfreich ist. Betrachten wir ein Beispiel, in dem $p(\mathbf{x})$ gleichverteilt ist. Wir möchten $f(\mathbf{x}) = \mathbb{E}[\mathbf{y} \mid \mathbf{x}]$ erlernen. Es liegt auf der Hand, dass wir durch Beobachten einer Trainingsdatenmenge mit \mathbf{x} -Werten keinerlei Informationen über $p(\mathbf{y} \mid \mathbf{x})$ erhalten.

Betrachten wir nun ein einfaches Beispiel dafür, wie halb-überwachtes Lernen zum Erfolg führt. Gegeben sei \mathbf{x} als Ergebnis einer Mischung mit einer Komponente der Mischung für jeden Wert von \mathbf{y} (vgl. Abbildung 15.4). Wenn die Komponenten der Mixture sauber getrennt sind, zeigt die Modellierung von $p(\mathbf{x})$ genau an, wo jede Komponente liegt; in diesem Fall genügt bereits ein einzelnes mit Labels gekennzeichnetes Beispiel für jede Klasse, um $p(\mathbf{y} \mid \mathbf{x})$ perfekt zu erlernen. Aber was könnte $p(\mathbf{y} \mid \mathbf{x})$ und $p(\mathbf{x})$ allgemein betrachtet miteinander verbinden?

Wenn \mathbf{y} eng mit einem der kausalen Faktoren von \mathbf{x} verknüpft ist, sind $p(\mathbf{x})$ und $p(\mathbf{y} \mid \mathbf{x})$ stark miteinander verbunden; in diesem Fall ist ein unüberwachtes Representation Learning, das versucht, die zugrunde liegenden Faktoren der Variation zu separieren, vermutlich als halb-überwachtes Lernverfahren nützlich.

Nehmen wir an, \mathbf{y} sei einer der kausalen Faktoren für \mathbf{x} und \mathbf{h} sei die Repräsentation aller dieser Faktoren. Der wahre generative Prozess lässt sich laut diesem gerichteten graphischen Modell als strukturiert ansehen; \mathbf{h} ist \mathbf{x} übergeordnet:

$$p(\mathbf{h}, \mathbf{x}) = p(\mathbf{x} \mid \mathbf{h})p(\mathbf{h}). \quad (15.1)$$

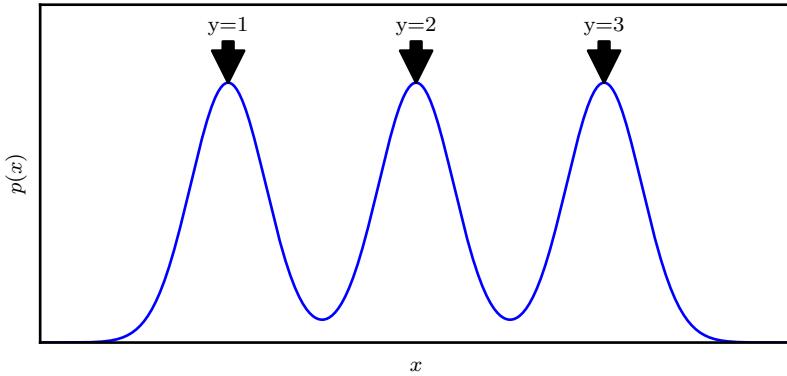


Abbildung 15.4: Mischmodell. Beispiele für eine Dichte über x , die eine Mischung über drei Komponenten darstellt. Die Komponentenidentität ist ein zugrunde liegender erklärender Faktor, y . Da die Komponenten der Mischung (z.B. die natürlichen Objektklassen in Bilddaten) statistisch charakteristisch sind, deckt das Modellieren von $p(x)$ auf unüberwachte Weise ohne Beispiel mit Labeln bereits den Faktor y auf.

Die Daten weisen dementsprechend folgende Randwahrscheinlichkeit auf:

$$p(\mathbf{x}) = \mathbb{E}_{\mathbf{h}} p(\mathbf{x} \mid \mathbf{h}). \quad (15.2)$$

Diese simple Beobachtung führt zu dem Schluss, dass das bestmögliche Modell für \mathbf{x} (im Sinne der Generalisierung) jenes ist, dass die obige »wahre« Struktur aufdeckt, mit \mathbf{h} als der latenten Variable, die die beobachteten Variationen in \mathbf{x} erklärt. Das oben behandelte »ideale« Representation Learning sollte daher diese latenten Faktoren wiederherstellen. Gehört \mathbf{y} dazu (oder ist eng damit verwandt), dann ist das Vorhersagen von \mathbf{y} aus einer solchen Repräsentation einfach zu erlernen. Wir sehen auch, dass die bedingte Verteilung von \mathbf{y} anhand von \mathbf{x} über den Satz von Bayes mit den Komponenten in der obigen Gleichung verbunden ist:

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \mathbf{y})p(\mathbf{y})}{p(\mathbf{x})}. \quad (15.3)$$

Somit ist die Randverteilung $p(\mathbf{x})$ sehr eng mit der bedingten Verteilung $p(\mathbf{y} \mid \mathbf{x})$ verbunden und das Wissen über die Struktur der ersten Verteilung hilft beim Erlernen der zweiten. Wenn diese Annahmen zutreffen, sollte halb-überwachtes Lernen daher die Leistung steigern.

Ein wichtiges Problem der Forschung betrifft die Tatsache, dass die meisten Beobachtungen aus einer extrem großen Anzahl zugrunde liegender Ursachen gebildet werden. Angenommen, es gilt $\mathbf{y} = \mathbf{h}_i$, aber der unüberwachte Klassifikator weiß nicht, um welches \mathbf{h}_i es sich handelt. Die Brute-

Force-Lösung lässt einen unüberwachten Klassifikator eine Repräsentation erlernen, die *alle* sinnvollen charakteristischen generativen Faktoren h_j erfasst und sie separiert, wodurch y problemlos aus \mathbf{h} vorhergesagt werden kann, und zwar ungeachtet dessen, welches h_i mit y verknüpft ist.

In der Praxis ist die Brute-Force-Lösung jedoch nicht realisierbar, da es nicht möglich ist, alle oder die meisten dieser Faktoren der Variation, die eine Beobachtung beeinflussen, zu erfassen. Sollte zum Beispiel in einer optischen Darstellung die Repräsentation jedes Mal auch noch die kleinsten Objekte im Bildhintergrund codieren? Es ist ein bekanntes psychologisches Phänomen, dass Menschen Änderungen in ihrem Umfeld nicht wahrnehmen, wenn diese für die aktuelle Aufgabe nicht von Bedeutung sind – siehe hierzu zum Beispiel *Simons und Levin* (1998). Ein unüberwindbares Problem in der Forschung beim halb-überwachten Lernen ist das Bestimmen dessen, *was* in jedem Einzelfall codiert werden sollte. Derzeit gibt es zwei wesentliche Vorgehensweisen für den Umgang mit vielen zugrunde liegenden Ursachen: Die eine verwendet die zeitgleich ein überwachtes und unüberwachtes Lernsignal, sodass das Modell sich dafür entscheidet, die Faktoren der Variation mit der höchsten Relevanz zu erfassen. Die andere verwendet sehr viel größere Repräsentationen in Verbindung mit rein unüberwachtem Lernen.

Ein jüngeres Verfahren für das unüberwachte Lernen ist, die Definition dafür zu ändern, welche der zugrunde liegenden Ursachen am charakteristischsten sind. In der Vergangenheit wurden Autoencoder und generative Modelle für das Optimieren eines unveränderlichen Kriteriums trainiert, das häufig dem mittleren quadratischen Fehler ähnelte. Diese unveränderlichen Kriterien bestimmen, welche Ursachen als charakteristisch gelten. Zum Beispiel gibt der mittlere quadratische Fehler bei Anwendung auf die Pixel eines Bildes implizit an, dass eine zugrunde liegende Ursache nur charakteristisch ist, wenn sie die Helligkeit einer großen Menge von Pixeln erheblich verändert. Das kann dann zu einem Problem werden, wenn wir im Rahmen der Aufgabe mit kleinen Objekten interagieren möchten. Abbildung 15.5 enthält ein Beispiel für eine Robotik-Aufgabe, in der ein Autoencoder es nicht geschafft hat, einen kleinen Tischtennisball zu codieren. Derselbe Roboter hat keine Probleme bei der Interaktion mit größeren Objekten wie Baseballs, die für den mittleren quadratischen Fehler charakteristischer sind.

Ob etwas charakteristisch ist, lässt sich auch anders definieren. Ein Beispiel: Wenn eine Gruppe von Pixeln einem sehr gut erkennbaren Muster folgt, obschon dieses Muster weder durch besondere Helligkeit noch besondere Dunkelheit hervorsticht, könnte dieses Muster als extrem charakteristisch und damit wichtig gewertet werden. Eine Möglichkeit zum Implementieren

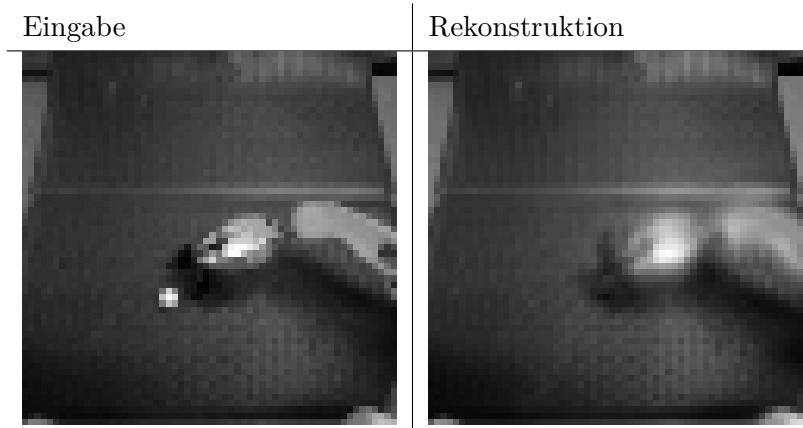


Abbildung 15.5: Ein anhand des mittleren quadratischen Fehlers für eine Robotik-Aufgabe trainierter Autoencoder konnte einen Tischtennisball nicht rekonstruieren. Die Existenz des Tischtennisballs und all seiner räumlichen Koordinaten sind wichtige zugrunde liegende kausale Faktoren für die Erzeugung des Bildes und für die Robotik-Aufgabe von Relevanz. Leider ist die Kapazität des Autoencoders eingeschränkt und das Training mit dem mittleren quadratischen Fehler hat den Tischtennisball nicht für charakteristisch genug gehalten, um ihn zu codieren. Die Bilder wurden freundlicherweise von Chelsea Finn zur Verfügung gestellt.

einer solchen Definition besteht in einem kürzlich entwickelten Ansatz, den **Generative-Adversarial-Netzen** oder GANs (*Goodfellow et al., 2014c*). Dabei wird ein generatives Modell so trainiert, dass es einen Feedforward-Klassifikator täuscht. Der Feedforward-Klassifikator versucht, alle Stichproben (Samples) des generativen Modells als Fälschungen und alle Stichproben aus der Trainingsdatenmenge als Originale zu erkennen. In diesem Framework ist jedes strukturierte Muster, das vom Feedforward-Netz erkannt wird, extrem charakteristisch. Das Generative-Adversarial-Netz wird in Abschnitt 20.10.4 näher beschrieben. Für den Moment reicht es aus, zu verstehen, dass die Netze *lernen* zu entscheiden, was charakteristisch ist. *Lotter et al. (2015)* haben gezeigt, dass Modelle, die darauf trainiert werden, Bilder menschlicher Köpfe auszugeben, häufig keine Ohren erzeugen, wenn sie mit dem mittleren quadratischen Fehler trainiert werden. Bei Verwendung des Adversarial Frameworks werden die Ohren dagegen gezeichnet. Da die Ohren sich von den umgebenden Hautpartien nicht besonders hell oder dunkel abheben, sind sie für den Verlust des mittleren quadratischen Fehlers nicht besonders charakteristisch. Aber ihre deutlich erkennbare Form und einheitliche Lage sorgen dafür, dass ein Feedforward-Netz problemlos erlernen kann, sie zu erkennen, sodass sie im Generative-Adversarial-

Framework (GAF) extrem charakteristisch sind. Abbildung 15.6 zeigt einige Beispielbilder hierzu. Generative-Adversarial-Netze sind lediglich ein Aspekt bei der Bestimmung der darzustellenden Faktoren. Wir erwarten, dass künftige Forschungsprojekte bessere Möglichkeiten zum Bestimmen der darzustellenden Faktoren aufdecken und Mechanismen zur Darstellung aufgabenabhängiger Faktoren bereitstellen werden.

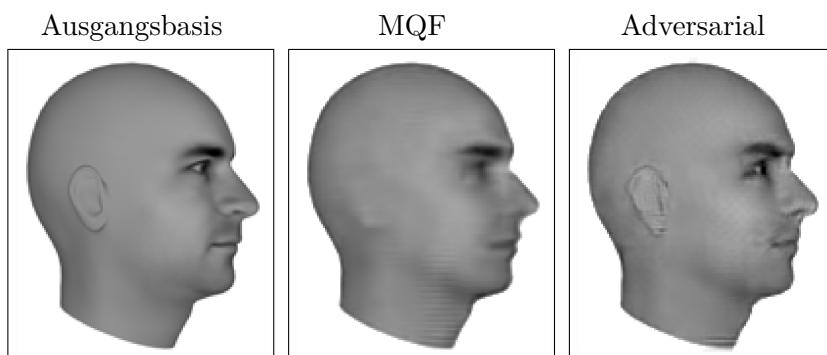


Abbildung 15.6: Prädiktive generative Netze zeigen, wie wichtig es ist, zu lernen, welche Merkmale charakteristisch sind. In diesem Beispiel wurde das prädiktive generative Netz darauf trainiert, das Erscheinungsbild eines 3-D-Modells eines menschlichen Kopfes für einen bestimmten Blickwinkel vorherzusagen. (*Links*) Ausgangsbasis. Dies ist das korrekte Bild, das vom Netz ausgegeben werden sollte. (*Mitte*) Dieses Bild wird von einem prädiktiven generativen Netz ausgegeben, das ausschließlich auf Basis des MQF trainiert wurde. Da die Ohren sich nicht sonderlich von den umgebenden Hautpartien abheben, hat das Modell sie nicht als charakteristisch genug für die Darstellung erachtet. (*Rechts*) Dieses Bild wird von einem Modell ausgegeben, das den MQF- und den sogenannten Adversarial-Verlust kombiniert. Mit dieser erlernten Kostenfunktion sind die Ohren charakteristisch, da sie einem vorhersagbaren Muster folgen. Zu lernen, welche zugrunde liegenden Ursachen wichtig und relevant genug für die Modellierung sind, ist ein wichtiges Gebiet der aktuellen Forschung. Die Abbildungen wurden freundlicherweise von *Lotter et al.* (2015) zur Verfügung gestellt.

Ein Vorteil beim Erlernen der zugrunde liegenden kausalen Faktoren, auf den *Schölkopf et al.* (2012) hingewiesen haben ist, dass der echte generative Prozess gegenüber Änderungen in $p(\mathbf{y})$ robust ist, wenn er \mathbf{x} als Effekt und \mathbf{y} als Ursache enthält. Wenn die Beziehung zwischen Ursache und Effekt umgekehrt wäre, gilt das nicht mehr, da $p(\mathbf{x} | \mathbf{y})$ laut dem Satz von Bayes empfindlich gegenüber Änderungen in $p(\mathbf{y})$ wäre. Bei Überlegungen hinsichtlich Änderungen in der Verteilung aufgrund unterschiedlicher Domänen, temporaler Instationarität oder Änderungen in der Art der Aufgabe *bleiben die kausalen Mechanismen häufig invariant* (»Die Gesetze des Universums

sind konstant.«), während sich die Randverteilung über die zugrunde liegenden Ursachen ändern kann. Daher können wir eine bessere Generalisierung und Robustheit gegenüber jeglichen Änderungen erwarten, wenn ein generatives Modell erlernt wird, das versucht, die kausalen Faktoren \mathbf{h} und $p(\mathbf{x} \mid \mathbf{h})$ wiederherzustellen.

15.4 Verteilte Repräsentation

Verteilte Repräsentationen von Konzepten – Repräsentationen, die aus vielen Elementen bestehen, die getrennt voneinander betrachtet werden können – gehören zu den wichtigsten Hilfsmitteln im Representation Learning. Verteilte Repräsentationen sind leistungsstark, da sie n Merkmale mit k Werten zur Beschreibung k^n unterschiedlicher Konzepte nutzen können. Wie Sie in diesem Buch bereits gesehen haben, nutzen neuronale Netze mit mehreren verdeckten Einheiten ebenso wie probabilistische Modelle mit mehreren latenten Variablen das Verfahren verteilter Repräsentationen. Wir führen nun eine weitere Beobachtung ein. Viele Deep-Learning-Algorithmen beruhen auf der Annahme, dass die verdeckten Einheiten das Repräsentieren der zugrunde liegenden kausalen Faktoren, mit denen die Daten erklärt werden, erlernen können (vgl. Abschnitt 15.3). Verteilte Repräsentationen sind für diesen Ansatz perfekt geeignet, da jede Richtung im Darstellungsraum dem Wert einer anderen zugrunde liegenden Konfigurationsvariablen entsprechen kann.

Ein Beispiel für eine verteilte Repräsentation ist ein Vektor mit n binären Merkmalen, die 2^n Konfigurationen annehmen können, von denen jede potenziell einem anderen Bereich im Eingaberaum entspricht, wie Abbildung 15.7 zeigt. Das lässt sich mit einer *symbolischen Darstellung* vergleichen, deren Eingabe mit einem einzelnen Symbol oder einer einzelnen Kategorie verknüpft ist. Wenn das Dictionary n Symbole enthält, sind n Merkmalsdetektoren vorstellbar, die jeweils erkennen, ob die zugehörige Kategorie vorhanden ist. In diesem Fall sind nur n unterschiedliche Konfigurationen des Darstellungsraums möglich, die n unterschiedliche Bereiche im Eingaberaum ausbilden (vgl. Abbildung 15.8). Eine solche symbolische Darstellung wird auch als One-hot-Repräsentation bezeichnet, da sie von einem binären Vektor mit n einander ausschließenden Bits (nur eines davon darf aktiv sein) erfasst werden kann. Eine symbolische Darstellung ist ein besonders Beispiel für die breitere Klasse der nicht verteilten Repräsentationen. Das sind Repräsentationen, die viele Einträge enthalten können, jedoch

ohne eine sonderlich bedeutsame, separate Kontrolle über die einzelnen Einträge.

Die folgenden Beispiele für Lernalgorithmen beruhen auf nicht verteilten Repräsentationen:

- *Clustering-Verfahren einschließlich des k-Means-Algorithmus*: Jeder Eingabepunkt wird genau einem Cluster zugewiesen.
- *k-Nearest-Neighbor-Algorithmen*: Eine oder einige wenige Templates oder prototypische Beispiele werden mit einer gegebenen Eingabe verknüpft. Für $k > 1$ beschreiben mehrere Werte jede der Eingaben; diese Werte können jedoch nicht separat voneinander gesteuert werden, sodass es sich hierbei nicht um eine echte verteilte Repräsentation handelt.
- *Entscheidungsbäume*: Bei einer vorliegenden Eingabe wird nur ein Blatt (und die Knoten auf dem Pfad von der Wurzel zum Blatt) aktiviert.
- *Gaußsche Mischverteilungen und Mixtures of Experts*: Die Templates (Cluster-Zentren) oder Experten werden hier mit einem *Grad* der Aktivierung gekoppelt. Wie beim *k*-Nearest-Neighbor-Algorithmus wird jede Eingabe durch mehrere Werte repräsentiert, die nicht direkt getrennt voneinander gesteuert werden können.
- *Kernel-Maschinen mit einem gaußschen Kernel (oder anderen, ähnlich lokalen Kerneln)*: Obwohl der Grad der Aktivierung für jeden »Stützvektor« oder jedes Vorlagenbeispiel nun stetigwertig ist, besteht dasselbe Problem wie bei der gaußschen Mischverteilung.
- *Sprach- oder Übersetzungsmodelle auf N-Gramm-Basis*: Die Menge der Kontexte (Sequenzen von Symbolen) wird mithilfe einer Baumstruktur der Suffixe unterteilt. Ein Blatt für die letzten beiden Wörter könnte zum Beispiel w_1 und w_2 entsprechen. Für jedes Blatt des Baums werden separate Parameter geschätzt (dabei ist eine gewisse gemeinsame Nutzung möglich).

Bei einigen dieser nicht verteilten Algorithmen erfolgt die Ausgabe nicht konstant nach Teilen, sondern interpoliert zwischen benachbarten Bereichen. Die Beziehung zwischen der Anzahl der Parameter (bzw. Beispiele) und der Anzahl der Bereiche, die damit definiert werden kann, bleibt linear.

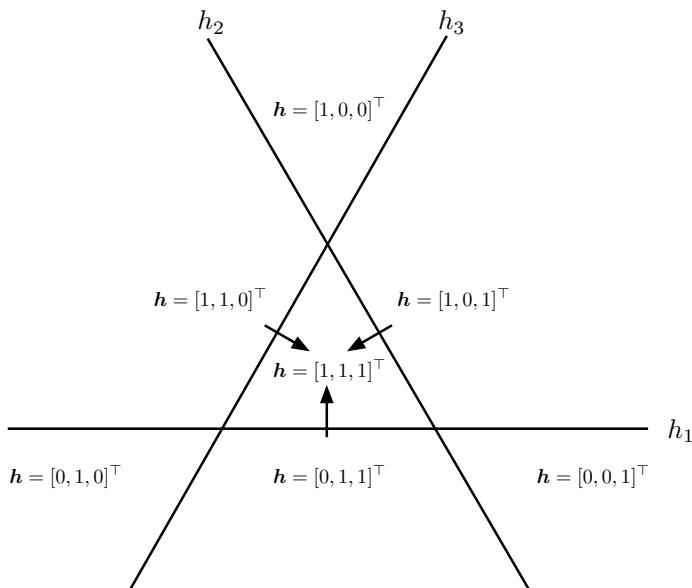


Abbildung 15.7: Darstellung der Aufteilung des Eingaberaums in Bereiche durch einen Lernalgorithmus auf Basis einer verteilten Repräsentation. In diesem Beispiel gibt es drei binäre Merkmale: h_1 , h_2 und h_3 . Jedes Merkmal wird über Schwellenwertbildung der Ausgabe einer erlernten linearen Transformation definiert. Jedes Merkmal unterteilt \mathbb{R}^2 in zwei Halbebenen. h_i^+ sei die Menge der Eingabepunkte, für die gilt $h_i = 1$, und h_i^- die Menge der Eingabepunkte, für die gilt $h_i = 0$. In dieser Abbildung stellt jede Linie die Entscheidungsgrenze für ein h_i dar, wobei der jeweilige Pfeil auf die Seite h_i^+ der Grenze weist. Die Repräsentation als Ganzes nimmt in jedem möglichen Schnittpunkt dieser Halbebenen einen eindeutigen Wert an. Zum Beispiel entspricht der Repräsentationswert $[1, 1, 1]^\top$ dem Bereich $h_1^+ \cap h_2^+ \cap h_3^+$. Vergleichen Sie dies mit der nicht verteilten Repräsentation aus Abbildung 15.8. Im allgemeinen Fall mit d Eingabedimensionen unterteilt eine verteilte Repräsentation \mathbb{R}^d durch Schneiden von Halbräumen anstelle von Halbebenen. Die verteilte Repräsentation mit n Merkmalen weist $O(n^d)$ verschiedenen Bereichen eindeutige Codes zu, während der Nearest-Neighbor-Algorithmus mit n Beispielen nur n Bereichen eindeutige Codes zuweist. Die verteilte Repräsentation ist daher in der Lage, exponentiell viel mehr Bereiche zu unterscheiden als die nicht verteilte Repräsentation. Bedenken Sie, dass nicht alle \mathbf{h} -Werte praktikabel sind (es gibt in diesem Beispiel kein $\mathbf{h} = \mathbf{0}$) und dass ein ergänzender linearer Klassifikator zur verteilten Repräsentation nicht in der Lage ist, jedem benachbarten Bereich unterschiedliche Klassenidentitäten zuzuweisen; selbst ein tiefes Netz mit linearem Schwellenwert weist eine VC-Dimension von lediglich $O(w \log w)$ auf, mit w als Anzahl der Gewichte (Sontag, 1998). Die Kombination aus einer leistungsstarken Repräsentationsschicht und einer schwachen Klassifizierungsschicht kann einen starken Regularisierer darstellen; ein Klassifikator, der versucht, das Konzept »Person« (im Gegensatz zu »keine Person«) zu erlernen, muss einer Eingabe, die als »Frau mit Brille« repräsentiert wird, keine andere Klasse zuweisen als einer Eingabe, die als »Mann ohne Brille« repräsentiert wird. Diese Einschränkung der Kapazität animiert jeden Klassifikator dazu, sich auf wenige h_i zu konzentrieren; und sie animiert \mathbf{h} dazu, zu lernen, die Klassen auf linear separierbare Weise zu

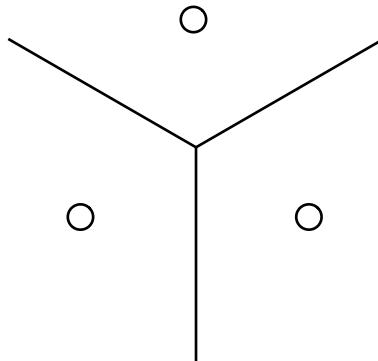


Abbildung 15.8: Darstellung der Aufteilung des Eingaberaums in verschiedene Bereiche durch den Nearest-Neighbor-Algorithmus. Der Nearest-Neighbor-Algorithmus ist ein Beispiel für einen Lernalgorithmus auf Basis einer nicht verteilten Repräsentation. Unterschiedliche, nicht verteilte Algorithmen können unterschiedliche Geometrien aufweisen, aber normalerweise teilen sie den Eingaberaum in Bereiche auf, *wobei es für jeden Bereich eine separate Parametermenge gibt*. Der Vorteil des nicht verteilten Ansatzes besteht darin, dass – genügend Parameter vorausgesetzt – die Trainingsdatenmenge ohne Lösen eines schwierigen Optimierungsalgorithmus angepasst werden kann, da die *unabhängige* Wahl einer für jeden Bereich anderen Ausgabe kein Problem darstellt. Der Nachteil ist, dass derartige nicht verteilte Modelle nur lokal mittels der Glattheitsannahme generalisieren, wodurch es schwierig wird, eine komplizierte Funktion mit mehr Höhen und Tiefen als der verfügbaren Anzahl von Beispielen zu erlernen. Vergleichen Sie dies mit einer verteilten Repräsentation (Abbildung 15.7).

Ein wichtiges verwandtes Konzept, das eine verteilte Repräsentation von einer symbolischen unterscheidet, ist, dass die *Generalisierung aufgrund der zwischen den unterschiedlichen Konzepten geteilten Attributen entsteht*. Als reine Symbole sind »Katze« und »Hund« genauso weit voneinander entfernt wie zwei beliebige andere Symbole. Wenn man sie jedoch mit einer aussagekräftigen verteilten Repräsentation verknüpft, lassen sich viele Dinge, die über Katzen gesagt werden, auf Hunde generalisieren und umgekehrt. Ein Beispiel: Unsere verteilte Repräsentation kann Einträge nach dem Muster »hat_Fell« oder »Anzahl_der_Beine« enthalten, die denselben Wert für das Embedding von »Katze« und von »Hund« aufweisen. Neuronale Sprachmodelle, die mit verteilten Repräsentationen von Wörtern arbeiten, generalisieren viel besser als andere Modelle, die direkt mit den One-hot-Repräsentationen von Wörtern arbeiten (vgl. Abschnitt 12.4). Verteilte Repräsentationen beinhalten einen reichhaltigen *Ähnlichkeitsraum*, in dem einander semantisch nahestehende Konzepte (oder Eingaben) einen geringen

Abstand zueinander aufweisen – eine Eigenschaft, die rein symbolischen Darstellungen abgeht.

Wann und warum kann sich ein statistischer Vorteil aus der Nutzung einer verteilten Repräsentation im Rahmen eines Lernalgorithmus ergeben? Verteilte Repräsentationen können einen statistischen Vorteil aufweisen, wenn eine offensichtlich komplexe Struktur mit einer geringen Anzahl an Parametern kompakt dargestellt werden kann. Einige klassische, nicht verteilte Lernalgorithmen generalisieren ausschließlich aufgrund der Glattheitsannahme, die besagt, dass für $u \approx v$ die zu erlernende Zielfunktion f die Eigenschaft aufweist, dass im Allgemeinen $f(u) \approx f(v)$ ist. Es gibt viele Möglichkeiten, um eine solche Annahme zu formalisieren, aber letztendlich wählen wir für ein Beispiel (x, y) , von dem wir wissen, dass $f(x) \approx y$ einen Schätzer \hat{f} , der diese Bedingungen näherungsweise erfüllt und sich zugleich möglichst wenig ändert, wenn wir uns zu einer nahegelegenen Eingabe $x + \epsilon$ begeben. Diese Annahme ist offenkundig sehr nützlich, leidet aber unter dem Fluch der Dimensionalität: Um eine Zielfunktion zu erlernen, die in vielen unterschiedlichen Bereichen häufig zu- und abnimmt,¹ benötigen wir eventuell mindestens so viele Beispiele, wie es unterscheidbare Bereiche gibt. Man kann sich jeden dieser Bereiche als eine Kategorie oder ein Symbol vorstellen: Durch einen anderen Freiheitsgrad für jedes Symbol (oder jeden Bereich) können wir eine beliebige Decoder-Zuordnung zwischen Symbolen und Werten erlernen. Allerdings erlaubt dies noch nicht die Generalisierung auf neue Symbole für neue Bereiche.

Mit etwas Glück ist die Zielfunktion nicht nur glatt, sondern weist auch eine gewisse Regelmäßigkeit auf. Zum Beispiel kann ein CNN mit Max-Pooling ein Objekt an jeder Position im Bild erkennen – auch dann, wenn die räumliche Verschiebung des Objekts nicht den glatten Transformationen im Eingaberaum entspricht.

Betrachten wir einen Sonderfall eines Lernalgorithmus für eine verteilte Repräsentation, der binäre Merkmale durch Schwellenwertbildung für lineare Funktionen der Eingabe extrahiert. Jedes binäre Merkmal in dieser Repräsentation unterteilt \mathbb{R}^d in ein Paar aus Halbräumen (vgl. Abbildung 15.7). Die exponentiell große Anzahl von Schnittpunkten von n der jeweiligen Halbräume bestimmt, wie viele Bereiche dieser Klassifikator für die verteilte Repräsentation (engl. *distributed representation learner*) unterscheiden kann. Wie viele Bereiche werden durch eine Anordnung von n Hyperebenen in \mathbb{R}^d

¹ Potenziell könnten wir eine Funktion erlernen wollen, deren Verhalten in exponentiell vielen Bereichen verschieden ist: In einem d -dimensionalen Raum mit mindestens 2 zu unterscheidenden verschiedenen Werten pro Dimension soll f in 2^d unterschiedlichen Bereichen verschieden sein, sodass $O(2^d)$ Trainingsbeispiele benötigt werden.

erzeugt? Durch Anwenden eines allgemeinen Ergebnisses hinsichtlich des Schnittpunkts der Hyperebenen (Zaslavsky, 1975) können wir die Anzahl der Bereiche, die diese binäre Merkmalsrepräsentation unterscheiden kann, bestimmen (Pascanu *et al.*, 2014b):

$$\sum_{j=0}^d \binom{n}{j} = O(n^d). \quad (15.4)$$

Daher gibt es ein exponentielles Wachstum der Eingabegröße und ein polynomiales Wachstum bei der Anzahl verdeckter Einheiten.

Wir gewinnen so ein geometrisches Argument zur Erläuterung der Fähigkeit zur Generalisierung der verteilten Repräsentation: Mit $O(nd)$ Parametern (für n lineare Schwellwertmerkmale in \mathbb{R}^d) können wir $O(n^d)$ Bereiche im Eingaberaum unterscheidbar darstellen. Wenn wir stattdessen keine Annahme über die Daten treffen und eine Repräsentation mit einem eindeutigen Symbol für jeden Bereich sowie separate Parameter für jedes Symbol zur Erkennung des zugehörigen Anteils von \mathbb{R}^d verwenden, benötigen wir zum Spezifizieren von $O(n^d)$ Bereichen $O(n^d)$ Beispiele. Allgemein formuliert: Das für die verteilte Repräsentation sprechende Argument lässt sich auf den Fall erweitern, in dem anstelle linearer Schwellwerteinheiten (engl. *linear threshold units*) nichtlineare, möglicherweise stetige Merkmalsextraktoren für jedes der Attribute in der verteilten Repräsentation verwendet werden. Das Argument lautet in diesem Fall, dass wir, wenn eine parametrische Transformation mit k Parametern über r Bereiche im Eingaberaum zu lernen vermag (für $k \ll r$) und das Erzielen einer solchen Repräsentation für die vorliegende Aufgabe nützlich war, auf diese Weise potenziell sehr viel besser generalisieren können als in einem nicht verteilten Kontext, in dem wir $O(r)$ Beispiele benötigen würden, um dieselben Merkmale und zugehörige Aufteilung des Eingaberaums in r Bereiche zu erreichen. Wenn wir weniger Parameter zur Repräsentation des Modells verwenden, müssen wir weniger Parameter anpassen und benötigen somit deutlich weniger Trainingsbeispiele für eine gute Generalisierung.

Ein weiterer Teil der Argumentation dafür, dass Modelle auf Basis verteilter Repräsentationen gut generalisieren, ist, dass ihre Kapazität eingeschränkt bleibt, obwohl sie in der Lage sind, so viele unterschiedliche Bereiche eindeutig zu codieren. So ist die Vapnik-Chervonenkis-Dimension eines neuronalen Netzes aus linearen Schwellwerteinheiten nur $O(w \log w)$, mit w als Anzahl der Gewichte (Sontag, 1998). Diese Einschränkung entsteht, da wir zwar sehr viele eindeutige Codes im Darstellungsraum zuweisen können, aber nicht den Code-Raum in seiner Gesamtheit nutzen können und auch keine beliebigen Funktionszuordnungen aus dem Darstellungsraum \mathbf{h}

zur Ausgabe \mathbf{y} mittels linearem Klassifikator erlernen können. Die Nutzung einer verteilten Repräsentation in Verbindung mit einem linearen Klassifikator drückt somit eine vorherige Überzeugung aus, dass die zu erkennenden Klassen als Funktion der zugrunde liegenden kausalen Faktoren, die von \mathbf{h} erfasst werden, linear separierbar sind. Wir möchten für gewöhnlich Kategorien erlernen wie die Menge aller Bilder von grünen Objekten oder die Menge aller Bilder mit Pkws. Kategorien, die eine nichtlineare XOR-Logik benötigen, interessieren uns dagegen nicht. Zum Beispiel möchten wir die Daten normalerweise nicht in eine Menge aller roten Pkws und grünen Lkws in der einen und eine Menge aller grünen Pkws und roten Lkws in einer anderen Klasse aufteilen.

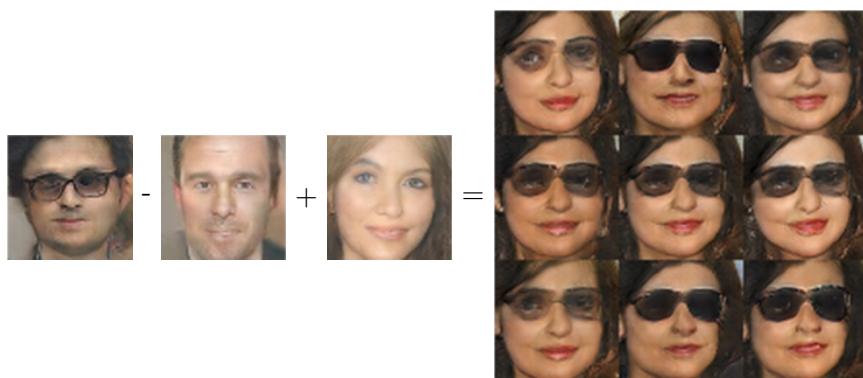


Abbildung 15.9: Ein generatives Modell hat eine verteilte Repräsentation erlernt, die das Konzept des Geschlechts vom Konzept des Brillentrags trennt. Wenn wir mit der Repräsentation des Konzepts für einen Mann mit Brille beginnen und dann den Vektor subtrahieren, der das Konzept für einen Mann ohne Brille darstellt, und abschließend den Vektor addieren, der das Konzept für eine Frau ohne Brille darstellt, erhalten wir den Vektor, der das Konzept einer Frau mit Brille darstellt. Das generative Modell decodiert all diese Repräsentationsvektoren korrekt für Bilder, die als der korrekten Klasse zugehörig erkannt werden können. (Abbildungen mit freundlicher Genehmigung von Radford et al. (2015))

Die bisher betrachteten Ideen waren abstrakt, können aber experimentell überprüft werden. Zhou et al. (2015) haben festgestellt, dass verdeckte Einheiten in einem tiefen CNN, das mit den ImageNet- und Places-Benchmark-Datensätzen trainiert wurde, Merkmale erlernen, die häufig interpretierbar sind und einer Kennzeichnung entsprechen, die im Normalfall auch von Menschen zugewiesen würde. In der Praxis ist es sicherlich nicht immer so, dass verdeckte Einheiten etwas erlernen, das einen einfachen linguistischen Namen trägt, aber es ist interessant zu sehen, wie dies auf den vorderen Plätzen der besten tiefen Netze für die Computer Vision geschieht. Solchen

Merkmale ist gemein, dass man sich vorstellen kann, *etwas über all diese Merkmale zu lernen, ohne alle Konfigurationen aller anderen Merkmale zu kennen*. Radford et al. (2015) haben gezeigt, dass ein generatives Modell eine Repräsentation von Fotos mit Gesichtern erlernen kann, wobei unterschiedliche Richtungen im Darstellungsraum unterschiedliche zugrunde liegende Faktoren der Variation erfassen. Abbildung 15.9 zeigt, dass eine Richtung im Darstellungsraum angibt, ob die Person männlich oder weiblich ist, eine andere dagegen, ob die Person eine Brille trägt. Diese Merkmale wurden automatisch erkannt, nicht etwa im Vorfeld festgelegt. Es ist nicht notwendig, Labels für die Klassifikatoren verdeckter Einheiten anzugeben: Das Gradientenabstiegsverfahren für eine untersuchte Zielfunktion erlernt natürlicherweise semantisch bedeutende Merkmale, sofern die Aufgabe diese Merkmale erfordert. Wir können etwas über die Unterscheidung zwischen Mann und Frau oder das Vorhandensein bzw. Fehlen von Brillen lernen, ohne alle Konfigurationen der $n - 1$ anderen Merkmale mit Beispielen für alle denkbaren Wertekombinationen charakterisieren zu müssen. Diese Art der statistischen Separierbarkeit erlaubt es, auf neue Konfigurationen der Merkmale einer Person zu generalisieren, die während des Trainings noch nicht aufgetreten sind.

15.5 Exponentielle Verbesserungen durch Tiefe

In Abschnitt 6.4.1 haben Sie gesehen, dass ein mehrschichtiges Perzeptron zur Klasse der universellen Approximatoren gehört und dass einige Funktionen durch gegenüber flachen Netzen exponentiell kleinere tiefe Netze dargestellt werden können. Diese Verringerung der Modellgröße führt zu einer verbesserten statistischen Effizienz. In diesem Abschnitt beschreiben wir, wie ähnliche Ergebnisse allgemeiner für andere Arten von Modellen mit verteilten verdeckten Repräsentationen gelten.

In Abschnitt 15.4 haben wir das Beispiel eines generativen Modells gezeigt, das etwas über die erklärenden Faktoren in Bildern von Gesichtern gelernt hat, darunter das Geschlecht einer Person und ob diese eine Brille trägt. Das generative Modell, das diese Aufgabe bewältigt hat, basierte auf einem tiefen neuronalen Netz. Es wäre nicht sinnvoll, anzunehmen, dass ein flaches Netz – beispielsweise ein lineares Netz – die komplizierte Beziehung zwischen diesen abstrakten erklärenden Faktoren und den Pixeln erlernen könnte. Bei dieser und anderen KI-Aufgaben sind die beinahe unabhängig voneinander wählbaren Faktoren, die dennoch aussagekräftigen Eingaben entsprechen, recht wahrscheinlich hochrangig und auf hochgradig nichtlineare

Arten mit der Eingabe verbunden. Wir sind der Meinung, dass hierfür *tiefe* verteilte Repräsentationen erforderlich sind, in denen die hochrangigeren Merkmale (die als Funktionen der Eingabe betrachtet werden) oder Faktoren (die als generative Ursachen betrachtet werden) durch die Zusammensetzung vieler Nichtlinearitäten ermittelt werden.

Es wurde für viele unterschiedliche Szenarios gezeigt, dass durch Organisieren der Berechnung durch Zusammensetzung vieler Nichtlinearitäten und eine Hierarchie wiederverwendeter Merkmale die statistische Effizienz erheblich verbessert werden kann – und zwar zusätzlich zur exponentiellen Verbesserung, die bereits durch den Einsatz einer verteilten Repräsentation entsteht. Für viele Arten von Netzen (z. B. mit sättigenden Nichtlinearitäten, booleschen Gates, Summen/Produkten oder Einheiten mit radialen Basisfunktionen) mit nur einer verdeckten Schicht kann ebenfalls gezeigt werden, dass es sich um universelle Approximatoren handelt. Eine Modellfamilie, die zu den universellen Approximatoren gehört, kann eine große Klasse von Funktionen (einschließlich aller stetigen Funktionen) bis zu einer von Null verschiedenen Toleranzschwelle approximieren, sofern genügend verdeckte Einheiten vorliegen. Allerdings werden möglicherweise sehr viele verdeckte Einheiten benötigt. Theoretische Ergebnisse hinsichtlich der Aussagekraft tiefer Architekturen besagen, dass es Funktionsfamilien gibt, die effizient durch eine Architektur der Tiefe k dargestellt werden können, die jedoch eine exponentielle Anzahl verdeckter Einheiten (bezüglich der Eingabegröße) mit unzureichender Tiefe (Tiefe 2 oder Tiefe $k - 1$) benötigen würden.

In Abschnitt 6.4.1 haben wir gesehen, dass deterministische Feedforward-Netze universelle Approximatoren von Funktionen sind. Viele strukturierte probabilistische Modelle mit nur einer verdeckten Schicht latenter Variablen, darunter RBMs (Restricted Boltzmann Machines) und DBNs (Deep-Belief-Netze) sind universelle Approximatoren von Wahrscheinlichkeitsverteilungen (*Le Roux und Bengio*, 2008, 2010; *Montúfar und Ay*, 2011; *Montúfar*, 2014; *Krause et al.*, 2013).

In Abschnitt 6.4.1 haben wir erfahren, dass ein hinreichend tiefes Feed-forward-Netz einen exponentiellen Vorteil gegenüber einem zu flachen Netz aufweisen kann. Derartige Ergebnisse lassen sich auch für andere Modelle wie probabilistische Modelle erzielen. Ein solches probabilistisches Modell ist das **Sum-Product-Netz**, kurz SPN (*Poon und Domingos*, 2011). Diese Modelle nutzen polynomiale Kreisläufe zum Berechnen der Wahrscheinlichkeitsverteilung über eine Menge von Zufallsvariablen. *Delalleau und Bengio* (2011) haben gezeigt, dass es Wahrscheinlichkeitsverteilungen gibt, für die ein SPN mit einer Mindesttiefe erforderlich ist, damit kein exponentiell großes Modell benötigt wird. Später haben *Martens und Medabalimi* (2014)

gezeigt, dass es erhebliche Unterschiede zwischen je zwei endlichen Tiefen von SPNs gibt und dass einige der Bedingungen, die eingesetzt werden, um SPNs effizient berechenbar zu machen, ihre Repräsentationsleistung beeinträchtigen können.

Eine weitere interessante Entwicklung ist eine Reihe theoretischer Ergebnisse für die Aussagekraft von Familien tiefer Kreisläufe in Verbindung mit CNNs; sie zeigen einen exponentiellen Vorteil tiefer Kreisläufe auch dann, wenn der flache Kreislauf die vom tiefen Kreislauf berechnete Funktion lediglich approximieren darf (*Cohen et al.*, 2015). Frühere theoretische Arbeiten hatten dagegen lediglich Fälle betrachtet, in denen der flache Kreislauf eine exakte Nachbildung bestimmter Funktionen sein musste.

15.6 Hinweise zum Aufdecken der zugrunde liegenden Ursachen

Am Ende dieses Kapitels kehren wir zu einer der anfangs gestellten Fragen zurück: Wodurch ist eine Repräsentation besser als eine andere? Eine Antwort, die wir zuerst in Abschnitt 15.3 gegeben haben, ist, dass eine ideale Repräsentation jene ist, die die zugrunde liegenden kausalen Faktoren der Variation, die die Daten generiert haben, separiert, und zwar vor allem die Faktoren, die für unsere Anwendungen relevant sind. Die meisten Verfahren beim Representation Learning beruhen auf anfänglichen Hinweisen, die beim Suchen dieser zugrunde liegenden Faktoren der Variation hilfreich sind. Die Hinweise können dem Klassifikator dabei helfen, diese beobachteten Faktoren von den anderen zu unterscheiden. Überwachtes Lernen bietet einen sehr starken Hinweis, nämlich ein Label y für jedes x , die meist den Wert mindestens einer der Faktoren der Variation direkt angibt. Allgemein nutzt Representation Learning, um eine große Menge Daten ohne Label zu verwenden, andere, weniger direkte Hinweise zu den zugrunde liegenden Faktoren. Diese Hinweise treten in Form impliziter Annahmen auf, die wir als Entwickler des Lernalgorithmus vorgeben, um den Klassifikator anzuleiten. Ergebnisse wie das No-Free-Lunch-Theorem zeigen, dass für eine gute Fähigkeit zur Generalisierung Regularisierungsverfahren notwendig sind. Es ist unmöglich, ein universell überlegenes Regularisierungsverfahren zu finden. Dennoch besteht eines der Ziele beim Deep Learning darin, eine Reihe recht generischer Regularisierungsverfahren zu ermitteln, die für eine Vielzahl von KI-Aufgaben einsetzbar sind – ähnlich den Aufgaben, die Menschen und Tiere lösen können.

Wir stellen hier eine Liste dieser generischen Regularisierungsverfahren vor. Natürlich ist diese Aufstellung nicht erschöpfend, aber sie enthält einige konkrete Beispiele dafür, wie Lernalgorithmen dazu animiert werden können, Merkmale aufzudecken, die den zugrunde liegenden Faktoren entsprechen. Die Liste stammt ursprünglich aus Abschnitt 3.1 von *Bengio et al.* (2013d) und wurde für dieses Buch erweitert.

- *Glättheit* (engl. *smoothness*): Dies ist die Annahme, dass $f(\mathbf{x} + \epsilon \mathbf{d}) \approx f(\mathbf{x})$ ist für Einheit \mathbf{d} und kleines ϵ . Diese Annahme erlaubt es dem Klassifikator, von Trainingsbeispielen auf nahegelegene Punkte im Eingaberaum zu generalisieren. Viele Machine-Learning-Algorithmen nutzen diese Idee, aber sie reicht nicht aus, um den Fluch der Dimensionalität zu überwinden.
- *Linearität*: Viele Lernalgorithmen gehen davon aus, dass Beziehungen zwischen einigen Variablen linear sind. Das erlaubt es dem Algorithmus, Vorhersagen auch weitab der beobachteten Daten zu treffen, kann aber manchmal zu besonders extremen Vorhersagen führen. Die meisten einfachen Machine-Learning-Algorithmen, die auf die Glättungsannahme verzichten, nutzen stattdessen die Linearitätsannahme. Tatsächlich handelt es sich um unterschiedliche Annahmen – lineare Funktionen mit großen Gewichtungen in hochdimensionalen Räumen sind eventuell nicht sehr glatt. *Goodfellow et al.* (2014b) enthält eine weitere Betrachtung der Einschränkungen, die sich aus der Annahme der Linearität ergeben.
- *Mehrere erklärende Faktoren*: Viele Representation-Learning-Algorithmen beruhen auf der Annahme, dass die Daten durch mehrere zugrunde liegende erklärende Faktoren generiert werden und dass die meisten Aufgaben sich problemlos lösen lassen, wenn der Zustand der einzelnen Faktoren bekannt ist. Abschnitt 15.3 beschreibt, wie diese Ansicht über das Representation Learning zum halb-überwachten Lernen führt. Für das Erlernen der Struktur von $p(\mathbf{x})$ müssen einige derselben Merkmale erlernt werden, die auch zum Modellieren von $p(\mathbf{y} | \mathbf{x})$ dienlich sind, da sich beide auf dieselben zugrunde liegenden erklärenden Faktoren beziehen. Abschnitt 15.4 beschreibt, wie diese Ansicht zum Einsatz verteilter Repräsentationen führt, mit separaten Richtungen im Darstellungsraum, die den einzelnen Faktoren der Variation entsprechen.
- *Kausale Faktoren*: Das Modell wird so aufgebaut, dass die mithilfe der erlernten Repräsentation \mathbf{h} beschriebenen Faktoren der Variation

als Ursachen der beobachteten Daten \mathbf{x} behandelt werden und nicht umgekehrt. Wie in Abschnitt 15.3 gezeigt, ist dies beim halb-überwachten Lernen von Vorteil und macht das erlernte Modell robuster, wenn sich die Verteilung über die zugrunde liegenden Ursachen ändert oder wir das Modell für eine neue Aufgabe nutzen.

- *Tiefe* oder *eine hierarchische Organisation der erklärenden Faktoren*: Hochrangige abstrakte Konzepte können anhand von einfachen Konzepten in Form einer Hierarchie definiert werden. Anders betrachtet drückt die Nutzung einer tiefen Architektur unsere Überzeugung aus, dass die Aufgabe mit einem mehrstufigen Programm erledigt werden könnte, wobei jeder Schritt auf das Verarbeitungsergebnis (Ausgabe) der vorherigen Schritte zurückgreift.
- *Aufgabenübergreifende Faktoren*: Wenn wir viele Aufgaben haben, die unterschiedlichen y_i -Variablen für dieselbe Eingabe \mathbf{x} entsprechen, oder wenn jede Aufgabe mit einer Teilmenge oder einer Funktion $f^{(i)}(\mathbf{x})$ einer globalen Eingabe \mathbf{x} verknüpft ist, gilt die Annahme, dass jedes y_i mit einer anderen Teilmenge aus einem gemeinsamen Pool relevanter Faktoren \mathbf{h} verknüpft ist. Da sich diese Teilmengen überschneiden, ermöglicht das Erlernen aller $P(y_i | \mathbf{x})$ mittels gemeinsam genutzter Zwischenrepräsentation $P(\mathbf{h} | \mathbf{x})$ die gemeinsame Nutzung der statistischen Aussagekraft zwischen den Aufgaben.
- *Mannigfaltigkeiten*: Die Ballung der Wahrscheinlichkeit(smasse) und die Bereiche, in denen diese Konzentration auftritt, sind lokal verbunden und nehmen nur einen winzigen Raum ein. Im stetigen Fall können diese Räume mittels geringdimensionaler Mannigfaltigkeiten mit einer gegenüber dem ursprünglichen Raum, der die Daten enthält, sehr viel kleineren Dimensionalität approximiert werden. Viele Machine-Learning-Algorithmen arbeiten nur auf dieser Mannigfaltigkeit vernünftig (Goodfellow et al., 2014b). Einige Machine-Learning-Algorithmen, insbesondere Autoencoder, versuchen, die Struktur der Mannigfaltigkeit explizit zu erlernen.
- *Natürliches Clustering*: Viele Machine-Learning-Algorithmen gehen davon aus, dass jede verbundene Mannigfaltigkeit im Eingaberaum einer einzelnen Klasse zugewiesen werden kann. Die Daten können auf vielen nicht verbundenen Mannigfaltigkeiten liegen, aber die Klasse bleibt in jeder dieser Mannigfaltigkeiten gleich. Diese Annahme führt zu einer Reihe von Lernalgorithmen, darunter Tangenten-Propagation,

Double Backpropagation, Mannigfaltigkeit-Tangentenklassifikator und Adversarial Training.

- *Temporale und räumliche Kohärenz:* Die Slow Feature Analysis (SFA) und verwandte Algorithmen gehen davon aus, dass die wichtigen erklärenden Faktoren sich im Laufe der Zeit langsam verändern oder dass es zumindest einfacher ist, die wahren zugrunde liegenden, erklärenden Faktoren vorherzusagen als die unbearbeiteten Eingaben (beispielsweise Pixelwerte). Abschnitt 13.3 geht näher auf diesen Ansatz ein.
- *Dünne Besetzung* (engl. *sparsity*): Die meisten Merkmale sollten für die Beschreibung der meisten Eingaben vermutlich nicht relevant sein – es ist nicht notwendig, ein Merkmal zu verwenden, das Elefantenrüssel erkennt, wenn das Bild einer Katze dargestellt wird. Es ist daher sinnvoll, vorzugeben, dass ein Merkmal, das entweder »vorhanden« oder »nicht vorhanden« sein kann, in den meisten Fällen nicht vorhanden ist.
- *Einfachheit der Faktorabhängigkeiten:* In guten hochrangigen Repräsentationen sind die Faktoren untereinander über einfache Abhängigkeiten verbunden. Die einfachste davon ist die marginale Unabhängigkeit, $P(\mathbf{h}) = \prod_i P(\mathbf{h}_i)$, aber auch lineare Abhängigkeiten oder solche, die von einem flachen Autoencoder erfasst werden, sind sinnvolle Annahmen. Das lässt sich in vielen physikalischen Gesetzen zeigen und wird angenommen, wenn ein linearer Prädiktor oder eine faktorierte A-priori-Wahrscheinlichkeit zusätzlich zu einer erlernten Repräsentation eingesetzt wird.

Das Konzept des Representation Learnings verbindet all die vielen Formen des Deep Learnings miteinander. Feedforward-Netze und RNNs, Autoencoder und tiefe probabilistische Modelle lernen und erkunden allesamt Repräsentationen. Das Erlernen der bestmöglichen Repräsentation ist nach wie vor ein aktives Forschungsgebiet.

16

Strukturierte probabilistische Modelle für Deep Learning

Deep Learning kennt viele Modellierungsformalismen, die in der Forschung als Richtlinien für den Entwurf und die Beschreibung von Algorithmen dienen können. Einer dieser Formalismen ist das Konzept der **strukturierten probabilistischen Modelle**. In Abschnitt 3.14 werden diese kurz behandelt. Die dortige kurze Vorstellung ist ausreichend, wenn es um den Einsatz strukturierter probabilistischer Modelle als Darstellungsform einiger der Algorithmen in Teil II geht. In Teil III sind sie allerdings eine Grundvoraussetzung für die meisten wichtigen Forschungsthemen im Deep Learning. Um die Abhandlung dieser Forschungskonzepte vorzubereiten, gehen wir in diesem Kapitel sehr viel detaillierter auf strukturierte probabilistische Modelle ein. Dieses Kapitel ist in sich abgeschlossen. Sie müssen die frühere Einführung nicht (nochmals) lesen, bevor Sie fortfahren.

Ein strukturiertes probabilistisches Modell ist eine Möglichkeit zum Beschreiben einer Wahrscheinlichkeitsverteilung. Dabei wird ein Graph verwendet, der angibt, welche Zufallsvariablen in der Wahrscheinlichkeitsverteilung direkt miteinander interagieren. Hier wird das Wort »Graph« im Sinne der Graphentheorie verwendet: eine Reihe von Ecken, die über Kanten miteinander verbunden sind. Da die Struktur des Modells über einen Graphen definiert wird, bezeichnet man diese Modelle häufig auch als **graphische Modelle**.

Es existiert eine große Forschungsgemeinde, die viele unterschiedliche Modelle, Trainingsalgorithmen und Inferenzalgorithmen entwickelt hat. In

diesem Kapitel legen wir die Grundlage für einige der zentralen Ideen für graphische Modelle, wobei wir uns besonders den Konzepten widmen, die sich für die Deep-Learning-Forschung als nützlich erwiesen haben. Wenn Sie bereits gut mit graphischen Modellen vertraut sind, können Sie dieses Kapitel wahrscheinlich überfliegen. Allerdings können selbst Experten für graphische Modelle vom letzten Teil des Kapitels, Abschnitt 16.7, profitieren, in dem wir einige der besonderen Einsatzarten graphischer Modelle für Deep-Learning-Algorithmen vorstellen. Diejenigen, die Deep Learning in der Praxis einsetzen, neigen dazu, Modellstrukturen, Lernalgorithmen und Inferenzverfahren zu verwenden, die sich stark von denen der restlichen Forschungsgemeinde für graphische Modelle unterscheiden. In diesem Kapitel benennen wir diese Unterschiede und erläutern die Gründe dafür.

Zunächst beschreiben wir die Herausforderungen, die beim Erstellen umfangreicher probabilistischer Modelle auftreten. Dann zeigen wir, wie die Struktur einer Wahrscheinlichkeitsverteilung mithilfe eines Graphen beschrieben werden kann. Zwar lassen sich damit viele Schwierigkeiten überwinden, aber der Ansatz ist selbst nicht ganz unproblematisch. Eine der größten Schwierigkeiten bei der graphischen Modellierung ist es, zu verstehen, welche Variablen direkt miteinander interagieren können müssen, welche Graphen-Strukturen also für ein bestimmtes Problem am besten geeignet sind. In Abschnitt 16.5 umreißen wir zwei Ansätze zum Lösen dieser Schwierigkeit durch das Lernen der Abhängigkeiten. Abschließend erfahren Sie in Abschnitt 16.7, welche Bedeutung diejenigen, die Deep Learning in der Praxis einsetzen, bestimmten Ansätzen für die graphische Modellierung beimessen.

16.1 Die Herausforderung der unstrukturierten Modellierung

Das Ziel von Deep Learning ist es, Machine Learning passend zu den Herausforderungen einzusetzen, die zum Erreichen Künstlicher Intelligenz bewältigt werden müssen. Dazu wird ein Verständnis hochdimensionaler Daten mit umfassender Struktur benötigt. Wir möchten zum Beispiel, dass KI-Algorithmen natürliche Bilder,¹ Audio-Wellenformen, die Sprache repräsentieren, sowie Dokumente, die mehrere Wörter und Satzzeichen enthalten, verstehen.

¹ Ein **natürliches Bild** ist ein Bild, das beispielsweise mit einer Kamera in einer hinreichend gewöhnlichen Umgebung aufgenommen wird. Das Gegenstück sind synthetisch generierte Bilder, Screenshots einer Webseite usw.

Klassifizierungsalgorithmen können eine Eingabe von einer solch umfangreichen hochdimensionalen Verteilung entgegennehmen und mit einem kategorischen Label zusammenfassen: das Objekt in einem Foto, das gesprochene Wort in einer Aufzeichnung, das Thema eines Dokuments. Bei der Klassifizierung werden die meisten Informationen aus der Eingabe verworfen und es wird eine einzelne Ausgabe (oder eine Wahrscheinlichkeitsverteilung über Werte dieser einzelnen Ausgabe) erzeugt. Der Klassifikator ist häufig auch in der Lage, viele Teile der Eingabe zu ignorieren. Bei der Objekterkennung in Fotografien kann der Hintergrund zum Beispiel im Normalfall ignoriert werden.

Es ist möglich, probabilistische Modelle auch für viele andere Aufgaben zu nutzen. Diese sind häufig aufwendiger als die Klassifizierung. Für einige müssen mehrere Ausgabewerte erzeugt werden. Die meisten setzen ein vollständiges Verständnis der gesamten Eingabestruktur voraus und bieten keine Möglichkeit, Teile davon zu ignorieren. Einige dieser Aufgaben haben wir hier aufgeführt:

- *Dichteschätzung*: Das Machine-Learning-System gibt für eine Eingabe \mathbf{x} eine Schätzung der wahren Dichte $p(\mathbf{x})$ unter der datengenerierenden Verteilung aus. Dazu wird nur eine Ausgabe benötigt, aber zusätzlich auch ein vollständiges Verständnis der gesamten Eingabe. Wenn nur ein Element des Vektors ungewöhnlich ist, muss das System ihm eine niedrige Wahrscheinlichkeit zuweisen.
- *Denoising (Entrauschen)*: Das Machine-Learning-System gibt für eine beschädigte oder fehlerhaft beobachtete Eingabe $\tilde{\mathbf{x}}$ einen Schätzwert für das ursprüngliche oder korrekte \mathbf{x} aus. Ein Beispiel wäre das Entfernen von Staub oder Kratzern auf alten Fotografien. Hier sind mehrere Ausgaben (jedes Element des geschätzten einwandfreien Beispiels \mathbf{x}) und ein Verständnis der gesamten Eingabe erforderlich (da bereits ein beschädigter Bereich dazu führt, dass der endgültige Schätzwert als beschädigt gilt).
- *Ersetzen fehlender Werte (auch Imputation)*: Das Modell muss aus den Beobachtungen einiger Elemente von \mathbf{x} Schätzwerte für oder eine Wahrscheinlichkeitsverteilung über alle nicht beobachteten Elemente von \mathbf{x} ausgeben. Hier werden mehrere Ausgaben benötigt. Da die Aufgabe des Modells in der Wiederherstellung beliebiger Elemente aus \mathbf{x} bestehen kann, muss es die gesamte Eingabe verstehen.
- *Stichprobenverfahren*: Das Modell erzeugt neue Stichproben aus der Verteilung $p(\mathbf{x})$. Eine Beispielanwendung ist die Sprachsynthese, also

das Erzeugen neuer Audio-Wellenformen, die wie menschliche Sprache klingen. Hierzu werden mehrere Ausgabewerte und ein gutes Modell der gesamten Eingabe benötigt. Wenn die Stichproben auch nur ein Element enthalten, das aus der falschen Verteilung gezogen wurde, ist die Stichprobenentnahme falsch.

Ein Beispiel für eine Aufgabe mit kleinen natürlichen Bildern finden Sie in Abbildung 16.1.

Das Modellieren einer umfangreichen Verteilung über Tausende oder Millionen von Zufallsvariablen ist eine echte Herausforderung – rechnerisch ebenso wie statistisch. Angenommen, wir möchten nur binäre Variablen modellieren. Das ist der einfachste denkbare Fall und doch erscheint er immens. Für kleine Farbbilder (RGB) mit einer Auflösung von 32×32 Pixeln gibt es 2^{3072} mögliche binäre Bilder dieser Form. Diese Zahl ist mehr als 10^{800} größer als die geschätzte Anzahl von Atomen im Universum.

Wenn wir eine Verteilung über einen Zufallsvektor \mathbf{x} mit n diskreten Variablen, die jeweils k Werte annehmen können, modellieren möchten, benötigen wir für den naiven Ansatz zur Darstellung von $P(\mathbf{x})$ durch Speichern einer Lookup-Tabelle mit einem Wahrscheinlichkeitswert pro möglichem Ausgang k^n Parameter!

Das ist aus mehreren Gründen nicht durchführbar:

- *Speicherbedarf – der Aufwand zum Speichern der Repräsentation:* Für alle bis auf sehr kleine Werte von n und k müssen zur Repräsentation der Verteilung als Tabelle zu viele Werte gespeichert werden.
- *Statistische Effizienz:* Mit wachsender Zahl der Parameter in einem Modell wächst auch die Menge der benötigten Trainingsdaten für die Auswahl der Werte dieser Parameter mit einem statistischen Schätzer. Da das tabellenbasierte Modell eine astronomische Anzahl von Parametern aufweist, wird eine astronomisch große Trainingsdatenmenge für eine exakte Anpassung benötigt. Ein solches Modell führt zu einer sehr nachteiligen Überanpassung der Trainingsdatenmenge, sofern nicht weitere Annahmen gemacht werden, die die verschiedenen Einträge in der Tabelle verknüpfen (wie in Backoff- oder geglätteten N -Gramm-Modellen, vgl. Abschnitt 12.4.1).
- *Laufzeit – die Kosten für die Inferenz:* Angenommen, wir möchten eine Inferenzaufgabe durchführen, bei der wir unser Modell der multivariaten Verteilung $P(\mathbf{x})$ zur Berechnung einer anderen Verteilung nutzen, bspw. der Randverteilung $P(x_1)$ oder der bedingten Verteilung

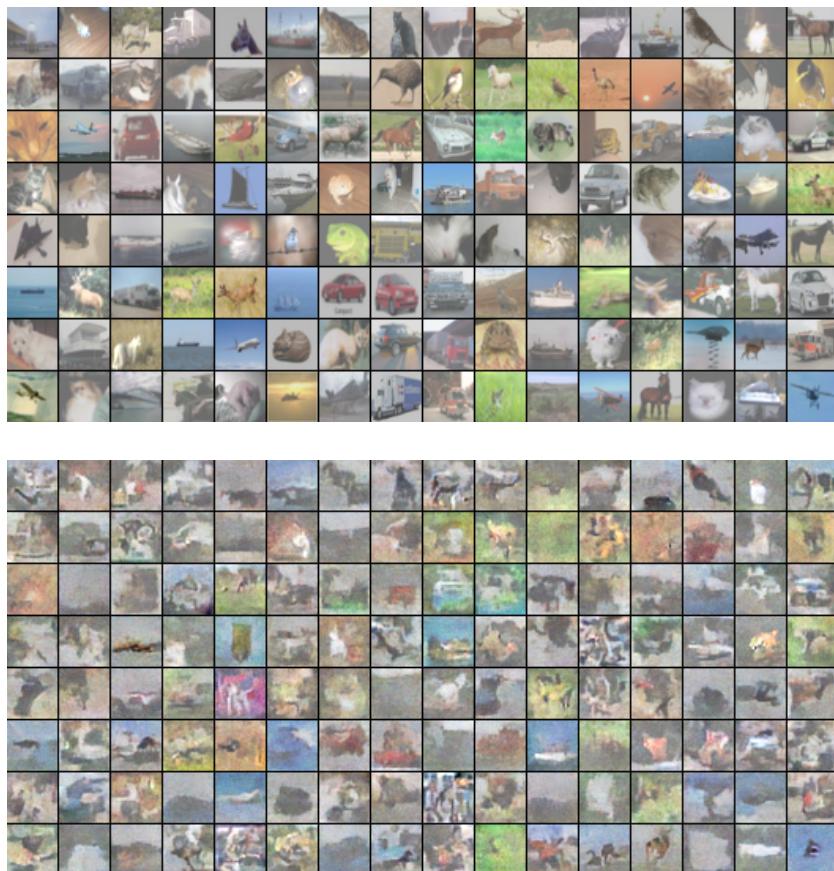


Abbildung 16.1: Probabilistische Modellierung natürlicher Bilder. (*Oben*) Beispieldbilder mit einer Auflösung von 32×32 Pixeln aus dem CIFAR-10-Datensatz (Krizhevsky und Hinton, 2009). (*Unten*) Aus einem strukturierten probabilistischen Modell, das mit diesem Datensatz trainiert wurde, gezogene Stichproben. Jede Stichprobe erscheint an derselben Position im Raster wie das Trainingsbeispiel, dem es im euklidischen Raum am nächsten liegt. Der Vergleich erlaubt uns zu erkennen, dass das Modell tatsächlich neue Bilder synthetisch erzeugt und nicht einfach Trainingsdaten aus dem Speicher wiedergibt. Der Kontrast der beiden Bildreihen wurde für die Darstellung angepasst. (Abbildung mit freundlicher Genehmigung von Courville et al. (2011))

$P(x_2 | x_1)$. Das Berechnen dieser Verteilung erfordert im ungünstigsten Fall eine Aufsummierung über die gesamte Tabelle, sodass die Laufzeit für diese Operationen ebenso hoch wie der Speicherbedarf zum Vorhalten des Modells ineffizient ist.

- *Laufzeit – die Kosten für die Stichprobenentnahme:* Nehmen wir ebenso an, dass wir eine Stichprobe aus dem Modell ziehen (sampeln) möchten. Der naive Weg besteht im Ziehen eines Werts $u \sim U(0, 1)$ und anschließender Iteration durch die Tabelle, wobei wir die Wahrscheinlichkeitswerte addieren, bis sie u übersteigen, und dann das Ergebnis für diese Position in der Tabelle ausgeben. Dazu müssen wir im schlechtesten Fall die gesamte Tabelle durchgehen, sodass derselbe exponentielle Aufwand wie bei den anderen Operationen entsteht.

Das Problem mit dem tabellenbasierten Ansatz besteht darin, dass wir jede mögliche Art der Interaktion zwischen allen möglichen Teilmengen der Variablen explizit modellieren. Die Wahrscheinlichkeitsverteilungen, mit denen wir es in der Realität zu tun haben, sind sehr viel einfacher gestrickt. Üblicherweise beeinflussen die meisten Variablen einander nur indirekt.

Betrachten Sie zum Beispiel das Modellieren der Zieleinlaufzeiten eines Teams in einem Staffellauf. Nehmen wir an, das Team besteht aus drei Läufern: Alice, Bob und Carol. Zu Beginn des Laufs hält Alice den Stab und beginnt mit ihrer Runde. Nachdem sie ihre Runde beendet hat, übergibt sie den Staffelstab an Bob. Bob läuft seine eigene Runde und übergibt den Stab an Carol, die die letzte Runde läuft. Wir können die einzelnen Zieleinlaufzeiten der Läufer als stetige Zufallsvariable modellieren. Die Zeit von Alice hängt von niemandem sonst ab, denn sie läuft als Erste. Bobs Einlaufzeit ist von der von Alice abhängig, denn er kann seine Runde erst beginnen, wenn Alice die ihre abgeschlossen hat. Wenn Alice schneller läuft, gelangt Bob – sofern alle anderen Umstände gleich bleiben – früher ans Ziel. Carols Zieleinlaufzeit schließlich ist von den Zeiten der beiden anderen Teammitglieder abhängig. Wenn Alice langsam ist, kommt auch Bob vermutlich später als gedacht an. Daraus folgt, dass Carol ihre Runde erst sehr spät beginnt, sodass sie wahrscheinlich zu einem späten Zeitpunkt die Ziellinie überquert. Allerdings hängt die Zieleinlaufzeit von Carol nur *indirekt* über Bobs Zeit von der Einlaufzeit von Alice ab. Wenn wir bereits wissen, wann Bob das Ziel erreicht hat, können wir Carols Zeit auch nach einem Blick auf die Zeit von Alice nicht besser schätzen. Damit lässt sich der Staffellauf mit lediglich zwei Interaktionen modellieren: Die Auswirkung von Alice auf Bob und die von Bob auf Carol. Die dritte, indirekte Interaktion zwischen Alice und Carol wird im Modell nicht benötigt.

Strukturierte probabilistische Modelle stellen ein formales Framework für die Modellierung ausschließlich unmittelbarer Interaktionen zwischen Zufallsvariablen zur Verfügung. Damit kommen die Modelle mit erheblich weniger Parametern aus und können auf der Grundlage einer geringeren Datenmenge zuverlässig geschätzt werden. Diese kleineren Modelle haben auch den Berechnungsaufwand hinsichtlich des Speicherbedarfs, des Durchföhrens von Inferenz im Modell und der Stichprobenentnahme aus dem Modell deutlich reduziert.

16.2 Verwenden von Graphen zum Beschreiben der Modellstruktur

Strukturierte probabilistische Modelle nutzen Graphen (im Sinne der Graphentheorie, also über Kanten verbundene »Knoten« oder »Ecken«) zur Repräsentation der Interaktionen zwischen Zufallsvariablen. Jeder Knoten steht für eine Zufallsvariable. Jede Kante steht für eine direkte Interaktion. Diese direkten Interaktionen implizieren andere indirekte Interaktionen, aber nur die direkten Interaktionen müssen explizit modelliert werden.

Es gibt mehrere Möglichkeiten zum Beschreiben der Interaktionen in einer Wahrscheinlichkeitsverteilung anhand eines Graphen. In den folgenden Abschnitten stellen wir einige der beliebtesten und nützlichsten Ansätze vor. Graphische Modelle lassen sich hauptsächlich in zwei Kategorien unterteilen: Modelle auf Basis gerichteter azyklischer Graphen und Modelle auf Basis ungerichteter Graphen.

16.2.1 Gerichtete Modelle

Eine Art des strukturierten probabilistischen Modells ist das **gerichtete graphische Modell**, auch bekannt als **Belief-Netz** oder **Bayes-Netz**² (Pearl, 1985).

Gerichtete graphische Modelle heißen »gerichtet«, weil ihre Kanten gerichtet sind, also von einer Ecke auf eine andere verweisen. Diese Richtung wird in der Abbildung durch einen Pfeil gekennzeichnet. Die Pfeilrichtung gibt an, zu welcher Variable die Wahrscheinlichkeitsverteilung gehört, die durch die Verteilung der anderen definiert wird. Ein Pfeil von a zu b bedeutet,

² Judea Pearl hat vorgeschlagen, den Begriff »Bayes-Netz« zu verwenden, wenn die beurteilende Natur der vom Netz berechneten Werte betont werden soll, d. h., um hervorzuheben, dass sie üblicherweise Grade der Überzeugung (engl. *degrees of belief*) und nicht Ereignishäufigkeiten ausdrücken.

dass wir die Wahrscheinlichkeitsverteilung über b mittels einer bedingten Verteilung definieren, wobei a eine der Variablen auf der rechten Seite des Bedingungszeichens ist. Anders ausgedrückt: Die Verteilung über b ist vom Wert für a abhängig.

Ziehen wir erneut den Staffellauf aus Abschnitt 16.1 als Beispiel heran und benennen wir die Zieleinlaufzeit von Alice mit t_0 , die von Bob mit t_1 und die von Carol mit t_2 . Wie wir bereits gezeigt haben, ist unsere Schätzung für t_1 abhängig von t_0 . Unser Schätzwert für t_2 ist direkt abhängig von t_1 , aber nur indirekt von t_0 . Wir können diese Beziehung als gerichtetes graphisches Modell wie in Abbildung 16.2 zeichnen.

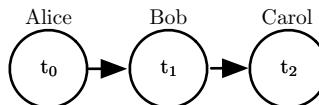


Abbildung 16.2: Ein gerichtetes graphisches Modell für unseren Staffellauf. Die Zieleinlaufzeit t_0 von Alice beeinflusst die Zieleinlaufzeit t_1 von Bob, da Bob erst mit dem Lauf beginnen kann, wenn der Lauf von Alice endet. Ebenso kann Carol erst loslaufen, wenn Bob ihr den Stab übergibt. Somit beeinflusst die Zieleinlaufzeit von Bob, t_1 , die Zeit t_2 von Carol direkt.

Formal wird ein für die Variablen \mathbf{x} definiertes gerichtetes graphisches Modell durch einen gerichteten azyklischen Graphen \mathcal{G} , dessen Ecken die Zufallsvariablen im Modell sind, sowie eine Menge **lokal bedingter Wahrscheinlichkeitsverteilungen** $p(x_i | Pa_{\mathcal{G}}(x_i))$ definiert, wobei $Pa_{\mathcal{G}}(x_i)$ die Eltern von x_i in \mathcal{G} bestimmt. Die Wahrscheinlichkeitsverteilung über \mathbf{x} ergibt sich aus

$$p(\mathbf{x}) = \prod_i p(x_i | Pa_{\mathcal{G}}(x_i)). \quad (16.1)$$

In unserem Staffellauf bedeutet dies für den Graphen aus Abbildung 16.2 Folgendes:

$$p(t_0, t_1, t_2) = p(t_0)p(t_1 | t_0)p(t_2 | t_1). \quad (16.2)$$

Gratulation! Sie haben soeben erstmals ein strukturiertes probabilistisches Modell in Aktion gesehen! Wir können den Aufwand für seine Anwendung untersuchen und feststellen, inwiefern die strukturierte Modellierung viele Vorteile gegenüber der unstrukturierten Modellierung aufweist.

Angenommen, wir würden den Zeitverlauf in den Minuten 0 bis 10 in 6 Sekunden große Blöcke einteilen. Damit sind t_0 , t_1 und t_2 jeweils diskrete Variablen mit 100 möglichen Werten. Wenn wir nun versuchen, $p(t_0, t_1, t_2)$ als Tabelle darzustellen, müssten wir 999.999 Werte speichern (100 Werte für $t_0 \times 100$ Werte für $t_1 \times 100$ Werte für t_2 minus 1, da die Wahrscheinlichkeit

einer der Konfigurationen redundant wird, denn es gilt die Bedingung, dass die Summe der Wahrscheinlichkeiten 1 ergeben muss). Wenn wir stattdessen eine Tabelle nur für die bedingten Wahrscheinlichkeitsverteilungen erstellen, werden für die Verteilung über t_0 genau 99 Werte benötigt; die Tabelle zur Definition von t_1 aus t_0 erfordert 9.900 Werte, ebenso die Tabelle für t_2 aus t_1 . Insgesamt kommen wir damit auf 19.899 Werte. Durch den Einsatz eines gerichteten graphischen Modells können wir also die Anzahl der Parameter um einen Faktor über 50 reduzieren!

Grundsätzlich liegt der Aufwand für die Modellierung von n diskreten Variablen mit jeweils k Werten in einer Tabelle im Bereich $O(k^n)$, wie wir bereits festgestellt haben. Nehmen wir nun an, dass wir ein gerichtetes graphisches Modell über diese Variablen erstellen. Wenn m die maximale Anzahl der in einer einzelnen Wahrscheinlichkeitsverteilung auftretenden Variablen (auf einer der Seiten des Bedingungszeichens) ist, beträgt der skalierte Aufwand für die Tabellen des gerichteten Modells $O(k^m)$. Sofern wir ein Modell mit $m \ll n$ erstellen können, führt dies zu erheblichen Einsparungen.

Mit anderen Worten: Sofern jede Variable wenige Elternknoten im Graphen besitzt, kann die Verteilung mit sehr wenigen Parametern dargestellt werden. Einige Einschränkungen der Graphenstruktur – wie »es muss sich um einen Baum handeln« – können außerdem garantieren, dass Operationen wie das Berechnen von Rand- oder bedingten Verteilungen über Teilmengen von Variablen effizient sind.

Es ist wichtig zu verstehen, welche Informationen im Graphen codiert werden können und welche nicht. Der Graph codiert nur vereinfachende Annahmen darüber, welche Variablen bedingt unabhängig voneinander sind. Es ist auch möglich, weitere Arten vereinfachender Annahmen zu treffen. Wir können zum Beispiel annehmen, dass Bobs Lauf stets gleich ist – egal wie Alice abschneidet. (In der Realität beeinflusst Alices Leistung wahrscheinlich die von Bob – wenn Alice zum Beispiel in einem Lauf besonders schnell unterwegs ist, könnte das, abhängig von Bobs Persönlichkeit, dazu führen, dass er sich ebenfalls besonders viel Mühe gibt, um mit ihrer hervorragenden Leistung gleichzuziehen – oder er wird zu selbstsicher und träge.) Damit gibt es nur noch eine Auswirkung, die Alice auf Bobs Zieleinlaufzeit hat, nämlich das Addieren ihrer Einlaufzeit zur Gesamtzeit, die wir von Bob für dessen Lauf erwarten. Diese Beobachtung führt dann zu einem Modell mit $O(k)$ anstelle von $O(k^2)$ Parametern. Beachten Sie jedoch, dass t_0 und t_1 auch für diese Annahme in einem direkten Abhängigkeitsverhältnis stehen, da t_1 für die Absolutzeit (Uhrzeit) steht, zu der Bob seinen Lauf abschließt, nicht für die Zeit, die er für seine Runde benötigt. Unser Graph muss also

noch immer einen Pfeil von t_0 zu t_1 enthalten. Die Annahme, dass Bobs persönliche Laufzeit unabhängig von allen anderen Faktoren ist, lässt sich in einem Graphen über t_0 , t_1 und t_2 nicht ausdrücken. Stattdessen codieren wir diese Information in der Definition der bedingten Verteilung selbst. Die bedingte Verteilung ist nicht länger eine Tabelle mit $k \times k - 1$ Elementen und Indizierung durch t_0 und t_1 , sondern eine etwa komplexere Formel, die lediglich $k - 1$ Parameter verwendet. Die Syntax für das gerichtete graphische Modell erlegt uns keine Bedingungen beim Definieren unserer bedingten Verteilungen auf. Sie definiert nur, welche Variablen als Argumente zulässig sind.

16.2.2 Ungerichtete Modelle

Gerichtete graphische Modelle stellen eine Repräsentationsform strukturierter probabilistischer Modelle dar. Eine weitere beliebte Repräsentationsform sind **ungerichtete Modelle**, die auch als **Markow-Zufallsfelder** oder **Markow-Netze** bezeichnet werden (engl. *Markov random fields*, MRF bzw. *Markov networks*, (Kindermann, 1980)). Wie der Name andeutet, verwenden ungerichtete Modelle Graphen mit ungerichteten Kanten.

Gerichtete Modelle sind besonders geeignet für Fälle, in denen es einen klaren Grund dafür gibt, jeden Pfeil in eine bestimmte Richtung zu zeichnen. Häufig geht es dabei um das Verständnis von Kausalität, die ja nur in eine Richtung wirkt. Ein Beispiel dafür ist unser Staffellauf. Frühe Läufer beeinflussen die Zieleinlaufzeit der späteren Läufer, aber spätere Läufer nicht die der früheren Läufer.

Dabei haben nicht alle Fälle, die wir möglicherweise modellieren wollen, eine so klare Richtung der Interaktionen. Wenn die Interaktionen keine intrinsische Richtung zu haben scheinen oder scheinbar in beiden Richtungen funktionieren, könnte ein ungerichtetes Modell die bessere Wahl sein.

Als Beispiel dafür dient das Modell einer Verteilung über drei binäre Variablen: ob Sie erkältet sind oder nicht, ob Ihr Kollege erkältet ist oder nicht und ob Ihr Mitbewohner erkältet ist oder nicht. Wie im Staffellauf können wir vereinfachende Annahmen über die Art der vorliegenden Interaktionen treffen. Unter der Annahme, dass Ihr Kollege und Ihr Mitbewohner einander nicht kennen, ist es sehr unwahrscheinlich, dass eine Krankheit, wie zum Beispiel ein Schnupfen, zwischen den beiden direkt übertragen wird. Dieses Ereignis kann als so selten betrachtet werden, dass wir berechtigerweise darauf verzichten können, es zu modellieren. Allerdings besteht durchaus die Möglichkeit, dass einer der beiden Sie ansteckt und Sie dann den anderen anstecken. Wir können die indirekte Übertragung des Schnupfens von Ihrem

Kollegen auf Ihren Mitbewohner als Übertragung eines Schnupfens von Ihrem Kollegen auf Sie und als Übertragung von Ihnen auf Ihren Mitbewohner modellieren.

In diesem Fall ist es für Sie ebenso leicht, Ihren Mitbewohner anzustecken, wie es umgekehrt für Ihren Mitbewohner leicht ist, Sie anzustecken – es gibt also keine eindeutige Richtung des Übertragungswegs. Damit ist ein ungerichtetes Modell nützlich. Wie bei gerichteten Modellen werden zwei Knoten im ungerichteten Modell über eine Kante miteinander verbunden. Die diesen Knoten entsprechenden Zufallsvariablen können dann direkt miteinander interagieren. Anders als in einem gerichteten Modell weist die Kante im ungerichteten Modell aber keine Pfeilspitze auf und ist nicht mit einer bedingten Wahrscheinlichkeitsverteilung verknüpft.

Wir verwenden als Zufallsvariable für Ihre Gesundheit h_y , für die Ihres Mitbewohners h_r und für die Ihres Kollegen h_c . Abbildung 16.3 zeigt den Graphen für dieses Szenario.

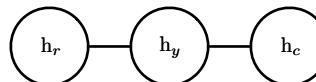


Abbildung 16.3: Ein ungerichteter Graph, der zeigt, wie die Gesundheit h_r Ihres Mitbewohners, Ihre eigene Gesundheit h_y und die Gesundheit h_c Ihres Kollegen einander beeinflussen. Sie und Ihr Mitbewohner können einander mit einem Schnupfen anstecken; dasselbe gilt für Sie und Ihren Kollegen. Aber unter der Voraussetzung, dass Ihr Mitbewohner und Ihr Kollege einander nicht kennen, ist zwischen den beiden nur der indirekte Ansteckungsweg über Sie möglich.

Formal ist ein ungerichtetes graphisches Modell ein strukturiertes probabilistisches Modell, das auf einem ungerichteten Graphen \mathcal{G} definiert ist. Für jede Clique \mathcal{C} in dem Graphen³ misst ein **Faktor** $\phi(\mathcal{C})$ (auch **Cliquen-Potenzial** genannt) die Affinität, dass die Variablen in dieser Clique sich in einem ihrer möglichen, übergreifenden Zustände befinden. Die Bedingung an die Faktoren ist, dass sie nicht-negativ sind. Gemeinsam definieren sie eine **nicht normalisierte Wahrscheinlichkeitsverteilung**

$$\tilde{p}(\mathbf{x}) = \prod_{\mathcal{C} \in \mathcal{G}} \phi(\mathcal{C}). \quad (16.3)$$

Die nicht normalisierte Wahrscheinlichkeitsverteilung ist effizient, sofern alle Cliques klein sind. Sie codiert die Idee, dass die Zustände mit höherer Affinität wahrscheinlicher sind. Anders als in einem Bayes-Netz

³ Eine Clique des Graphen ist eine Teilmenge von Knoten, die alle miteinander über eine Kante des Graphen verbunden sind.

gibt es allerdings wenig Struktur bei der Definition der Cliques, sodass es keine Garantie dafür gibt, dass ihr Produkt eine gültige Wahrscheinlichkeitsverteilung ergibt. Abbildung 16.4 zeigt ein Beispiel für das Auslesen der Faktorisierungsinformationen aus einem ungerichteten Graphen.

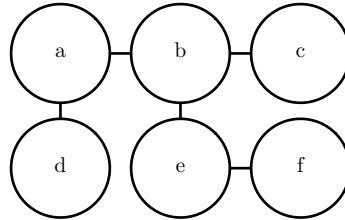


Abbildung 16.4: Dieser Graph gibt an, dass $p(a, b, c, d, e, f)$ für eine passende Wahl der ϕ -Funktionen auch $\frac{1}{Z} \phi_{a,b}(a, b)\phi_{b,c}(b, c)\phi_{a,d}(a, d)\phi_{b,e}(b, e)\phi_{e,f}(e, f)$ geschrieben werden kann.

Unser Schnupfenbeispiel mit drei Parteien (Sie, Ihr Mitbewohner und Ihr Kollege) enthält zwei Cliques. Eine Clique enthält h_y und h_c . Der Faktor für diese Clique kann mithilfe einer Tabelle definiert werden und Werte wie die folgenden annehmen:

		$h_y = 0$	$h_y = 1$
		2	1
$h_c = 0$	2	1	
	1	10	

Ein Zustand von 1 gibt an, dass Sie sich guter Gesundheit erfreuen. Ein Zustand von 0 dagegen weist auf schlechte Gesundheit hin (Sie haben einen Schnupfen). Meist sind Sie beide gesund. Deshalb weist der zugehörige Zustand die höchste Affinität auf. Der Zustand, in dem nur einer von Ihnen krank ist, hat die niedrigste Affinität, da er sehr selten vorliegt. Der Zustand, in dem beide von Ihnen krank sind (weil einer den anderen angesteckt hat), weist eine höhere Affinität auf, ist aber nicht so häufig wie der Zustand, in dem Sie beide gesund sind. Um das Modell zu vervollständigen, müssen wir einen ähnlichen Faktor für die Clique bestehend aus h_y und h_r definieren.

16.2.3 Die Partitionsfunktion

Während die nicht normalisierte Wahrscheinlichkeitsverteilung garantiert überall nicht-negativ ist, gibt es keine Garantie dafür, dass sie zu 1 summiert

oder integriert. Um eine gültige Wahrscheinlichkeitsverteilung zu erhalten, müssen wir die entsprechende normalisierte Wahrscheinlichkeitsverteilung⁴

$$p(\mathbf{x}) = \frac{1}{Z} \tilde{p}(\mathbf{x}) \quad (16.4)$$

verwenden, wobei Z der Wert ist, der beim Aufsummieren oder Integrieren der Wahrscheinlichkeitsverteilung zu 1 entsteht:

$$Z = \int \tilde{p}(\mathbf{x}) d\mathbf{x}. \quad (16.5)$$

Sie können sich Z als Konstante vorstellen, wenn die ϕ -Funktionen konstant gesetzt werden. Beachten Sie, dass für ϕ -Funktionen mit Parametern Z eine Funktion dieser Parameter ist. Es ist in der Literatur üblich, Z ohne seine Argumente zu schreiben, um Platz zu sparen. Die Normalisierungskonstante Z ist als **Partitionsfunktion** bekannt, ein Begriff aus der statistischen Physik.

Da Z ein Integral oder eine Summe über alle möglichen multivariaten Zuweisungen des Zustands \mathbf{x} ist, ist es oft nicht effizient berechenbar (engl. *intractable*). Um die normalisierte Wahrscheinlichkeitsverteilung eines ungerichteten Modells zu ermitteln, müssen die Modellstruktur und die Definitionen der ϕ -Funktionen zu einer effizienten Berechnung von Z beitragen. Im Deep-Learning-Kontext ist Z normalerweise nicht effizient berechenbar. Aus diesem Grund müssen wir auf Approximationen zurückgreifen. Solche Approximationsalgorithmen werden in Kapitel 18 behandelt.

Eine wichtige Überlegung beim Entwerfen ungerichteter Modelle ist die Möglichkeit, die Faktoren so anzugeben, dass Z nicht existiert. Dies ist der Fall, wenn einige der Variablen im Modell stetig sind und das Integral von \tilde{p} über ihrem Definitionsbereich divergiert. Angenommen, wir möchten eine einzelne skalare Variable $x \in \mathbb{R}$ mit einem einzelnen Cliques-Potenzial $\phi(x) = x^2$ modellieren. In diesem Fall gilt

$$Z = \int x^2 dx. \quad (16.6)$$

Da das Integral divergiert, gibt es keine Wahrscheinlichkeitsverteilung für diese Wahl von $\phi(x)$. Manchmal entscheidet die Auswahl eines Parameters der ϕ -Funktionen darüber, ob die Wahrscheinlichkeitsverteilung definiert ist. So bestimmt für $\phi(x; \beta) = \exp(-\beta x^2)$ der β -Parameter, ob Z existiert. Ein positives β führt zu einer Normalverteilung über x , aber alle anderen Werte für β führen dazu, dass ϕ nicht normalisiert werden kann.

⁴ Eine Verteilung, die durch Normalisieren eines Produkts der Cliques-Potenziale definiert wird, ist bekannt als **Gibbs-Verteilung**.

Ein entscheidender Unterschied zwischen der gerichteten und der ungerichteten Modellierung ist, dass gerichtete Modelle im Sinne der Wahrscheinlichkeitsverteilungen von Anfang an direkt definiert werden, wohingegen ungerichtete Modelle freier über ϕ -Funktionen definiert werden, die dann in Wahrscheinlichkeitsverteilungen umgewandelt werden. Daher müssen Sie für die Arbeit mit diesen Modellen anders vorgehen und ein anderes Gespür entwickeln. Denken Sie im Zusammenhang mit ungerichteten Modellen immer daran, dass der Definitionsbereich jeder der Variablen sich erheblich auf die Art der Wahrscheinlichkeitsverteilung auswirkt, der eine bestimmte Menge von ϕ -Funktionen entspricht. Betrachten Sie beispielsweise eine n -dimensionale vektorwertige Zufallsvariable \mathbf{x} und ein ungerichtetes Modell, das über einen Vektor aus Verzerrungen \mathbf{b} parametrisiert wird. Für jedes Element aus \mathbf{x} , $\phi^{(i)}(\mathbf{x}_i) = \exp(b_i \mathbf{x}_i)$ gebe es eine Clique. Welche Art von Wahrscheinlichkeitsverteilung ergibt sich daraus? Die Antwort ist, dass wir nicht genug Informationen haben, denn wir haben ja den Definitionsbereich von \mathbf{x} noch gar nicht festgelegt. Für $\mathbf{x} \in \mathbb{R}^n$ divergiert das Integral zur Definition von Z , sodass es keine Wahrscheinlichkeitsverteilung gibt. Für $\mathbf{x} \in \{0, 1\}^n$ wird $p(\mathbf{x})$ in n unabhängige Verteilungen faktorisiert, mit $p(\mathbf{x}_i = 1) = \text{sigmoid}(b_i)$. Ist der Definitionsbereich von \mathbf{x} die Menge der elementaren Basisvektoren ($\{[1, 0, \dots, 0], [0, 1, \dots, 0], \dots, [0, 0, \dots, 1]\}$), dann gilt $p(\mathbf{x}) = \text{softmax}(\mathbf{b})$; ein großer Wert für b_i reduziert also $p(\mathbf{x}_j = 1)$ für $j \neq i$. Häufig ist es möglich, die Auswirkung eines sorgfältig ausgewählten Definitionsbereichs einer Variable zu nutzen, um für eine recht einfache Menge mit ϕ -Funktionen ein kompliziertes Verhalten zu erhalten. Wir beschäftigen uns in Abschnitt 20.6 mit einer praktischen Anwendung dieses Konzepts.

16.2.4 Energiebasierte Modelle

Viele interessante theoretische Ergebnisse zu ungerichteten Modellen beruhen auf der Annahme, dass $\forall \mathbf{x}, \tilde{p}(\mathbf{x}) > 0$ ist. Ein praktischer Weg, diese Bedingung zu erzwingen, besteht in der Nutzung eines **energiebasierten Modells** (engl. *energy-based model*, EBM) mit

$$\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x})), \quad (16.7)$$

wobei $E(\mathbf{x})$ die **Energiefunktion** ist. Da $\exp(z)$ für alle z positiv ist, führt garantiert keine Energiefunktion zu einer Wahrscheinlichkeit von Null für einen beliebigen Zustand \mathbf{x} . Die völlige Freiheit bei der Auswahl der Energiefunktion erleichtert das Lernen. Würden wir die Cliques-Potenziale direkt erlernen, müssten wir die Optimierung unter Nebenbedingungen verwenden,

um willkürlich einen spezifischen, minimalen Wahrscheinlichkeitswert aufzuwerten. Durch Erlernen der Energiefunktion können wir die Optimierung ohne Nebenbedingungen nutzen.⁵ Die Wahrscheinlichkeiten in einem energiebasierten Modell können sich beliebig weit an Null annähern, erreichen diesen Wert jedoch nie.

Jede Verteilung der Form aus Gleichung 16.7 stellt ein Beispiel für eine **Boltzmann-Verteilung** dar. Daher werden viele energiebasierte Modelle auch **Boltzmann-Maschinen** genannt (*Fahlman et al.*, 1983; *Ackley et al.*, 1985; *Hinton et al.*, 1984; *Hinton und Sejnowski*, 1986). Es gibt keine allgemeine Richtlinie dazu, wann ein Modell als energiebasiertes Modell und wann als Boltzmann-Maschine bezeichnet wird. Der Begriff *Boltzmann-Maschine* wurde zuerst eingeführt, um ein Modell mit ausschließlich binären Variablen zu beschreiben, aber heute enthalten viele Modelle – darunter die Restricted Boltzmann Machine mit mittlerer Kovarianz (engl. *mean-covariance restricted Boltzmann machine*) – auch reellwertige Variablen. Boltzmann-Maschinen umfassen zwar laut der ursprünglichen Definition sowohl Modelle mit als auch ohne latente Variablen, aber heute bezeichnet der Begriff meist Modelle mit latenten Variablen. Boltzmann-Maschinen ohne latente Variablen werden dagegen meist als Markow-Zufallsfelder oder logarithmisch-lineare Modelle bezeichnet.

Cliquen in einem ungerichteten Graphen entsprechen den Faktoren der nicht normalisierten Wahrscheinlichkeitsfunktion. Da $\exp(a) \exp(b) = \exp(a + b)$ ist, entsprechen unterschiedliche Cliques im ungerichteten Graphen den unterschiedlichen Termen der Energiefunktion. Anders ausgedrückt: Ein energiebasiertes Modell ist eine besondere Art von Markow-Netz: Die Potenzierung setzt jeden Term der Energiefunktion mit einem Faktor für eine andere Clique gleich. Abbildung 16.5 enthält ein Beispiel dafür, wie die Form der Energiefunktion aus einer ungerichteten Graphenstruktur abgelesen wird. Man kann ein energiebasiertes Modell mit mehreren Termen in der Energiefunktion als **Product of Experts** (PoE) betrachten (*Hinton*, 1999). Jeder Term der Energiefunktion entspricht einem anderen Faktor in der Wahrscheinlichkeitsverteilung. Jeder Term der Energiefunktion stellt somit eine Art »Experten« dar, der entscheidet, ob eine bestimmte weiche Bedingung erfüllt ist. Jeder Experte kann nur eine Bedingung durchsetzen, die nur eine geringdimensionale Projektion der Zufallsvariablen betrifft. Doch bei einer Kombination mittels Multiplikation der Wahrscheinlichkeiten setzen die Experten gemeinsam eine komplizierte, hochdimensionale Bedingung durch.

⁵ Bei einigen Modellen müssen wir nach wie vor eine Optimierung unter Nebenbedingungen einsetzen, um sicherzustellen, dass Z existiert.

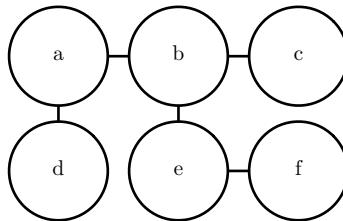


Abbildung 16.5: Dieser Graph gibt an, dass $E(a, b, c, d, e, f)$ für eine passende Wahl der Energiefunktion pro Clique auch $E_{a,b}(a, b) + E_{b,c}(b, c) + E_{a,d}(a, d) + E_{b,e}(b, e) + E_{e,f}(e, f)$ geschrieben werden kann. Beachten Sie, dass wir die ϕ -Funktionen aus Abbildung 16.4 erhalten können, indem wir jedes ϕ dem Exponentialwert der entsprechenden negativen Energie gleichsetzen, zum Beispiel $\phi_{a,b}(a, b) = \exp(-E(a, b))$.

Ein Teil der Definition eines energiebasierten Modells dient aus Sicht des Machine Learnings keinem funktionalen Zweck: das Minuszeichen – in Gleichung 16.7. Dieses –Zeichen könnte auch in die Definition von E aufgenommen werden. Für viele Wahlmöglichkeiten der Funktion E ist der Lernalgorithmus in der Wahl des Vorzeichens der Energie sowieso frei. Das Zeichen – ist in erster Linie vorhanden, um die Kompatibilität zwischen Literatur über Machine Learning und Literatur über Physik zu erhalten. Viele Errungenschaften in der probabilistischen Modellierung wurden ursprünglich von statistischen Physikern entwickelt, für die E eine tatsächliche physikalische Energie beschreibt und kein beliebiges Vorzeichen aufweist. Begriffe wie »Energie« und »Partitionsfunktion« sind weiterhin mit diesen Verfahren verknüpft, auch wenn ihre mathematische Anwendbarkeit viel breiter als im ursprünglichen Physik-Kontext ist. Einige Machine-Learning-Forscher (darunter *Smolensky* [1986], der negative Energie als **Harmonie** bezeichnete) haben die Negation gewählt – aber es ist keine allgemeine Übereinkunft.

Viele Algorithmen für probabilistische Modelle müssen $p_{\text{model}}(\mathbf{x})$ gar nicht berechnen, sondern nur $\log \tilde{p}_{\text{model}}(\mathbf{x})$. Für energiebasierte Modelle mit latenten Variablen \mathbf{h} werden diese Algorithmen manchmal hinsichtlich der Negation dieser Größe ausgedrückt, der sogenannten **freien Energie**:

$$\mathcal{F}(\mathbf{x}) = -\log \sum_{\mathbf{h}} \exp(-E(\mathbf{x}, \mathbf{h})). \quad (16.8)$$

In diesem Buch ziehen wir normalerweise die allgemeinere Formulierung $\log \tilde{p}_{\text{model}}(\mathbf{x})$ vor.

16.2.5 Separation und d-Separation

Die Kanten in einem graphischen Modell geben an, welche Variablen direkt miteinander interagieren. Häufig müssen wir auch wissen, welche Variablen *indirekt* interagieren. Einige dieser indirekten Interaktionen können durch Beobachtung anderer Variablen aktiviert oder deaktiviert werden. Formal betrachtet möchten wir wissen, welche Teilmengen der Variablen bedingt unabhängig voneinander sind, und zwar abhängig von den Werten anderer Teilmengen der Variablen.

Das Bestimmen der bedingten Unabhängigkeit in einem Graphen ist für ungerichtete Modelle einfach. In diesem Fall wird die vom Graphen implizierte bedingte Unabhängigkeit als **Separation** bezeichnet. Eine Menge von Variablen \mathbb{S} **separiert** die Menge von Variablen \mathbb{A} von einer anderen Menge von Variablen \mathbb{B} , wenn die Graphenstruktur impliziert, dass \mathbb{A} für \mathbb{S} unabhängig von \mathbb{B} ist. Wenn zwei Variablen a und b über einen Pfad miteinander verbunden sind, der nur unbeobachtete Variablen umfasst, sind diese Variablen nicht separiert. Wenn kein Pfad zwischen ihnen existiert oder alle Pfade eine beobachtete Variable enthalten, sind sie separiert. Pfade, die nur unbeobachtete Variablen umfassen, heißen »aktiv«, solche, die nur beobachtete Variablen umfassen, heißen »inaktiv«.

Wenn wir einen Graphen zeichnen, können wir beobachtete Variablen durch Einfärben markieren. Abbildung 16.6 zeigt, wie auf diese Weise aktive und inaktive Pfade in einem ungerichteten Modell dargestellt werden. Abbildung 16.7 zeigt ein Beispiel für das Auslesen der Separation aus einem ungerichteten Graphen.

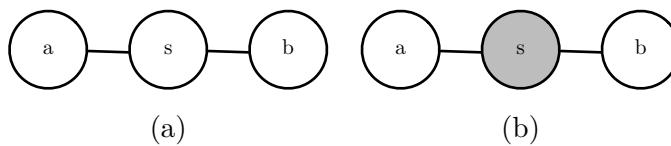


Abbildung 16.6: (a) Der Pfad zwischen der Zufallsvariable a und der Zufallsvariable b durch s ist aktiv, da s unbeobachtet ist. Daher sind a und b nicht separiert. (b) Hier ist s eingefärbt, um anzudeuten, dass es beobachtet ist. Da der einzige Pfad zwischen a und b durch s verläuft und dieser Pfad inaktiv ist, können wir folgern, dass a und b für s separiert sind.

Ähnliche Konzepte gibt es auch für gerichtete Modelle, wobei in diesem Fall von der **d-Separation** gesprochen wird. Das »d« steht für das englische Wort »dependence«, also »Abhängigkeit«. Die d-Separation für gerichtete Graphen wird auf dieselbe Weise wie die Separation für ungerichtete Graphen definiert: Eine Menge von Variablen \mathbb{S} d-separiert eine Menge von Variablen

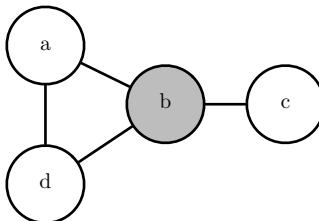


Abbildung 16.7: Ein Beispiel für das Auslesen der Separationseigenschaften aus einem ungerichteten Graphen. Hier ist b eingefärbt, um anzusehen, dass es beobachtet ist. Da die Beobachtung von b den einzigen Pfad zwischen a und c blockiert, sind a und c voneinander für b separiert. Die Beobachtung von b blockiert auch einen Pfad zwischen a und d, aber es gibt einen zweiten, aktiven Pfad zwischen diesen. Daher sind a und d für b nicht separiert.

\mathbb{A} von einer anderen Menge von Variablen \mathbb{B} , wenn die Graphenstruktur impliziert, dass \mathbb{A} für \mathbb{S} unabhängig von \mathbb{B} ist.

Wie bei ungerichteten Modellen können wir die durch den Graphen implizierten Unabhängigkeiten untersuchen, indem wir die aktiven Pfade im Graphen betrachten. Wie zuvor sind zwei Variablen abhängig, wenn es einen aktiven Pfad zwischen diesen gibt. Existiert kein solcher Pfad, sind sie d-separiert. In gerichteten Netzen ist es etwas komplizierter, zu ermitteln, ob ein Pfad aktiv ist. Abbildung 16.8 bietet einen Leitfaden zum Bestimmen aktiver Pfade in einem gerichteten Modell. Abbildung 16.9 enthält ein Beispiel zum Auslesen einiger Eigenschaften aus einem Graphen.

Es ist wichtig, daran zu denken, dass Separation und d-Separation uns nur jene bedingten Unabhängigkeiten angeben, *die vom Graphen impliziert sind*. Es ist nicht vorgeschrieben, dass der Graph alle vorhandenen Unabhängigkeiten impliziert. Insbesondere ist es immer legitim, den vollständigen Graphen (mit allen möglichen Kanten) zu verwenden, um eine Verteilung darzustellen. Tatsächlich weisen einige Verteilungen Unabhängigkeiten auf, die mit der derzeitigen graphischen Notation nicht dargestellt werden können. **Kontextspezifische Unabhängigkeiten** sind Unabhängigkeiten, die abhängig vom Wert bestimmter Variablen im Netz vorhanden sind. Als Beispiel dient ein Modell mit den drei binären Variablen a, b und c. Wenn a gleich 0 ist, seien b und c unabhängig, aber wenn a gleich 1 ist, sei b deterministisch gleich c. Um das Verhalten für a = 1 zu codieren, wird eine Kante zwischen b und c benötigt. Der Graph kann dann nicht anzeigen, dass b und c unabhängig sind, wenn a = 0 ist.

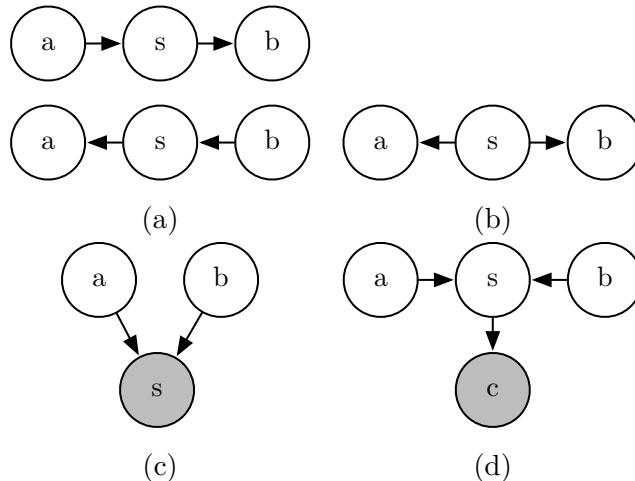


Abbildung 16.8: Alle möglichen aktiven Pfade der Länge 2, die zwischen den Zufallsvariablen a und b existieren können. (a) Jeder Pfad mit Pfeilen direkt von a zu b oder umgekehrt. Diese Art von Pfad wird blockiert, wenn s beobachtet wird. Wir haben dies bereits im Beispiel für den Staffellauf gesehen. (b) Die Variablen a und b sind über eine *gemeinsame Ursache* s miteinander verbunden. Ein Beispiel: s sei eine Variable, die angibt, ob momentan ein Orkan tobt. a und b messen die Windgeschwindigkeit an zwei verschiedenen Wetterstationen in der Nähe. Wenn wir eine hohe Windgeschwindigkeit an Station a messen, dürfen wir auch mit starkem Wind an Station b rechnen. Dieser Pfad kann durch Beobachtung von s blockiert werden. Wenn wir bereits wissen, dass ein Orkan tobt, erwarten wir eine hohe Windgeschwindigkeit an Station b, und zwar ungeachtet der Beobachtung an Station a. Wäre die Windgeschwindigkeit an Station a geringer als (für einen Orkan) erwartet, ändert sich unsere Erwartung hinsichtlich der Windgeschwindigkeit an Station b nicht (denn wir wissen ja, dass der Orkan da ist). Wird s allerdings nicht beobachtet, sind a und b abhängig, das heißt, der Pfad ist aktiv. (d) Die Variablen a und b sind Eltern von s. Dies wird als **V-Struktur** oder **Collider-Fall** bezeichnet. Die V-Struktur führt zu einer Verknüpfung von a und b mit dem **Explaining-Away-Effekt**. In diesem Fall ist der Pfad tatsächlich dann aktiv, wenn s beobachtet wird. Ein Beispiel: s ist eine Variable, die angibt, dass Ihre Kollegin nicht auf der Arbeit ist. Die Variable a gibt an, dass sie krank ist, aber b bedeutet, dass sie im Urlaub ist. Wenn Sie feststellen, dass sie nicht anwesend ist, können Sie vermuten, dass sie wahrscheinlich krank oder im Urlaub ist – aber es ist recht unwahrscheinlich, dass beides gleichzeitig zutrifft. Wenn Sie herausfinden, dass sie im Urlaub ist, reicht dieses Tatsache aus, um ihre Abwesenheit zu erklären. Sie können schließen, dass sie höchstwahrscheinlich nicht auch krank ist. (c) Der Explaining-Away-Effekt tritt sogar auf, wenn ein »Nachkomme« von s beobachtet wird! Ein Beispiel: c gibt an, ob Sie einen Bericht Ihrer Kollegin erhalten haben. Wenn Sie bemerken, dass Sie den Bericht nicht erhalten haben, verstärkt sich Ihre Vermutung, dass sie wahrscheinlich heute nicht auf der Arbeit ist. Dadurch wird es wiederum wahrscheinlicher, dass sie entweder krank oder im Urlaub ist. Die einzige Möglichkeit, einen Pfad durch eine V-Struktur zu blockieren, besteht darin, keinen der Nachkommen des gemeinsamen Kind-Elements zu beobachten.

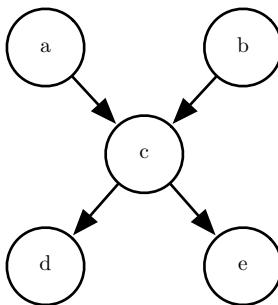


Abbildung 16.9: Aus diesem Graphen können wir mehrere Eigenschaften der d-Separation ablesen. Dazu gehören diese:

- a und b sind für die leere Menge d-separiert.
- a und e sind für c d-separiert.
- d und e sind für c d-separiert.

Wir können außerdem erkennen, dass einige Variablen nicht mehr d-separiert sind, sobald wir andere Variablen beobachten:

- a und b sind für c nicht d-separiert.
- a und b sind für d nicht d-separiert.

Generell wird ein Graph niemals eine Unabhängigkeit implizieren, wenn es keine gibt. Allerdings kann er daran scheitern, eine Unabhängigkeit zu codieren.

16.2.6 Umwandlung zwischen ungerichteten und gerichteten Graphen

Häufig bezeichnen wir ein bestimmtes Machine-Learning-Modell als ungerichtet oder gerichtet. Zum Beispiel nennen wir RBMs normalerweise ungerichtet, Sparse Coding dagegen gerichtet. Diese Wortwahl kann in die Irre führen, da kein probabilistisches Modell von Anfang an gerichtet oder ungerichtet ist. Stattdessen sind einige Modelle am einfachsten mit einem gerichteten Graphen *beschreibbar*, andere dagegen mit einem ungerichteten Graphen.

Gerichtete Modelle und ungerichtete Modelle haben jeweils eigene Vorteile und Nachteile. Kein Ansatz ist dem anderen deutlich überlegen oder wird allgemein bevorzugt. Stattdessen sollten wir uns für die geeignete Repräsentationsform für die jeweilige Aufgabe entscheiden. Diese Wahl hängt zum Teil davon ab, welche Wahrscheinlichkeitsverteilung wir beschreiben

möchten. Wir können uns entweder für die gerichtete oder die ungerichtete Modellierung entscheiden – je nachdem, welcher Ansatz die meisten Unabhängigkeit in der Wahrscheinlichkeitsverteilung erfasst oder die wenigsten Kanten zum Beschreiben der Verteilung benötigt. Außerdem können andere Faktoren die Wahl der Repräsentationsform beeinflussen. Selbst wenn wir mit nur einer Wahrscheinlichkeitsverteilung arbeiten, kann es vorkommen, dass wir zwischen den verschiedenen Modellierungssprachen wechseln. Manchmal ist eine andere Repräsentationsform geeigneter, wenn wir eine bestimmte Teilmenge von Variablen beobachten. Oder wir möchten eine andere Berechnungsaufgabe durchführen. Zum Beispiel bietet die Beschreibung als gerichtetes Modell häufig einen geradlinigen Ansatz zum effizienten Ziehen von Stichproben aus dem Modell (siehe Abschnitt 16.3), während das ungerichtete Modell für die Ableitung approximativer Inferenzverfahren nützlich ist (mehr dazu in Kapitel 19, und dort speziell Gleichung 19.56 zur Rolle ungerichteter Modelle).

Jede Wahrscheinlichkeitsverteilung kann entweder als gerichtetes oder ungerichtetes Modell dargestellt werden. Im ungünstigsten Fall lässt sich jede Verteilung immer noch als »vollständiger Graph« darstellen. Bei einem gerichteten Modell entspricht der vollständige Graph jedem gerichteten azyklischen Graphen, in dem wir die Reihenfolge der Zufallsvariablen bestimmen, und jede Variable alle anderen Variablen, die ihr in der Reihenfolge vorausgehen, als Vorgänger im Graphen hat. Für ein ungerichtetes Modell ist der vollständige Graph einfach ein Graph, der eine einzelne Clique enthält, die alle Variablen umfasst. Abbildung 16.10 zeigt ein Beispiel.

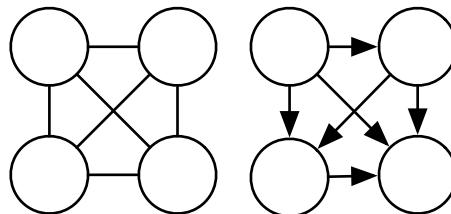


Abbildung 16.10: Beispiel für vollständige Graphen zur Beschreibung beliebiger Wahrscheinlichkeitsverteilungen. Hier zeigen wir Beispiele mit vier Zufallsvariablen. (*Links*) Der vollständige ungerichtete Graph. Im ungerichteten Fall ist der vollständige Graph eindeutig. (*Rechts*) Ein vollständiger gerichteter Graph. Im gerichteten Fall gibt es keinen eindeutigen vollständigen Graphen. Wir wählen eine Sortierung der Variablen und zeichnen einen Bogen von jeder Variable zu allen Variablen, die in der Reihenfolge später kommen. Es gibt somit eine faktorielle Anzahl vollständiger Graphen für jede Menge von Zufallsvariablen. In diesem Beispiel haben wir die Variablen von links nach rechts und oben nach unten angeordnet.

Natürlich besteht der Sinn eines graphischen Modells darin, dass der Graph impliziert, dass einige Variablen nicht direkt miteinander interagieren. Der vollständige Graph ist nicht besonders nützlich, da er keine Unabhängigkeit impliziert.

Wenn wir eine Wahrscheinlichkeitsverteilung als Graphen darstellen, möchten wir einen Graphen wählen, der möglichst viele Unabhängigkeiten impliziert, ohne dabei nicht wirklich vorhandene Unabhängigkeiten zu implizieren.

Von diesem Standpunkt aus betrachtet können einige Verteilungen effizienter mit gerichteten Modellen dargestellt werden, andere dagegen mit ungerichteten Modellen. Anders ausgedrückt: Gerichtete Modelle können einige Unabhängigkeiten codieren, die ungerichtete Modelle nicht codieren können – und umgekehrt.

Gerichtete Modelle können eine besondere Art Unterstruktur nutzen, die ungerichtete Modelle nicht perfekt darstellen können. Diese Unterstruktur wird als **Immoralität** (engl. *immorality*) bezeichnet. Die Struktur tritt auf, wenn zwei Zufallsvariablen a und b beide Eltern einer Zufallsvariable c sind und es keine Kante gibt, die a und b in einer beliebigen Richtung direkt miteinander verbindet. (Die Bezeichnung »Immoralität« mag seltsam erscheinen – sie wurde in der Fachliteratur zu graphischen Modellen als humorvolle Anspielung auf unverheiratete Eltern geprägt.) Um ein gerichtetes Modell mit dem Graphen \mathcal{D} in ein ungerichtetes Modell umzuwandeln, müssen wir einen neuen Graphen \mathcal{U} erstellen. Für jedes Variablenpaar x und y können wir eine ungerichtete Kante zwischen x und y zu \mathcal{U} hinzufügen, wenn es eine gerichtete Kante (in beliebiger Richtung) zwischen x und y in \mathcal{D} gibt oder wenn x und y in \mathcal{D} beide einer Eltern einer dritten Variable z sind. Der resultierende \mathcal{U} wird als **moralisierter Graph** bezeichnet. Abbildung 16.11 zeigt Beispiele für die Umwandlung gerichteter in ungerichtete Modelle mittels Moralisierung (engl. *moralization*).

Ebenso können ungerichtete Modelle Unterstrukturen aufweisen, die kein gerichtetes Modell perfekt darstellen kann. Insbesondere kann ein gerichteter Graph \mathcal{D} nicht alle bedingten Unabhängigkeiten erfassen, die ein ungerichteter Graph \mathcal{U} impliziert, falls \mathcal{U} eine **Schleife** mit einer Länge größer 3 enthält, es sei denn, die Schleife enthält auch ein sogenannter **Chord**. Eine Schleife ist eine Sequenz von Variablen, die über ungerichtete Kanten miteinander verbunden sind, wobei die letzte Variable der Sequenz eine abschließende Verbindung zur ersten Variable der Sequenz aufweist. Ein Chord ist eine Verbindung zwischen zwei beliebigen, nicht aufeinanderfolgenden Variablen in der Sequenz, die eine Schleife definiert. Wenn \mathcal{U}

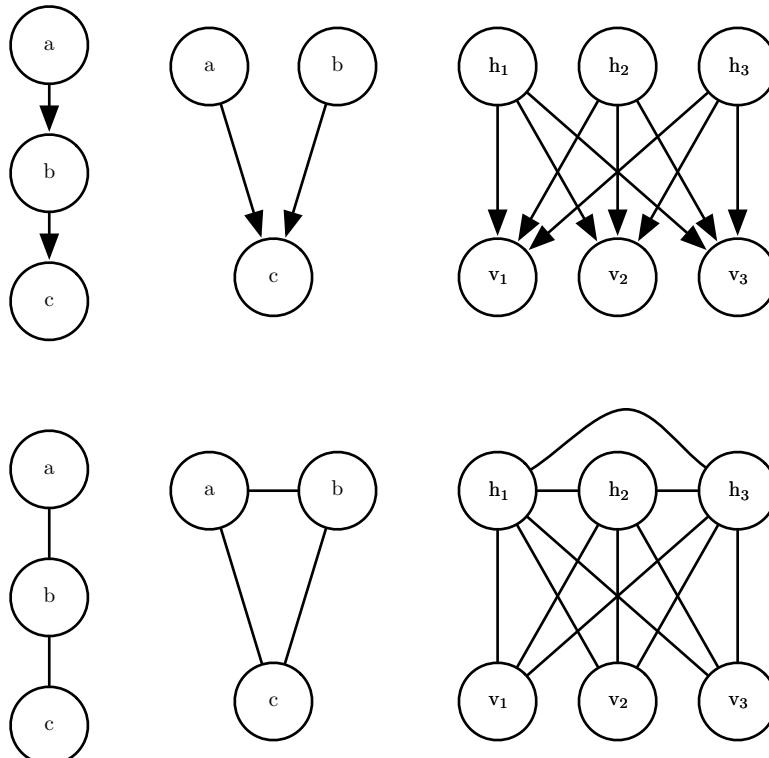


Abbildung 16.11: Beispiele für die Umwandlung gerichteter Modelle (obere Zeile) in ungerichtete Modelle (untere Zeile) durch Konstruieren moralisierter Graphen. (*Links*) Diese einfache Kette kann ganz leicht in einen moralisierten Graphen umgewandelt werden. Dazu müssen lediglich die gerichteten Kanten durch ungerichtete Kanten ersetzt werden. Das resultierende ungerichtete Modell impliziert exakt dieselbe Menge von Unabhängigkeiten und bedingten Unabhängigkeiten. (*Mitte*) Dieser Graph ist das einfachste gerichtete Modell, das sich nicht in ein ungerichtetes Modell umwandeln lässt, ohne einige Unabhängigkeiten zu verlieren. Dieser Graph besteht zur Gänze aus einer einzelnen Immoralität. Da a und b Eltern von c sind, sind sie bei Beobachtung von c durch einen aktiven Pfad verbunden. Um diese Abhängigkeit zu erfassen, muss das ungerichtete Modell eine Clique enthalten, die alle drei Variablen umfasst. Diese Clique kann $a \perp b$ nicht codieren. (*Rechts*) Generell kann die Moralisierung viele Kanten zum Graphen hinzufügen, sodass in diesem Zuge viele implizierte Unabhängigkeiten verloren gehen. Zum Beispiel müssen für diesen Sparse-Coding-Graphen moralisierende Kanten zwischen allen Paaren verdeckter Einheiten hinzugefügt werden, wodurch eine quadratische Anzahl neuer direkter Abhängigkeiten eingeführt wird.

Schleifen der Länge 4 oder größer aufweist und diese Schleifen keine Chords enthalten, müssen wir die Chords hinzufügen, bevor die Umwandlung in ein gerichtetes Modell möglich ist. Beim Hinzufügen der Chords gehen einige Informationen über Abhängigkeiten verloren, die in \mathcal{U} codiert sind. Der durch das Hinzufügen von Chords zu \mathcal{U} gebildete Graph wird als **chordaler** oder **triangulierter Graph** bezeichnet, da alle Schleifen nun als kleinere Schleifen mit drei Ecken beschrieben werden können. Um einen gerichteten Graphen \mathcal{D} aus dem chordalen Graphen zu erstellen, müssen wir den Kanten außerdem Richtungen zuweisen. Dabei dürfen wir keinen gerichteten Kreislauf in \mathcal{D} erzeugen, da das Ergebnis ansonsten kein gültiges gerichtetes probabilistisches Modell definiert. Eine Möglichkeit, die Richtungen für die Kanten in \mathcal{D} zuzuweisen, besteht darin, die Zufallsvariablen in einer bestimmten Reihenfolge anzurufen, um dann jede Kante vom in der Anordnung vorausgehenden Knoten auf den in der Anordnung folgenden Knoten zu richten. Abbildung 16.12 zeigt dies beispielhaft.

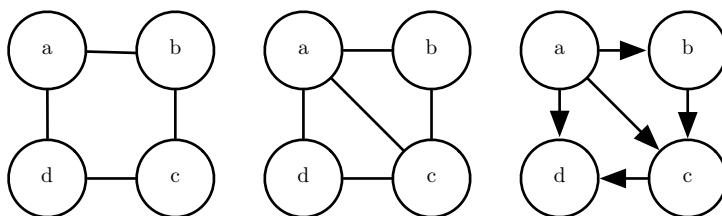


Abbildung 16.12: Umwandlung eines ungerichteten in ein gerichtetes Modell. (*Links*) Dieses ungerichtete Modell kann nicht in ein gerichtetes Modell umgewandelt werden, da es eine Schleife der Länge 4 ohne Sehnen aufweist. Das ungerichtete Modell codiert zwei unterschiedliche Abhängigkeiten, die von einem gerichteten Modell nicht gleichzeitig erfasst werden können: $a \perp c \mid \{b, d\}$ und $b \perp d \mid \{a, c\}$. (*Mitte*) Um das ungerichtete Modell in ein gerichtetes Modell umzuwandeln, müssen wir den Graphen triangulieren (in Dreiecke zerlegen), indem wir sicherstellen, dass jede Schleife mit einer Länge größer 3 eine Sehne enthält. Dazu können wir entweder eine Kante zwischen a und c oder eine Kante zwischen b und d hinzufügen. In diesem Beispiel haben wir die Kante zwischen a und c gewählt. (*Rechts*) Um die Umwandlung abzuschließen, müssen wir jeder Kante eine Richtung zuweisen. Dabei dürfen keine gerichteten Kreisläufe entstehen. Um gerichtete Kreisläufe zu vermeiden, können wir die Knoten in einer bestimmten Reihenfolge anordnen und darauf achten, dass die Kanten stets vom in der Anordnung vorausgehenden Knoten auf den darauffolgenden Knoten weisen. In diesem Beispiel haben wir die Variablennamen alphabetisch geordnet.

16.2.7 Faktorgraphen

Faktorgraphen sind eine weitere Möglichkeit zum Erstellen ungerichteter Modelle, die eine Mehrdeutigkeit in der graphischen Darstellung der üblichen, ungerichteten Modellsyntax auflösen. In einem ungerichteten Modell muss der Bereich jeder ϕ -Funktion eine *Teilmenge* einer Clique des Graphen sein. Es kommt zu Mehrdeutigkeiten, da nicht klar ist, ob jeder Clique wirklich ein entsprechender Faktor zugeordnet ist, dessen Bereich die gesamte Clique umfasst – zum Beispiel könnte eine Clique mit drei Knoten einem Faktor über alle drei Knoten oder aber drei Faktoren, die jeweils ein Knotenpaar enthalten, entsprechen. Faktorgraphen lösen diese Mehrdeutigkeit auf, indem sie den Bereich jeder ϕ -Funktion explizit darstellen. Bei Faktorgraphen handelt es sich somit um eine graphische Darstellung eines ungerichteten Modells, das aus einem zweiteiligen ungerichteten Graphen besteht. Einige der Knoten werden als Kreise gezeichnet. Diese entsprechen wie in einem gewöhnlichen ungerichteten Modell den Zufallsvariablen. Die restlichen Knoten werden als Quadrate dargestellt. Diese entsprechen den Faktoren ϕ der nicht normalisierten Wahrscheinlichkeitsverteilung. Variablen und Faktoren können über ungerichtete Kanten miteinander verbunden werden. Eine Variable und ein Faktor werden im Graphen genau dann miteinander verbunden, wenn die Variable in der nicht normalisierten Wahrscheinlichkeitsverteilung eines der Argumente für den Faktor ist. Kein Faktor darf mit einem anderen Faktor im Graphen verbunden werden, und keine Variable mit einer anderen Variable. Abbildung 16.13 enthält ein Beispiel dafür, wie Faktorgraphen die Mehrdeutigkeit bei der Interpretation ungerichteter Netze auflösen können.

16.3 Stichprobenentnahme aus graphischen Modellen

Graphische Modelle ermöglichen es auch, Stichproben aus einem Modell zu ziehen (engl. *sampling*).

Ein Vorteil gerichteter graphischer Modelle ist, dass ein einfaches und effizientes Verfahren namens **Ancestral Sampling** (auf Vorfahren/Abstammung basierendes Stichprobenverfahren) Stichproben aus der durch das Modell dargestellten multivariaten Verteilung erzeugen kann.

Die Grundidee besteht darin, die Variablen x_i im Graphen topologisch zu sortieren, sodass für alle i und j gilt, dass j größer ist als i , sofern x_i ein Elternknoten von x_j ist. Die Variablen können dann in dieser Reihenfolge gezogen werden. Mit anderen Worten: Wir ziehen zunächst $x_1 \sim P(x_1)$, dann

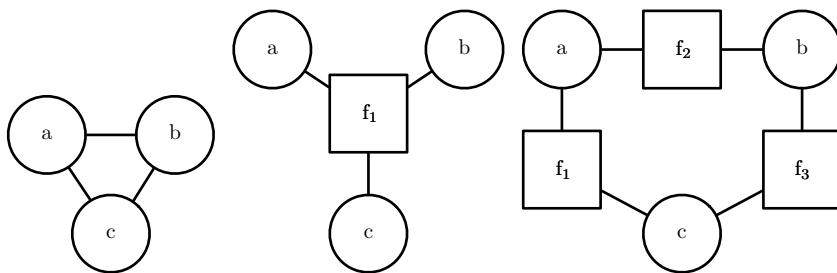


Abbildung 16.13: Beispiel für einen Faktorgraphen zum Lösen der Mehrdeutigkeit bei der Interpretation ungerichteter Netze. (*Links*) Ein ungerichtetes Netz mit einer Clique, die drei Variablen enthält: a, b und c. (*Mitte*) Ein Faktorgraph, der dem ungerichteten Modell entspricht. Dieser Faktorgraph nutzt einen Faktor für alle drei Variablen. (*Rechts*) Ein weiterer gültiger Faktorgraph für dasselbe ungerichtete Modell. Dieser Faktorgraph kennt drei Faktoren, die jeweils zwei Variablen zugeordnet sind. Repräsentation, Inferenz und Lernen sind in diesem Faktorgraphen asymptotisch günstiger als im mittleren Graphen, obwohl beide auf demselben ungerichteten Graphen für die Darstellung basieren.

$P(x_2 | Pa_{\mathcal{G}}(x_2))$ usw., bis wir abschließend $P(x_n | Pa_{\mathcal{G}}(x_n))$ ziehen. Sofern aus jeder bedingten Verteilung $p(x_i | Pa_{\mathcal{G}}(x_i))$ einfach Stichproben gezogen werden können, ist auch die Stichprobenentnahme aus dem gesamten Modell einfach. Die topologische Sortierung garantiert, dass wir die bedingten Verteilungen in Gleichung 16.1 lesen und in der Reihenfolge aus ihnen ziehen können. Ohne topologische Sortierung könnten wir versuchen, als Stichprobe eine Variable zu ziehen, bevor ihre Eltern verfügbar sind.

Für einige Graphen sind mehrere topologische Anordnungen möglich. Das Ancestral Sampling kann mit jeder dieser Anordnungen verwendet werden.

Es ist außerdem im Allgemeinen sehr schnell (sofern die Stichprobenentnahme aus den einzelnen bedingten Verteilungen einfach ist) und praktisch.

Ein Nachteil des Ancestral Samplings ist, dass es nur für gerichtete graphische Modelle nutzbar ist. Ein weiterer Nachteil ist, dass nicht alle bedingten Stichprobenentnahmen unterstützt werden. Wenn wir Stichproben aus einer Teilmenge der Variablen in einem gerichteten graphischen Modell ziehen möchten, sollen häufig alle bedingten Variablen zeitlich vor den Variablen, die gezogen werden, im geordneten Graphen liegen. In diesem Fall können wir aus den lokalen bedingten Wahrscheinlichkeitsverteilungen Stichproben ziehen, die durch die Modellverteilung angegeben werden. Ansonsten sind die bedingten Verteilungen, aus denen wir Stichproben ziehen müssen, die A-posteriori-Verteilungen für die beobachteten Variablen. Diese

A-posteriori-Verteilungen werden im Modell meist nicht explizit angegeben und parametrisiert. Das Schließen auf diese A-posteriori-Verteilungen kann aufwendig sein. In Modellen, in denen dies der Fall ist, ist Ancestral Sampling nicht länger effizient.

Leider ist das Ancestral Sampling nur für gerichtete Modelle nutzbar. Wir können Stichproben aus ungerichteten Modellen ziehen, indem wir diese zuvor in gerichtete Modelle umwandeln, aber dazu müssen häufig nicht effizient lösbare Inferenzprobleme gelöst werden (um die Randverteilung über die Wurzelknoten des neuen gerichteten Graphen zu ermitteln) oder so viele Kanten eingeführt werden, dass das resultierende gerichtete Modell nicht effizient berechenbar (engl. *intractable*) wird. Die Stichprobenentnahme aus einem ungerichteten Modell ohne vorherige Umwandlung in ein gerichtetes Modell scheint das Auflösen zyklischer Abhängigkeiten vorauszusetzen. Jede Variable interagiert mit jeder anderen Variable, sodass es keinen eindeutigen Startpunkt für die Stichprobenentnahme gibt. Leider ist das Ziehen von Stichproben aus einem ungerichteten graphischen Modell ein aufwendiger Prozess mit mehreren Durchläufen. Der konzeptuell einfachste Ansatz ist das **Gibbs-Sampling**. Gegeben sei ein graphisches Modell über einem n -dimensionalen Vektor der Zufallsvariablen \mathbf{x} . Wir suchen iterativ jede Variable x_i auf und ziehen aus $p(x_i | \mathbf{x}_{-i})$ eine Stichprobe, die von allen anderen Variablen abhängig ist. Aufgrund der Separationseigenschaften des graphischen Modells können wir das nur gleichwertig für die Nachbarn von x_i abhängig machen. Leider steht auch nach dem ersten Durchgang durch das graphische Modell und Ziehen von Stichproben aller n Variablen noch immer keine angemessene Stichprobe aus $p(\mathbf{x})$ zur Verfügung. Stattdessen müssen wir den Vorgang wiederholen und eine Stichprobewiederholung (engl. *resampling*) aller n Variablen mithilfe der aktualisierten Werte ihrer Nachbarn vornehmen. Dieser Prozess konvergiert nach vielen Wiederholungen asymptotisch gegen die Stichprobenentnahme aus der korrekten Verteilung. Es kann schwierig sein, zu bestimmen, wann die Stichproben eine ausreichend genaue Approximation der gesuchten Verteilung erreicht haben. Stichprobenverfahren für ungerichtete Modelle sind ein komplexeres Thema, auf das wir in Kapitel 17 genauer eingehen.

16.4 Vorteile der strukturierten Modellierung

Der Hauptvorteil der Anwendung strukturierter probabilistischer Modelle besteht darin, dass wir damit den Aufwand für die Repräsentation von Wahrscheinlichkeitsverteilungen, für das Lernen und für die Inferenz erheblich

senken können. Im Falle gerichteter Modelle wird auch die Stichprobenentnahme beschleunigt; bei ungerichteten Modellen kann die Situation kompliziert werden. Der wesentliche Mechanismus für die geringere Laufzeit und den geringeren Speicherbedarf all dieser Operationen ist die Entscheidung, bestimmte Interaktionen nicht zu modellieren. Graphische Modelle vermitteln Informationen durch das Weglassen von Kanten. Wo immer keine Kante vorhanden ist, drückt das Modell die Annahme aus, dass wir keine direkte Interaktion modellieren müssen.

Ein weniger quantifizierbarer Vorteil der Anwendung strukturierter probabilistischer Modelle besteht darin, dass sie uns ermöglichen, die Repräsentation von Wissen explizit vom Lernen des Wissens oder der Inferenz aus bestehendem Wissen zu separieren. Das erleichtert die Entwicklung und das Debugging unserer Modelle. Wir können Lernalgorithmen und Inferenzalgorithmen entwerfen, untersuchen und bewerten, die für viele Klassen von Graphen nutzbar sind. Unabhängig davon können wir Modelle konzipieren, die die Beziehungen erfassen, die wir in unseren Daten für wichtig halten. Anschließend können wir diese unterschiedlichen Algorithmen und Strukturen kombinieren und ein kartesisches Produkt der unterschiedlichen Möglichkeiten erhalten. Es wäre sehr viel schwieriger, Ende-zu-Ende-Algorithmen für jedes mögliche Szenario zu entwerfen.

16.5 Lernen anhand von Abhängigkeiten

Ein gutes generatives Modell muss die Verteilung über die beobachteten oder »sichtbaren« Variablen \mathbf{v} genau erfassen. Häufig sind unterschiedliche Elemente von \mathbf{v} extrem abhängig voneinander. Im Deep-Learning-Kontext werden zum Modellieren dieser Abhängigkeiten meist mehrere latente oder »verdeckte« Variablen \mathbf{h} eingeführt. Das Modell kann dann Abhängigkeiten zwischen beliebigen Variablenpaaren v_i und v_j indirekt über direkte Abhängigkeiten zwischen v_i und \mathbf{h} erfassen, ebenso direkte Abhängigkeiten zwischen \mathbf{h} und v_j .

Ein gutes Modell von \mathbf{v} , das keine latenten Variablen enthält, müsste sehr viel mehr Eltern für jeden Knoten in einem Bayes-Netz oder sehr große Cliques in einem Markow-Netz aufweisen. Bereits die Repräsentation dieser Interaktionen höherer Ordnung ist aufwendig – sowohl rechnerisch, da die Anzahl der Parameter, die im Speicher vorgehalten werden müssen, exponentiell zur Anzahl der Elemente einer Clique ansteigt, als auch statistisch, da diese exponentielle Anzahl an Parametern für eine genaue Schätzung Unmengen an Daten erfordert.

Wenn das Modell die Abhängigkeiten zwischen sichtbaren Variablen mit direkten Verbindungen erfassen soll, ist es normalerweise unmöglich, alle Variablen miteinander zu verbinden, sodass der Graph so ausgelegt werden muss, dass die eng gekoppelten Variablen verbunden sind, aber die Kanten zwischen anderen Variablen weggelassen werden. Diesem Problem widmet sich ein eigenes Feld des Machine Learnings: das **Structure Learning** (auch **Strukturlernen** genannt). Eine gute Referenz zum Structure Learning bietet (*Koller und Friedman*, 2009). Die meisten Structure-Learning-Verfahren stellen eine Art der Greedy-Suche dar. Nachdem eine Struktur vorgeschlagen und ein Modell mit dieser Struktur trainiert wurde, wird dem Modell ein Score (Bewertung oder Punktzahl) zugewiesen. Der Score belohnt eine hohe Korrektklassifikationsrate der Trainingsdatenmenge und bestraft die Modellkomplexität. Mögliche Strukturen werden dann um eine kleine Anzahl von Kanten erweitert oder gekürzt, bevor sie für die nächste Suchphase vorgeschlagen werden. Die Suche wird mit einer neuen Struktur fortgeführt, die den Score verbessern soll.

Durch Verwendung latenter Variablen anstelle einer adaptiven Struktur müssen keine diskreten Suchen durchgeführt oder mehrere Trainingsrunden durchlaufen werden. Eine unveränderliche Struktur über sichtbaren und verdeckten Variablen kann direkte Interaktionen zwischen sichtbaren und verdeckten Einheiten verwenden, um indirekte Interaktionen zwischen sichtbaren Einheiten zu erzwingen. Mithilfe einfacher Verfahren des Parameter Learnings können wir ein Modell mit einer festen Struktur erlernen, das die richtige Struktur auf der Randverteilung $p(\mathbf{v})$ vorschreibt.

Latente Variablen haben neben der effizienten Erfassung von $p(\mathbf{v})$ noch weitere Vorteile. Die neuen Variablen \mathbf{h} bieten auch eine alternative Repräsentation für \mathbf{v} . Zum Beispiel lernt das gaußsche Mischmodell (GMM), wie in Abschnitt 3.9.6 gezeigt, eine latente Variable, die der Kategorie der Beispiele entspricht, aus denen die Eingabe stammt. Das bedeutet, dass die latente Variable in einem GMM für die Klassifizierung verwendet werden kann. In Kapitel 14 haben wir gesehen, wie einfache probabilistische Modelle wie Sparse Coding latente Variablen erlernen, die als Eingabemerkmale für einen Klassifikator oder als Koordinaten entlang einer Mannigfaltigkeit dienen können. Andere Modelle können auf dieselbe Weise genutzt werden, aber tiefere Modelle und Modelle mit unterschiedlichen Arten von Interaktionen können noch umfassendere Beschreibungen der Eingabe erzeugen. Viele Ansätze erlernen Merkmale durch das Erlernen latenter Variablen. Häufig zeigen experimentelle Beobachtungen für ein Modell \mathbf{v} und \mathbf{h} , dass $\mathbb{E}[\mathbf{h} | \mathbf{v}]$ oder $\text{argmax}_{\mathbf{h}} p(\mathbf{h}, \mathbf{v})$ eine gute Merkmalszuordnung für \mathbf{v} ist.

16.6 Inferenz und approximative Inferenz

Eine der vielen Möglichkeiten zur Verwendung eines probabilistischen Modells besteht darin, Fragen über die Verbindung der Variablen untereinander zu stellen. Bei einer Menge medizinischer Tests können wir zum Beispiel fragen, an welcher Krankheit ein Patient leiden könnte. In einem Modell mit latenten Variablen könnten wir die Merkmale $\mathbb{E}[\mathbf{h} \mid \mathbf{v}]$ extrahieren, die die beobachteten Variablen \mathbf{v} beschreiben. Manchmal müssen wir solche Probleme lösen, um andere Aufgaben zu bewältigen. Häufig trainieren wir unsere Modelle mittels Maximum-Likelihood-Prinzip. Da

$$\log p(\mathbf{v}) = \mathbb{E}_{\mathbf{h} \sim p(\mathbf{h} \mid \mathbf{v})} [\log p(\mathbf{h}, \mathbf{v}) - \log p(\mathbf{h} \mid \mathbf{v})] \quad (16.9)$$

ist, möchten wir häufig $p(\mathbf{h} \mid \mathbf{v})$ berechnen, um eine Lernregel zu implementieren. All dies sind Beispiele für **Inferenzprobleme**, bei denen wir den Wert einiger Variablen oder die Wahrscheinlichkeitsverteilung über einige Variablen anhand der Werte anderer Variablen vorhersagen müssen.

Leider sind diese Inferenzprobleme für die interessantesten tiefen Modelle nicht effizient lösbar, selbst wenn wir sie mit einem strukturierten graphischen Modell vereinfachen. Die Graphenstruktur ermöglicht es uns, komplizierte hochdimensionale Verteilungen mit einer annehmbaren Anzahl von Parametern zu repräsentieren, aber die für das Deep Learning verwendeten Graphen sind meist nicht restriktiv genug, um auch eine effiziente Inferenz zu ermöglichen.

Es ist offenkundig, dass das Berechnen der Randwahrscheinlichkeit eines allgemeinen graphischen Modells #P-hart ist. Die Komplexitätsklasse #P (sprich *Sharp-P* oder *Number-P*) ist eine Generalisierung der Komplexitätsklasse NP. Für NP-Probleme muss nur entschieden werden, ob ein Problem eine Lösung hat; sofern es eine Lösung gibt, muss diese gesucht werden. Für #P-Probleme muss die Anzahl der Lösungen gezählt werden. Zum Konstruieren des graphischen Modells für den Worst Case stellen wir uns ein graphisches Modell über den binären Variablen in einem 3-SAT-Problem vor. Wir können eine Gleichverteilung über diese Variablen verwenden. Anschließend können wir für jede Klausel eine binäre latente Variable hinzufügen, die angibt, ob jede Klausel erfüllt ist. Wir können dann eine weitere latente Variable hinzufügen, die angibt, ob alle Klauseln erfüllt sind. Das lässt sich ohne Bilden einer großen Clique erreichen, indem wir einen Reduktionsbaum der latenten Variablen erstellen, wobei jeder Knoten im Baum angibt, ob zwei andere Variablen erfüllt sind. Die Blätter des Baums sind die Variablen der einzelnen Klauseln. Die Wurzel des Baums gibt an, ob das gesamte Problem gelöst ist. Aufgrund der Gleichverteilung über die Literale gibt die

Randverteilung über die Wurzel des Reduktionsbaums an, welcher Teil der Zuweisungen das Problem löst. Obwohl dies ein fiktives Worst-Case-Beispiel ist, treffen wir auch in der Praxis immer wieder auf NP-schwere Graphen.

Das führt zur Nutzung der approximierten Inferenz. Im Deep-Learning-Kontext bezeichnet dieser Begriff meist die Variational Inference, in der wir die wahre Verteilung $p(\mathbf{h} | \mathbf{v})$ approximieren, indem wir eine approximative Verteilung $q(\mathbf{h} | \mathbf{v})$ suchen, die möglichst nah an der wahren liegt. Diese und andere Verfahren werden in Kapitel 19 genauer behandelt.

16.7 Der Deep-Learning-Ansatz für strukturierte probabilistische Modelle

In der Deep-Learning-Praxis werden grundsätzlich dieselben Tools für Berechnungen verwendet wie für strukturierte probabilistische Modelle in anderen Bereichen des Machine Learnings. Im Deep-Learning-Kontext treffen wir jedoch meist andere Designentscheidungen bezüglich der Kombination dieser Tools, sodass sich die resultierenden Algorithmen und Modelle stark von den traditionellen graphischen Modellen unterscheiden.

Beim Deep Learning spielen nicht immer besonders tiefe graphische Modelle eine Rolle. Im Kontext der graphischen Modelle können wir die Tiefe eines Modells eher hinsichtlich des Graphen eines graphischen Modells als hinsichtlich eines Berechnungsgraphen definieren. Wir können uns eine latente Variable h_i als in der Tiefe j befindlich vorstellen, wenn der kürzeste Pfad von h_i zu einer beobachteten Variable j Schritte beträgt. Normalerweise geben wir als Tiefe des Modells die größte Tiefe für ein solches h_i an. Diese Art der Tiefe unterscheidet sich von der Tiefe im Berechnungsgraphen. Viele generative Modelle für das Deep Learning enthalten keine latenten Variablen oder nur eine Schicht mit latenten Variablen, nutzen aber tiefe Berechnungsgraphen, um die bedingten Verteilungen in einem Modell zu definieren.

Deep Learning nutzt praktisch immer das Konzept der verteilten Repräsentationen. Selbst flache Modelle, die für Deep-Learning-Zwecke eingesetzt werden (zum Beispiel beim Pretraining flacher Modelle, die später zu tiefen Modellen zusammengesetzt werden), weisen fast immer eine einzelne große Schicht latenter Variablen auf. Deep-Learning-Modelle enthalten üblicherweise mehr latente als beobachtete Variablen. Komplizierte nichtlineare Interaktionen zwischen Variablen werden durch indirekte Verbindungen erzielt, die durch mehrere latente Variablen verlaufen.

Im Gegensatz dazu enthalten traditionelle graphische Modelle meist größtenteils Variablen, die zumindest gelegentlich beobachtet werden – selbst wenn viele der Variablen zufällig in einigen Trainingsbeispielen fehlen. Klassische Modelle nutzen meist Terme höherer Ordnung und Structure Learning zum Erfassen komplexer nichtlinearer Interaktionen zwischen Variablen. Wenn es latente Variablen gibt, ist ihre Anzahl meist geringer.

Auch die Art, wie latente Variablen entwickelt werden, ist im Deep Learning anders. Diejenigen, die Deep Learning in der Praxis einsetzen, legen normalerweise keine spezifische Semantik für die latenten Variablen im Voraus fest – der Trainingsalgorithmus kann die Konzepte, die zum Modellieren eines bestimmten Datensatzes benötigt werden, selbst erstellen. Die latenten Variablen sind im Nachhinein für Menschen kaum zu interpretieren, obschon Visualisierungsverfahren eine gewisse Einordnung erlauben können. Wenn latente Variablen im Rahmen traditioneller graphischer Modelle eingesetzt werden, steckt oft eine spezifische Semantik dahinter – das Thema eines Dokuments, die Intelligenz einer Studentin, die Krankheit, die die Symptome eines Patienten verursacht, usw. Diese Modelle sind oft sehr viel leichter von Menschen interpretierbar und enthalten häufig mehr theoretische Garantien, können aber weniger leicht für komplexe Probleme skaliert und nicht in so vielen verschiedenen Zusammenhängen wie tiefe Modelle wiederverwendet werden.

Ein weiterer offensichtlicher Unterschied ist die Art der im Deep-Learning-Ansatz üblicherweise verwendeten Konnektivität. Tiefe graphische Modelle weisen meist große Gruppen von Einheiten auf, die alle mit anderen Gruppen von Einheiten verbunden sind, sodass die Interaktionen zwischen zwei Gruppen mit nur einer Matrix beschrieben werden können. Traditionelle graphische Modelle weisen sehr wenige Verbindungen auf und die Wahl der Verbindungen für jede Variable kann individuell angepasst sein. Das Design der Modellstruktur ist eng mit der Wahl des Inferenzalgorithmus verwoben. Klassische Ansätze für graphische Modelle zielen meist darauf ab, die effiziente Lösbarkeit der exakten Inferenz beizubehalten. Wenn diese Bedingung zu stark einschränkt, wird ein beliebter Algorithmus für approximative Inferenz als **Loopy Belief Propagation** bezeichnet. Beide Ansätze funktionieren oft gut mit geringfügig miteinander verbundenen Graphen. Im Vergleich neigen im Deep Learning eingesetzte Modelle dazu, jede sichtbare Einheit v_i mit vielen verdeckten Einheiten h_j zu verbinden, sodass \mathbf{h} eine verteilte Repräsentation von v_i bereitstellen kann (und möglicherweise noch weitere beobachtete Variablen). Verteilte Repräsentationen bieten viele Vorteile, haben aber unter den Gesichtspunkten der graphischen Modelle und rechnerischen Komplexität auch einen Nachteil: Die resultierenden Graphen

sind nur selten hinreichend geringfügig miteinander verbunden, damit die traditionellen Techniken der exakten Inferenz und die Loopy Belief Propagation relevant sind. Aus diesem Grund ist einer der auffälligsten Unterschiede zwischen der größeren Gemeinschaft, die graphische Modelle verwendet, und der Gemeinschaft, die tiefe graphische Modelle verwendet, dass die Loopy Belief Propagation für das Deep Learning nahezu nie eingesetzt wird. Die meisten tiefen Modelle werden stattdessen für ein effizientes Gibbs-Sampling oder Algorithmen für Variational Inference entwickelt. Eine weitere Überlegung ist, dass Deep-Learning-Modelle sehr viele latente Variablen enthalten, sodass ein effizienter numerischer Code unabdingbar ist. Dies bietet – neben der Wahl eines hochrangigen Inferenzalgorithmus – einen zusätzlichen Anreiz für die Gruppierung der Einheiten in Schichten mit einer Matrix, die die Interaktion zwischen zwei Schichten beschreibt. Damit können einzelne Schritte des Algorithmus mit effizienten Matrixprodukt-Operationen oder mit geringfügig miteinander verbundenen Generalisierungen wie Produkten von Blockdiagonalmatrizen oder Faltungen implementiert werden.

Schließlich zeichnet sich der Deep-Learning-Ansatz für die graphische Modellierung durch eine markante Toleranz des Unbekannten aus. Statt das Modell so weit zu vereinfachen, bis alle eventuell gesuchten Größen exakt berechnet werden können, erhöhen wir einfach die Leistung des Modells, bis es gerade noch trainiert oder verwendet werden kann. Wir verwenden häufig Modelle, deren Randverteilung nicht berechnet werden kann und denen man einfach durch Ziehen approximativer Stichproben aus diesen Modellen gerecht wird. Wir trainieren häufig Modelle mit einer nicht effizient berechenbaren Zielfunktion, die wir nicht einmal in einer angemessenen Zeit approximieren können, aber wir sind dennoch in der Lage, das Modell näherungsweise zu trainieren, wenn wir auf effiziente Weise eine Gradientenschätzung für eine solche Funktion bestimmen können. Der Deep-Learning-Ansatz besteht häufig darin herauszufinden, welche Menge an Informationen wir unbedingt benötigen; anschließend wird ermittelt, wie sich so schnell wie möglich eine hinreichende Approximation dieser Informationen bestimmen lässt.

16.7.1 Beispiel: Die Restricted Boltzmann Machine (RBM)

Die **Restricted Boltzmann Machine** (RBM, dt. *eingeschränkte Boltzmann-Maschine*) (Smolensky, 1986), auch **Harmonium**, ist das Musterbeispiel für den Einsatz graphischer Modelle für das Deep Learning. Bei der RBM handelt es sich nicht um ein tiefes Modell. Stattdessen weist sie eine

Schicht mit latenten Variablen auf, die zum Erlernen einer Repräsentation für die Eingabe genutzt werden können. In Kapitel 20 zeigen wir, wie RBMs zum Erstellen vieler tiefer Modelle verwendet werden können. Hier geht es darum, wie die RBM viele der Methoden beispielhaft umsetzt, die in vielen tiefen graphischen Modellen zum Einsatz kommen: Ihre Einheiten sind in große Gruppen, die Schichten, eingeteilt. Die Konnektivität zwischen den Schichten wird mithilfe einer Matrix beschrieben und ist relativ dicht. Das Modell wurde für ein effizientes Gibbs-Sampling konzipiert und die Betonung liegt beim Modelldesign darauf, dem Trainingsalgorithmus die Freiheit zu geben, latente Variablen zu erlernen, deren Semantik vom Designer nicht festgelegt wurde. In Abschnitt 20.2 widmen wir uns der RBM noch detaillierter.

Die kanonische RBM ist ein energiebasiertes Modell mit binären sichtbaren und verdeckten Einheiten. Ihre Energiefunktion lautet

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h} - \mathbf{v}^\top \mathbf{W} \mathbf{h}, \quad (16.10)$$

wobei \mathbf{b} , \mathbf{c} und \mathbf{W} reellwertige erlernbare Parameter ohne Nebenbedingungen sind. Wir sehen, dass das Modell in zwei Einheitengruppen unterteilt ist: \mathbf{v} und \mathbf{h} . Die Interaktion zwischen diesen wird über eine Matrix \mathbf{W} beschrieben. Das Modell ist in Abbildung 16.14 zu sehen. Wie diese Abbildung zeigt, besteht ein wichtiger Aspekt des Modells darin, dass es keine direkten Interaktionen zwischen zwei beliebigen sichtbaren Einheiten oder zwischen zwei beliebigen verdeckten Einheiten gibt (daher nennt man sie auch »restricted«, also »eingeschränkt« – eine allgemeine Boltzmann-Maschine darf über beliebige Verbindungen verfügen).

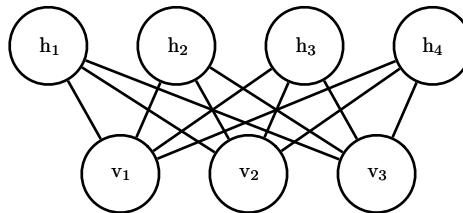


Abbildung 16.14: Eine RBM, gezeichnet als Markow-Netz

Die Einschränkungen der RBM-Struktur führen zu den guten Eigenschaften

$$p(\mathbf{h} \mid \mathbf{v}) = \prod_i p(h_i \mid \mathbf{v}) \quad (16.11)$$

und

$$p(\mathbf{v} \mid \mathbf{h}) = \prod_i p(v_i \mid \mathbf{h}). \quad (16.12)$$

Die einzelnen bedingten Verteilungen sind ebenfalls einfach zu berechnen. Für die binäre RBM erhalten wir

$$P(h_i = 1 | \mathbf{v}) = \sigma(\mathbf{v}^\top \mathbf{W}_{:,i} + b_i), \quad (16.13)$$

$$P(h_i = 0 | \mathbf{v}) = 1 - \sigma(\mathbf{v}^\top \mathbf{W}_{:,i} + b_i). \quad (16.14)$$

Zusammen ermöglichen diese Eigenschaften ein effizientes **Block-Gibbs-Sampling**, das zwischen dem gleichzeitigen Ziehen von Stichproben aller \mathbf{h} und dem gleichzeitigen Ziehen von Stichproben aller \mathbf{v} wechselt. Die durch das Gibbs-Sampling aus einem RBM generierten Stichproben sind in Abbildung 16.15 abgebildet.



Abbildung 16.15: Stichproben einer trainierten RBM und ihre Gewichten. (*Links*) Stichproben aus einem mit MNIST trainierten Modell, gezogen mittels Gibbs-Sampling. Jede Spalte ist ein separater Gibbs-Sampling-Prozess. Jede Zeile steht für die Ausgabe weiterer 1.000 Schritte des Gibbs-Samplings. Aufeinanderfolgende Stichproben korrelieren stark miteinander. (*Rechts*) Die entsprechenden Gewichtungsvektoren. Vergleichen Sie die Darstellung mit den Stichproben und Gewichten eines linearen Faktorenmodells aus Abbildung 13.2. Die Stichproben hier sind viel besser, da die A-priori-Wahrscheinlichkeit $p(\mathbf{h})$ der RBM, nicht darauf eingeschränkt ist, faktoriell zu sein. Die RBM kann lernen, welche Merkmale bei der Stichprobenentnahme gemeinsam erscheinen sollten. Andererseits ist die A-posteriori-Wahrscheinlichkeit $p(\mathbf{h} | \mathbf{v})$ der RBM faktoriell, die A-posteriori-Wahrscheinlichkeit $p(\mathbf{h} | \mathbf{v})$ von Sparse Coding aber nicht, sodass sich das Modell mit Sparse Coding möglicherweise besser für die Merkmalsextraktion eignet. Andere Modelle können sowohl nichtfaktorielle $p(\mathbf{h})$ als auch nichtfaktorielle $p(\mathbf{h} | \mathbf{v})$ aufweisen. (Abbildungen mit freundlicher Genehmigung von LISA (2008).)

Da die Energiefunktion selbst lediglich eine lineare Funktion der Parameter ist, sind die Ableitungen schnell erledigt. Zum Beispiel:

$$\frac{\partial}{\partial W_{i,j}} E(\mathbf{v}, \mathbf{h}) = -v_i h_j. \quad (16.15)$$

Diese beiden Eigenschaften – effizientes Gibbs-Sampling und effiziente Ableitungen – erleichtern das Training. In Kapitel 18 zeigen wir, dass unge-

richtete Modelle durch das Berechnen solcher Ableitungen für Stichproben aus dem Modell trainiert werden können.

Das Trainieren des Modells führt zu einer Repräsentation \mathbf{h} der Daten \mathbf{v} . Wir können $\mathbb{E}_{\mathbf{h} \sim p(\mathbf{h}|\mathbf{v})}[\mathbf{h}]$ häufig als eine Menge der Merkmale zur Beschreibung von \mathbf{v} verwenden.

Insgesamt verdeutlicht die RBM den typischen Deep-Learning-Ansatz für graphische Modelle: Representation Learning mithilfe von Schichten mit latenten Variablen, kombiniert mit effizienten Interaktionen zwischen Schichten, die mittels Matrizen parametrisiert werden.

Graphische Modelle sind eine elegante, flexible und klare Repräsentationsform zur Abbildung probabilistischer Modelle. In den folgenden Kapiteln nutzen wir diese Repräsentationsform unter anderem, um eine Vielzahl tiefer probabilistischer Modelle zu beschreiben.

17

Monte-Carlo-Verfahren

Randomisierte Algorithmen lassen sich grob in zwei Kategorien unterteilen: Las-Vegas-Algorithmen und Monte-Carlo-Algorithmen. Las-Vegas-Algorithmen geben immer genau die korrekte Antwort zurück (oder geben einen Fehler zurück). Diese Algorithmen benötigen eine zufällige Menge von Ressourcen, meist Speicherplatz oder Zeit. Im Gegensatz dazu geben Monte-Carlo-Algorithmen Antworten mit einer zufälligen Höhe des Fehlers zurück. Die Höhe des Fehlers lässt sich meist durch Aufwenden zusätzlicher Ressourcen (meist Laufzeit und Speicher) reduzieren. Für ein festes rechentechnisches Budget kann ein Monte-Carlo-Algorithmus eine approximative Antwort liefern.

Viele Machine-Learning-Probleme sind so schwierig, dass wir gar keine genaue Antwort erwarten dürfen – mit Ausnahme von deterministischen Algorithmen und Las-Vegas-Algorithmen. Stattdessen müssen wir deterministische Approximationsalgorithmen oder Monte-Carlo-Approximationen verwenden. Beide Ansätze werden im Machine Learning ständig eingesetzt. In diesem Kapitel befassen wir uns hauptsächlich mit Monte-Carlo-Verfahren.

17.1 Stichprobenentnahme und Monte-Carlo-Verfahren

Viele wichtige Technologien zum Erreichen von Machine-Learning-Zielen beruhen auf dem Ziehen von Stichproben (Sampling) aus einer Wahrscheinlichkeitsverteilung, mit denen dann ein Monte-Carlo-Schätzwert einer gewünschten Größe gebildet wird.

17.1.1 Gründe für das Stichprobenverfahren

Es kann verschiedene Gründe haben, Stichproben aus einer Wahrscheinlichkeitsverteilung ziehen zu wollen. Das Stichprobenverfahren bietet eine flexible Möglichkeit, viele Summen und Integrale mit geringem Aufwand zu approximieren. Manchmal verwenden wir es, um eine beträchtliche Beschleunigung für eine aufwendige, aber effizient berechenbare (engl. *tractable*) Summe zu erzielen, zum Beispiel indem wir den gesamten Trainingsaufwand anhand von Mini-Batches in Teilstichproben unterteilen. In anderen Fällen müssen wir für den Lernalgorithmus eine nicht effizient berechenbare Summe oder ein ebensolches Integral approximieren, zum Beispiel den Gradienten der Log-Partitionsfunktion eines ungerichteten Modells. In vielen weiteren Fällen ist das Ziehen von Stichproben unser eigentliches Ziel – in dem Sinne, dass wir ein Modell trainieren möchten, das Stichproben aus einer Trainingsverteilung ziehen kann.

17.1.2 Grundlagen des Monte-Carlo-Stichprobenverfahrens

Wenn eine Summe oder ein Integral nicht exakt berechnet werden kann (weil die Summe zum Beispiel eine exponentielle Anzahl von Termen aufweist und keine exakte Vereinfachung bekannt ist), ist es häufig möglich, sie mithilfe des Monte-Carlo-Stichprobenverfahrens zu approximieren. Dabei wird die Summe oder das Integral so betrachtet, als wäre sie bzw. es ein Erwartungswert unter einer Verteilung; *dieser Erwartungswert wird dann durch einen entsprechenden Durchschnittswert approximiert*.

Sei

$$s = \sum_{\mathbf{x}} p(\mathbf{x}) f(\mathbf{x}) = E_p[f(\mathbf{x})] \quad (17.1)$$

oder

$$s = \int p(\mathbf{x}) f(\mathbf{x}) d\mathbf{x} = E_p[f(\mathbf{x})] \quad (17.2)$$

die zu schätzende Summe oder das zu schätzende Integral, notiert als Erwartungswert mit der Bedingung, dass p eine Wahrscheinlichkeitsverteilung (für die Summe) oder eine Wahrscheinlichkeitsdichte (für das Integral) über die Zufallsvariable \mathbf{x} darstellt.

Wir können s approximieren, indem wir n Stichproben $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}$ aus p ziehen und anschließend das empirische Mittel bilden:

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^n f(\mathbf{x}^{(i)}). \quad (17.3)$$

Diese Approximation ist durch einige unterschiedliche Eigenschaften begründet. Die erste triviale Beobachtung ist, dass der Schätzer \hat{s} erwartungstreu ist, da

$$\mathbb{E}[\hat{s}_n] = \frac{1}{n} \sum_{i=1}^n \mathbb{E}[f(\mathbf{x}^{(i)})] = \frac{1}{n} \sum_{i=1}^n s = s \quad (17.4)$$

ist. Aber zusätzlich besagt das **Gesetz der großen Zahlen**, dass bei u.i.v. Stichproben $\mathbf{x}^{(i)}$ der Durchschnittswert fast sicher gegen den Erwartungswert konvergiert:

$$\lim_{n \rightarrow \infty} \hat{s}_n = s, \quad (17.5)$$

sofern die Varianz der einzelnen Terme, $\text{Var}[f(\mathbf{x}^{(i)})]$, eingeschränkt ist. Um dies deutlicher zu zeigen, betrachten wir die Varianz von \hat{s}_n für zunehmendes n . Die Varianz $\text{Var}[\hat{s}_n]$ nimmt ab und konvergiert gegen 0, sofern $\text{Var}[f(\mathbf{x}^{(i)})] < \infty$ ist:

$$\text{Var}[\hat{s}_n] = \frac{1}{n^2} \sum_{i=1}^n \text{Var}[f(\mathbf{x})] \quad (17.6)$$

$$= \frac{\text{Var}[f(\mathbf{x})]}{n}. \quad (17.7)$$

Das Ergebnis zeigt uns, wie wir die Unsicherheit in einem Monte-Carlo-Durchschnittswert bzw. äquivalent den Betrag des erwarteten Fehlers der Monte-Carlo-Approximation schätzen können. Wir berechnen den empirischen Durchschnittswert der $f(\mathbf{x}^{(i)})$ und deren empirische Varianz,¹ und teilen anschließend die geschätzte Varianz durch die Anzahl der Stichproben n , um einen Schätzer für $\text{Var}[\hat{s}_n]$ zu erhalten. Der **zentrale Grenzwertsatz** gibt an, dass die Verteilung des Durchschnittswerts, \hat{s}_n , gegen eine Normalverteilung mit dem Mittel s und der Varianz $\frac{\text{Var}[f(\mathbf{x})]}{n}$ konvergiert. Damit können wir Konfidenzintervalle um den Schätzwert \hat{s}_n schätzen, indem wir die kumulative Verteilung der normalen Dichte heranziehen.

All dies hängt von unserer Fähigkeit ab, problemlos Stichproben aus der Basisverteilung $p(\mathbf{x})$ zu ziehen, aber das ist nicht immer möglich. Wenn aus p keine Stichproben gezogen werden können, können wir auch auf das Importance Sampling (Stichprobenentnahme nach Wichtigkeit) zurückgreifen, das in Abschnitt 17.2 vorgestellt wird. Ein allgemeinerer Ansatz besteht darin, eine Sequenz von Schätzern zu bilden, die gegen die gesuchte Verteilung konvergieren. Dies ist der Ansatz der Monte-Carlo-Markow-Ketten (Abschnitt 17.3).

¹ Der erwartungstreue Schätzer der Varianz wird oft bevorzugt; hier wird die Summe des Quadrats der Differenzen durch $n - 1$ statt durch n geteilt.

17.2 Importance Sampling

Ein wichtiger Schritt bei der Zerlegung des Integranden (oder Summanden) im Monte-Carlo-Verfahren aus Gleichung 17.2 ist die Entscheidung, welcher Teil des Integranden die Rolle der Wahrscheinlichkeitsverteilung $p(\mathbf{x})$ und welcher Teil die Rolle der Größe $f(\mathbf{x})$, deren Erwartungswert (unter der Wahrscheinlichkeitsverteilung) geschätzt werden soll, spielen soll. Es gibt keine eindeutige Zerlegung, da $p(\mathbf{x})f(\mathbf{x})$ stets geschrieben werden kann als

$$p(\mathbf{x})f(\mathbf{x}) = q(\mathbf{x}) \frac{p(\mathbf{x})f(\mathbf{x})}{q(\mathbf{x})}, \quad (17.8)$$

wobei wir nun Stichproben aus q ziehen und $\frac{pf}{q}$ mitteln. In vielen Fällen möchten wir einen Erwartungswert für ein bestimmtes p und ein f berechnen; die Tatsache, dass das Problem von Anfang an als Erwartungswert spezifiziert wird, macht p und f zur natürlichen Wahl für die Zerlegung. Allerdings stellt die ursprüngliche Spezifikation des Problems vielleicht gar nicht die optimale Wahl hinsichtlich der Anzahl der Stichproben dar, die für ein bestimmtes Genauigkeitsniveau benötigt werden. Zum Glück lässt sich die Form der optimalen Wahl q^* leicht ableiten. Das optimale q^* entspricht dem sogenannten optimalen Importance Sampling (Stichprobenentnahme nach Wichtigkeit).

Aufgrund der Identität aus Gleichung 17.8 kann jeder Monte-Carlo-Schätzer

$$\hat{s}_p = \frac{1}{n} \sum_{i=1, \mathbf{x}^{(i)} \sim p}^n f(\mathbf{x}^{(i)}) \quad (17.9)$$

in einen Importance-Sampling-Schätzer transformiert werden:

$$\hat{s}_q = \frac{1}{n} \sum_{i=1, \mathbf{x}^{(i)} \sim q}^n \frac{p(\mathbf{x}^{(i)})f(\mathbf{x}^{(i)})}{q(\mathbf{x}^{(i)})}. \quad (17.10)$$

Wir sehen gleich, dass der Erwartungswert des Schätzers nicht von q abhängt:

$$\mathbb{E}_q[\hat{s}_q] = \mathbb{E}_q[\hat{s}_p] = s. \quad (17.11)$$

Die Varianz eines Importance-Sampling-Schätzers kann jedoch besonders empfindlich auf die Wahl von q reagieren. Die Varianz ergibt sich aus

$$\text{Var}[\hat{s}_q] = \text{Var}\left[\frac{p(\mathbf{x})f(\mathbf{x})}{q(\mathbf{x})}\right]/n. \quad (17.12)$$

Die minimale Varianz tritt auf, wenn q ist:

$$q^*(\mathbf{x}) = \frac{p(\mathbf{x})|f(\mathbf{x})|}{Z}, \quad (17.13)$$

wobei Z die Normalisierungskonstante ist, die so gewählt wird, dass $q^*(\mathbf{x})$ entsprechend aufsummiert oder integriert 1 ergibt. Bessere Importance-Sampling-Verteilungen weisen an Stellen mit einem größeren Integranden ein höheres Gewicht auf. Tatsächlich gilt für $f(\mathbf{x})$ ohne Vorzeichenwechsel $\text{Var}[\hat{s}_{q^*}] = 0$, sodass *eine einzige Stichprobe ausreicht*, wenn die optimale Verteilung genutzt wird. Natürlich ist der Grund dafür, dass die Berechnung von q^* praktisch das ursprüngliche Problem löst, sodass es meist nicht praktikabel ist, diesen Ansatz zum Ziehen einer einzelnen Stichprobe aus der optimalen Verteilung zu nutzen.

Jede Wahl der Stichprobenverteilung q ist gültig (hinsichtlich des korrekten Erwartungswerts), und q^* ist die optimale Wahl (hinsichtlich der minimalen Varianz). Eine Stichprobenentnahme aus q^* ist normalerweise nicht durchführbar, aber andere Optionen für q können sich als durchführbar erweisen und dennoch zu einer gewissen Reduzierung der Varianz führen.

Ein weiterer Ansatz ist das **Biased Importance Sampling** (verzerrte Stichprobenentnahme nach Wichtigkeit), für das keine Normalisierung von p oder q benötigt wird. Im Falle von diskreten Variablen ergibt sich der Biased-Importance-Sampling-Schätzer aus

$$\hat{s}_{BIS} = \frac{\sum_{i=1}^n \frac{p(\mathbf{x}^{(i)})}{q(\mathbf{x}^{(i)})} f(\mathbf{x}^{(i)})}{\sum_{i=1}^n \frac{p(\mathbf{x}^{(i)})}{q(\mathbf{x}^{(i)})}} \quad (17.14)$$

$$= \frac{\sum_{i=1}^n \frac{p(\mathbf{x}^{(i)})}{\tilde{q}(\mathbf{x}^{(i)})} f(\mathbf{x}^{(i)})}{\sum_{i=1}^n \frac{p(\mathbf{x}^{(i)})}{\tilde{q}(\mathbf{x}^{(i)})}} \quad (17.15)$$

$$= \frac{\sum_{i=1}^n \frac{\tilde{p}(\mathbf{x}^{(i)})}{\tilde{q}(\mathbf{x}^{(i)})} f(\mathbf{x}^{(i)})}{\sum_{i=1}^n \frac{\tilde{p}(\mathbf{x}^{(i)})}{\tilde{q}(\mathbf{x}^{(i)})}}, \quad (17.16)$$

wobei \tilde{p} und \tilde{q} die nicht normalisierten Formen von p sowie q sind und $\mathbf{x}^{(i)}$ Stichproben aus q sind. Dieser Schätzer ist verzerrt, da $\mathbb{E}[\hat{s}_{BIS}] \neq s$ ist, ausgenommen asymptotisch, wenn $n \rightarrow \infty$ und der Nenner von Gleichung 17.14 gegen 1 konvergiert. Daher wird dieser Schätzer als asymptotisch erwartungstreu bezeichnet.

Zwar kann eine gute Wahl von q die Effizienz der Monte-Carlo-Schätzung erheblich steigern, aber eine schlechte Wahl von q führt zu einer sehr viel

schlechteren Effizienz. In Gleichung 17.12 erkennen wir, dass für Stichproben aus q mit großem $\frac{p(\mathbf{x})|f(\mathbf{x})|}{q(\mathbf{x})}$ die Varianz des Schätzers sehr groß werden kann. Das kann passieren, wenn $q(\mathbf{x})$ winzig ist und weder $p(\mathbf{x})$ noch $f(\mathbf{x})$ klein genug sind, um das aufzuheben. Die q -Verteilung wird normalerweise als einfache Verteilung gewählt, sodass leicht Stichproben daraus gezogen werden können. Wenn \mathbf{x} hochdimensional ist, führt diese Einfachheit in q dazu, dass es schlecht zu p oder $p|f|$ passt. Für $q(\mathbf{x}^{(i)}) \gg p(\mathbf{x}^{(i)})|f(\mathbf{x}^{(i)})|$ sammelt das Importance Sampling unbrauchbare Stichproben (Aufsummieren winziger Zahlen oder Nullen). Andererseits kann das Verhältnis für $q(\mathbf{x}^{(i)}) \ll p(\mathbf{x}^{(i)})|f(\mathbf{x}^{(i)})|$ – was seltener geschieht – riesig sein. Da die letztgenannten Ereignisse sehr selten sind, kommen sie in einer typischen Stichprobe eventuell nicht vor, was zur typischen Unterschätzung von s führt, die nur selten durch eine extreme Überschätzung kompensiert wird. Solche sehr großen oder sehr kleinen Zahlen sind typisch, wenn \mathbf{x} hochdimensional ist, da der dynamische Bereich der multivariaten Wahrscheinlichkeiten in der hohen Dimension sehr groß sein kann.

Trotz dieses Risikos haben sich das Importance Sampling und seine Varianten in vielen Machine-Learning- sowie Deep-Learning-Algorithmen als sehr nützlich erwiesen. Betrachten Sie zum Beispiel die Funktion des Importance Samplings zum Beschleunigen des Trainings in neuronalen Sprachmodellen mit einem großen Wortschatz (Abschnitt 12.4.3.3) oder in anderen neuronalen Netzen mit einer Vielzahl von Ausgaben. Beachten Sie auch, wie das Importance Sampling zum Schätzen einer Partitionsfunktion (die Normalisierungskonstante einer Wahrscheinlichkeitsverteilung) in Abschnitt 18.7 verwendet wurde oder zum Schätzen der Log-Likelihood in tiefen gerichteten Modellen, beispielsweise dem VAE (Variational Autoencoder) in Abschnitt 20.10.3. Das Importance Sampling kann auch zum Verbessern der Gradientenschätzung für die Kostenfunktion beim Trainieren von Modellparametern im stochastischen Gradientenabstiegsverfahren genutzt werden, insbesondere für Modelle wie Klassifikatoren, in denen eine kleine Anzahl falsch klassifizierter Beispiele den größten Teil des Funktionswerts der Kostenfunktion ausmacht. Häufigeres Ziehen von schwierigeren Beispielen kann die Varianz des Gradienten in diesen Fällen reduzieren (*Hinton, 2006*).

17.3 Markow-Ketten-Monte-Carlo-Verfahren

In vielen Fällen soll ein Monte-Carlo-Verfahren verwendet werden, aber es gibt keine effizient durchführbare Methode zum Ziehen exakter Stichproben aus der Verteilung $p_{\text{model}}(\mathbf{x})$ oder einer guten (niedrige Varianz) Importance-

Sampling-Verteilung $q(\mathbf{x})$. Im Deep-Learning-Kontext geschieht dies am häufigsten, wenn $p_{\text{model}}(\mathbf{x})$ durch ein ungerichtetes Modell repräsentiert wird. In diesen Fällen nutzen wir ein mathematisches Werkzeug namens **Markow-Kette**, um eine approximative Stichprobe aus $p_{\text{model}}(\mathbf{x})$ zu entnehmen. Die Familie der Algorithmen, die Markow-Ketten zum Durchführen von Monte-Carlo-Schätzungen nutzen, wird **Markow-Ketten-Monte-Carlo-Verfahren** (engl. *Markov Chain Monte Carlo methods*, MCMC) genannt. Markow-Ketten-Monte-Carlo-Verfahren für das Machine Learning werden genauer in *Koller und Friedman* (2009) behandelt. Die gängigsten generischen Garantien für MCMC-Verfahren gelten nur, wenn das Modell keinem der Zustände eine Wahrscheinlichkeit von Null zuweist. Daher ist es sehr praktisch, dieses Verfahren als Stichprobenentnahme aus einem energiebasierten Modell (EBM) $p(\mathbf{x}) \propto \exp(-E(\mathbf{x}))$ zu präsentieren (vgl. Abschnitt 16.2.4). Im EBM-Ansatz ist für jeden Zustand eine von Null verschiedene Wahrscheinlichkeit garantiert. MCMC-Verfahren sind tatsächlich breiter anwendbar und können für viele Wahrscheinlichkeitsverteilungen genutzt werden, die Zustände mit einer Wahrscheinlichkeit von Null enthalten. Allerdings müssen die theoretischen Garantien hinsichtlich des Verhaltens der MCMC-Verfahren für unterschiedliche Familien solcher Verteilungen jedes Mal nachgewiesen werden. Im Deep-Learning-Kontext ist es üblich, sich auf allgemeine theoretische Garantien zu verlassen, die naturgemäß für alle energiebasierten Modelle gelten.

Um zu verstehen, warum das Ziehen von Stichproben aus einem energiebasierten Modell schwierig ist, betrachten Sie ein EBM über lediglich zwei Variablen; definiert wird eine Verteilung $p(a, b)$. Um die Stichprobe a zu erhalten, müssen wir a aus $p(a | b)$ ziehen; und um die Stichprobe b zu erhalten, müssen wir es aus $p(b | a)$ ziehen. Das scheint ein unlösbares Henne-Ei-Problem darzustellen. Gerichtete Modelle umgehen dieses Problem, da ihr Graph gerichtet und azyklisch ist. Für das **Ancestral Sampling** (auf Vorfahren/Abstammung basierendes Stichprobenverfahren) wird einfach jede der Variablen in topologischer Reihenfolge gezogen, die bestimmt ist durch die Eltern von jeder Variablen, die bereits als Stichproben gezogen wurden (Abschnitt 16.3). Ancestral Sampling definiert ein effizientes Verfahren zum Ermitteln einer Stichprobe in nur einem Durchgang.

In einem energiebasierten Modell (engl. *energy-based model*, EBM) können wir das Henne-Ei-Problem umgehen, indem wir das Ziehen von Stichproben mithilfe einer Markow-Kette durchführen. Die Grundidee einer Markow-Kette ist, dass es einen Zustands \mathbf{x} gibt, der als beliebiger Wert beginnt. Im Laufe der Zeit aktualisieren wir \mathbf{x} immer wieder nach dem Zufallsprinzip. Irgendwann wird \mathbf{x} (nahezu) eine angemessene Stichprobe aus $p(\mathbf{x})$. Formal

wird eine Markow-Kette definiert durch eine Stichprobe des Zustands \mathbf{x} und eine Übergangsverteilung $T(\mathbf{x}' | \mathbf{x})$ die die Wahrscheinlichkeit angibt, mit der die Anpassung einer Stichprobe zu Zustand \mathbf{x}' übergeht, wenn der Ausgangszustand \mathbf{x} ist. Während der Ausführung der Markow-Kette wird der Zustand \mathbf{x} wiederholt für einen Wert \mathbf{x}' angepasst, der aus $T(\mathbf{x}' | \mathbf{x})$ gezogen wurde.

Um ein theoretisches Verständnis für die Funktionsweise von MCMC-Verfahren zu erlangen, können wir das Problem neu parametrisieren. Zunächst beschränken wir uns auf den Fall, in dem die Zufallsvariable \mathbf{x} abzählbar viele Zustände aufweist. Wir können den Zustand nun ganz einfach durch eine positive ganze Zahl x darstellen. Unterschiedliche ganzzahlige Werte von x verweisen auf unterschiedliche Zustände \mathbf{x} des ursprünglichen Problems.

Betrachten wir nun, was geschieht, wenn wir unendlich viele Markow-Ketten parallel ausführen. Alle Zustände der unterschiedlichen Markow-Ketten werden aus einer Verteilung $q^{(t)}(x)$ gezogen, wobei t die Anzahl der verstrichenen Zeitschritte angibt. Zu Beginn ist $q^{(0)}$ eine Verteilung, die wir zum willkürlichen Initialisieren von x für jede Markow-Kette eingesetzt haben. Später wird $q^{(t)}$ von allen bisher ausgeführten Markow-Ketten-Schritten beeinflusst. Unser Ziel ist, dass $q^{(t)}(x)$ gegen $p(x)$ konvergiert.

Da wir das Problem als positive ganze Zahl x neu parametrisiert haben, können wir die Wahrscheinlichkeitsverteilung q anhand eines Vektors \mathbf{v} beschreiben mit

$$q(\mathbf{x} = i) = v_i. \quad (17.17)$$

Betrachten wir nun, was geschieht, wenn wir den Zustand x einer einzelnen Markow-Kette aktualisieren, sodass ein neuer Zustand x' entsteht. Die Wahrscheinlichkeit, dass ein einzelner Zustand zum Zustand x' wird, ergibt sich aus

$$q^{(t+1)}(x') = \sum_x q^{(t)}(x)T(x' | x). \quad (17.18)$$

Mittels unserer ganzzahligen Parametrisierung können wir die Auswirkung des Übergangsoperators T anhand einer Matrix \mathbf{A} darstellen. Wir definieren \mathbf{A} so, dass

$$A_{i,j} = T(\mathbf{x}' = i | \mathbf{x} = j) \quad (17.19)$$

ist. Mithilfe dieser Definition können wir Gleichung 17.18 nun neu schreiben. Statt dabei q und T für die Aktualisierung eines einzelnen Zustands zu nutzen, können wir nun \mathbf{v} und \mathbf{A} verwenden, um zu beschreiben, wie die

gesamte Verteilung über all die unterschiedlichen (parallel ausgeführten) Markow-Ketten bei einem Update verschoben wird:

$$\mathbf{v}^{(t)} = \mathbf{A}\mathbf{v}^{(t-1)}. \quad (17.20)$$

Das wiederholte Anwenden des Updates der Markow-Kette entspricht der wiederholten Multiplikation mit der Matrix \mathbf{A} . Anders formuliert: Wir können uns den Prozess als Potenzierung der Matrix \mathbf{A} vorstellen:

$$\mathbf{v}^{(t)} = \mathbf{A}^t \mathbf{v}^{(0)}. \quad (17.21)$$

Die Matrix \mathbf{A} weist eine spezielle Struktur auf, da jede ihrer Spalten für eine Wahrscheinlichkeitsverteilung steht. Solche Matrizen werden **Übergangsmatrizen** genannt. Wenn es eine von Null verschiedene Wahrscheinlichkeit für den Übergang von einem beliebigen Zustand x in einen beliebigen anderen Zustand x' für eine Potenz t gibt, garantiert der Satz von Perron-Frobenius (*Perron*, 1907; *Frobenius*, 1908), dass der größte Eigenwert reell und gleich 1 ist. Im Laufe der Zeit können wir sehen, dass alle Eigenwerte potenziert werden:

$$\mathbf{v}^{(t)} = (\mathbf{V} \text{diag}(\boldsymbol{\lambda}) \mathbf{V}^{-1})^t \mathbf{v}^{(0)} = \mathbf{V} \text{diag}(\boldsymbol{\lambda})^t \mathbf{V}^{-1} \mathbf{v}^{(0)}. \quad (17.22)$$

Dieser Prozess führt dazu, dass alle Eigenwerte, die nicht gleich 1 sind, zu 0 zerfallen. Unter einigen weiteren schwachen Voraussetzungen hat \mathbf{A} garantiert nur einen Eigenvektor mit dem Eigenwert 1. Der Prozess konvergiert somit gegen eine **stationäre Verteilung**, die manchmal auch als **Gleichgewichtsverteilung** bezeichnet wird. Bei Konvergenz gilt

$$\mathbf{v}' = \mathbf{A}\mathbf{v} = \mathbf{v}, \quad (17.23)$$

und diese Voraussetzung ist auch für jeden weiteren Schritt zutreffend. Dies ist eine Eigenvektorgleichung. Damit \mathbf{v} ein stationärer Punkt ist, muss es sich um einen Eigenvektor mit zugehörigem Eigenwert 1 handeln. Diese Voraussetzung garantiert, dass wiederholte Anwendungen des Transition Samplings (Übergangsstichprobenverfahren) nach dem Erreichen der stationären Verteilung die *Verteilung* über die Zustände der diversen Markow-Ketten nicht verändert (obwohl der Übergangsoperator natürlich jeden einzelnen Zustand verändert).

Wenn wir T korrekt gewählt haben, ist die stationäre Verteilung q gleich der Verteilung p , aus der wir Stichproben ziehen möchten. Mehr zur Auswahl von T finden Sie in Abschnitt 17.4.

Die meisten Eigenschaften von Markow-Ketten mit abzählbar vielen Zuständen lassen sich auf stetige Variablen generalisieren. In dieser Situation bezeichnen einige Autoren die Markow-Kette als **Harris-Kette**, aber wir nutzen für beide Voraussetzungen den Begriff Markow-Kette. Grundsätzlich konvergiert eine Markow-Kette mit dem Übergangsoperator T unter schwachen Voraussetzungen gegen einen Fixpunkt, der durch die Gleichung

$$q'(\mathbf{x}') = \mathbb{E}_{\mathbf{x} \sim q} T(\mathbf{x}' | \mathbf{x}) \quad (17.24)$$

beschrieben wird; für den diskreten Fall handelt es sich dabei lediglich um eine andere Version von Gleichung 17.23. Ist \mathbf{x} diskret, entspricht der Erwartungswert einer Summe, ist \mathbf{x} stetig, einem Integral.

Ob der Zustand nun stetig oder diskret ist: Alle Markow-Ketten-Verfahren wenden wiederholt stochastische Anpassungen an, bis der Zustand irgendwann zu Stichproben aus der Gleichgewichtsverteilung führt. Das Ausführen der Markow-Kette bis zum Erreichen der Gleichgewichtsverteilung wird als **Burn-In** oder Konvergenzphase der Markow-Kette bezeichnet. Nachdem die Kette den Gleichgewichtszustand erreicht hat, kann eine Reihe unendlich vieler Stichproben aus der Gleichgewichtsverteilung gezogen werden. Sie sind identisch verteilt, aber zwei beliebige aufeinanderfolgende Stichproben werden immer stark miteinander korrelieren. Eine endliche Reihe von Stichproben mag somit nicht sonderlich repräsentativ für die Gleichgewichtsverteilung sein. Eine Möglichkeit, dieses Problem abzuschwächen, besteht darin, nur alle n -ten aufeinanderfolgenden Stichproben auszugeben, sodass unsere Schätzung der statistischen Größen der Gleichgewichtsverteilung nicht durch die Korrelation zwischen einer MCMC-Stichprobe und den nächsten Stichproben verzerrt wird. Der Aufwand für den Einsatz von Markow-Ketten ist somit beträchtlich, da das Burn-In bis zur Gleichgewichtsverteilung viel Zeit in Anspruch nimmt – ebenso wie der Übergang von einer Stichprobe zu einer anderen angemessen dekorrelierten Stichprobe nach Erreichen des Gleichgewichts. Wenn wirklich unabhängige Stichproben gewünscht sind, können mehrere Markow-Ketten parallel ausgeführt werden. Dieser Ansatz eliminiert durch zusätzliche parallele Berechnung die Latenz. Der Einsatz nur einer Markow-Kette für die Generierung aller Stichproben und der Einsatz einer Markow-Kette pro Stichprobe sind die beiden Extreme. Diejenigen, die Deep Learning in der Praxis einsetzen, verwenden meist in etwa so viele Ketten, wie ein Mini-Batch Beispiele enthält, und ziehen dann die benötigte Anzahl von Stichproben aus dieser festen Anzahl von Markow-Ketten. Häufig werden 100 Markow-Ketten verwendet.

Eine weitere Schwierigkeit ist, dass wir im Voraus nicht wissen, wie viele Schritte eine Markow-Kette absolvieren muss, bevor die Gleichgewichtsver-

teilung erreicht wird. Diese Zeitspanne wird **Mischzeit** (engl. *mixing time*) genannt.

Auch die Kontrolle, ob eine Markow-Kette bereits das Gleichgewicht erreicht hat, ist komplex. Zur Beantwortung dieser Frage gibt es keine ausreichend exakte Theorie. Wir wissen lediglich, dass die Kette konvergiert, aber kaum mehr. Wenn wir die Markow-Kette aus Sicht einer Matrix \mathbf{A} , die auf einen Vektor der Wahrscheinlichkeiten \mathbf{v} agiert, betrachten, wissen wir, dass die Kette durchmischt ist, wenn \mathbf{A}^t effektiv alle Eigenwerte von \mathbf{A} verloren hat – bis auf den eindeutigen Eigenwert 1. Dementsprechend bestimmt der Betrag des zweitgrößten Eigenwerts die Mischzeit. In der Praxis können wir unsere Markow-Kette leider nicht als Matrix darstellen. Die Anzahl der Zustände, die unser probabilistisches Modell aufzusuchen kann, ist hinsichtlich der Anzahl der Variablen exponentiell groß, sodass es unmöglich ist, \mathbf{v} , \mathbf{A} oder die Eigenwerte von \mathbf{A} darzustellen. Wegen dieser und anderer Hindernisse wissen wir für gewöhnlich nicht, ob die Markov-Kette gemischt hat. Stattdessen führen wir die Markow-Kette einfach so lange aus, wie wir für notwendig halten, und wenden heuristische Verfahren an, um zu bestimmen, ob die Kette durchmischt ist. Diese heuristischen Verfahren umfassen die manuelle Untersuchung von Stichproben oder das Messen von Korrelationen zwischen aufeinanderfolgenden Stichproben.

17.4 Gibbs-Sampling

Bisher haben wir gezeigt, wie Stichproben aus einer Verteilung $q(\mathbf{x})$ durch wiederholtes Aktualisieren von $\mathbf{x} \leftarrow \mathbf{x}' \sim T(\mathbf{x}' | \mathbf{x})$ gezogen werden. Wir haben nicht erklärt, wie sichergestellt wird, dass $q(\mathbf{x})$ eine nützliche Verteilung ist. In diesem Buch werden zwei grundlegende Ansätze berücksichtigt: Der erste leitet T aus einer gegebenen erlernten p_{model} ab (weiter unten für das Ziehen von Stichproben aus EBMs beschrieben). Der zweite Ansatz parametrisiert T direkt und lernt daraus, sodass die stationäre Verteilung die gesuchte p_{model} implizit definiert. Beispiele für diesen zweiten Ansatz werden in den Abschnitten 20.12 und 20.13 behandelt.

Im Deep-Learning-Kontext verwenden wir Markow-Ketten üblicherweise, um Stichproben aus einem energiebasierten Modell, das eine Verteilung $p_{\text{model}}(\mathbf{x})$ definiert, zu ziehen. In diesem Fall möchten wir, dass $q(\mathbf{x})$ für die Markow-Kette $p_{\text{model}}(\mathbf{x})$ ist. Um das gewünschte $q(\mathbf{x})$ zu erhalten, müssen wir ein passendes $T(\mathbf{x}' | \mathbf{x})$ auswählen.

Ein recht einfacher und effektiver Ansatz zum Erstellen einer Markow-Kette, die Stichproben aus $p_{\text{model}}(\mathbf{x})$ zieht, ist das **Gibbs-Sampling**, bei

dem die Stichprobenentnahme aus $T(\mathbf{x}' \mid \mathbf{x})$ erfolgt, indem eine Variable x_i ausgewählt und aus p_{model} gezogen wird – unter der Voraussetzung, dass ihre Nachbarn im ungerichteten Graphen \mathcal{G} die Struktur des energiebasierten Modells definieren. Wir können auch mehrere Variablen gleichzeitig ziehen, sofern sie hinsichtlich all ihrer Nachbarn bedingt unabhängig sind. Wie im Beispiel der RBM aus Abschnitt 16.7.1 können alle verdeckten Einheiten einer RBM zeitgleich gezogen werden, da sie hinsichtlich aller sichtbaren Einheiten bedingt unabhängig voneinander sind. Ebenso können alle sichtbaren Einheiten zeitgleich gezogen werden, da sie hinsichtlich aller verdeckten Einheiten bedingt unabhängig voneinander sind. Gibbs-Sampling-Ansätze, die auf diese Weise viele Variablen zeitgleich aktualisieren, werden **Block-Gibbs-Sampling** genannt.

Es gibt alternative Herangehensweisen zum Konzipieren von Markow-Ketten, um Stichproben aus p_{model} zu ziehen. Zum Beispiel ist in anderen Disziplinen der Metropolis-Hastings-Algorithmus weit verbreitet. Im Deep-Learning-Kontext wird bei ungerichteten Modellen nur selten auf das Gibbs-Sampling verzichtet. Verbesserte Stichprobenverfahren sind ein aktueller Gegenstand der Forschung.

17.5 Die Herausforderung, zwischen getrennten Modi zu mischen

Die größte Schwierigkeit im Zusammenhang mit MCMC-Verfahren ist, dass sie zu einer schlechten **Mischung** neigen. Idealerweise wären aufeinanderfolgende Stichproben aus einer Markow-Kette, die dafür konzipiert wurde, Stichproben aus $p(\mathbf{x})$ zu ziehen, vollständig unabhängig voneinander und würden viele unterschiedliche Bereiche im \mathbf{x} -Raum proportional zu ihrer Wahrscheinlichkeit aufsuchen. Stattdessen – und speziell in hochdimensionalen Fällen – sind MCMC-Stichproben stark korreliert. Wir sprechen hier von langsamem Mischen (engl. *slow mixing*) oder sogar dem Scheitern des Mischens. MCMC-Verfahren mit langsamem Mischen führen quasi unabsehbar eine Art verrauchtes Gradientenabstiegsverfahren (engl. *noisy gradient descent*) auf der Energiefunktion aus oder äquivalent den sogenannten Noisy-Hill-Climbing-Algorithmus auf der Wahrscheinlichkeit, und zwar bezüglich des Zustands der Kette (der Zufallsvariablen, die gezogen werden). Die Kette neigt dazu, kleine Schritte (im Zustandsraum der Markow-Kette) von einer Konfiguration $\mathbf{x}^{(t-1)}$ zu einer Konfiguration $\mathbf{x}^{(t)}$ vorzunehmen, wobei die Energie $E(\mathbf{x}^{(t)})$ allgemein niedriger oder näherungsweise gleich der Energie $E(\mathbf{x}^{(t-1)})$ ist; dabei werden Bewegungen bevorzugt, die

zu Konfigurationen mit niedrigerer Energie führen. Wenn man mit einer ziemlich unwahrscheinlichen Konfiguration beginnt (höhere Energie als für $p(\mathbf{x})$ üblich), neigt die Kette dazu, nach und nach die Energie des Zustands zu reduzieren und nur gelegentlich in einen anderen Modus zu wechseln. Sobald die Kette einen Bereich mit niedriger Energie gefunden hat (zum Beispiel könnte es sich, wenn die Variablen Pixel in einem Bild sind, bei einem Bereich mit niedriger Energie um eine verbundene Mannigfaltigkeit von Bildern desselben Objekts handeln), den wir als Modus bezeichnen, neigt die Kette dazu, sich um diesen Modus herum zu bewegen (in einer Art Random Walk). Hin und wieder verlässt sie diesen Modus und kehrt dann wieder zurück oder wechselt in einen anderen Modus (falls sie eine Möglichkeit findet, aus dem Modus auszutreten, also einen Ausweg aus dem Modus findet, engl. *escape route*). Das Problem ist, dass es für viele interessante Verteilungen nur wenige Auswege gibt, sodass die Markow-Kette länger als sollte Stichproben im selben Modus zieht.

Das wird offensichtlich, wenn wir uns den Gibbs-Sampling-Algorithmus ansehen (Abschnitt 17.4). In diesem Kontext betrachten wir die Wahrscheinlichkeit für den Wechsel aus einem Modus binnen einer bestimmten Anzahl von Schritten in einen nahegelegenen Modus. Die Wahrscheinlichkeit wird anhand der Form der »Energiebarriere« zwischen diesen Modi bestimmt. Übergänge zwischen zwei Modi, die durch eine hohe Energiebarriere (einen Bereich mit niedriger Wahrscheinlichkeit) getrennt sind, sind exponentiell weniger wahrscheinlich (hinsichtlich der Höhe der Energiebarriere). Abbildung 17.1 stellt dies dar. Das Problem tritt auf, wenn es mehrere Modi mit hoher Wahrscheinlichkeit gibt, die durch Bereiche mit niedriger Wahrscheinlichkeit voneinander getrennt sind, insbesondere wenn jeder Schritt des Gibbs-Samplings nur eine kleine Teilmenge der Variablen aktualisieren muss, deren Werte hauptsächlich durch andere Variablen bestimmt werden.

Betrachten Sie als einfaches Beispiel ein energiebasiertes Modell über zwei Variablen a und b , die beide binär mit einem Vorzeichen sind, also die Werte -1 und 1 annehmen. Ist $E(a, b) = -wab$ für eine große positive Zahl w , dann drückt das Modell eine starke Überzeugung dahingehend aus, dass a und b dasselbe Vorzeichen aufweisen. Aktualisieren Sie nun b unter Verwendung des Gibbs-Sampling-Schritts mit $a = 1$. Die bedingte Verteilung über b ergibt sich aus $P(b = 1 | a = 1) = \sigma(w)$. Ist w groß, sättigt die Sigmoidfunktion und die Wahrscheinlichkeit, dass auch b gleich 1 wird, ist nahezu 1 . Ebenso ist für $a = -1$ die Wahrscheinlichkeit, dass b ebenfalls -1 wird, nahezu 1 . Laut $P_{\text{model}}(a, b)$ sind beide Vorzeichen beider Variablen gleich wahrscheinlich. Laut $P_{\text{model}}(a | b)$ sollten beide Variablen dasselbe

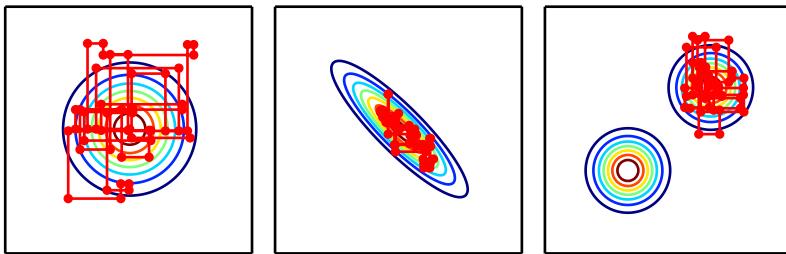


Abbildung 17.1: Pfade des Gibbs-Sampling für drei Verteilungen. In beiden Fällen wird die Markow-Kette im Modus initialisiert. (*Links*) Eine multivariate Normalverteilung mit zwei unabhängigen Variablen. Das Gibbs-Sampling mischt gut, da die Variablen unabhängig sind. (*Mitte*) Eine multivariate Normalverteilung mit stark korrelierten Variablen. Die Korrelation zwischen den Variablen erschwert der Markow-Kette das Mischen. Da die Aktualisierung für jede Variable von der anderen Variable abhängt, reduziert die Korrelation die Rate, mit der die Markow-Kette sich vom Startpunkt wegbewegen kann. (*Rechts*) Eine gaußsche Mischverteilung mit breit verteilten Modi ohne übereinstimmende Achsen. Das Gibbs-Sampling mischt sehr langsam, da es schwierig ist, den Modus zu wechseln, wenn jede einzelne Variable nacheinander verändert wird.

Vorzeichen aufweisen. Das bedeutet, dass Gibbs-Sampling nur sehr selten das Vorzeichen dieser Variablen umkehrt.

In praxisrelevanteren Szenarios ist die Herausforderung noch größer, da wir nicht nur Übergänge zwischen zwei Modi anstreben, sondern zwischen möglichst allen Modi, die ein echtes Modell enthalten könnte. Wenn mehrere solcher Übergänge infolge der Schwierigkeiten zwischen Modi zu mischen schwierig sind, wird es sehr aufwendig, eine zuverlässige Menge von Stichproben zu erhalten, die die meisten der Modi abdeckt; auch die Konvergenz der Ketten gegen ihre stationäre Verteilung erfolgt sehr langsam.

Manchmal lässt sich das Problem lösen, indem Gruppen stark abhängiger Einheiten gesucht und blockweise gleichzeitig aktualisiert werden. Leider kann es bei komplexen Abhängigkeiten rechnerisch nicht effizient möglich sein, ein Beispiel aus der Gruppe zu ziehen. Schließlich ist das Problem, zu deren Lösung die Markow-Kette ursprünglich eingeführt wurde, genau das der Stichprobenentnahme aus einer großen Gruppe von Variablen.

Bei Modellen mit latenten Variablen, die eine multivariate Verteilung $p_{\text{model}}(\mathbf{x}, \mathbf{h})$ definieren, ziehen wir oft Stichproben von \mathbf{x} durch Hin- und Herwechseln zwischen dem Ziehen von Stichproben aus $p_{\text{model}}(\mathbf{x} | \mathbf{h})$ und aus $p_{\text{model}}(\mathbf{h} | \mathbf{x})$. Für ein schnelles Mischen soll $p_{\text{model}}(\mathbf{h} | \mathbf{x})$ eine hohe Entropie aufweisen. Für das Erlernen einer nützlichen Repräsentation von \mathbf{h} soll \mathbf{h} genügend Informationen über \mathbf{x} codieren, um es gut zu rekonstruieren;

hierzu müssen h und x über viele gemeinsame Informationen verfügen. Diese beiden Ziele widersprechen einander. Wir erlernen häufig generative Modelle, die x sehr präzise in h codieren, sich aber nicht gut mischen lassen. Dieser Fall tritt häufig bei Boltzmann-Maschinen auf – je steiler die Verteilung ist, die eine Boltzmann-Maschine lernt, desto schwieriger ist es für eine Markow-Kette, die Stichproben aus der Modellverteilung zieht, gut zu mischen. Das Problem ist in Abbildung 17.2 dargestellt.

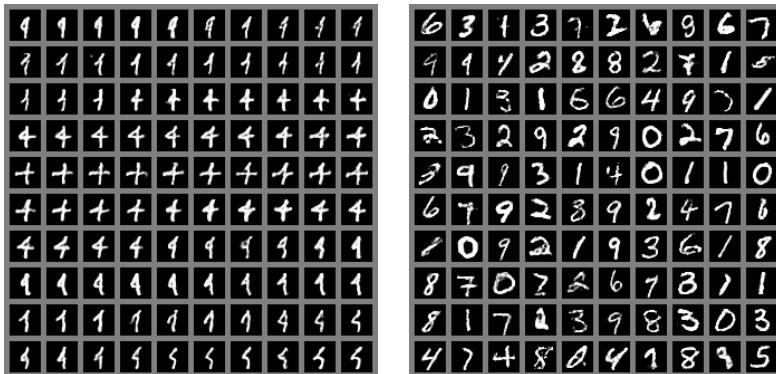


Abbildung 17.2: Eine Darstellung des Problems des langsamen Mischens in tiefen probabilistischen Modellen. Jedes Feld muss von links nach rechts und von oben nach unten gelesen werden. (*Links*) Aufeinanderfolgende Stichproben aus einem Gibbs-Sampling, angewandt auf eine DBM (Deep Boltzmann Machine), die mit dem MNIST-Datensatz trainiert wurde. Aufeinanderfolgende Stichproben sind einander ähnlich. Da das Gibbs-Sampling in einem tiefen graphischen Modell stattfindet, ist diese Ähnlichkeit eher semantisch als optisch. Dennoch ist es für die Gibbs-Kette immer noch schwierig, zwischen den Modi zu wechseln – zum Beispiel durch Ändern der Ziffernidentität. (*Rechts*) Aufeinanderfolgende Ancestral Samples (auf Vorfahren/Abstammung basierende Stichproben) aus einem Generative-Adversarial-Netz. Da das Ancestral Sampling jede Stichprobe unabhängig von den anderen generiert, gibt es kein Mischproblem.

All dies könnte dazu führen, dass MCMC-Verfahren weniger nützlich sind, wenn die betrachtete Verteilung eine Mannigfaltigkeitsstruktur mit separaten Mannigfaltigkeiten für jede Klasse aufweist: Die Verteilung konzentriert sich auf viele Modi und diese sind durch große Bereiche hoher Energie voneinander getrennt. Diese Art Verteilung erwarten wir bei vielen Klassifizierungsproblemen; sie würde zu einer langsamen Konvergenz von MCMC-Verfahren infolge der schlechten Mischung zwischen Modi führen.

17.5.1 Tempering zum Mischen zwischen Modi

Wenn eine Verteilung steile Gipfel von hoher Wahrscheinlichkeit aufweist, die von Bereichen mit niedriger Wahrscheinlichkeit umgeben sind, ist das Mischen zwischen den unterschiedlichen Modi der Verteilung schwierig. Mehrere Verfahren für ein schnelleres Mischen beruhen auf dem Erstellen alternativer Versionen der Zielverteilung, in denen die Gipfel nicht so hoch und die umgebenden Täler nicht so tief sind. Energiebasierte Modelle bieten eine besonders einfache Methode hierfür. Bisher haben wir ein energiebasiertes Modell als eine Wahrscheinlichkeitsverteilung definierend beschrieben:

$$p(\mathbf{x}) \propto \exp(-E(\mathbf{x})). \quad (17.25)$$

Energiebasierte Modelle können um einen zusätzlichen Parameter β ergänzt werden, der steuert, wie steil die Verteilung um die Gipfel ansteigt und wieder abfällt:

$$p_\beta(\mathbf{x}) \propto \exp(-\beta E(\mathbf{x})). \quad (17.26)$$

Der Parameter β wird häufig als Kehrwert der **Temperatur** beschrieben und spiegelt den Ursprung energiebasierter Modelle in der statistischen Physik wider. Wenn die Temperatur auf Null fällt und β gegen Unendlichkeit steigt, wird das energiebasierte Modell deterministisch. Wenn die Temperatur ins Unendliche steigt und β auf Null fällt, wird die Verteilung (bei diskretem \mathbf{x}) zu einer Gleichverteilung.

Üblicherweise wird ein Modell für eine Evaluation mit $\beta = 1$ trainiert. Allerdings können wir auch andere Temperaturen verwenden, insbesondere solche, für die $\beta < 1$ ist. **Tempering** ist ein allgemeines Verfahren zum schnellen Mischen zwischen Modi von p_1 durch Ziehen von Stichproben mit $\beta < 1$.

Markow-Ketten auf Basis von **Tempered Transitions** (Neal, 1994) ziehen vorübergehend Stichproben aus Verteilungen mit höherer Temperatur, um unterschiedliche Modi zu mischen, bevor wieder Stichproben aus der Verteilung mit Einheitstemperatur gezogen werden. Diese Verfahren wurden für Modelle wie RBMs verwendet (Salakhutdinov, 2010). Ein weiterer Ansatz ist **Parallel Tempering** (Iba, 2001), bei dem die Markow-Kette viele unterschiedliche Zustände parallel simuliert, die unterschiedliche Temperaturen aufweisen. Die Zustände mit den höchsten Temperaturen werden nur langsam durchmischt, die mit der niedrigsten Temperatur (Temperatur 1) stellen dagegen akkurate Stichproben aus dem Modell zur Verfügung. Der Übergangsoperator umfasst das stochastische Tauschen von Zuständen zwischen zwei verschiedenen Temperaturniveaus, sodass eine Stichprobe mit hinreichend hoher Wahrscheinlichkeit aus einem Slot mit hoher Temperatur

in einen Slot mit niedriger Temperatur springen kann. Dieser Ansatz wurden ebenfalls für RBMs verwendet (*Desjardins et al.*, 2010; *Cho et al.*, 2010). Obwohl das Tempering ein vielversprechender Ansatz ist, konnte die Forschung bisher noch keine großen Fortschritte bei der Lösung des Problems der Stichprobenentnahme aus komplexen EBMs machen. Ein möglicher Grund dafür ist, dass es **kritische Temperaturen** gibt, bei denen der Temperaturübergang sehr langsam erfolgen muss (da die Temperatur nach und nach reduziert wird), damit das Tempering effizient ist.

17.5.2 Besseres Mischen durch Tiefe

Beim Ziehen von Stichproben aus einem Modell $p(\mathbf{h}, \mathbf{x})$ mit latenten Variablen hat sich gezeigt, dass – wenn \mathbf{x} durch $p(\mathbf{h} | \mathbf{x})$ zu gut codiert wird – die Stichprobenentnahme aus $p(\mathbf{x} | \mathbf{h})$ kaum zu einer Änderung von \mathbf{x} führt, sodass eine schlechte Mischung entsteht. Eine Möglichkeit zum Lösen dieses Problems besteht darin, \mathbf{h} zu einer tiefen Repräsentation zu machen und \mathbf{x} so in \mathbf{h} zu codieren, dass eine Markow-Kette im Raum von \mathbf{h} leichter mischen kann. Viele Representation-Learning-Algorithmen wie Autoencoder und RBMs neigen dazu, eine Randverteilung über \mathbf{h} auszugeben, die gleichförmiger und unimodaler als die ursprüngliche Datenverteilung über \mathbf{x} ist. Man kann argumentieren, dass dies eine Folge des Versuchs ist, den Rekonstruktionsfehler zu minimieren und gleichzeitig den gesamten verfügbaren Darstellungsraum zu nutzen, da das Minimieren des Rekonstruktionsfehlers über den Trainingsbeispielen sich besser erreichen lässt, wenn unterschiedliche Trainingsbeispiele voneinander im \mathbf{h} -Raum leicht zu unterscheiden und somit gut separiert sind. *Bengio et al.* (2013a) haben festgestellt, dass tiefere Stapel regularisierter Autoencoder oder RBMs Randverteilungen im oberen Bereich des \mathbf{h} -Raums ausgeben, die weiter verteilt und gleichförmiger erscheinen und eine kleinere Lücke zwischen den Bereichen aufweisen, die den unterschiedlichen Modi entsprechen (in den Experimenten: Kategorien). Das Trainieren einer RBM in diesem höherrangigen Raum ermöglicht dem Gibbs-Sampling ein schnelleres Mischen zwischen Modi. Es bleibt jedoch unklar, wie diese Beobachtung dabei helfen kann, ein besseres Training und Ziehen von Stichproben mit tiefen generativen Modellen zu erzielen.

Trotz der Schwierigkeiten beim Mischen sind Monte-Carlo-Verfahren nützlich und oft auch das beste verfügbare Hilfsmittel. Tatsächlich sind sie das das wichtigste Tool im Umgang mit der nicht effizient berechenbaren Partitionsfunktion ungerichteter Modelle, um die es im nächsten Kapitel geht.

18

Die Partitionsfunktion

In Abschnitt 16.2.2 haben wir gezeigt, dass viele probabilistische Modelle (häufig auch als ungerichtete graphische Modelle bezeichnet) durch eine nicht normalisierte Wahrscheinlichkeitsverteilung $\tilde{p}(\mathbf{x}; \boldsymbol{\theta})$ definiert werden. Wir müssen \tilde{p} normalisieren, indem wir durch eine Partitionsfunktion $Z(\boldsymbol{\theta})$ dividieren, sodass wir eine gültige Wahrscheinlichkeitsverteilung erhalten:

$$p(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \tilde{p}(\mathbf{x}; \boldsymbol{\theta}). \quad (18.1)$$

Die Partitionsfunktion ist ein Integral (für stetige Variablen) oder eine Summe (für diskrete Variablen) über der nicht normalisierten Wahrscheinlichkeit aller Zustände:

$$\int \tilde{p}(\mathbf{x}) d\mathbf{x} \quad (18.2)$$

oder

$$\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}). \quad (18.3)$$

Diese Operation ist für viele interessante Modelle nicht effizient durchführbar (engl. *intractable*).

Wie wir in Kapitel 20 noch zeigen werden, sind diverse Deep-Learning-Modelle dazu gedacht, eine effizient berechenbare Normalisierungskonstante zu enthalten, oder sie werden so eingesetzt, dass $p(\mathbf{x})$ überhaupt nicht berechnet werden muss. Andere Modelle werden sogar speziell für den Fall der nicht effizient berechenbaren Partitionsfunktion eingesetzt. In diesem Kapitel beschreiben wir Verfahren zum Trainieren und Bewerten von Modellen, die nicht effizient berechenbare Partitionsfunktionen enthalten.

18.1 Der Log-Likelihood-Gradient

Das Erlernen ungerichteter Modelle mittels Maximum Likelihood ist deswegen so schwierig, weil die Partitionsfunktion von den Parametern abhängig ist. Der Gradient der Log-Likelihood bezüglich der Parameter weist einen Term auf, der dem Gradienten der Partitionsfunktion entspricht:

$$\nabla_{\theta} \log p(\mathbf{x}; \boldsymbol{\theta}) = \nabla_{\theta} \log \tilde{p}(\mathbf{x}; \boldsymbol{\theta}) - \nabla_{\theta} \log Z(\boldsymbol{\theta}). \quad (18.4)$$

Dies ist eine bekannte Zerlegung in die **positive Phase** und die **negative Phase** des Lernprozesses.

Für die meisten ungerichteten Modelle, die wir betrachten, ist die negative Phase schwierig. Modelle ohne latente Variablen oder mit wenigen Interaktionen zwischen latenten Variablen weisen meist eine effizient durchführbare positive Phase auf. Das Musterbeispiel für ein Modell mit geradliniger positiver Phase und schwieriger negativer Phase ist die RBM, deren verdeckte Einheiten bezüglich der sichtbaren Einheiten bedingt unabhängig voneinander sind. Der Fall einer schwierigen positiven Phase mit komplizierten Interaktionen zwischen latenten Variablen wird hauptsächlich in Kapitel 19 behandelt. Dieses Kapitel rückt die Schwierigkeiten der negativen Phase in den Fokus.

Sehen wir uns den Gradienten von $\log Z$ genauer an:

$$\nabla_{\theta} \log Z \quad (18.5)$$

$$= \frac{\nabla_{\theta} Z}{Z} \quad (18.6)$$

$$= \frac{\nabla_{\theta} \sum_{\mathbf{x}} \tilde{p}(\mathbf{x})}{Z} \quad (18.7)$$

$$= \frac{\sum_{\mathbf{x}} \nabla_{\theta} \tilde{p}(\mathbf{x})}{Z}. \quad (18.8)$$

Für Modelle, in denen $p(\mathbf{x}) > 0$ für alle \mathbf{x} garantiert ist, können wir $\exp(\log \tilde{p}(\mathbf{x}))$ für $\tilde{p}(\mathbf{x})$ einsetzen:

$$\frac{\sum_{\mathbf{x}} \nabla_{\theta} \exp(\log \tilde{p}(\mathbf{x}))}{Z} \quad (18.9)$$

$$= \frac{\sum_{\mathbf{x}} \exp(\log \tilde{p}(\mathbf{x})) \nabla_{\theta} \log \tilde{p}(\mathbf{x})}{Z} \quad (18.10)$$

$$= \frac{\sum_{\mathbf{x}} \tilde{p}(\mathbf{x}) \nabla_{\theta} \log \tilde{p}(\mathbf{x})}{Z} \quad (18.11)$$

$$= \sum_{\mathbf{x}} p(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}) \quad (18.12)$$

$$= \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}). \quad (18.13)$$

Diese Ableitung nutzt die Aufsummierung über diskretem \mathbf{x} , aber ein ähnliches Ergebnis gilt bei der Integration über stetigem \mathbf{x} . In der stetigen Version dieser Ableitung nutzen wir die Leibnizregeln zur Differentiation unter dem Integralzeichen, um die Identität zu bestimmen:

$$\nabla_{\boldsymbol{\theta}} \int \tilde{p}(\mathbf{x}) d\mathbf{x} = \int \nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x}) d\mathbf{x}. \quad (18.14)$$

Diese Identität ist nur unter bestimmten Regularitätsbedingungen für \tilde{p} und $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$ anwendbar. In der Maßtheorie würde man die Bedingungen wie folgt formulieren: (1) Die nicht normalisierte Verteilung \tilde{p} muss eine Lebesgue-integrierbare Funktion von \mathbf{x} für jeden Wert von $\boldsymbol{\theta}$ sein. (2) Der Gradient $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$ muss für alle $\boldsymbol{\theta}$ und fast alle \mathbf{x} vorliegen. (3) Es muss eine integrierbare Funktion $R(\mathbf{x})$ geben, die $\nabla_{\boldsymbol{\theta}} \tilde{p}(\mathbf{x})$ insofern einschränkt, als $\max_i |\frac{\partial}{\partial \theta_i} \tilde{p}(\mathbf{x})| \leq R(\mathbf{x})$ für alle $\boldsymbol{\theta}$ und fast alle \mathbf{x} gilt. Zum Glück weisen die meisten betrachteten Machine-Learning-Modelle diese Eigenschaften auf.

Diese Identität

$$\nabla_{\boldsymbol{\theta}} \log Z = \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}) \quad (18.15)$$

bildet die Grundlage für diverse Monte-Carlo-Verfahren zur approximativen Maximierung der Likelihood von Modellen mit nicht effizient berechenbaren Partitionsfunktionen.

Der Monte-Carlo-Ansatz zum Erlernen ungerichteter Modelle bietet ein intuitives Framework, in dem wir die positive Phase und die negative Phase betrachten können. In der positiven Phase erhöhen wir $\log \tilde{p}(\mathbf{x})$ für aus den Daten als Stichproben gezogene \mathbf{x} . In der negativen Phase verringern wir die Partitionsfunktion durch Verringern des aus der Modellverteilung gezogenen $\log \tilde{p}(\mathbf{x})$.

In der Deep-Learning-Literatur ist es üblich, $\log \tilde{p}$ als Energiefunktion zu parametrisieren (Gleichung 16.7). In diesem Fall können wir die positive Phase als Herabdrücken der Energie der Trainingsbeispiele betrachten und die negative Phase als Hinaufdrücken der Energie der Stichproben, die aus dem Modell gezogen wurden (vgl. Abbildung 18.1).

18.2 Stochastische Maximum Likelihood und kontrastive Divergenz

Die naive Implementierung von Gleichung 18.15 besteht darin, sie im Rahmen eines Burn-Ins einer Reihe von Markow-Ketten aus einer Zufallsinitialisierung bei jeder Nutzung des Gradienten zu berechnen. Wenn das Lernen mit dem stochastischen Gradientenabstiegsverfahren erfolgt, müssen die Ketten somit in jedem Gradientenschritt einmal einem Burn-In unterzogen werden. Dieser Ansatz führt zum Trainingsverfahren aus Algorithmus 18.1. Der hohe Aufwand für das Burn-In der Markow-Ketten in der inneren Schleife macht dieses Verfahren rechnerisch undurchführbar, aber es dient durchaus als Ausgangspunkt für die Approximation praxistauglicherer Algorithmen.

Algorithmus 18.1 Ein naiver MCMC-Algorithmus zum Maximieren der Log-Likelihood mit einer nicht effizient berechenbaren Partitionsfunktion, der den Gradientenanstieg verwendet

Setze ϵ , die Schrittweite, auf eine kleine positive Zahl.

Setze k , die Anzahl der Gibbs-Schritte, auf einen Wert, der hoch genug für ein Burn-In ist, möglicherweise 100 zum Trainieren einer RBM mit einem kleinen Bildbereich.

while nicht konvergiert **do**

Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ aus der Trainingsdatenmenge

$$\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} \log \tilde{p}(\mathbf{x}^{(i)}; \theta).$$

Initialisieren einer Menge mit m Stichproben $\{\tilde{\mathbf{x}}^{(1)}, \dots, \tilde{\mathbf{x}}^{(m)}\}$ auf zufällige Werte (z. B. aus einer Gleich- bzw. Normalverteilung oder möglicherweise aus einer Verteilung mit Randwahrscheinlichkeiten, die zu den Randwahrscheinlichkeiten des Modells passen).

for $i = 1$ to k **do**

for $j = 1$ to m **do**

$$\tilde{\mathbf{x}}^{(j)} \leftarrow \text{Gibbs_Update}(\tilde{\mathbf{x}}^{(j)}).$$

end for

end for

$$\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \theta).$$

$$\theta \leftarrow \theta + \epsilon \mathbf{g}.$$

end while

Wir können den MCMC-Ansatz zur Maximum Likelihood als Versuch betrachten, zwei Kräfte auszutarieren, nämlich zum einen die Kraft, die

die Modellverteilung für die Daten nach oben drückt, und zum anderen die Kraft, die die Modellverteilung mit den zugehörigen Stichproben nach unten drückt. Abbildung 18.1 stellt dies dar. Die beiden Kräfte entsprechen dem Maximieren von $\log \tilde{p}$ und dem Minimieren von $\log Z$. Es sind mehrere Approximationen an die negative Phase möglich. Jede dieser Approximationen sorgt dafür, dass die negative Phase rechnerisch zwar weniger aufwendig wird, gleichzeitig aber an den falschen Stellen nach unten drückt.

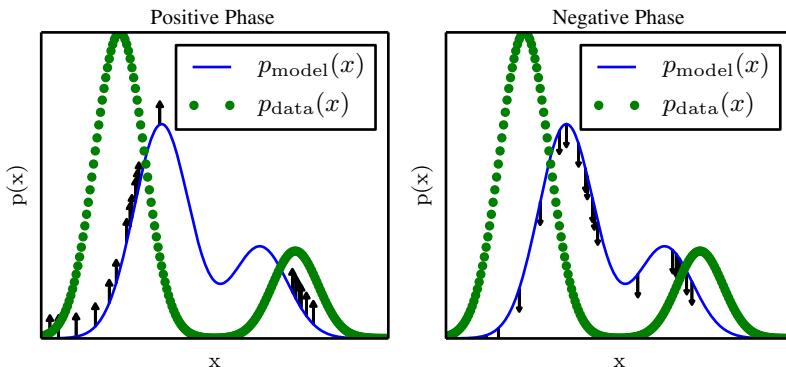


Abbildung 18.1: Die Darstellung des Algorithmus 18.1 mit einer »positiven Phase« und einer »negativen Phase«. (*Links*) In der positiven Phase ziehen wir Punkte aus der Datenverteilung und drücken deren nicht normalisierte Wahrscheinlichkeit nach oben. Somit werden in den Daten wahrscheinliche Punkte stärker nach oben gedrückt. (*Rechts*) In der negativen Phase ziehen wir Punkte aus der Modellverteilung und drücken deren nicht normalisierte Wahrscheinlichkeit nach unten. Das ist die »Gegenkraft« zur Tendenz der positiven Phase, einfach überall eine große Konstante zur nicht normalisierten Wahrscheinlichkeit zu addieren. Wenn die Datenverteilung und die Modellverteilung gleich sind, ist die Chance der positiven Phase, an einem Punkt nach oben zu drücken, gleich der Chance der negativen Phase, nach unten zu drücken. Wenn dies geschieht, gibt es keinen Gradienten (im Erwartungswert) mehr und das Training muss enden.

Da die negative Phase die Stichprobenentnahme aus der Verteilung des Modells beinhaltet, können wir sie so verstehen, dass sie Punkte sucht, denen das Modell besonders stark vertraut. Da die negative Phase die Wahrscheinlichkeit dieser Punkte verringert, sagt man, dass sie die fehlerhaften Vorstellungen des Modells von der Welt darstellen. In der Literatur werden sie oft als »Halluzinationen« oder »Fantasy Particles« (erfundene Partikel) bezeichnet. Tatsächlich wurde die negative Phase als mögliche Erklärung für das Träumen bei Menschen und anderen Tieren vorgeschlagen (*Crick und Mitchison, 1983*); dahinter steckt die Idee, dass das Gehirn ein probabilistisches Modell der Welt unterhält und dem Gradienten von $\log \tilde{p}$ folgt, wenn es reale Ereignisse im wachen Zustand erlebt, aber dem negativen Gradienten

von $\log \tilde{p}$ folgt, um $\log Z$ während des Schlafs zu minimieren und dabei Ereignisse erlebt, die quasi als Stichproben aus dem aktuellen Modell gezogen wurden. Diese Ansicht erklärt viel über die Repräsentationsform, die zur Abbildung von Algorithmen mit positiver und negativer Phase verwendet wird; sie wurde aber nicht durch neurowissenschaftliche Experimente verifiziert. In Machine-Learning-Modellen müssen positive und negative Phase meist zeitgleich verwendet werden; es gibt also keine separaten Zeiträume des Wachseins und des REM-Schlafs. Wie wir in Abschnitt 19.5 zeigen, ziehen andere Machine-Learning-Algorithmen für andere Zwecke Stichproben aus der Modellverteilung. Solche Algorithmen könnten ebenfalls Erklärungen für die Funktion des Traumschlafs liefern.

Auf Basis dieses Verständnisses, welche Rolle die positive und negative Phase des Lernens spielen, können wir versuchen, eine weniger aufwendige Alternative zu Algorithmus 18.1 zu erstellen. Der Hauptaufwand beim naiven MCMC-Algorithmus entsteht beim Burn-In der Markow-Ketten ausgehend von einer Zufallsinitialisierung in jedem Schritt. Eine natürliche Lösung besteht darin, die Markow-Ketten mittels einer Verteilung zu initialisieren, die der Modellverteilung sehr nahe liegt, sodass der Burn-In-Prozess nicht so viele Schritte benötigt.

Die **kontrastive Divergenz** (engl. *contrastive divergence*, kurz CD oder CD- k für eine CD mit k Gibbs-Schritten) ist ein Algorithmus zum Initialisieren der Markow-Kette in jedem Schritt anhand von Stichproben aus der Datenverteilung (Hinton, 2000, 2010). Dieser Ansatz ist in Algorithmus 18.2 dargestellt. Das Ermitteln von Beispielen aus der Datenverteilung ist nicht mit Aufwand verbunden, da sie bereits im Datensatz enthalten sind. Zunächst liegt die Datenverteilung nicht nahe bei der Modellverteilung, sodass die negative Phase nicht sonderlich genau ist. Die positive Phase kann aber die Wahrscheinlichkeit der Daten für das Modell dennoch exakt erhöhen. Nachdem die positive Phase einige Zeit aktiv war, liegt die Modellverteilung näher an der Datenverteilung; nun wird auch die negative Phase genauer.

Natürlich stellt auch die kontrastive Divergenz lediglich eine Approximation an die korrekte negative Phase dar. Der Hauptgrund für ein qualitatives Versagen bei der Implementierung der korrekten negativen Phase durch die kontrastive Divergenz besteht darin, dass der Algorithmus keine Bereiche hoher Wahrscheinlichkeit unterdrückt, die weitab der eigentlichen Trainingsbeispiele liegen. Diese Bereiche haben unter dem Modell eine hohe Wahrscheinlichkeit, aber nur eine geringe Wahrscheinlichkeit unter der datengenerierenden Verteilung; man nennt sie **Störmodi**. Abbildung 18.2 zeigt, wieso es dazu kommt. Grundsätzlich werden Modi in der Modellverteilung, die weitab der Datenverteilung liegen, von Markow-Ketten, die an

Algorithmus 18.2 Der CD-Algorithmus (kontrastive Divergenz) nutzt den Gradientenanstieg als Optimierungsverfahren.

Setze ϵ , die Schrittweite, auf eine kleine positive Zahl.

Setze k , die Anzahl der Gibbs-Schritte, auf einen Wert, der hoch genug ist, damit eine Markow-Kette Stichproben aus $p(\mathbf{x}; \boldsymbol{\theta})$ ziehen kann, um zu mischen, wenn die Initialisierung aus p_{data} erfolgte. Möglicherweise 1–20 zum Trainieren einer RBM mit einem kleinen Bildbereich.

while nicht konvergiert **do**

Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ aus der Trainingsdatenmenge
 $\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta}).$
for $i = 1$ to m **do**
 $\tilde{\mathbf{x}}^{(i)} \leftarrow \mathbf{x}^{(i)}.$
end for
for $i = 1$ to k **do**
 for $j = 1$ to m **do**
 $\tilde{\mathbf{x}}^{(j)} \leftarrow \text{Gibbs_Update}(\tilde{\mathbf{x}}^{(j)}).$
 end for
end for
 $\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta}).$
 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \mathbf{g}.$
end while

Trainingspunkten initialisiert werden, nicht aufgesucht, es sei denn, k ist sehr groß.

Carreira-Perpiñan und Hinton (2005) haben experimentell nachgewiesen, dass der CD-Schätzer hinsichtlich RBMs und vollständig sichtbarer Boltzmann-Maschinen (engl. *fully visible Boltzmann machines*) insofern verzerrt ist, als er gegen andere Punkte konvergiert als der Maximum-Likelihood-Schätzer. Sie argumentieren, dass die kontrastive Divergenz aufgrund der kleinen Verzerrung als günstige Methode zur Initialisierung eines Modells dienen kann, das später mithilfe aufwendiger MCMC-Verfahren feinabgestimmt werden könnte. *Bengio und Delalleau (2009)* zeigen, dass die kontrastive Divergenz quasi die kleinsten Terme des korrekten MCMC-Update-Gradienten verwirft, wodurch sich die Verzerrung erklären lässt.

Die kontrastive Divergenz ist beim Trainieren flacher Modelle wie RBMs nützlich. Diese können wiederum zur Initialisierung tieferer Modelle (DBNs, DBMs) gestapelt werden. Aber die kontrastive Divergenz ist nicht besonders hilfreich, wenn tiefere Modelle direkt trainiert werden sollen. Das liegt daran,

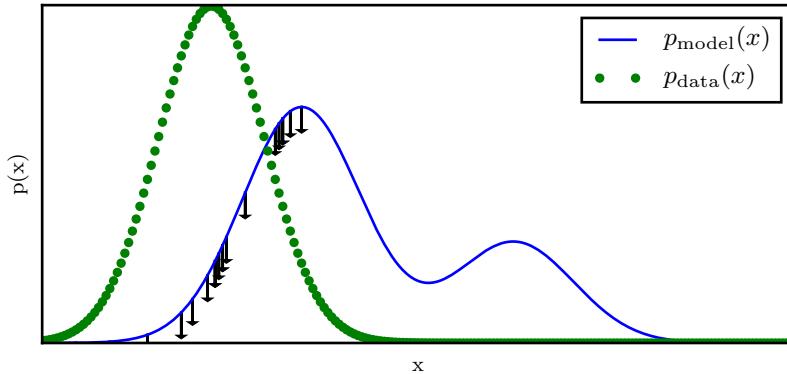


Abbildung 18.2: Ein Störmodus. Die Darstellung zeigt, wie die negative Phase der kontrastiven Divergenz (Algorithmus 18.2) beim Unterdrücken von Störmodi versagt. Ein Störmodus ist ein Modus, der in der Modellverteilung gezeigt wird, aber in der Datenverteilung fehlt. Da die kontrastive Divergenz ihre Markow-Ketten anhand von Datenpunkten initialisiert und die Markow-Kette nur wenige Schritte ausgeführt wird, ist es unwahrscheinlich, dass Modi, die weitab der Datenpunkte liegen, im Modell gefunden werden. Daher erhalten wir bei der Stichprobenentnahme aus dem Modell manchmal Stichproben, die den Daten nicht ähneln. Außerdem wird ein Teil der Wahrscheinlichkeit(smasse) an diese Modi vergeben, sodass das Modell Probleme bekommt, den korrekten Modi eine hohe Wahrscheinlichkeit(smasse) zuzuweisen. Für Zwecke der Visualisierung nutzt diese Abbildung ein vereinfachtes Konzept des Abstands – der Störmodus liegt weit entfernt vom korrekten Modus entlang der Zahlengerade \mathbb{R} . Dies entspricht einer Markow-Kette, die auf lokalen Bewegungen mit einer einzelnen x -Variable in \mathbb{R} beruht. Für die meisten tiefen probabilistischen Modelle beruhen die Markow-Ketten auf dem Gibbs-Sampling und können nichtlokale Bewegungen einzelner Variablen ausführen, aber nicht alle Variablen zeitgleich bewegen. Für diese Probleme ist es meist besser, anstelle des euklidischen Abstands die Levenshtein-Distanz (Editierdistanz) zwischen den Modi zu betrachten. Allerdings lässt sich die Levenshtein-Distanz in einem hochdimensionalen Raum schwierig in einem 2-D-Plot darstellen.

dass es schwierig ist, Stichproben von sichtbaren Einheiten bei gegebenen verdeckten Einheiten zu erhalten. Da die verdeckten Einheiten in den Daten nicht enthalten sind, löst auch die Initialisierung an Trainingspunkten das Problem nicht. Selbst wenn wir die sichtbaren Einheiten anhand der Daten initialisieren, müssen wir noch immer ein Burn-In für eine Markow-Ketten-Stichprobenentnahme aus der Verteilung über die verdeckten Einheiten durchführen, die durch diese sichtbaren Stichproben vorgegeben sind.

Der CD-Algorithmus bestraft sozusagen das Modell dafür, dass es eine Markow-Kette enthält, die die Eingabe sehr schnell verändert, sobald die Eingabe aus den Daten stammt. Somit ähnelt das Trainieren mit der kon-

trastiven Divergenz bis zu einem gewissen Grad dem Autoencoder-Training. Obwohl die kontrastive Divergenz eine höhere Verzerrung als einige andere Trainingsverfahren aufweist, kann sie für das Pretraining flacher Modelle, die später gestapelt werden, nützlich sein. Das liegt daran, dass die frühesten Modelle im Stapel dazu angeregt werden, mehr Informationen in ihre latenten Variablen zu kopieren, damit diese den späteren Modellen zur Verfügung stehen. Sie können sich dies wie eine häufig genutzte Nebenwirkung des CD-Trainings vorstellen, nicht so sehr als grundsätzlichen Designvorteil.

Sutskever und Tieleman (2010) haben gezeigt, dass die Aktualisierungsrichtung der kontrastiven Divergenz nicht der Gradient einer Funktion ist. Es kann somit dazu kommen, dass die kontrastive Divergenz endlos läuft – in der Praxis stellt dies allerdings kein ernsthaftes Problem dar.

Eine andere Vorgehensweise, die viele der Probleme mit der kontrastiven Divergenz löst, besteht im Initialisieren der Markow-Ketten in jedem Gradientenschritt mit den Zuständen aus dem vorhergehenden Gradientenschritt. Dieser Ansatz wurde zuerst unter der Bezeichnung **stochastische Maximum Likelihood** (engl. *stochastic maximum likelihood*, SML) in der angewandten Mathematik und Statistik entdeckt (*Younes*, 1998) und später unabhängig davon nochmals unter der Bezeichnung **persistente kontrastive Divergenz** (engl. *persistent contrastive divergence*, PCD oder PCD- k mit Angabe der k Gibbs-Schritte für jedes Update) in der Deep-Learning-Forschungsgemeinde »entdeckt« (*Tieleman*, 2008), siehe Algorithmus 18.3. Die Grundidee des Ansatzes besagt, dass – sofern die Schritte des stochastischen Gradientenalgorithmus klein sind – das Modell des vorherigen Schritts dem Modell aus dem aktuellen Schritt ähnlich ist. Daraus folgt, dass die Stichproben aus der Verteilung des vorherigen Modells sehr nahe an angemessenen Stichproben aus der Verteilung des aktuellen Modells liegen, sodass eine Markow-Kette, die mit diesen Stichproben initialisiert wird, keine sonderlich lange Mischzeit benötigt.

Da jede Markow-Kette während des Lernprozesses kontinuierlich aktualisiert wird (und nicht in jedem Gradientenschritt neu gestartet wird), können sich die Ketten auch ungehindert weit genug bewegen, um alle Modi des Modells zu finden. Die SML neigt somit deutlich weniger als die kontrastive Divergenz dazu, Modelle mit Störmodi zu bilden. Da der Zustand aller als Stichproben gezogenen Variablen – sowohl sichtbar als auch latent – gespeichert werden kann, bietet die SML außerdem einen Initialisierungspunkt für die verdeckten **und** die sichtbaren Einheiten. Die kontrastive Divergenz ist dazu nur für die sichtbaren Einheiten in der Lage, weshalb für tiefe Modelle ein Burn-In benötigt wird. Die SML kann tiefe Modelle effizient trainieren. *Marlin et al.* (2010) haben die SML mit vielen anderen in diesem Kapitel

vorgestellten Kriterien verglichen. Sie haben festgestellt, dass die SML zur besten Log-Likelihood der Testdatenmenge für eine RBM führt. Wenn die verdeckten Einheiten der RBM als Merkmale für einen Support-Vektor-Machine-Klassifikator verwendet werden, führt die SML auch zur besten Korrektklassifikationsrate.

Allerdings ist die SML anfällig dafür, ungenau zu werden, wenn der stochastische Gradientenalgorithmus das Modell schneller bewegen kann, als die Markow-Kette zwischen den Schritten mischen kann. Dies kann geschehen, wenn k zu klein oder ϵ zu groß ist. Der zulässige Wertebereich ist leider stark aufgabenabhängig. Es gibt keine bekannte Methode, um formal zu prüfen, ob die Kette zwischen den Schritten erfolgreich mischt. Ist die Lernrate für die Anzahl der Gibbs-Schritte zu hoch, kann der Entwickler subjektiv viel mehr Varianz in den Stichproben der negativen Phase bei den Gradientenschritten als bei den unterschiedlichen Markow-Ketten beobachten. Zum Beispiel könnte ein mittels MNIST trainiertes Modell in einem Schritt ausschließlich Stichproben bestehend aus Varianten der Ziffer 7 ziehen. Der Lernprozess senkt dann stark den Modus, der der Ziffer 7 entspricht, sodass das Modell im nächsten Schritt möglicherweise nur Stichproben bestehend aus Varianten der Ziffer 9 zieht.

Die Auswertung der Stichproben aus einem mit SML trainierten Modell muss sorgfältig erfolgen. Die Stichproben müssen beginnend mit einer neuen Markow-Kette, die von einem zufälligen Startpunkt nach Abschluss des Modelltrainings aus initialisiert wurde, gezogen werden. Die Stichproben in den persistent negativen Ketten, die für das Training verwendet wurden, sind durch mehrere vorige Versionen des Modells beeinflusst worden und können somit dazu führen, dass es so aussieht, als ob das Modell eine größere Kapazität aufweist, als dies tatsächlich der Fall ist.

Berglund und Raiko (2013) haben diese Verzerrung und diese Varianz in Experimenten für die Gradientenschätzung untersucht, die von der kontrastiven Divergenz und SML bereitgestellt wird. Die kontrastive Divergenz weist demzufolge eine geringere Varianz als der Schätzer auf Basis einer exakten Stichprobenentnahme auf. Die SML weist eine höhere Varianz auf. Der Grund für die geringere Varianz in der kontrastiven Divergenz liegt darin, dass sie dieselben Trainingspunkte in der positiven und in der negativen Phase nutzt. Wird die negative Phase an unterschiedlichen Trainingspunkten initialisiert, steigt die Varianz über die des Schätzers auf Basis einer exakten Stichprobenentnahme an.

Alle Verfahren, in denen MCMC für die Stichprobenentnahme aus dem Modell verwendet wird, lassen sich prinzipiell mit nahezu allen MCMC-

Algorithmus 18.3 Der Algorithmus für die stochastische Maximum Likelihood/persistente kontrastive Divergenz nutzt den Gradientenanstieg als Optimierungsverfahren

Setze ϵ , die Schrittweite, auf eine kleine positive Zahl.

Setze k , die Anzahl der Gibbs-Schritte, auf einen Wert, der hoch genug ist, damit für das Burn-In eine Markow-Ketten-Stichprobenentnahme aus $p(\mathbf{x}; \boldsymbol{\theta} + \epsilon \mathbf{g})$ erfolgen kann, beginnend mit Stichproben aus $p(\mathbf{x}; \boldsymbol{\theta})$. Möglicherweise 1 für eine RBM mit einem kleinen Bildbereich oder 5–50 für komplexere Modelle wie eine DBM.

Initialisieren einer Menge mit m Stichproben $\{\tilde{\mathbf{x}}^{(1)}, \dots, \tilde{\mathbf{x}}^{(m)}\}$ auf zufällige Werte (z. B. aus einer Gleich- bzw. Normalverteilung oder möglicherweise aus einer Verteilung mit Randwahrscheinlichkeiten, die zu den Randwahrscheinlichkeiten des Modells passen).

while nicht konvergiert **do**

Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ aus der Trainingsdatenmenge

$$\mathbf{g} \leftarrow \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x}^{(i)}; \boldsymbol{\theta}).$$

for $i = 1$ to k **do**

for $j = 1$ to m **do**

$$\tilde{\mathbf{x}}^{(j)} \leftarrow \text{Gibbs_Update}(\tilde{\mathbf{x}}^{(j)}).$$

end for

end for

$$\mathbf{g} \leftarrow \mathbf{g} - \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\tilde{\mathbf{x}}^{(i)}; \boldsymbol{\theta}).$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \mathbf{g}.$$

end while

Varianten einsetzen. Verfahren wie die SML können also mit beliebigen der in Kapitel 17 vorgestellten erweiterten MCMC-Verfahren optimiert werden, wie zum Beispiel mit dem Parallel Tempering (*Desjardins et al.*, 2010; *Cho et al.*, 2010).

Ein Ansatz zum Beschleunigen des Mischens während der Lernphase ändert nicht etwa die Methoden für das Monte-Carlo-Stichprobenverfahren, sondern vielmehr die Parametrisierung des Modells und der Kostenfunktion. Bei der **Fast PCD** (kurz FPCD, dt. *schnelle persistente kontrastive Divergenz*) (*Tieleman und Hinton*, 2009) werden die Parameter $\boldsymbol{\theta}$ eines klassischen Modells durch den folgenden Ausdruck ersetzt:

$$\boldsymbol{\theta} = \boldsymbol{\theta}^{(\text{slow})} + \boldsymbol{\theta}^{(\text{fast})}. \quad (18.16)$$

Nun gibt es doppelt so viele Parameter wie zuvor. Diese werden elementweise addiert, um die in der ursprünglichen Modelldefinition verwendeten

Parameter anzugeben. Die schnelle Kopie der Parameter wird mit einer viel größeren Lernrate trainiert, sodass sie schnell auf die negative Phase des Lernprozesses reagieren und die Markow-Kette in ein neues Gebiet drücken kann. Das zwingt die Markow-Kette zum schnellen Mischen, obschon dieser Effekt nur während der Lernens auftritt, solange die schnellen Gewichte (engl. *fast weights*) sich beliebig ändern können. Üblicherweise wird auf die schnellen Gewichte auch ein erheblicher Weight Decay angewendet, damit sie gegen kleine Werte konvergieren, nachdem sie vorübergehend hohe Werte angenommen haben, um die Markow-Kette zu animieren, den Modus zu wechseln.

Ein wesentlicher Vorteil der in diesem Abschnitt beschriebenen MCMC-Verfahren ist, dass sie eine Gradientenschätzung für $\log Z$ liefern und wir somit das Problem in den Beitrag $\log \tilde{p}$ und den Beitrag $\log Z$ zerlegen können. Anschließend können wir $\log \tilde{p}(\mathbf{x})$ mit beliebigen anderen Verfahren angehen und einfach unseren Gradienten der negativen Phase zum Gradienten des anderen Verfahrens addieren. Das bedeutet konkret, dass unsere positive Phase Verfahren nutzen kann, die nur eine untere Schranke für \tilde{p} liefern. Die meisten anderen in diesem Kapitel vorgestellten Verfahren, die sich mit $\log Z$ befassen, sind inkompatibel mit Verfahren für positive Phasen auf Basis von Schranken.

18.3 Pseudo-Likelihood

Monte-Carlo-Approximationen der Partitionsfunktion und ihres Gradienten widmen sich direkt der Partitionsfunktion. Andere Ansätze umgehen das Problem, indem das Modell ohne Berechnung der Partitionsfunktion trainiert wird. Die meisten dieser Ansätze beruhen auf der Feststellung, dass die Verhältnisse der Wahrscheinlichkeiten in einem ungerichteten probabilistischen Modell recht einfach berechnet werden können. Das liegt daran, dass die Partitionsfunktion sowohl im Zähler als auch im Nenner des Verhältnisses auftaucht und sich somit herauskürzt:

$$\frac{p(\mathbf{x})}{p(\mathbf{y})} = \frac{\frac{1}{Z}\tilde{p}(\mathbf{x})}{\frac{1}{Z}\tilde{p}(\mathbf{y})} = \frac{\tilde{p}(\mathbf{x})}{\tilde{p}(\mathbf{y})}. \quad (18.17)$$

Die Pseudo-Likelihood beruht auf der Feststellung, dass bedingte Wahrscheinlichkeiten diese Form (auf Basis eines Verhältnisses) annehmen und somit ohne Kenntnis der Partitionsfunktion berechnet werden können. Angenommen, wir zerlegen \mathbf{x} in \mathbf{a} , \mathbf{b} und \mathbf{c} , wobei \mathbf{a} die Variablen enthält, für

die wir die bedingte Verteilung suchen, **b** die Variablen, die als Bedingung gelten sollen, und **c** die Variablen, die nicht Teil unserer Abfrage sind:

$$p(\mathbf{a} \mid \mathbf{b}) = \frac{p(\mathbf{a}, \mathbf{b})}{p(\mathbf{b})} = \frac{p(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a}, \mathbf{c}} p(\mathbf{a}, \mathbf{b}, \mathbf{c})} = \frac{\tilde{p}(\mathbf{a}, \mathbf{b})}{\sum_{\mathbf{a}, \mathbf{c}} \tilde{p}(\mathbf{a}, \mathbf{b}, \mathbf{c})}. \quad (18.18)$$

Diese Größe erfordert das Entfernen von **a**; dabei kann es sich um eine sehr effiziente Operation handeln, sofern **a** und **c** nicht viele Variablen enthalten. Im Extremfall kann **a** eine einzige Variable und **c** leer sein, sodass für diese Operation nur so viele Berechnungen von \tilde{p} erforderlich sind, wie es Werte einer einzelnen Zufallsvariable gibt.

Leider müssen wir zum Berechnen der Log-Likelihood große Sätze von Variablen entfernen. Für insgesamt n Variablen müssen wir einen Satz der Größe $n - 1$ marginalisieren. Laut der Produktregel für Wahrscheinlichkeiten gilt

$$\log p(\mathbf{x}) = \log p(x_1) + \log p(x_2 \mid x_1) + \cdots + p(x_n \mid \mathbf{x}_{1:n-1}). \quad (18.19)$$

In diesem Fall haben wir **a** maximal klein gemacht, aber **c** kann genauso so groß wie $\mathbf{x}_{2:n}$ sein. Was geschieht, wenn wir **c** einfach in **b** verschieben, um den Berechnungsaufwand zu senken? Wir erhalten die **Pseudo-Likelihood-Zielfunktion** (*Besag, 1975*), basierend auf der Vorhersage des Werts des Merkmals x_i auf Grundlage aller anderen Merkmale \mathbf{x}_{-i} :

$$\sum_{i=1}^n \log p(x_i \mid \mathbf{x}_{-i}). \quad (18.20)$$

Wenn jede Zufallsvariable k unterschiedliche Werte aufweist, benötigen wir nur $k \times n$ Berechnungen von \tilde{p} zur Berechnung anstelle der k^n Auswertungen zur Berechnung der Partitionsfunktion.

Das sieht vielleicht aus wie ein Regelverstoß, aber es lässt sich nachweisen, dass die Schätzung durch Maximieren der Pseudo-Likelihood asymptotisch konsistent ist (*Mase, 1995*). Natürlich kann die Pseudo-Likelihood für Datensätze, die sich nicht dem großen Stichproben-Limit nähern, ein anderes Verhalten als der Maximum-Likelihood-Schätzer aufweisen.

Es ist möglich, rechnerische Komplexität mit einem von der Maximum Likelihood abweichenden Verhalten zu ersetzen, indem wir den **generalisierten Pseudo-Likelihood-Schätzer** (engl. *generalized pseudolikelihood estimator*) verwenden (*Huang und Ogata, 2002*). Der generalisierte Pseudo-Likelihood-Schätzer nutzt m unterschiedliche Mengen $\mathbb{S}^{(i)}$, $i = 1, \dots, m$ von Variablen-Indizes, die gemeinsam auf der linken Seite des Bedingungszeichens

stehen. Im Extremfall $m = 1$ und $\mathbb{S}^{(1)} = 1, \dots, n$ stellt die generalisierte Pseudo-Likelihood die Log-Likelihood wieder her. Im Extremfall $m = n$ und $\mathbb{S}^{(i)} = \{i\}$ stellt die generalisierte Pseudo-Likelihood die Pseudo-Likelihood wieder her. Die generalisierte Pseudo-Likelihood-Zielfunktion ergibt sich aus

$$\sum_{i=1}^m \log p(\mathbf{x}_{\mathbb{S}^{(i)}} \mid \mathbf{x}_{-\mathbb{S}^{(i)}}). \quad (18.21)$$

Die Leistung von Ansätzen auf Basis der Pseudo-Likelihood ist stark davon abhängig, wie das Modell verwendet wird. Die Pseudo-Likelihood neigt dazu, bei Aufgaben, die ein gutes Modell der gesamten Multivariaten $p(\mathbf{x})$ benötigen, schlecht abzuschneiden. Beispiele dafür sind Dichteschätzung und Stichprobenverfahren. Sie arbeitet besser als die Maximum Likelihood, wenn die Aufgaben nur die bedingten Verteilungen aus dem Training benötigen. Beispiele hierfür sind das Ermitteln kleiner Mengen fehlender Werte. Generalisierte Pseudo-Likelihood-Methoden sind besonders leistungsstark, wenn die Daten eine regelmäßige Struktur aufweisen, sodass die \mathbb{S} -Indexmengen so gestaltet werden können, dass sie die wichtigsten Korrelationen erfassen, während Variablengruppen mit vernachlässigbarer Korrelation ausgelassen werden. Zum Beispiel weisen räumlich weit voneinander entfernte Pixel in natürlichen Bildern eine schwache Korrelation auf, sodass bei Anwendung der generalisierten Pseudo-Likelihood jede \mathbb{S} -Menge ein kleines räumlich begrenztes Fenster darstellt.

Eine Schwachstelle des Pseudo-Likelihood-Schätzers ist, dass er nicht mit anderen Approximationen eingesetzt werden kann, die nur eine untere Schranke für $\tilde{p}(\mathbf{x})$ liefern, darunter die Variational Inference (siehe Kapitel 19). Das liegt daran, dass \tilde{p} Teil des Nenners ist. Eine untere Schranke für den Nenner liefert nur eine obere Schranke für den Gesamtausdruck – und das Maximieren einer oberen Schranke bietet keinen Vorteil. Das macht es schwierig, die Pseudo-Likelihood in tiefen Modellen wie DBMs (Deep Boltzmann Machines) anzuwenden, da Variationsmethoden einer der primären Ansätze zum approximativen Entfernen der vielen Schichten mit verdeckten Variablen sind, die miteinander interagieren. Nichtsdestotrotz ist die Pseudo-Likelihood für das Deep Learning von Belang, da sich damit einschichtige Modelle oder tiefe Modelle mit Verfahren der approximativen Inferenz trainieren lassen, die nicht auf unteren Schranken basieren.

Der Aufwand pro Gradientenschritt ist bei der Pseudo-Likelihood deutlich höher als bei der SML, da alle bedingten Verteilungen explizit berechnet werden müssen. Aber die generalisierte Pseudo-Likelihood und ähnliche Kriterien können noch immer gut abschneiden, wenn nur eine zufällig ausgewählte bedingte Verteilung pro Beispiel berechnet wird (*Goodfellow et al.*,

2013b), sodass der Berechnungsaufwand gesenkt wird und dem für die SML entspricht.

Obwohl der Pseudo-Likelihood-Schätzer $\log Z$ nicht explizit minimiert, ähnelt er doch in gewisser Weise einer negativen Phase. Die Nenner der einzelnen bedingten Verteilungen führen dazu, dass der Lernalgorithmus die Wahrscheinlichkeit sämtlicher Zustände unterdrückt, die sich in nur einer Variable von einem Trainingsbeispiel unterscheiden.

Marlin und de Freitas (2011) enthält eine theoretische Analyse der asymptotischen Effizienz der Pseudo-Likelihood.

18.4 Score Matching und Ratio Matching

Score Matching (*Hyrvärinen*, 2005) ist eine weitere konsistente Möglichkeit zum Trainieren eines Modells ohne Schätzung von Z oder seiner Ableitungen. Der Name *Score Matching* bezieht sich auf die Bezeichnung von Ableitungen einer Log-Dichte bezüglich ihres Arguments, $\nabla_{\mathbf{x}} \log p(\mathbf{x})$, als **Score**. Beim Score Matching wird das Quadrat der erwarteten Differenz zwischen den Ableitungen der Log-Dichte des Modells bezüglich der Eingabe und den Ableitungen der Log-Dichte der Daten bezüglich der Eingabe minimiert:

$$L(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{2} \|\nabla_{\mathbf{x}} \log p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}), -\nabla_{\mathbf{x}} \log p_{\text{data}}(\mathbf{x})\|_2^2, \quad (18.22)$$

$$J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{p_{\text{data}}(\mathbf{x})} L(\mathbf{x}, \boldsymbol{\theta}), \quad (18.23)$$

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}). \quad (18.24)$$

Diese Zielfunktion umgeht die Schwierigkeiten, die mit der Differentiation der Partitionsfunktion Z assoziiert sind, da Z keine Funktion von \mathbf{x} ist und somit $\nabla_{\mathbf{x}} Z = 0$. Anfänglich scheint das Score Matching ein Problem durch ein anderes zu ersetzen: Zum Berechnen des Scores für die Datenverteilung muss die wahre Verteilung zur Generierung der Trainingsdaten, p_{data} , bekannt sein. Zum Glück entspricht das Minimieren des Erwartungswerts für $L(\mathbf{x}, \boldsymbol{\theta})$ dem Minimieren des Erwartungswerts für

$$\tilde{L}(\mathbf{x}, \boldsymbol{\theta}) = \sum_{j=1}^n \left(\frac{\partial^2}{\partial x_j^2} \log p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}) + \frac{1}{2} \left(\frac{\partial}{\partial x_j} \log p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}) \right)^2 \right), \quad (18.25)$$

wobei n die Dimensionalität von \mathbf{x} angibt.

Da für das Score Matching Ableitungen bezüglich \mathbf{x} benötigt werden, kann es nicht für Modelle mit diskreten Daten genutzt werden; allerdings dürfen die latenten Variablen im Modell diskret sein.

Wie die Pseudo-Likelihood funktioniert auch das Score Matching nur, wenn wir $\log \tilde{p}(\mathbf{x})$ und seine Ableitungen direkt ermitteln können. Es ist nicht kompatibel mit Verfahren, die nur eine untere Schranke für $\log \tilde{p}(\mathbf{x})$ angeben, da das Score Matching die Ableitungen und die zweiten Ableitungen von $\log \tilde{p}(\mathbf{x})$ benötigt; eine untere Schranke vermittelt keine Angaben über ihre Ableitungen. Damit lässt sich Score Matching nicht zum Schätzen von Modellen mit komplizierten Interaktionen zwischen den verdeckten Einheiten nutzen, darunter Modelle mit Sparse Coding oder DBMs. Score Matching kann zwar für das Pretraining der ersten verdeckten Schicht eines größeren Modells eingesetzt werden, es wurde aber nicht als Pretraining-Verfahren für die tieferen Schichten eines größeren Modells verwendet. Dies liegt vermutlich daran, dass die verdeckten Schichten solcher Modelle meist einige diskrete Variablen enthalten.

Obwohl das Score Matching nicht explizit eine negative Phase aufweist, kann es als Variante der kontrastiven Divergenz mittels spezieller Markow-Kette betrachtet werden (*Hyvärinen*, 2007a). Die Markow-Kette ist hierbei kein Gibbs-Sampling, sondern ein anderer Ansatz, der lokale Bewegungen durch den Gradienten geführt vornimmt. Score Matching entspricht der kontrastiven Divergenz mit einer solchen Markow-Kette, wenn die Größe der lokalen Bewegungen gegen Null geht.

Lyu (2009) hat das Score Matching für den diskreten Fall generalisiert (aber in der Ableitung einen Fehler gemacht, der durch *Marlin et al.* [2010] korrigiert wurde). *Marlin et al.* (2010) haben herausgefunden, dass das **generalisierte Score Matching** (GSM) in hochdimensionalen diskreten Räumen, in denen die beobachtete Wahrscheinlichkeit vieler Ereignisse 0 ist, nicht funktioniert.

Ein erfolgreicher Ansatz bei der Erweiterung der Grundkonzepte des Score Matchings für diskrete Daten ist das **Ratio Matching** (*Hyvärinen*, 2007b). Das Ratio Matching findet ganz besonders für binäre Daten Anwendung. Dabei wird der Durchschnittswert über Beispielen der folgenden Zielfunktion minimiert:

$$L^{(\text{RM})}(\mathbf{x}, \boldsymbol{\theta}) = \sum_{j=1}^n \left(\frac{1}{1 + \frac{p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})}{p_{\text{model}}(f(\mathbf{x}, j); \boldsymbol{\theta})}} \right)^2, \quad (18.26)$$

wobei $f(\mathbf{x}, j)$ den Wert \mathbf{x} mit an der Position j geflipptem Bit zurückgibt. Ratio Matching umgeht die Partitionsfunktion auf dieselbe Weise wie der

Pseudo-Likelihood-Schätzer: In einem Verhältnis von zwei Wahrscheinlichkeiten wird die Partitionsfunktion herausgekürzt. *Marlin et al.* (2010) haben festgestellt, dass Ratio Matching der stochastischen Maximum Likelihood (SML), der Pseudo-Likelihood und dem generalisierten Score Matching insoweit überlegen ist, als es die Fähigkeit der mit Ratio Matching trainierten Modelle zum Denoising (Entrauschen) der Bilder einer Testdatenmenge betrifft.

Wie der Pseudo-Likelihood-Schätzer müssen für das Ratio Matching n Berechnungen von \tilde{p} für jeden Datenpunkt erfolgen, sodass der Berechnungsaufwand pro Update etwa n -mal höher als bei der SML ist.

Wie der Pseudo-Likelihood-Schätzer drückt auch das Ratio Matching quasi alle erfundenen Zustände nach unten, die sich in nur einer Variable von einem Trainingsbeispiel unterscheiden. Da das Ratio Matching insbesondere für binäre Daten gilt, wirkt es auf sämtliche erfundenen Zustände im Hamming-Abstand 1 der Daten.

Ratio Matching kann auch eine nützliche Basis für den Umgang mit hochdimensionalen dünnbesetzten Daten sein, zum Beispiel Wortzählungsvektoren. Diese Art von Daten stellt für Verfahren auf MCMC-Basis ein Problem dar, da sie einen extremen Aufwand zur Darstellung in einem dichtbesetzten Format erfordern, zumal der MCMC-Sampler überhaupt erst kleine Werte erzeugt, nachdem das Modell gelernt hat, eine Verteilung mit dünnbesetzten Daten darzustellen. *Dauphin und Bengio* (2013) haben als Lösung eine erwartungstreue stochastische Approximation für das Ratio Matching entwickelt. Die Approximation wertet nur eine zufällig ausgewählte Teilmenge der Terme des Ziels aus und benötigt kein Modell zur Generierung vollständiger erfundener Stichproben.

Marlin und de Freitas (2011) enthält eine theoretische Analyse der asymptotischen Effizienz des Ratio Matchings.

18.5 Denoising Score Matching

In einigen Fällen möchten wir das Score Matching regularisieren, indem wir eine Verteilung

$$p_{\text{smoothed}}(\mathbf{x}) = \int p_{\text{data}}(\mathbf{y})q(\mathbf{x} \mid \mathbf{y})d\mathbf{y} \quad (18.27)$$

anstelle der wahren Verteilung p_{data} anpassen. Die Verteilung $q(\mathbf{x} \mid \mathbf{y})$ ist ein Beschädigungsprozess, und zwar meist derart, dass \mathbf{x} durch Addieren einer kleinen Menge Rauschen zu \mathbf{y} gebildet wird.

Das Denoising Score Matching ist besonders nützlich, da wir in der Praxis normalerweise keinen Zugang zur echten Verteilung p_{data} haben, sondern nur zu einer empirischen Verteilung, die mithilfe von Stichproben daraus definiert ist. Jeder konsistente Schätzer mit hinreichender Kapazität macht p_{model} zu einer Menge von Dirac-Verteilungen, deren Zentrum die Trainingspunkte sind. Durch Glätten von q wird dieses Problem abgeschwächt, allerdings zulasten der asymptotischen Konsistenz (vgl. Abschnitt 5.4.5). *Kingma und LeCun (2010)* haben ein Verfahren für ein regularisiertes Score Matching vorgestellt, bei dem die Glättungsverteilung q normalverteiltes Rauschen ist.

Sie wissen aus Abschnitt 14.5.1, dass diverse Autoencoder-Trainingsalgorithmen dem Score Matching oder Denoising Score Matching gleichwertig sind. Diese Autoencoder-Trainingsalgorithmen sind daher eine Möglichkeit, der Partitionsfunktion aus dem Weg zu gehen.

18.6 Noise-Contrastive Estimation

Die meisten Verfahren zur Schätzung von Modellen mit nicht effizient berechenbaren Partitionsfunktionen liefern keine Schätzung der Partitionsfunktion. SML und kontrastive Divergenz schätzen nur den Gradienten der Log-Partitionsfunktion, nicht aber die Partitionsfunktion selbst. Score Matching und Pseudo-Likelihood vermeiden das Berechnen von Größen in Bezug auf die Partitionsfunktion vollständig.

Die **Noise-Contrastive Estimation** (NCE, *Rausch-kontrastive Schätzung*) (*Gutmann und Hyvarinen, 2010*) nutzt einen anderen Ansatz. Dabei wird die vom Modell geschätzte Wahrscheinlichkeitsverteilung explizit repräsentiert als

$$\log p_{\text{model}}(\mathbf{x}) = \log \tilde{p}_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}) + c, \quad (18.28)$$

wobei c explizit als Approximation von $-\log Z(\boldsymbol{\theta})$ eingeführt wird. Statt nur $\boldsymbol{\theta}$ zu schätzen, behandelt die NCE c als weiteren Parameter und schätzt gleichzeitig $\boldsymbol{\theta}$ und c mithilfe desselben Algorithmus. Der resultierende $\log p_{\text{model}}(\mathbf{x})$ entspricht somit nicht unbedingt exakt einer gültigen Wahrscheinlichkeitsverteilung, aber er nähert sich der Gültigkeit immer weiter an, während die Schätzung von c besser wird.¹

¹ Die NCE ist auch auf Probleme mit einer effizient berechenbaren Partitionsfunktion anzuwenden, wenn der zusätzliche Parameter c nicht eingeführt werden muss. Allerdings ist sie vor allem als Mittel zur Schätzung von Modellen mit schwierigen Partitionsfunktionen bekannt.

Ein solcher Ansatz wäre bei Verwendung der Maximum Likelihood als Kriterium des Schätzers nicht möglich. Das Maximum-Likelihood-Kriterium würde c willkürlich hoch setzen, statt c so zu setzen, dass eine gültige Wahrscheinlichkeitsverteilung erzeugt wird.

Die NCE reduziert das unüberwachte Lernproblem der Schätzung von $p(\mathbf{x})$ auf das Problem, einen probabilistischen binären Klassifikator zu erlernen, in dem eine der Kategorien mit den vom Modell generierten Daten korrespondiert. Dieses überwachte Lernproblem ist so aufgebaut, dass die Maximum-Likelihood-Schätzung einen asymptotisch konsistenten Schätzer des ursprünglichen Problems definiert.

Dabei führen wir eine zweite Verteilung ein, die **Rauschverteilung** $p_{\text{noise}}(\mathbf{x})$. Die Rauschverteilung muss effizient berechenbar sein und es müssen sich effizient Stichproben daraus entnehmen lassen. Wir können nun ein Modell über \mathbf{x} und einer neuen binären Klassenvariable y konstruieren. Im neuen Kombimodell geben wir an, dass

$$p_{\text{joint}}(y = 1) = \frac{1}{2}, \quad (18.29)$$

$$p_{\text{joint}}(\mathbf{x} \mid y = 1) = p_{\text{model}}(\mathbf{x}), \quad (18.30)$$

und

$$p_{\text{joint}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x}) \quad (18.31)$$

ist. Anders ausgedrückt ist y eine Schaltvariable, die bestimmt, ob wir \mathbf{x} aus der Modell- oder Rauschverteilung generieren.

Wir können ein ähnliches Kombimodell für Trainingsdaten konstruieren. In diesem Fall bestimmt die Schaltvariable, ob wir \mathbf{x} aus der Daten- oder der Rauschverteilung ziehen. Formal ausgedrückt: $p_{\text{train}}(y = 1) = \frac{1}{2}$, $p_{\text{train}}(\mathbf{x} \mid y = 1) = p_{\text{data}}(\mathbf{x})$ und $p_{\text{train}}(\mathbf{x} \mid y = 0) = p_{\text{noise}}(\mathbf{x})$.

Wir können nun das normale Maximum Likelihood Learning auf das **überwachte** Lernproblem zur Anpassung von p_{joint} an p_{train} anwenden:

$$\theta, c = \arg \max_{\theta, c} \mathbb{E}_{\mathbf{x}, y \sim p_{\text{train}}} \log p_{\text{joint}}(y \mid \mathbf{x}). \quad (18.32)$$

Die Verteilung p_{joint} ist praktisch ein logistisches Regressionsmodell, das auf die Differenz der Log-Wahrscheinlichkeiten der Modell- und der Rauschverteilung angewandt wird:

$$p_{\text{joint}}(y = 1 \mid \mathbf{x}) = \frac{p_{\text{model}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x}) + p_{\text{noise}}(\mathbf{x})} \quad (18.33)$$

$$= \frac{1}{1 + \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}} \quad (18.34)$$

$$= \frac{1}{1 + \exp\left(\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right)} \quad (18.35)$$

$$= \sigma\left(-\log \frac{p_{\text{noise}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x})}\right) \quad (18.36)$$

$$= \sigma(\log p_{\text{model}}(\mathbf{x}) - \log p_{\text{noise}}(\mathbf{x})). \quad (18.37)$$

Die NCE ist somit leicht anwendbar, sofern die Backpropagation durch log \tilde{p}_{model} einfach ist und, wie oben gezeigt, die Berechnung von p_{noise} (zur Evaluation von p_{joint}) und auch die Stichprobenentnahme daraus (zum Generieren der Trainingsdaten) einfach ist.

Die NCE ist dann am erfolgreichsten, wenn sie auf Probleme mit wenigen Zufallsvariablen angewendet wird, aber sie kann auch dann gut funktionieren, wenn diese Zufallsvariablen eine hohe Anzahl von Werten annehmen. Zum Beispiel wurde sie erfolgreich auf das Modellieren der bedingten Verteilung über ein Wort anhand des Wortkontexts angewandt (*Mnih und Kavukcuoglu, 2013*). Obwohl das Wort aus einem großen Wortschatz stammen kann, gibt es nur ein Wort.

Wenn die NCE für Aufgaben mit vielen Zufallsvariablen benutzt wird, ist sie weniger effizient. Der logistische Regressionsklassifikator kann ein verrauschttes Beispiel verwerfen, indem eine beliebige Variable identifiziert wird, deren Wert unwahrscheinlich ist. Damit geht das Lernen deutlich langsamer voran, sobald p_{model} die grundlegenden Randstatistiken erlernt hat. Stellen Sie sich vor, dass ein Modell von Bildern mit Gesichtern unter Einsatz unstrukturierten normalverteilten Rauschens als p_{noise} erlernt wird. Wenn p_{model} etwas über Augen lernt, können nahezu alle unstrukturierten verrauschten Beispiele verworfen werden, ohne dass es etwas über andere Gesichtsmerkmale wie Münder erlernt hat.

Die Bedingung, dass es einfach sein muss, p_{noise} zu berechnen und Stichproben daraus zu ziehen, kann sich als übermäßig restriktiv erweisen. Wenn p_{noise} einfach ist, unterscheiden sich die meisten Stichproben vermutlich zu offensichtlich von den Daten, als dass sie zu einer merklichen Verbesserung von p_{model} führen würden.

Wie Score Matching und Pseudo-Likelihood funktioniert die NCE nicht, wenn es für \tilde{p} nur eine untere Schranke gibt. Eine solche untere Schranke könnte zum Konstruieren einer unteren Schranke für $p_{\text{joint}}(y = 1 | \mathbf{x})$ verwendet werden, aber sie kann nur zum Konstruieren einer oberen Schranke

für $p_{\text{joint}}(y = 0 | \mathbf{x})$ verwendet werden, die in der Hälfte der Terme der NCE-Zielfunktion vorkommt. Ebenso ist eine untere Schranke für p_{noise} nicht nützlich, da sie nur eine obere Schranke für $p_{\text{joint}}(y = 1 | \mathbf{x})$ liefert.

Wenn die Modellverteilung vor jedem Gradientenschritt zum Definieren einer neuen Rauschverteilung kopiert wird, definiert die NCE ein Verfahren namens **Self-Contrastive Estimation** (selbst-kontrastive Schätzung), dessen erwarteter Gradient gleich dem erwarteten Gradienten der Maximum Likelihood ist (*Goodfellow*, 2014). Der Sonderfall der NCE mit vom Modell generierten verrauschten Beispielen deutet an, dass die Maximum Likelihood als Verfahren betrachtet werden kann, das ein Modell dazu zwingt, ständig zu lernen, zwischen der Realität und den eigenen wachsenden Überzeugungen zu entscheiden, wohingegen die NCE eine gewisse Reduzierung des Berechnungsaufwands erreicht, indem das Modell gezwungen wird, zwischen der Realität und einer unveränderlichen Baseline (dem Rauschmodell) zu unterscheiden.

Die Verwendung der überwachten Klassifizierung (engl. *supervised task of classifying*) zwischen Trainingsstichproben und generierten Stichproben (mit der Modell-Energiefunktion, die zum Definieren des Klassifikators verwendet wird) zum Angeben eines Gradienten auf dem Modell wurde schon früher in unterschiedlicher Form vorgestellt (*Welling et al.*, 2003b; *Bengio*, 2009).

Die NCE beruht auf der Idee, dass ein gutes generatives Modell in der Lage sein sollte, Daten und Rauschen auseinanderzuhalten. Eine eng damit verwandte Idee besagt, dass ein gutes generatives Modell in der Lage sein sollte, Stichproben zu generieren, die kein Klassifikator von Daten unterscheiden kann. Diese Idee ist der Ursprung der Generative-Adversarial-Netze (GANs, siehe Abschnitt 20.10.4).

18.7 Schätzen der Partitionsfunktion

Ein Großteil dieses Kapitels widmet sich dem Beschreiben von Verfahren, die das Berechnen der nicht effizient lösbarer Partitionsfunktion $Z(\boldsymbol{\theta})$ in Verbindung mit einem ungerichteten graphischen Modell umgehen. In diesem Abschnitt wenden wir uns allerdings diversen Verfahren zu, mit denen die Partitionsfunktion direkt geschätzt werden kann.

Das Schätzen der Partitionsfunktion kann sich als wichtig erweisen, da sie zum Berechnen der normalisierten Likelihood der Daten benötigt wird. Dies ist oft wichtig, wenn das Modell *evaluiert* wird, die Trainingsleistung überwacht oder Modelle miteinander verglichen werden.

Ein Beispiel: Stellen Sie sich zwei Modelle vor: Modell \mathcal{M}_A definiert eine Wahrscheinlichkeitsverteilung $p_A(\mathbf{x}; \boldsymbol{\theta}_A) = \frac{1}{Z_A} \tilde{p}_A(\mathbf{x}; \boldsymbol{\theta}_A)$ und Modell \mathcal{M}_B definiert eine Wahrscheinlichkeitsverteilung $p_B(\mathbf{x}; \boldsymbol{\theta}_B) = \frac{1}{Z_B} \tilde{p}_B(\mathbf{x}; \boldsymbol{\theta}_B)$. Ein gängiger Weg zum Vergleichen der Modelle besteht darin, die Likelihood, mit der beide Modelle die Zuordnung zu einer u.i.v. Testdatenmenge vornehmen, zu bestimmen und zu vergleichen. Die Testdatenmenge bestehe aus m Beispielen $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$. Ist $\prod_i p_A(\mathbf{x}^{(i)}; \boldsymbol{\theta}_A) > \prod_i p_B(\mathbf{x}^{(i)}; \boldsymbol{\theta}_B)$ oder (gleichwertig)

$$\sum_i \log p_A(\mathbf{x}^{(i)}; \boldsymbol{\theta}_A) - \sum_i \log p_B(\mathbf{x}^{(i)}; \boldsymbol{\theta}_B) > 0, \quad (18.38)$$

sagen wir, dass \mathcal{M}_A ein besseres Modell als \mathcal{M}_B ist (oder zumindest ein besseres Modell der Testdatenmenge), und zwar in dem Sinne, dass die Log-Likelihood zum Testen besser ist. Leider benötigen wir zur Prüfung, ob diese Bedingung gilt, Kenntnis über die Partitionsfunktion. Gleichung 18.38 scheint das Bestimmen der Log-Wahrscheinlichkeit, die das Modell jedem Punkt zuweist, vorauszusetzen, was wiederum die Auswertung der Partitionsfunktion voraussetzt. Wir können die Situation ein wenig vereinfachen, indem wir Gleichung 18.38 so umstellen, dass wir nur das **Verhältnis** der Partitionsfunktionen der beiden Modelle kennen müssen:

$$\begin{aligned} \sum_i \log p_A(\mathbf{x}^{(i)}; \boldsymbol{\theta}_A) - \sum_i \log p_B(\mathbf{x}^{(i)}; \boldsymbol{\theta}_B) &= \\ \sum_i \left(\log \frac{\tilde{p}_A(\mathbf{x}^{(i)}; \boldsymbol{\theta}_A)}{\tilde{p}_B(\mathbf{x}^{(i)}; \boldsymbol{\theta}_B)} \right) - m \log \frac{Z(\boldsymbol{\theta}_A)}{Z(\boldsymbol{\theta}_B)}. \end{aligned} \quad (18.39)$$

Wir können nun bestimmen, ob \mathcal{M}_A ein besseres Modell als \mathcal{M}_B ist, ohne dass wir die Partitionsfunktion eines der Modelle kennen – es reicht ihr Verhältnis. Wie wir gleich noch zeigen werden, können wir dieses Verhältnis mittels Importance Sampling schätzen, sofern die beiden Modelle ähnlich sind.

Wenn wir jedoch die tatsächliche Wahrscheinlichkeit der Testdaten unter \mathcal{M}_A oder \mathcal{M}_B berechnen möchten, müssen wir den tatsächlichen Wert der Partitionsfunktionen berechnen. Unter dieser Prämisse könnten wir, wenn wir das Verhältnis der beiden Partitionsfunktionen, $r = \frac{Z(\boldsymbol{\theta}_B)}{Z(\boldsymbol{\theta}_A)}$, und den tatsächlichen Wert einer der beiden, zum Beispiel $Z(\boldsymbol{\theta}_A)$ kennen, den Wert der anderen bestimmen:

$$Z(\boldsymbol{\theta}_B) = r Z(\boldsymbol{\theta}_A) = \frac{Z(\boldsymbol{\theta}_B)}{Z(\boldsymbol{\theta}_A)} Z(\boldsymbol{\theta}_A). \quad (18.40)$$

Eine einfache Möglichkeit zum Schätzen der Partitionsfunktion besteht im Einsatz eines Monte-Carlo-Verfahrens wie dem Importance Sampling. Wir zeigen den Ansatz mit stetigen Variablen und Integralen, aber er lässt sich auch für diskrete Variablen verwenden. Dazu werden die Integrale einfach durch Summenbildung ersetzt. Wir verwenden eine Proposal-Verteilung $p_0(\mathbf{x}) = \frac{1}{Z_0} \tilde{p}_0(\mathbf{x})$, die eine effizient durchführbare Stichprobenentnahme und Berechnung der Partitionsfunktion Z_0 und der nicht normalisierten Verteilung $\tilde{p}_0(\mathbf{x})$ unterstützt.

$$Z_1 = \int \tilde{p}_1(\mathbf{x}) d\mathbf{x} \quad (18.41)$$

$$= \int \frac{p_0(\mathbf{x})}{\tilde{p}_0(\mathbf{x})} \tilde{p}_1(\mathbf{x}) d\mathbf{x} \quad (18.42)$$

$$= Z_0 \int p_0(\mathbf{x}) \frac{\tilde{p}_1(\mathbf{x})}{\tilde{p}_0(\mathbf{x})} d\mathbf{x} \quad (18.43)$$

$$\hat{Z}_1 = \frac{Z_0}{K} \sum_{k=1}^K \frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} \quad \text{s.t. : } \mathbf{x}^{(k)} \sim p_0 \quad (18.44)$$

In der letzten Zeile erstellen wir einen Monte-Carlo-Schätzer, \hat{Z}_1 , des Integrals mit aus $p_0(\mathbf{x})$ gezogenen Stichproben und gewichten dann jede Stichprobe mit dem Verhältnis der nicht normalisierten \tilde{p}_1 und der Proposal-Verteilung p_0 .

Dieser Ansatz ermöglicht uns außerdem die Schätzung des Verhältnisses zwischen den Partitionsfunktionen als

$$\frac{1}{K} \sum_{k=1}^K \frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} \quad \text{s.t. : } \mathbf{x}^{(k)} \sim p_0. \quad (18.45)$$

Dieser Wert kann dann direkt zum Vergleichen der beiden Modelle genutzt werden (siehe Gleichung 18.39).

Wenn die Verteilung p_0 nahe bei p_1 liegt, kann Gleichung 18.44 eine effektive Möglichkeit zum Schätzen der Partitionsfunktion darstellen (*Minka, 2005*). Leider ist p_1 meist kompliziert (häufig multimodal) und über einem hochdimensionalen Raum definiert. Es ist schwierig, eine effizient berechenbare Verteilung p_0 zu finden, die einfach genug ist, aber gleichzeitig nahe genug an p_1 liegt, um eine Approximation von hoher Güte zu ermöglichen. Liegen p_0 und p_1 nicht nahe beieinander, haben die meisten Stichproben aus p_0 unter p_1 eine geringe Wahrscheinlichkeit und tragen somit (relativ) wenig zur Summe in Gleichung 18.44 bei.

Wenige Stichproben mit signifikanten Gewichten in dieser Summe führen aufgrund großer Varianz zu einem minderwertigen Schätzer. Das lässt sich

quantitativ durch eine Schätzung der Varianz für unseren Schätzwert \hat{Z}_1 zeigen:

$$\hat{\text{Var}}(\hat{Z}_1) = \frac{Z_0}{K^2} \sum_{k=1}^K \left(\frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})} - \hat{Z}_1 \right)^2. \quad (18.46)$$

Diese Quantität ist dann am größten, wenn es eine beträchtliche Abweichung in den Werten der Gewichte nach Wichtigkeit $\frac{\tilde{p}_1(\mathbf{x}^{(k)})}{\tilde{p}_0(\mathbf{x}^{(k)})}$ gibt.

Wir wenden uns nun zwei verwandten Verfahren zu, die für die herausfordernde Aufgabe der Schätzung der Partitionsfunktionen für komplexe Verteilungen über hochdimensionale Räume entwickelt wurden: Annealed Importance Sampling und Bridge Sampling. Beiden liegt die einfache Methode des Importance Samplings zugrunde, die wir oben vorgestellt haben. Und beide versuchen, das Problem einer Proposal-Verteilung p_0 , die zu weit von p_1 entfernt ist, zu umgehen, indem sie Zwischenverteilungen einführen, die versuchen, die Lücke zwischen p_0 und p_1 zu schließen.

18.7.1 Annealed Importance Sampling

Wenn $D_{\text{KL}}(p_0 \| p_1)$ groß ist (es also kaum eine Überschneidung zwischen p_0 und p_1 gibt), wird eine Methode namens **Annealed Importance Sampling** (AIS, dt. *abgekühlte Stichprobenentnahme nach Wichtigkeit*) verwendet, um die Lücke durch Einführung von Zwischenverteilungen zu schließen (Jarzynski, 1997; Neal, 2001). Stellen Sie sich eine Sequenz von Verteilungen $p_{\eta_0}, \dots, p_{\eta_n}$ vor, mit $0 = \eta_0 < \eta_1 < \dots < \eta_{n-1} < \eta_n = 1$, sodass die erste und die letzte Verteilung in der Sequenz p_0 bzw. p_1 sind.

Dieser Ansatz ermöglicht das Schätzen der Partitionsfunktion einer multimodalen Verteilung, die über einen hochdimensionalen Raum definiert ist (zum Beispiel der Verteilung, die von einer trainierten RBM definiert wird). Wir beginnen mit einem einfacheren Modell mit bekannter Partitionsfunktion (zum Beispiel einer RBM mit Nullen als Gewichten) und schätzen das Verhältnis zwischen den Partitionsfunktionen der beiden Modelle. Der Schätzwert dieses Verhältnisses beruht auf der Schätzung der Verhältnisse einer Sequenz vieler ähnlicher Verteilungen, beispielsweise der Sequenz von RBMs mit Gewichten, die zwischen Null und den erlernten Gewichten interpoliert werden.

Wir können das Verhältnis $\frac{Z_1}{Z_0}$ nun wie folgt schreiben:

$$\frac{Z_1}{Z_0} = \frac{Z_1}{Z_0} \frac{Z_{\eta_1}}{Z_{\eta_1}} \dots \frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-1}}} \quad (18.47)$$

$$= \frac{Z_{\eta_1}}{Z_0} \frac{Z_{\eta_2}}{Z_{\eta_1}} \dots \frac{Z_{\eta_{n-1}}}{Z_{\eta_{n-2}}} \frac{Z_1}{Z_{\eta_{n-1}}} \quad (18.48)$$

$$= \prod_{j=0}^{n-1} \frac{Z_{\eta_{j+1}}}{Z_{\eta_j}}. \quad (18.49)$$

Sofern die Verteilungen p_{η_j} und $p_{\eta_{j+1}}$ einander für alle $0 \leq j \leq n - 1$ hinreichend nah sind, können wir jeden der Faktoren $\frac{Z_{\eta_{j+1}}}{Z_{\eta_j}}$ zuverlässig mittels Importance Sampling schätzen und sie anschließend verwenden, um einen Schätzwert für $\frac{Z_1}{Z_0}$ zu ermitteln.

Woher stammen diese Zwischenverteilungen? Wie die ursprüngliche Proposal-Verteilung p_0 so ist auch die Sequenz der Verteilungen $p_{\eta_1} \dots p_{\eta_{n-1}}$ eine bewusste Designentscheidung. Sie kann also gezielt passend für einen bestimmten Aufgabenbereich konstruiert werden. Eine für viele Fälle passende und beliebte Wahl für die Zwischenverteilungen ist das gewichtete geometrische Mittel der Zielverteilung p_1 und die anfängliche Proposal-Verteilung (deren Partitionsfunktion bekannt ist) p_0 zu verwenden:

$$p_{\eta_j} \propto p_1^{\eta_j} p_0^{1-\eta_j}. \quad (18.50)$$

Um Stichproben aus diesen Zwischenverteilungen zu ziehen, definieren wir eine Reihe von Markow-Ketten-Übergangsfunktionen $T_{\eta_j}(\mathbf{x}' | \mathbf{x})$, mit denen die bedingte Wahrscheinlichkeitsverteilung für den Übergang zu \mathbf{x}' von \mathbf{x} definiert wird. Der Übergangsoperator $T_{\eta_j}(\mathbf{x}' | \mathbf{x})$ ist so definiert, dass $p_{\eta_j}(\mathbf{x})$ invariant bleibt:

$$p_{\eta_j}(\mathbf{x}) = \int p_{\eta_j}(\mathbf{x}') T_{\eta_j}(\mathbf{x} | \mathbf{x}') d\mathbf{x}'. \quad (18.51)$$

Diese Übergänge können als beliebige Markow-Ketten-Monte-Carlo-Verfahren konstruiert werden (z. B. Metropolis-Hastings, Gibbs), einschließlich von Verfahren mit mehreren Durchläufen durch alle Zufallsvariablen oder anderen Arten von Iterationen.

Das Verfahren des Annealed Importance Samplings besteht dann darin, Stichproben aus p_0 zu generieren und die Übergangsoperatoren zu verwenden, um sequenziell Stichproben aus den Zwischenverteilungen zu generieren, bis wir zu den Stichproben aus der Zielverteilung p_1 gelangen:

- for $k = 1 \dots K$
 - Stichprobe $\mathbf{x}_{\eta_1}^{(k)} \sim p_0(\mathbf{x})$
 - Stichprobe $\mathbf{x}_{\eta_2}^{(k)} \sim T_{\eta_1}(\mathbf{x}_{\eta_2}^{(k)} | \mathbf{x}_{\eta_1}^{(k)})$

- ...
- Stichprobe $\mathbf{x}_{\eta_{n-1}}^{(k)} \sim T_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-1}}^{(k)} | \mathbf{x}_{\eta_{n-2}}^{(k)})$
- Stichprobe $\mathbf{x}_{\eta_n}^{(k)} \sim T_{\eta_{n-1}}(\mathbf{x}_{\eta_n}^{(k)} | \mathbf{x}_{\eta_{n-1}}^{(k)})$
- end

Für Stichprobe k können wir die Gewichte nach Wichtigkeit ableiten, indem wir die Gewichte nach Wichtigkeit für die Sprünge zwischen den Zwischenverteilungen aus Gleichung 18.49 verketten:

$$w^{(k)} = \frac{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\mathbf{x}_{\eta_1}^{(k)})} \frac{\tilde{p}_{\eta_2}(\mathbf{x}_{\eta_2}^{(k)})}{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_2}^{(k)})} \cdots \frac{\tilde{p}_1(\mathbf{x}_1^{(k)})}{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_{\eta_n}^{(k)})}. \quad (18.52)$$

Um rechnerische Probleme wie einen Überlauf zu verhindern, ist es vermutlich am besten, $\log w^{(k)}$ durch Addieren und Subtrahieren der Log-Wahrscheinlichkeiten zu berechnen und auf die Berechnung von $w^{(k)}$ mittels Multiplikation und Division von Wahrscheinlichkeiten zu verzichten.

Mit dem so definierten Verfahren zur Stichprobenentnahme und den Gewichten nach Wichtigkeit aus Gleichung 18.52 ergibt sich der Schätzwert für das Verhältnis der Partitionsfunktionen aus:

$$\frac{Z_1}{Z_0} \approx \frac{1}{K} \sum_{k=1}^K w^{(k)}. \quad (18.53)$$

Um zu verifizieren, dass dieses Vorgehen ein gültiges Importance-Sampling-System definiert, können wir zeigen (Neal, 2001), dass das AIS-Verfahren einem einfachen Importance Sampling für einen erweiterten Zustandsraum entspricht, wobei die Datenpunkte als Stichproben aus dem Produktraum $[\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1]$ gezogen werden. Dazu definieren wir die Verteilung über den erweiterten Raum als

$$\tilde{p}(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) \quad (18.54)$$

$$= \tilde{p}_1(\mathbf{x}_1) \tilde{T}_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}} | \mathbf{x}_1) \tilde{T}_{\eta_{n-2}}(\mathbf{x}_{\eta_{n-2}} | \mathbf{x}_{\eta_{n-1}}) \dots \tilde{T}_{\eta_1}(\mathbf{x}_{\eta_1} | \mathbf{x}_{\eta_2}), \quad (18.55)$$

wobei \tilde{T}_a die Umkehrung des durch T_a definierten Übergangsoptators ist (mittels Anwendung des Satzes von Bayes):

$$\tilde{T}_a(\mathbf{x}' | \mathbf{x}) = \frac{p_a(\mathbf{x}')}{p_a(\mathbf{x})} T_a(\mathbf{x} | \mathbf{x}') = \frac{\tilde{p}_a(\mathbf{x}')}{\tilde{p}_a(\mathbf{x})} T_a(\mathbf{x} | \mathbf{x}'). \quad (18.56)$$

Wenn wir dies in den Ausdruck für die multivariate Verteilung für den erweiterten Zustandsraum aus Gleichung 18.55 einsetzen, erhalten wir:

$$\tilde{p}(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) \quad (18.57)$$

$$= \tilde{p}_1(\mathbf{x}_1) \frac{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}})}{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_1)} T_{\eta_{n-1}}(\mathbf{x}_1 | \mathbf{x}_{\eta_{n-1}}) \prod_{i=1}^{n-2} \frac{\tilde{p}_{\eta_i}(\mathbf{x}_{\eta_i})}{\tilde{p}_{\eta_i}(\mathbf{x}_{\eta_{i+1}})} T_{\eta_i}(\mathbf{x}_{\eta_{i+1}} | \mathbf{x}_{\eta_i}) \quad (18.58)$$

$$= \frac{\tilde{p}_1(\mathbf{x}_1)}{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_1)} T_{\eta_{n-1}}(\mathbf{x}_1 | \mathbf{x}_{\eta_{n-1}}) \tilde{p}_{\eta_1}(\mathbf{x}_{\eta_1}) \prod_{i=1}^{n-2} \frac{\tilde{p}_{\eta_{i+1}}(\mathbf{x}_{\eta_{i+1}})}{\tilde{p}_{\eta_i}(\mathbf{x}_{\eta_{i+1}})} T_{\eta_i}(\mathbf{x}_{\eta_{i+1}} | \mathbf{x}_{\eta_i}). \quad (18.59)$$

Wir verfügen nun über Mittel, um Stichproben aus der multivariaten Proposal-Verteilung q über die erweiterte Stichprobe zu generieren mittels des oben angegebenen Systems zur Stichprobenentnahme, mit der multivariaten Verteilung aus

$$q(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1) = p_0(\mathbf{x}_{\eta_1}) T_{\eta_1}(\mathbf{x}_{\eta_2} | \mathbf{x}_{\eta_1}) \dots T_{\eta_{n-1}}(\mathbf{x}_1 | \mathbf{x}_{\eta_{n-1}}). \quad (18.60)$$

Wir haben eine multivariate Verteilung für den erweiterten Raum, der sich aus Gleichung 18.59 ergibt. Mit $q(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)$ als Proposal-Verteilung für den erweiterten Zustandsraum, aus dem wir die Stichproben ziehen, verbleibt noch das Bestimmen der Gewichten nach Wichtigkeit:

$$w^{(k)} = \frac{\tilde{p}(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)}{q(\mathbf{x}_{\eta_1}, \dots, \mathbf{x}_{\eta_{n-1}}, \mathbf{x}_1)} = \frac{\tilde{p}_1(\mathbf{x}_1^{(k)})}{\tilde{p}_{\eta_{n-1}}(\mathbf{x}_{\eta_{n-1}}^{(k)})} \dots \frac{\tilde{p}_{\eta_2}(\mathbf{x}_{\eta_2}^{(k)})}{\tilde{p}_1(\mathbf{x}_{\eta_1}^{(k)})} \frac{\tilde{p}_{\eta_1}(\mathbf{x}_{\eta_1}^{(k)})}{\tilde{p}_0(\mathbf{x}_0^{(k)})}. \quad (18.61)$$

Diese Gewichte stimmen mit den für das AIS vorgeschlagenen überein. Wir können das AIS somit als einfaches Importance Sampling für einen erweiterten Zustand betrachten, dessen Gültigkeit sich direkt aus der Gültigkeit des Importance Samplings ergibt.

Das Annealed Importance Sampling (AIS) wurde zuerst von *Jarzynski* (1997) und später nochmals – unabhängig – von *Neal* (2001) entdeckt. Es ist die derzeit gängigste Art zur Schätzung der Partitionsfunktion für ungerichtete probabilistische Modelle. Die Gründe hierfür haben mehr mit der Veröffentlichung einer einflussreichen Abhandlung zu tun (*Salakhutdinov und Murray*, 2008), in der seine Anwendung zum Abschätzen der Partitionsfunktion für RBMs und DBNs beschrieben wird, als mit einem innewohnenden Vorteil des Verfahrens gegenüber dem unten beschriebenen Verfahren.

Eine Abhandlung über die Eigenschaften des AIS-Schätzers (hinsichtlich seiner Varianz und Effizienz) bietet *Neal* (2001).

18.7.2 Bridge Sampling

Bridge Sampling (Bennett, 1976) ist ein weiteres Verfahren, das – wie AIS – auf die Defizite des Importance Samplings eingeht. Statt eine Reihe von Zwischenverteilungen zu verketten, baut das Bridge Sampling auf einer einzelnen Verteilung p_* auf, der sogenannten Bridge (dt. *Brücke*). Diese wird zum Interpolieren zwischen einer Verteilung mit bekannter Partitionsfunktion, p_0 , und einer Verteilung p_1 , deren Partitionsfunktion Z_1 geschätzt werden soll, verwendet.

Bridge Sampling schätzt das Verhältnis Z_1/Z_0 als Verhältnis der erwarteten Gewichte nach Wichtigkeit zwischen \tilde{p}_0 und \tilde{p}_* sowie zwischen \tilde{p}_1 und \tilde{p}_* :

$$\frac{Z_1}{Z_0} \approx \left(\sum_{k=1}^K \frac{\tilde{p}_*(\mathbf{x}_0^{(k)})}{\tilde{p}_0(\mathbf{x}_0^{(k)})} \right) \Bigg/ \left(\sum_{k=1}^K \frac{\tilde{p}_*(\mathbf{x}_1^{(k)})}{\tilde{p}_1(\mathbf{x}_1^{(k)})} \right). \quad (18.62)$$

Wenn die Bridge-Verteilung p_* sorgfältig ausgewählt wird und eine große Überlappung zur Unterstützung der Funktionen p_0 und p_1 aufweist, darf beim Bridge Sampling der Abstand zwischen den beiden Verteilungen (formal ausgedrückt: $D_{KL}(p_0\|p_1)$) viel größer als beim normalen Importance Sampling sein.

Es lässt sich zeigen, dass die optimale Bridge-Verteilung sich aus $p_*^{(opt)}(\mathbf{x}) \propto \frac{\tilde{p}_0(\mathbf{x})\tilde{p}_1(\mathbf{x})}{r\tilde{p}_0(\mathbf{x})+\tilde{p}_1(\mathbf{x})}$ ergibt, wobei $r = Z_1/Z_0$ ist. Zunächst erscheint diese Lösung undurchführbar, da anscheinend genau die Größe benötigt wird, die wir schätzen möchten: Z_1/Z_0 . Allerdings können wir mit einem groben Schätzwert für r beginnen und die resultierende Bridge-Verteilung verwenden, um unseren Schätzwert iterativ zu verfeinern (Neal, 2005). Wir führen also iterative Neuabschätzungen des Verhältnisses durch und verwenden jede Iteration, um den Wert von r zu aktualisieren.

Linked Importance Sampling Sowohl AIS als auch Bridge Sampling haben ihre Vorteile. Wenn $D_{KL}(p_0\|p_1)$ nicht zu groß ist (da p_0 und p_1 nahe genug beieinander liegen), kann das Bridge Sampling das Verhältnis der Partitionsfunktionen effektiver schätzen als AIS. Wenn die beiden Verteilungen jedoch zu weit auseinanderliegen und eine einzelne Verteilung p_* nicht zum Schließen der Lücke ausreicht, lässt sich der Abstand zwischen p_0 und p_1 zumindest mittels AIS mit eventuell vielen Zwischenverteilungen überbrücken. Neal (2005) hat gezeigt, wie dieses Linked Importance Sampling (verknüpfte Stichprobenentnahme nach Wichtigkeit) die Vorteile des Bridge Samplings zum Überbrücken der Zwischenverteilungen aus dem AIS nutzt und so die Gesamtschätzung der Partitionsfunktion deutlich verbessert.

Schätzen der Partitionsfunktion während des Trainings Obwohl AIS mittlerweile als Standardverfahren zur Schätzung der Partitionsfunktion für viele ungerichtete Modelle akzeptiert ist, ist es doch rechnerisch so aufwendig, dass es während des Trainings nicht genutzt werden kann. Alternative Verfahren wurden untersucht, um einen Schätzwert der Partitionsfunktion während des Trainings zu aktualisieren.

Mit einer Kombination aus Bridge Sampling, kurzkettigem AIS und Parallel Tempering haben *Desjardins et al.* (2011) ein System entwickelt, das die Partitionsfunktion einer RBM während des Trainingsverfahrens verfolgen kann. Das Verfahren beruht darauf, unabhängige Schätzwerte der Partitionsfunktionen der RBM bei jeder Temperatur im Rahmen des Parallel Temperings aufrechtzuerhalten. Die Autoren haben Bridge-Sampling-Schätzwerte der Verhältnisse von Partitionsfunktionen benachbarter Ketten (d. h. aus Parallel Tempering) mit den AIS-Schätzwerten über die Zeit kombiniert, um einen Schätzwert mit geringer Varianz für die Partitionsfunktionen in jeder Iteration des Lernens zu bestimmen.

Die in diesem Kapitel beschriebenen Tools bieten viele unterschiedliche Möglichkeiten zum Überwinden des Problems der nicht effizient berechenbaren Partitionsfunktionen, aber es kann noch diverse andere Schwierigkeiten beim Trainieren und Verwenden generativer Modelle geben. Das wohl größte davon ist die nicht effizient durchführbare Inferenz, der wir uns im nächsten Kapitel widmen.

19

Approximative Inferenz

Viele probabilistische Modelle sind schwierig zu trainieren, da das Durchführen von Inferenz darin schwierig ist. Im Deep-Learning-Kontext haben wir es normalerweise mit einer Menge sichtbarer Variablen v und einer Menge latenter Variablen h zu tun. Die Herausforderung der Inferenz bezieht sich für gewöhnlich auf das Problem der Berechnung von $p(h | v)$ oder das Ermitteln von Erwartungswerten hierfür. Solche Operationen sind häufig für Aufgaben wie das Maximum Likelihood Learning erforderlich.

Viele einfache graphische Modelle mit nur einer verdeckten Schicht – wie RBMs und die probabilistische Hauptkomponentenanalyse (engl. *principal components analysis*, PCA) – sind auf eine Weise definiert, die Inferenzoperationen wie das Berechnen von $p(h | v)$ oder das Ermitteln von Erwartungswerten hierzu einfach macht. Leider weisen die meisten graphischen Modelle mit mehreren Schichten verdeckter Variablen nicht effizient berechenbare A-posteriori-Verteilungen auf. Für eine exakte Inferenz müsste in diesen Modellen eine exponentiell viel Zeit aufgewendet werden. Selbst einige Modelle mit nur einer Schicht, zum Beispiel Sparse Coding, weisen dieses Problem auf.

In diesem Kapitel stellen wir mehrere Methoden für diese nicht effizient lösbarer Inferenzprobleme vor. In Kapitel 20 beschreiben wir dann, wie diese Methoden zum Trainieren probabilistischer Modelle verwendet werden können, die sonst nicht effizient lösbar wären, darunter Deep-Belief-Netze (DBNs) und Deep Boltzmann Machines (DBMs).

Nicht effizient lösbarer Inferenzprobleme im Deep Learning entstehen meist aus Interaktionen zwischen latenten Variablen in einem strukturierten graphischen Modell. Abbildung 19.1 zeigt hierfür einige Beispiele. Diese Interaktionen können zurückzuführen sein auf direkte Interaktionen in un-

gerichteten Modellen oder auf »Explaining-Away«-Interaktionen zwischen gemeinsamen Vorgängern derselben sichtbaren Einheit in gerichteten Modellen.

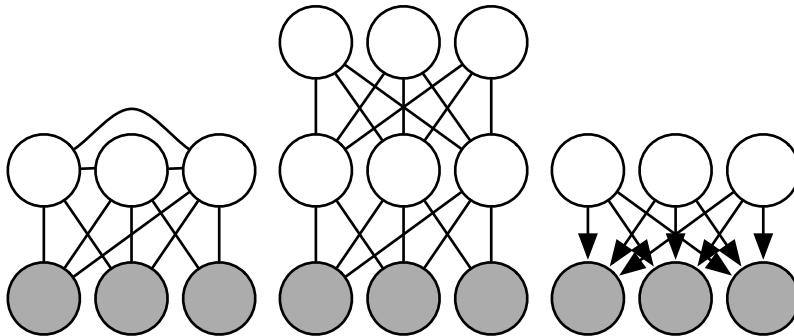


Abbildung 19.1: Nicht effizient lösbare Inferenzprobleme im Deep Learning sind meist das Ergebnis von Interaktionen zwischen latenten Variablen in einem strukturierten graphischen Modell. Diese Interaktionen können zurückzuführen sein auf Kanten, die eine latente Variable direkt mit einer anderen verbinden, oder auf längere Pfade, die aktiviert werden, wenn der Kindknoten einer V-Struktur beobachtet wird. (*Links*) Eine **semi-restricted Boltzmann Machine** (halbeingeschränkte Boltzmann-Maschine) (*Osindero und Hinton, 2008*) mit Verbindungen zwischen verdeckten Einheiten. Diese direkten Verbindungen zwischen latenten Variablen führen dazu, dass die A-posteriori-Verteilung nicht effizient berechenbar ist; Grund dafür sind große Cliques latenter Variablen. (*Mitte*) Eine DBM mit zwei Variablenschichten ohne Verbindungen in den Schichten selbst. Sie weist dennoch eine nicht effizient berechenbare A-posteriori-Verteilung auf, und zwar aufgrund der Verbindungen zwischen den Schichten. (*Rechts*) Dieses gerichtete Modell weist Interaktionen zwischen latenten Variablen auf, wenn die sichtbaren Variablen beobachtet werden, da alle zwei latenten Variablen gemeinsam übergeordnet sind. Einige probabilistische Modelle ermöglichen eine effizient durchführbare Inferenz über die latenten Variablen, obwohl sie eine der hier dargestellten Graphenstrukturen aufweisen. Das ist möglich, wenn die bedingten Wahrscheinlichkeitsverteilungen so gewählt werden, dass über die vom Graphen beschriebenen Unabhängigkeiten hinaus zusätzliche Unabhängigkeiten eingeführt werden. Zum Beispiel hat die probabilistische PCA trotz der rechts dargestellten Graphenstruktur eine einfache Inferenz; der Grund dafür sind die besonderen Eigenschaften, die die spezifischen bedingten Verteilungen nutzen (linear bedingte Normalverteilungen mit paarweise orthogonalen Basisvektoren).

19.1 Inferenz als Optimierung

Viele Ansätze, die sich mit dem Problem der schwierigen Inferenz beschäftigen, nutzen die Erkenntnis, dass die exakte Inferenz als Optimierungsproblem beschrieben werden kann. Algorithmen zur approximativen Inferenz können dann durch Approximation des zugrunde liegenden Optimierungsproblems hergeleitet werden.

Um das Optimierungsproblem zu konstruieren, gehen wir von einem probabilistischen Modell mit beobachteten Variablen \mathbf{v} und latenten Variablen \mathbf{h} aus. Wir möchten die Log-Wahrscheinlichkeit der beobachteten Daten berechnen, $\log p(\mathbf{v}; \boldsymbol{\theta})$. Manchmal ist die Berechnung von $\log p(\mathbf{v}; \boldsymbol{\theta})$ zu schwierig, wenn das Entfernen von \mathbf{h} aufwendig ist. Stattdessen können wir eine untere Schranke $\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, q)$ für $\log p(\mathbf{v}; \boldsymbol{\theta})$ berechnen. Diese Schranke wird als **ELBO** (Evidence Lower Bound) bezeichnet. Eine weitere gängige Bezeichnung für diese untere Schranke ist auch negative **Variational Free Energy** oder **Variational Lower Bound**. Die ELBO wird definiert als

$$\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, q) = \log p(\mathbf{v}; \boldsymbol{\theta}) - D_{\text{KL}}(q(\mathbf{h} | \mathbf{v}) \| p(\mathbf{h} | \mathbf{v}; \boldsymbol{\theta})), \quad (19.1)$$

wobei q eine beliebige Wahrscheinlichkeitsverteilung über \mathbf{h} ist.

Da sich die Differenz zwischen $\log p(\mathbf{v})$ und $\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, q)$ aus der Kullback-Leibler-Divergenz (KL-Divergenz) ergibt und da die KL-Divergenz stets nicht-negativ ist, kann \mathcal{L} höchstens denselben Wert wie die gesuchte Log-Wahrscheinlichkeit annehmen. Die beiden sind genau dann gleich, wenn q dieselbe Verteilung wie $p(\mathbf{h} | \mathbf{v})$ ist.

Überraschenderweise lässt sich \mathcal{L} für einige Verteilungen q deutlich leichter berechnen. Einfache Algebra reicht aus, um zu zeigen, dass wir \mathcal{L} auf sehr viel einfachere Weise umstellen können:

$$\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, q) = \log p(\mathbf{v}; \boldsymbol{\theta}) - D_{\text{KL}}(q(\mathbf{h} | \mathbf{v}) \| p(\mathbf{h} | \mathbf{v}; \boldsymbol{\theta})) \quad (19.2)$$

$$= \log p(\mathbf{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \log \frac{q(\mathbf{h} | \mathbf{v})}{p(\mathbf{h} | \mathbf{v})} \quad (19.3)$$

$$= \log p(\mathbf{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} \log \frac{q(\mathbf{h} | \mathbf{v})}{\frac{p(\mathbf{h}, \mathbf{v}; \boldsymbol{\theta})}{p(\mathbf{v}; \boldsymbol{\theta})}} \quad (19.4)$$

$$= \log p(\mathbf{v}; \boldsymbol{\theta}) - \mathbb{E}_{\mathbf{h} \sim q} [\log q(\mathbf{h} | \mathbf{v}) - \log p(\mathbf{h}, \mathbf{v}; \boldsymbol{\theta}) + \log p(\mathbf{v}; \boldsymbol{\theta})] \quad (19.5)$$

$$= -\mathbb{E}_{\mathbf{h} \sim q} [\log q(\mathbf{h} | \mathbf{v}) - \log p(\mathbf{h}, \mathbf{v}; \boldsymbol{\theta})]. \quad (19.6)$$

Das führt zur kanonischeren Definition der ELBO, nämlich

$$\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, q) = \mathbb{E}_{\mathbf{h} \sim q} [\log p(\mathbf{h}, \mathbf{v})] + H(q). \quad (19.7)$$

Für eine angemessene Wahl von q lässt sich \mathcal{L} effizient berechnen. Für jede Wahl von q bietet \mathcal{L} eine untere Schranke der Likelihood. Für Verteilungen $q(\mathbf{h} \mid \mathbf{v})$, die eine bessere Approximation von $p(\mathbf{h} \mid \mathbf{v})$ sind, ist die untere Schranke \mathcal{L} enger, liegt also näher an $\log p(\mathbf{v})$. Für $q(\mathbf{h} \mid \mathbf{v}) = p(\mathbf{h} \mid \mathbf{v})$ ist die Approximation perfekt und es gilt $\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, q) = \log p(\mathbf{v}; \boldsymbol{\theta})$.

Wir können uns die Inferenz als Verfahren vorstellen, bei dem das q gesucht wird, mit dem \mathcal{L} maximiert werden kann. Die exakte Inferenz maximiert \mathcal{L} perfekt durch die Suche über eine Funktionsfamilie q , die $p(\mathbf{h} \mid \mathbf{v})$ einschließt. In diesem Kapitel zeigen wir, wie unterschiedliche Formen der approximativen Inferenz durch approximative Optimierung bei der Suche nach q hergeleitet werden können. Das Optimierungsverfahren ist weniger rechenaufwändig aber dennoch approximativ, wenn wir die Familie der Verteilungen q , in denen wir die Optimierung suchen, beschränken oder wenn wir ein unvollkommenes Optimierungsverfahren einsetzen, in dem \mathcal{L} vielleicht nicht vollständig maximiert, sondern nur erheblich erhöht wird.

Wie unsere Wahl für q auch aussieht, \mathcal{L} ist immer eine untere Schranke. Wir können engere oder weniger enge Schranken erhalten, die weniger oder mehr Rechenaufwand erfordern – je nach Herangehensweise an dieses Optimierungsproblem. Wir können ein schlecht passendes q ermitteln und den Berechnungsaufwand reduzieren, indem wir ein unvollkommenes Optimierungsverfahren einsetzen. Oder wir können ein perfektes Optimierungsverfahren auf eine eingeschränkte Familie der q -Verteilungen anwenden.

19.2 Erwartungsmaximierung

Der erste von uns vorgestellte Algorithmus, der auf dem Maximieren einer unteren Schranke \mathcal{L} basiert, ist der **Expectation-Maximization-Algorithmus** (EM-Algorithmus, dt. *Erwartungsmaximierungsalgorithmus*), der vor allem zum Trainieren von Modellen mit latenten Variablen eingesetzt wird. Wir beschreiben hier den von *Neal und Hinton* (1999) entwickelten EM-Algorithmus. Anders als die meisten anderen in diesem Kapitel gezeigten Algorithmen handelt es sich bei der Erwartungsmaximierung nicht um einen Ansatz zur approximativen Inferenz, sondern um einen Ansatz zum Lernen mittels approximativer A-posteriori-Wahrscheinlichkeit.

Der EM-Algorithmus wechselt bis zur Konvergenz zwischen zwei Schritten hin und her:

- Der **E-Schritt** (Erwartungsschritt): $\boldsymbol{\theta}^{(0)}$ sei der Wert der Parameter zu Beginn des Schritts. Sei $q(\mathbf{h}^{(i)} \mid \mathbf{v}) = p(\mathbf{h}^{(i)} \mid \mathbf{v}^{(i)}; \boldsymbol{\theta}^{(0)})$ für alle

Indizes i der Trainingsbeispiele $\mathbf{v}^{(i)}$, die wir für das Training verwenden möchten (sowohl Batch- als auch Mini-Batch-Varianten sind gültig). Dadurch geben wir an, dass q anhand des *aktuellen* Parameterwerts für $\boldsymbol{\theta}^{(0)}$ definiert wird; wenn wir $\boldsymbol{\theta}$ verändern, ändert sich $p(\mathbf{h} \mid \mathbf{v}; \boldsymbol{\theta})$, aber $q(\mathbf{h} \mid \mathbf{v})$ bleibt gleich $p(\mathbf{h} \mid \mathbf{v}; \boldsymbol{\theta}^{(0)})$.

- Der **M-Schritt** (Maximierungsschritt):

$$\sum_i \mathcal{L}(\mathbf{v}^{(i)}, \boldsymbol{\theta}, q) \quad (19.8)$$

wird bezüglich $\boldsymbol{\theta}$ mittels des gewählten Optimierungsalgorithmus vollständig oder teilweise maximiert.

Dies kann als Coordinate-Ascent-Algorithmus (Koordinatenanstiegsalgorithmus) zum Maximieren von \mathcal{L} angesehen werden. Im ersten Schritt maximieren wir \mathcal{L} bezüglich q und im zweiten Schritt maximieren wir \mathcal{L} bezüglich $\boldsymbol{\theta}$.

Der stochastische Gradientenanstieg (engl. *stochastic gradient ascent*) für Modelle mit latenten Variablen ist ein Sonderfall des EM-Algorithmus, in dem der M-Schritt nur einen einzelnen Gradientenschritt macht. Andere Varianten des EM-Algorithmus können viel größere Schritte machen. Für einige Modellfamilien kann der M-Schritt sogar analytisch durchgeführt werden und direkt zur optimalen Lösung für $\boldsymbol{\theta}$ auf Grundlage des aktuellen q springen.

Obwohl der E-Schritt die exakte Inferenz nutzt, können wir uns dennoch vorstellen, dass der EM-Algorithmus auf gewisse Weise die approximative Inferenz einsetzt. So geht der M-Schritt davon aus, dass derselbe Wert für q für alle Werte von $\boldsymbol{\theta}$ genutzt werden kann. Dadurch entsteht eine Lücke zwischen \mathcal{L} und dem wahren $\log p(\mathbf{v})$, während sich der M-Schritt weiter und weiter vom Wert $\boldsymbol{\theta}^{(0)}$ entfernt, der im E-Schritt genutzt wird. Zum Glück reduziert der E-Schritt die Lücke wieder auf Null, sobald die Schleife das nächste Mal beginnt.

Der EM-Algorithmus enthält einige verschiedene Erkenntnisse: Zunächst haben wir die grundlegende Struktur des Lernprozesses, in der wir die Modellparameter anpassen, um die Likelihood eines abgeschlossenen Datensatzes zu verbessern, in der alle fehlenden Variablen ihre Werte aus einer Schätzung der A-posteriori-Verteilung erhalten. Diese spezielle Erkenntnis gilt nicht allein für den EM-Algorithmus. Zum Beispiel weist auch das Gradientenabstiegsverfahren zum Maximieren der Log-Likelihood dieselbe Eigenschaft auf; die Gradientenberechnungen für die Log-Likelihood nutzen

Erwartungswerte bezüglich der A-posteriori-Verteilung über die verdeckten Einheiten. Eine weitere wichtige Erkenntnis im EM-Algorithmus ist, dass wir einen Wert für q auch dann noch verwenden können, wenn für θ ein neuer Wert gilt. Diese spezielle Erkenntnis wird überall im klassischen Machine Learning genutzt, um große M-Schritt-Aktualisierungen herzuleiten. Im Deep-Learning-Kontext sind die meisten Modelle zu komplex, um eine effizient durchführbare Lösung für eine optimal große M-Schritt-Aktualisierung zuzulassen, sodass diese zweite Erkenntnis, die eher ein Alleinstellungsmerkmal des EM-Algorithmus ist, kaum genutzt wird.

19.3 MAP-Inferenz und Sparse Coding

Wir verwenden den Begriff *Inferenz* Inferenz meist für die Berechnung der Wahrscheinlichkeitsverteilung über einen Satz von Variablen, wenn einen anderer Satz von Variablen gegeben ist. Beim Trainieren probabilistischer Modelle mit latenten Variablen sind wir normalerweise an der Berechnung von $p(\mathbf{h} \mid \mathbf{v})$ interessiert. Eine alternative Form der Inferenz besteht in der Berechnung des einzelnen wahrscheinlichsten Werts der fehlenden Variablen, anstatt auf die gesamte Verteilung über die möglichen Werten zu schließen. Im Zusammenhang mit Modellen mit latenten Variablen bedeutet dies die Berechnung von

$$\mathbf{h}^* = \arg \max_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{v}). \quad (19.9)$$

Dies ist als **Maximum-a-posteriori**-Inferenz oder kurz MAP-Inferenz bekannt.

Die MAP-Inferenz wird nur selten als approximative Inferenz angesehen, da sie den exakten wahrscheinlichsten Wert für \mathbf{h}^* berechnet. Wenn wir jedoch einen Lernprozess auf Grundlage der Maximierung von $\mathcal{L}(\mathbf{v}, \mathbf{h}, q)$ entwickeln möchten, ist es hilfreich, sich die MAP-Inferenz als Verfahren vorzustellen, das einen Wert für q liefert. In diesem Sinne ist die MAP-Inferenz eine approximative Inferenz, da sie nicht das optimale q liefert

In Abschnitt 19.1 haben wir gezeigt, dass die exakte Inferenz aus dem Maximieren von

$$\mathcal{L}(\mathbf{v}, \theta, q) = \mathbb{E}_{\mathbf{h} \sim q} [\log p(\mathbf{h}, \mathbf{v})] + H(q) \quad (19.10)$$

bezüglich q über einer uneingeschränkten Familie von Wahrscheinlichkeitsverteilungen besteht, indem ein exakter Optimierungsalgorithmus verwendet wird. Wir können die MAP-Inferenz als eine Form der approximativen Inferenz verstehen.

renz ableiten, indem wir die Familie der Verteilungen, aus denen q gezogen werden kann, einschränken. Dazu muss q eine Dirac-Verteilung annehmen:

$$q(\mathbf{h} \mid \mathbf{v}) = \delta(\mathbf{h} - \boldsymbol{\mu}). \quad (19.11)$$

Das heißt wir können q nun vollständig mit $\boldsymbol{\mu}$ steuern. Durch Entfernen der Terme von \mathcal{L} , die nicht mit $\boldsymbol{\mu}$ variieren, gelangen wir zum Optimierungsproblem

$$\boldsymbol{\mu}^* = \arg \max_{\boldsymbol{\mu}} \log p(\mathbf{h} = \boldsymbol{\mu}, \mathbf{v}), \quad (19.12)$$

das dem MAP-Inferenzproblem entspricht:

$$\mathbf{h}^* = \arg \max_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{v}). \quad (19.13)$$

Wir können somit ein der Erwartungsmaximierung ähnliches Lernverfahren begründen, bei dem wir abwechselnd die MAP-Inferenz zum Schließen auf \mathbf{h}^* durchführen und dann $\boldsymbol{\theta}$ anpassen, um $\log p(\mathbf{h}^*, \mathbf{v})$ zu vergrößern. Wie bei der Erwartungsmaximierung handelt es sich um eine Art Coordinate Ascent (Koordinatenanstieg) auf \mathcal{L} , bei dem wir abwechselnd die Inferenz zum Optimieren von \mathcal{L} bezüglich q durchführen und Parameteranpassungen zum Optimieren von \mathcal{L} bezüglich $\boldsymbol{\theta}$ anwenden. Das gesamte Verfahren kann dadurch begründet werden, dass \mathcal{L} eine untere Schranke für $\log p(\mathbf{v})$ darstellt. Im Falle der MAP-Inferenz ist diese Begründung recht nichtssagend, da die Schranke in keiner Weise fest ist (eine Folge der differentiellen Entropie der negativen Unendlichkeit in der Dirac-Verteilung). Durch Einspeisen von Rauschen in $\boldsymbol{\mu}$ wird die Schranke wieder aussagekräftig.

Die MAP-Inferenz wird im Deep Learning häufig als Merkmalsextraktor und auch als Lernmechanismus verwendet. In erster Linie kommt sie für Sparse-Coding-Modelle zum Einsatz.

Aus Abschnitt 13.4 wissen Sie, dass Sparse Coding ein lineares Faktorenmodell ist, das auch eine dünne Besetzung der A-priori-Wahrscheinlichkeit für die eigenen verdeckten Einheiten veranlasst. Häufig wird eine faktorielle Laplace-A-priori-Wahrscheinlichkeit verwendet mit

$$p(h_i) = \frac{\lambda}{2} e^{-\lambda|h_i|}. \quad (19.14)$$

Die sichtbaren Einheiten werden dann mittels linearer Transformation und Hinzufügen von Rauschen erzeugt:

$$p(\mathbf{x} \mid \mathbf{h}) = \mathcal{N}(\mathbf{v}; \mathbf{W}\mathbf{h} + \mathbf{b}, \beta^{-1}\mathbf{I}). \quad (19.15)$$

Das Berechnen und sogar das Darstellen von $p(\mathbf{h} \mid \mathbf{v})$ ist schwierig. Jeweils zwei Variablen h_i und h_j sind Eltern von \mathbf{v} . Bei Beobachtung von \mathbf{v} enthält das graphische Modell also einen aktiven Pfad zwischen h_i und h_j . Alle verdeckten Einheiten sind damit Teil einer gewaltigen Clique in $p(\mathbf{h} \mid \mathbf{v})$. Wäre das Modell normalverteilt, dann könnten diese Interaktionen effizient anhand der Kovarianzmatrix modelliert werden – aber die dünnbesetzte A-priori-Wahrscheinlichkeit führt dazu, dass diese Interaktionen nicht normalverteilt sind.

Da $p(\mathbf{h} \mid \mathbf{v})$ nicht effizient berechenbar ist, können auch die Log-Likelihood und ihr Gradient nicht berechnet werden. Der Einsatz des exakten Maximum Likelihood Learnings bleibt uns somit verwehrt. Stattdessen nutzen wir die MAP-Inferenz und erlernen die Parameter durch Maximieren der ELBO, die mittels Dirac-Verteilung um den MAP-Schätzwert für \mathbf{h} definiert ist.

Wenn wir alle \mathbf{h} -Vektoren in der Trainingsdatenmenge zu einer Matrix \mathbf{H} verknüpfen und alle \mathbf{v} -Vektoren zu einer Matrix \mathbf{V} , dann besteht der Lernprozess mit Sparse Coding aus dem Minimieren von

$$J(\mathbf{H}, \mathbf{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (\mathbf{V} - \mathbf{H}\mathbf{W}^\top)_{i,j}^2. \quad (19.16)$$

Meist kommt bei der Anwendung von Sparse Coding auch Weight Decay oder eine Bedingung an die Norm der Spalten von \mathbf{W} zum Einsatz, um die pathologische Lösung mit extrem kleinem \mathbf{H} und großem \mathbf{W} zu verhindern.

Wir können J durch Wechsel zwischen der Minimierung bezüglich \mathbf{H} und der Minimierung bezüglich \mathbf{W} minimieren. Beide Unterprobleme sind konvex. Tatsächlich handelt es sich bei der Minimierung bezüglich \mathbf{W} lediglich um ein lineares Regressionsproblem. Die Minimierung von J bezüglich beider Argumente stellt jedoch im Normalfall kein konkaves Problem dar.

Die Minimierung bezüglich \mathbf{H} erfordert spezialisierte Algorithmen wie den Feature-Sign-Suchalgorithmus (Lee et al., 2007).

19.4 Variational Inference und Variational Learning

Wir haben gesehen, dass die ELBO $\mathcal{L}(\mathbf{v}; \boldsymbol{\theta}, q)$ eine untere Schranke für $\log p(\mathbf{v}; \boldsymbol{\theta})$ ist und wie die Inferenz \mathcal{L} quasi bezüglich q maximiert. Außerdem haben wir gesehen, wie das Lernen als Maximieren von \mathcal{L} bezüglich $\boldsymbol{\theta}$ betrachtet werden kann. Der EM-Algorithmus ermöglicht es uns, große Lernschritte mit einem unveränderlichen q zu machen. Wie gezeigt, ermöglichen

Lernalgorithmen auf Basis der MAP-Inferenz das Erlernen einer Punktschätzung für $p(\mathbf{h} | \mathbf{v})$ anstelle des Schließens auf die gesamte Verteilung. Nun entwickeln wir den allgemeineren Ansatz für das Variational Learning (variantenreiches Lernen oder Variationslernen).

Die Grundidee hinter dem Variational Learning ist, dass wir \mathcal{L} über einer eingeschränkten Familie von Verteilungen q maximieren können. Diese Familie muss so gewählt sein, dass $\mathbb{E}_q \log p(\mathbf{h}, \mathbf{v})$ einfach zu berechnen ist. Dazu können wir Annahmen über die Faktorisierung von q einführen.

Ein häufiger Ansatz beim Variational Learning besteht darin, die Einschränkung aufzuerlegen, dass q eine faktorielle Verteilung ist:

$$q(\mathbf{h} | \mathbf{v}) = \prod_i q(h_i | \mathbf{v}). \quad (19.17)$$

Man bezeichnet dies als **Molekularfeld**-Ansatz. Allgemein ausgedrückt, können wir jede beliebige graphische Modellstruktur für q vorschreiben, um die Anzahl der Interaktionen, die unsere Approximation erfassen soll, flexibel festzulegen. Dieser ganz allgemeine graphische Modellansatz wird als **strukturierte Variational Inference** bezeichnet (*Saul und Jordan, 1996*).

Das Schöne an diesem Ansatz ist, dass wir keine bestimmte parametrische Form für q vorgeben müssen. Wir geben genau an, wie die Faktorisierung von q aussehen soll, aber dann bestimmt das Optimierungsproblem die optimale Wahrscheinlichkeitsverteilung im Rahmen dieser Bedingungen an die Faktorisierung. Für diskrete latente Variablen bedeutet das also, dass wir herkömmliche Optimierungsmethoden nutzen, um eine endliche Anzahl von Variablen zur Beschreibung der q -Verteilung zu optimieren. Für stetige latente Variablen nutzen wir einen Zweig der Mathematik namens Variationsrechnung. Damit führen wir die Optimierung über einem Funktionsraum durch und bestimmen dabei, welche Funktion für die Darstellung von q verwendet werden soll. Von der Bezeichnung »Variationsrechnung« (engl. *calculus of variations*) sind auch die Begriffe »Variational Learning« und »Variational Inference« abgeleitet, obwohl diese auch verwendet werden, wenn die latenten Variablen diskret sind und die Variationsrechnung nicht benötigt wird. Bei stetigen latenten Variablen ist die Variationsrechnung ein leistungsstarkes Werkzeug, das dem menschlichen Entwickler des Modells einen Großteil seiner Verantwortung abnimmt. Er muss lediglich bestimmen, wie q faktorisiert, und nicht schätzen, wie ein bestimmtes q aussehen muss, das die A-posteriori-Wahrscheinlichkeit exakt approximieren kann.

Da $\mathcal{L}(\mathbf{v}; \boldsymbol{\theta}, q)$ definiert ist als $\log p(\mathbf{v}; \boldsymbol{\theta}) - D_{\text{KL}}(q(\mathbf{h} | \mathbf{v}) \| p(\mathbf{h} | \mathbf{v}; \boldsymbol{\theta}))$, können wir uns das Maximieren von \mathcal{L} bezüglich q als Minimieren von

$D_{\text{KL}}(q(\mathbf{h} | \mathbf{v}) \| p(\mathbf{h} | \mathbf{v}))$ vorstellen. In diesem Sinne passen wir q an p an. Dabei verwenden wir allerdings die umgekehrte Richtung der Kullback-Leibler-Divergenz (KL-Divergenz), die wir normalerweise für das Anpassen einer Approximation nutzen. Wenn wir Maximum Likelihood Learning zum Anpassen eines Modells an Daten einsetzen, minimieren wir $D_{\text{KL}}(p_{\text{data}} \| p_{\text{model}})$. Wie in Abbildung 3.6 dargestellt, animiert die Maximum Likelihood das Modell dazu, überall dort eine hohe Wahrscheinlichkeit aufzuweisen, wo die Daten eine hohe Wahrscheinlichkeit haben. Unser Inferenzverfahren auf Optimierungsbasis dagegen animiert q dazu, überall dort eine niedrige Wahrscheinlichkeit anzunehmen, wo die tatsächliche A-posteriori-Wahrscheinlichkeit eine niedrige Wahrscheinlichkeit aufweist. Beide Richtungen der KL-Divergenz können wünschenswerte und unerwünschte Eigenschaften aufweisen. Die Wahl der Richtung richtet sich nach den Eigenschaften, die für die jeweilige Anwendung die höchste Priorität haben. Im Inferenz-Optimierungsproblem wählen wir aus rechnerischen Gründen $D_{\text{KL}}(q(\mathbf{h} | \mathbf{v}) \| p(\mathbf{h} | \mathbf{v}))$. Vor allem die Berechnung von $D_{\text{KL}}(q(\mathbf{h} | \mathbf{v}) \| p(\mathbf{h} | \mathbf{v}))$ erfordert die Berechnung von Erwartungswerten bezüglich q ; indem wir q so bestimmen, dass diese einfach ist, können wir die erforderlichen Erwartungswerte vereinfachen. Die umgekehrte Richtung der KL-Divergenz würde die Berechnung der Erwartungswerte bezüglich der wahren A-posteriori-Wahrscheinlichkeit erfordern. Da die Form der wahren A-posteriori-Wahrscheinlichkeit von der Wahl des Modells abhängt, können wir keinen weniger aufwendigen Ansatz zur exakten Berechnung von $D_{\text{KL}}(p(\mathbf{h} | \mathbf{v}) \| q(\mathbf{h} | \mathbf{v}))$ entwerfen.

19.4.1 Diskrete latente Variablen

Variational Inference mit diskreten latenten Variablen ist verhältnismäßig unkompliziert. Wir definieren eine Verteilung q – üblicherweise so, dass jeder Faktor von q mittels Lookup-Tabelle über diskreten Zuständen definiert wird. Im einfachsten Fall ist \mathbf{h} binär und wir treffen die Molekularfeld-Annahme, dass q über jedes einzelne h_i faktorisiert. In diesem Fall können wir q mit einem Vektor $\hat{\mathbf{h}}$ parametrisieren, dessen Einträge Wahrscheinlichkeiten sind. Dann ist $q(h_i = 1 | \mathbf{v}) = \hat{h}_i$.

Nachdem wir bestimmt haben, wie q repräsentiert wird, optimieren wir einfach seine Parameter. Mit diskreten latenten Variablen ist dies ein normales Optimierungsproblem. Prinzipiell könnte die Auswahl von q mit einem beliebigen Optimierungsalgorithmus wie dem Gradientenabstiegsverfahren erfolgen.

Da diese Optimierung in der inneren Schleife eines Lernalgorithmus stattfinden muss, muss sie sehr schnell sein. Hierzu nutzen wir meist spe-

zielle Optimierungsalgorithmen, die zum Lösen vergleichsweise kleiner und einfacher Probleme in wenigen Schritten konzipiert sind. Eine beliebte Wahl ist die Iteration von Fixpunktgleichungen, also das Lösen von

$$\frac{\partial}{\partial \hat{h}_i} \mathcal{L} = 0 \quad (19.18)$$

für \hat{h}_i . Wir aktualisieren wiederholt unterschiedliche Elemente von $\hat{\mathbf{h}}$, bis wir ein Konvergenzkriterium erfüllen.

Wir zeigen die Anwendung der Variational Inference konkret für das **binäre Sparse-Coding-Modell** (es wurde von *Henniges et al. [2010]* entwickelt und nutzt das klassische generische Molekularfeld für das Modell, wobei sie einen anderen spezialisieren Algorithmus einführen). Diese Herleitung steigt tief in mathematische Details ein. Sie ist für Leser gedacht, die jegliche Unklarheiten in der bisherigen allgemeinen konzeptuellen Beschreibung der Variational Inference und des Variational Learnings beseitigen möchten. Leser, die keine Algorithmen für das Variational Learning entwickeln oder implementieren möchten, können den nächsten Abschnitt problemlos überspringen, da darin keine neuen wichtigen Konzepte eingeführt werden. Leser, die sich mit dem Beispiel für binäres Sparse Coding beschäftigen möchten, sollten die Liste der nützlichen Eigenschaften von Funktionen, die häufig in probabilistischen Modellen vorkommen, nochmals in Abschnitt 3.10 nachschlagen. Wir verwenden diese Eigenschaften in den folgenden Herleitungen immer wieder, ohne dabei jeweils gesondert darauf hinzuweisen.

Im binären Sparse-Coding-Modell wird die Eingabe $\mathbf{v} \in \mathbb{R}^n$ aus dem Modell durch Hinzufügen von normalverteiltem Rauschen zur Summe der m unterschiedlichen Komponenten, die vorhanden sein oder fehlen können, erzeugt. Jede Komponente wird durch die zugehörige verdeckte Einheit in $\mathbf{h} \in \{0, 1\}^m$ aktiviert oder deaktiviert:

$$p(h_i = 1) = \sigma(b_i), \quad (19.19)$$

$$p(\mathbf{v} | \mathbf{h}) = \mathcal{N}(\mathbf{v}; \mathbf{W}\mathbf{h}, \boldsymbol{\beta}^{-1}), \quad (19.20)$$

wobei \mathbf{b} eine erlernbare Menge von Verzerrungen ist, \mathbf{W} eine erlernbare Gewichtungsmatrix und $\boldsymbol{\beta}$ eine erlernbare Diagonal-Präzisionsmatrix.

Das Trainieren dieses Modells mittels Maximum Likelihood erfordert das Verwenden der Ableitung bezüglich der Parameter. Betrachten wir die Ableitung bezüglich einer der Verzerrungen:

$$\frac{\partial}{\partial b_i} \log p(\mathbf{v}) \quad (19.21)$$

$$= \frac{\frac{\partial}{\partial b_i} p(\mathbf{v})}{p(\mathbf{v})} \quad (19.22)$$

$$= \frac{\frac{\partial}{\partial b_i} \sum_{\mathbf{h}} p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})} \quad (19.23)$$

$$= \frac{\frac{\partial}{\partial b_i} \sum_{\mathbf{h}} p(\mathbf{h}) p(\mathbf{v} | \mathbf{h})}{p(\mathbf{v})} \quad (19.24)$$

$$= \frac{\sum_{\mathbf{h}} p(\mathbf{v} | \mathbf{h}) \frac{\partial}{\partial b_i} p(\mathbf{h})}{p(\mathbf{v})} \quad (19.25)$$

$$= \sum_{\mathbf{h}} p(\mathbf{h} | \mathbf{v}) \frac{\frac{\partial}{\partial b_i} p(\mathbf{h})}{p(\mathbf{h})} \quad (19.26)$$

$$= \mathbb{E}_{\mathbf{h} \sim p(\mathbf{h} | \mathbf{v})} \frac{\partial}{\partial b_i} \log p(\mathbf{h}). \quad (19.27)$$

Dazu müssen die Erwartungswerte von $p(\mathbf{h} | \mathbf{v})$ bestimmt werden. Leider ist $p(\mathbf{h} | \mathbf{v})$ eine komplexe Verteilung. Abbildung 19.2 enthält die Graphenstruktur für $p(\mathbf{h}, \mathbf{v})$ und $p(\mathbf{h} | \mathbf{v})$. Die A-posteriori-Verteilung entspricht dem vollständigen Graphen über den verdeckten Einheiten; daher beschleunigen Algorithmen zur Variablenelimination die Berechnung der benötigten Erwartungswerte gegenüber dem Brute-Force-Ansatz nicht.

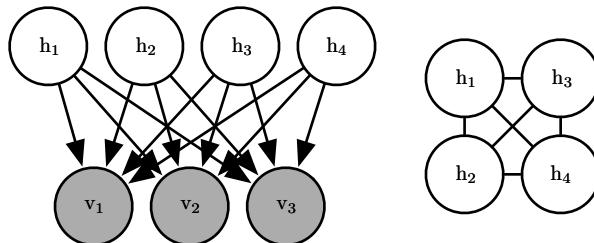


Abbildung 19.2: Die Graphenstruktur eines binären Sparse-Coding-Modells mit vier verdeckten Einheiten. (*Links*) Die Graphenstruktur für $p(\mathbf{h}, \mathbf{v})$. Beachten Sie, dass die Kanten gerichtet sind und dass jeweils zwei verdeckte Einheiten jeder sichtbaren Einheit gemeinsam übergeordnet sind. (*Rechts*) Die Graphenstruktur für $p(\mathbf{h} | \mathbf{v})$. Um die aktiven Pfade zwischen den gemeinsamen Elternknoten zu berücksichtigen, benötigt die A-posteriori-Verteilung eine Kante zwischen allen verdeckten Einheiten.

Wir können dieses Problem stattdessen mithilfe der Variational Inference und des Variational Learnings lösen.

Wir können eine Molekularfeld-Approximation vornehmen:

$$q(\mathbf{h} \mid \mathbf{v}) = \prod_i q(h_i \mid \mathbf{v}). \quad (19.28)$$

Die latenten Variablen des binären Sparse-Coding-Modells sind binär; daher müssen wir zur Repräsentation eines faktoriellen q lediglich m Bernoulli-Verteilungen $q(h_i \mid \mathbf{v})$ modellieren. Eine natürliche Art zur Darstellung der Mittelwerte der Bernoulli-Verteilungen ist ein Vektor $\hat{\mathbf{h}}$ der Wahrscheinlichkeiten, mit $q(h_i = 1 \mid \mathbf{v}) = \hat{h}_i$. Wir geben die Einschränkung vor, dass \hat{h}_i niemals gleich 0 oder 1 ist, um Fehler bei der Berechnung zu vermeiden, zum Beispiel $\log \hat{h}_i$.

Wir werden sehen, dass die Gleichungen für die Variational Inference analytisch niemals 0 oder 1 für \hat{h}_i zuweisen. In einer Softwareimplementierung kann der maschinelle Rundungsfehler jedoch zu den Werten 0 oder 1 führen. Im Softwarekontext könnte es wünschenswert sein, das binäre Sparse Coding zu implementieren, indem wir einen uneingeschränkten Vektor der Variationsparameterrameter \mathbf{z} verwenden und $\hat{\mathbf{h}}$ über die Beziehung $\hat{\mathbf{h}} = \sigma(\mathbf{z})$ ermitteln. Wir können $\log \hat{h}_i$ daher problemlos auf einem Computer berechnen, indem wir die Identität $\log \sigma(z_i) = -\zeta(-z_i)$ zum Verknüpfen von Sigmoid und Softplus verwenden.

Um unsere Herleitung des Variational Learnings im binären Sparse-Coding-Modell zu beginnen, zeigen wir, dass die Verwendung dieser Molekularfeld-Approximation das Lernen effizient durchführbar macht.

Die ELBO ergibt sich aus

$$\mathcal{L}(\mathbf{v}, \theta, q) \quad (19.29)$$

$$= \mathbb{E}_{\mathbf{h} \sim q} [\log p(\mathbf{h}, \mathbf{v})] + H(q) \quad (19.30)$$

$$= \mathbb{E}_{\mathbf{h} \sim q} [\log p(\mathbf{h}) + \log p(\mathbf{v} \mid \mathbf{h}) - \log q(\mathbf{h} \mid \mathbf{v})] \quad (19.31)$$

$$= \mathbb{E}_{\mathbf{h} \sim q} \left[\sum_{i=1}^m \log p(h_i) + \sum_{i=1}^n \log p(v_i \mid \mathbf{h}) - \sum_{i=1}^m \log q(h_i \mid \mathbf{v}) \right] \quad (19.32)$$

$$= \sum_{i=1}^m \left[\hat{h}_i (\log \sigma(b_i) - \log \hat{h}_i) + (1 - \hat{h}_i) (\log \sigma(-b_i) - \log(1 - \hat{h}_i)) \right] \quad (19.33)$$

$$+ \mathbb{E}_{\mathbf{h} \sim q} \left[\sum_{i=1}^n \log \sqrt{\frac{\beta_i}{2\pi}} \exp \left(-\frac{\beta_i}{2} (v_i - \mathbf{W}_{i,:} \mathbf{h})^2 \right) \right] \quad (19.34)$$

$$= \sum_{i=1}^m \left[\hat{h}_i (\log \sigma(b_i) - \log \hat{h}_i) + (1 - \hat{h}_i) (\log \sigma(-b_i) - \log(1 - \hat{h}_i)) \right] \quad (19.35)$$

$$+ \frac{1}{2} \sum_{i=1}^n \left[\log \frac{\beta_i}{2\pi} - \beta_i \left(v_i^2 - 2v_i \mathbf{W}_{i,:} \hat{\mathbf{h}} + \sum_j \left[W_{i,j}^2 \hat{h}_j + \sum_{k \neq j} W_{i,j} W_{i,k} \hat{h}_j \hat{h}_k \right] \right) \right]. \quad (19.36)$$

Obwohl diese Gleichungen ästhetisch nicht besonders ansprechend sind, zeigen sie doch, dass \mathcal{L} durch eine kleine Anzahl einfacher arithmetischer Operationen dargestellt werden kann. Die ELBO \mathcal{L} ist somit effizient berechenbar. Wir können \mathcal{L} als Ersatz für die nicht effizient berechenbare Log-Likelihood verwenden.

Im Prinzip könnten wir einfach einen Gradientenanstieg für v und \mathbf{h} durchführen – das wäre ein ganz und gar zulässiger kombinierter Inferenz- und Trainingsalgorithmus. Normalerweise nehmen wir davon aber aus zwei Gründen Abstand: Erstens müssten wir $\hat{\mathbf{h}}$ für jedes v speichern. Wir bevorzugen allerdings Algorithmen, die keinen Speicherplatz für jedes Beispiel benötigen. Es ist schwierig, Lernalgorithmen für Milliarden von Beispielen zu skalieren, wenn wir uns einen dynamisch angepassten Vektor für jedes Beispiel merken müssen. Zweitens möchten wir in der Lage sein, die Merkmale $\hat{\mathbf{h}}$ sehr schnell zu extrahieren, um den Inhalt von v zu erkennen. In einer produktiv eingesetzten Umgebung müssen wir in der Lage sein, $\hat{\mathbf{h}}$ in Echtzeit zu berechnen.

Aus diesen beiden Gründen verwenden wir das Gradientenabstiegsverfahren normalerweise nicht zum Berechnen der Molekularfeld-Parameter $\hat{\mathbf{h}}$. Stattdessen schätzen wir sie schnell mittels Fixpunktgleichungen.

Die Idee hinter Fixpunktgleichungen ist die Suche nach einem lokalen Maximum für $\hat{\mathbf{h}}$, wobei $\nabla_{\mathbf{h}} \mathcal{L}(v, \theta, \hat{\mathbf{h}}) = \mathbf{0}$ ist. Wir können diese Gleichung nicht effizient für alle $\hat{\mathbf{h}}$ zeitgleich lösen. Allerdings können wir die Lösung für eine einzelne Variable bestimmen:

$$\frac{\partial}{\partial \hat{h}_i} \mathcal{L}(v, \theta, \hat{\mathbf{h}}) = 0. \quad (19.37)$$

Anschließend können wir die Lösung iterativ auf die Gleichung für $i = 1, \dots, m$ anwenden und den Zyklus wiederholen, bis ein Konvergenzkriterium erfüllt ist. Übliche Konvergenzkriterien sind ein Abbruch, wenn ein vollständiger Zyklus von Aktualisierungen nicht zu einer Verbesserung von \mathcal{L} um mehr als einen bestimmten Toleranzbetrag führt, oder ein Abbruch, wenn der Zyklus $\hat{\mathbf{h}}$ nicht mindestens um mehr als eine bestimmte Höhe ändert.

Die Iteration von Molekularfeld-Fixpunktgleichungen ist eine allgemeine Vorgehensweise, die in einer Vielzahl von Modellen für eine schnelle Varia-

tional Inference sorgt. Als konkretes Beispiel zeigen wir die Herleitung der Aktualisierungen für das binäre Sparse-Coding-Modell.

Zunächst müssen wir einen Ausdruck für die Ableitungen bezüglich \hat{h}_i notieren. Dazu setzen wir Gleichung 19.36 in die linke Seite von Gleichung 19.37 ein:

$$\frac{\partial}{\partial \hat{h}_i} \mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, \hat{\mathbf{h}}) \quad (19.38)$$

$$= \frac{\partial}{\partial \hat{h}_i} \left[\sum_{j=1}^m \left[\hat{h}_j (\log \sigma(b_j) - \log \hat{h}_j) + (1 - \hat{h}_j) (\log \sigma(-b_j) - \log(1 - \hat{h}_j)) \right] \right] \quad (19.39)$$

$$+ \frac{1}{2} \sum_{j=1}^n \left[\log \frac{\beta_j}{2\pi} - \beta_j \left(v_j^2 - 2v_j \mathbf{W}_{j,:} \hat{\mathbf{h}} + \sum_k \left[W_{j,k}^2 \hat{h}_k + \sum_{l \neq k} W_{j,k} W_{j,l} \hat{h}_k \hat{h}_l \right] \right) \right] \quad (19.40)$$

$$= \log \sigma(b_i) - \log \hat{h}_i - 1 + \log(1 - \hat{h}_i) + 1 - \log \sigma(-b_i) \quad (19.41)$$

$$+ \sum_{j=1}^n \left[\beta_j \left(v_j W_{j,i} - \frac{1}{2} W_{j,i}^2 - \sum_{k \neq i} W_{j,k} W_{j,i} \hat{h}_k \right) \right] \quad (19.42)$$

$$= b_i - \log \hat{h}_i + \log(1 - \hat{h}_i) + \mathbf{v}^\top \boldsymbol{\beta} \mathbf{W}_{:,i} - \frac{1}{2} \mathbf{W}_{:,i}^\top \boldsymbol{\beta} \mathbf{W}_{:,i} - \sum_{j \neq i} \mathbf{W}_{:,j}^\top \boldsymbol{\beta} \mathbf{W}_{:,i} \hat{h}_j. \quad (19.43)$$

Um die Inferenzregel für Fixpunktaktualisierungen anzuwenden, stellen wir nach dem \hat{h}_i um, das Gleichung 19.43 zu 0 löst:

$$\hat{h}_i = \sigma \left(b_i + \mathbf{v}^\top \boldsymbol{\beta} \mathbf{W}_{:,i} - \frac{1}{2} \mathbf{W}_{:,i}^\top \boldsymbol{\beta} \mathbf{W}_{:,i} - \sum_{j \neq i} \mathbf{W}_{:,j}^\top \boldsymbol{\beta} \mathbf{W}_{:,i} \hat{h}_j \right). \quad (19.44)$$

An diesem Punkt sehen wir, dass es eine enge Verbindung zwischen RNNs und der Inferenz in graphischen Modellen gibt, denn die Molekularfeld-Fixpunktgleichungen haben ein RNN definiert. Die Aufgabe dieses Netzes ist das Durchführen von Inferenz. Wir haben beschrieben, wie dieses Netz aus einer Modellbeschreibung hergeleitet wird, aber das Inferenznetz kann auch direkt trainiert werden. Einige diesbezügliche Ideen werden in Kapitel 20 beschrieben.

Für den Fall des binären Sparse Codings zeigt sich, dass die rekurrente Netzverbindung aus Gleichung 19.44 aus der wiederholten Anpassung der verdeckten Einheiten auf Basis der sich verändernden Werte der benachbarten verdeckten Einheiten besteht. Die Eingabe übergibt stets eine

unveränderliche Botschaft $\mathbf{v}^\top \boldsymbol{\beta} \mathbf{W}$ an die verdeckten Einheiten, aber die verdeckten Einheiten aktualisieren kontinuierlich die Botschaft, die sie aneinander senden. So hemmen zwei Einheiten \hat{h}_i und \hat{h}_j einander, wenn ihre Gewichtungsvektoren übereinstimmen. Dies ist eine Art von Wettstreit zwischen zwei verdeckten Einheiten, die beide die Eingabe erklären – aber nur die Einheit, die die beste Erklärung für die Eingabe liefert, darf aktiv bleiben. Dieser Wettstreit ist der Versuch der Molekularfeld-Approximation, die Explaining-Away-Interaktionen in der A-posteriori-Verteilung des binären Sparse Codings zu erfassen. Der Explaining-Away-Effekt sollte eigentlich zu einer multimodalen A-posteriori-Verteilung führen: Wenn wir Stichproben aus der A-posteriori-Verteilung ziehen, ist bei einigen davon eine Einheit aktiv, bei anderen dagegen die andere Einheit. Bei nur sehr wenigen Stichproben sind beide Einheiten aktiv. Leider können Explaining-Away-Interaktionen nicht durch das für das Molekularfeld verwendete faktorielle q modelliert werden, sodass die Molekularfeld-Approximation einen Modus für die Modellierung auswählen muss. Dieses Verhalten wird in Abbildung 3.6 dargestellt.

Wir können Gleichung 19.44 in gleichwertiger Form neu schreiben, um einige weitere Erkenntnisse zu gewinnen:

$$\hat{h}_i = \sigma \left(b_i + \left(\mathbf{v} - \sum_{j \neq i} \mathbf{W}_{:,j} \hat{h}_j \right)^\top \boldsymbol{\beta} \mathbf{W}_{:,i} - \frac{1}{2} \mathbf{W}_{:,i}^\top \boldsymbol{\beta} \mathbf{W}_{:,i} \right). \quad (19.45)$$

In dieser Neuformulierung sehen wir, dass die Eingabe in jedem Schritt aus $\mathbf{v} - \sum_{j \neq i} \mathbf{W}_{:,j} \hat{h}_j$ anstelle von \mathbf{v} besteht. Wir können uns vorstellen, dass die Einheit i versucht, den Restfehler in \mathbf{v} anhand des Codes der anderen Einheiten zu codieren. Sparse Coding stellt somit eine Art iterativen Autoencoder dar, der wiederholt seine Eingabe codiert und decodiert, um nach jeder Iteration die Fehler in der Rekonstruktion zu beheben.

In diesem Beispiel haben wir eine Update-Regel abgeleitet, die zu jedem Zeitpunkt eine einzelne Einheit aktualisiert. Es wäre vorteilhaft, mehrere Einheiten zeitgleich zu aktualisieren. Einige graphische Modelle wie DBMs sind so aufgebaut, dass wir viele Einträge von $\hat{\mathbf{h}}$ zeitgleich lösen können. Leider erlaubt das binäre Sparse Coding derartige Anpassungen eines Blocks nicht. Stattdessen können wir ein heuristisches Verfahren namens **Damping** zum Durchführen von Block-Updates einsetzen. Im Rahmen des Dampings bestimmen wir die für sich genommen optimalen Werte für jedes Element aus $\hat{\mathbf{h}}$ und verschieben anschließend alle Werte um einen kleinen Schritt in diese Richtung. Dieser Ansatz garantiert nicht länger, dass \mathcal{L} in jedem Schritt zunimmt, aber er funktioniert in der Praxis in vielen Modellen hinreichend

gut. Bei *Koller und Friedman* (2009) finden Sie mehr Informationen über die Auswahl des Grads der Synchronität und Verfahren des Dampings für Algorithmen, die Botschaften weitergeben.

19.4.2 Variationsrechnung

Bevor wir mit unserer Darstellung des Variational Learnings fortfahren, müssen wir kurz eine Reihe wichtiger mathematischer Werkzeuge für das Variational Learning einführen – die **Variationsrechnung**.

Viele Machine-Learning-Verfahren minimieren eine Funktion $J(\boldsymbol{\theta})$ durch Bestimmen des Eingabevektors $\boldsymbol{\theta} \in \mathbb{R}^n$, für den sie den kleinsten Wert annimmt. Das lässt sich mittels multivariater Analyse und linearer Algebra durch Bestimmen der kritischen Punkte, in denen $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbf{0}$ ist, erreichen. In einigen Fällen möchten wir eine Funktion $f(\mathbf{x})$ bestimmen, zum Beispiel wenn eine Wahrscheinlichkeitsdichtefunktion über eine Zufallsvariable gesucht ist. Dies ermöglicht uns die Variationsrechnung.

Eine Funktion einer Funktion f heißt **Funktional** $J[f]$. Ebenso wie es partielle Ableitungen einer Funktion bezüglich der Elemente ihres vektorwertigen Arguments gibt, gibt es auch Funktionalableitungen (engl. auch *variational derivatives* genannt) eines Funktionalen $J[f]$ bezüglich einzelner Werte der Funktion $f(\mathbf{x})$ für jeden spezifischen Wert von \mathbf{x} . Die Funktionalableitung des Funktionalen J bezüglich des Werts der Funktion f im Punkt \mathbf{x} wird $\frac{\delta}{\delta f(\mathbf{x})} J$ geschrieben.

Eine vollständige formale Entwicklung von Funktionalableitungen sprengt den Rahmen dieses Buchs. Für unsere Zwecke reicht es aus festzuhalten, dass für differenzierbare Funktionen $f(\mathbf{x})$ und differenzierbare Funktionen $g(y, \mathbf{x})$ mit stetigen Ableitungen

$$\frac{\delta}{\delta f(\mathbf{x})} \int g(f(\mathbf{x}), \mathbf{x}) d\mathbf{x} = \frac{\partial}{\partial y} g(f(\mathbf{x}), \mathbf{x}) \quad (19.46)$$

ist. Um ein Gespür für diese Identität zu bekommen, können Sie sich $f(\mathbf{x})$ als einen Vektor mit überabzählbar vielen Elementen vorstellen, wobei ein reeller Vektor \mathbf{x} zur Indizierung dient. In dieser (nicht ganz vollständigen) Vorstellung ist die Identität für die Funktionalableitungen dieselbe, die wir für einen Vektor $\boldsymbol{\theta} \in \mathbb{R}^n$ erhalten würden, der mit positiven ganzen Zahlen indiziert ist:

$$\frac{\partial}{\partial \theta_i} \sum_j g(\theta_j, j) = \frac{\partial}{\partial \theta_i} g(\theta_i, i). \quad (19.47)$$

Viele Ergebnisse in anderen Machine-Learning-Veröffentlichungen werden mithilfe der allgemeineren **Euler-Lagrange-Gleichung** dargestellt, in der

g sowohl von den Ableitungen von f als auch vom Wert für f abhängig sein kann. Wir nutzen diese ganz allgemeine Form für die Ergebnisse in diesem Buch allerdings nicht.

Um eine Funktion bezüglich eines Vektors zu optimieren, nehmen wir den Gradienten der Funktion bezüglich des Vektors und lösen nach dem Punkt auf, in dem jedes Element des Gradienten gleich Null ist. Ebenso können wir ein Funktional optimieren, indem wir die Funktion bestimmen, in der die Funktionalableitung in jedem Punkt gleich Null ist.

Als Beispiel dafür, wie dieses Verfahren funktioniert, betrachten wir das Problem, dass wir die Wahrscheinlichkeitsverteilungsfunktion über $x \in \mathbb{R}$ finden wollen, die eine maximale differentielle Entropie aufweist. Sie wissen bereits, dass die Entropie einer Wahrscheinlichkeitsverteilung $p(x)$ definiert ist als

$$H[p] = -\mathbb{E}_x \log p(x). \quad (19.48)$$

Für stetige Werte ist der Erwartungswert ein Integral:

$$H[p] = - \int p(x) \log p(x) dx. \quad (19.49)$$

Wir können nicht einfach $H[p]$ bezüglich der Funktion $p(x)$ maximieren, da das Ergebnis eventuell keine Wahrscheinlichkeitsverteilung ist. Stattdessen müssen wir Lagrange-Multiplikatoren verwenden, um eine Bedingung hinzuzufügen, dass $p(x)$ zu 1 integriert. Außerdem muss die Entropie mit zunehmender Varianz uneingeschränkt zunehmen. Somit wird die Frage, welche Verteilung die größte Entropie aufweist, uninteressant. Stattdessen fragen wir, welche Verteilung für eine feste Varianz σ^2 die maximale Entropie aufweist. Letztendlich ist das Problem unterbestimmt, da die Verteilung beliebig verschoben werden kann, ohne die Entropie zu verändern. Um eine eindeutige Lösung zu ermöglichen, fügen wir die Bedingung hinzu, dass der Mittelwert der Verteilung μ ist. Das Lagrange-Funktional für dieses Optimierungsproblem lautet:

$$\mathcal{L}[p] = \lambda_1 \left(\int p(x) dx - 1 \right) + \lambda_2 (\mathbb{E}[x] - \mu) + \lambda_3 (\mathbb{E}[(x - \mu)^2] - \sigma^2) + H[p] \quad (19.50)$$

$$= \int \left(\lambda_1 p(x) + \lambda_2 p(x)x + \lambda_3 p(x)(x - \mu)^2 - p(x) \log p(x) \right) dx - \lambda_1 - \mu \lambda_2 - \sigma^2 \lambda_3. \quad (19.51)$$

Um das Lagrange-Funktional bezüglich p zu minimieren, setzen wir die Funktionalableitungen gleich 0:

$$\forall x, \frac{\delta}{\delta p(x)} \mathcal{L} = \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2 - 1 - \log p(x) = 0. \quad (19.52)$$

Diese Bedingung gibt nun die funktionale Form von $p(x)$ an. Durch algebraische Umstellung der Gleichung erhalten wir

$$p(x) = \exp\left(\lambda_1 + \lambda_2 x + \lambda_3(x - \mu)^2 - 1\right). \quad (19.53)$$

Wir haben niemals direkt angenommen, dass $p(x)$ diese funktionale Form annehmen würde; wir haben den Ausdruck selbst durch analytisches Minimieren eines Funktionals ermittelt. Um das Minimierungsproblem abzuschließen, müssen wir die λ -Werte so wählen, dass all unsere Bedingungen erfüllt sind. Wir können beliebige λ -Werte auswählen, da der Gradient des Lagrange-Funktionals bezüglich der λ -Variablen Null ist, sofern die Bedingungen erfüllt sind. Um alle Bedingungen zu erfüllen, können wir $\lambda_1 = 1 - \log \sigma \sqrt{2\pi}$, $\lambda_2 = 0$ und $\lambda_3 = -\frac{1}{2\sigma^2}$ setzen, um dies zu erhalten:

$$p(x) = \mathcal{N}(x; \mu, \sigma^2). \quad (19.54)$$

Das ist ein Grund für die Verwendung der Normalverteilung, wenn wir die wahre Verteilung nicht kennen. Da die Normalverteilung maximale Entropie aufweist, geben wir mit dieser Annahme die geringstmögliche Struktur vor.

Beim Prüfen der kritischen Punkte des Lagrange-Funktionals auf Entropie haben wir nur einen kritischen Punkt gefunden, in dem die Entropie für eine feste Varianz maximiert wird. Was ist aber mit der Wahrscheinlichkeitsverteilungsfunktion, die die Entropie *minimiert*? Warum haben wir keinen zweiten kritischen Punkt gefunden, der dem Minimum entspricht? Der Grund ist, dass keine bestimmte Funktion minimale Entropie erreicht. Während Funktionen den beiden Punkten $x = \mu + \sigma$ und $x = \mu - \sigma$ eine höhere Wahrscheinlichkeitsdichte auferlegen und allen anderen Werten von x eine geringere, verlieren sie bei Beibehaltung der gewünschten Varianz an Entropie. Allerdings integriert jede Funktion, die eine Masse von exakt Null auf alle bis auf zwei Punkte platziert, nicht zu 1 und ist keine gültige Wahrscheinlichkeitsverteilung. Somit gibt es keine einzelne Wahrscheinlichkeitsverteilungsfunktion für minimale Entropie, ebenso wie es keine einzelne kleinste positive reelle Zahl gibt. Stattdessen können wir sagen, dass es eine Folge von Wahrscheinlichkeitsverteilungen gibt, die dahingehend konvergieren, dass nur zwei Punkte gewichtet werden. Dieser Sonderfall lässt sich als Mischung von Dirac-Verteilungen beschreiben. Da Dirac-Verteilungen nicht durch eine einzelne Wahrscheinlichkeitsverteilungsfunktion beschrieben werden, entspricht keine Dirac-Verteilung oder Mischung von Dirac-Verteilungen einem einzelnen spezifischen Punkt im Funktionsraum. Diese Verteilungen sind somit für unsere Verfahren zum Lösen für einen bestimmten Punkt, in dem die Funktionalableitungen Null sind, unsichtbar. Das stellt eine

Einschränkung des Verfahrens dar. Verteilungen wie die Dirac-Verteilung müssen auf andere Weise gesucht werden, zum Beispiel durch Erraten der Lösung und anschließenden Beweis der Korrektheit.

19.4.3 Stetige latente Variablen

Auch wenn unser graphisches Modell stetige latente Variablen enthält, können wir Variational Inference und Variational Learning durch Maximieren von \mathcal{L} anwenden. Allerdings müssen wir nun die Variationsrechnung beim Maximieren von \mathcal{L} bezüglich $q(\mathbf{h} \mid \mathbf{v})$ einsetzen.

In den meisten Fällen müssen Entwickler die Probleme der Variationsrechnung nicht selbst lösen. Stattdessen gibt es eine allgemeine Gleichung für die Aktualisierung des Molekularfeld-Fixpunktes. Wenn wir die Molekularfeld-Approximation

$$q(\mathbf{h} \mid \mathbf{v}) = \prod_i q(h_i \mid \mathbf{v}) \quad (19.55)$$

vornehmen und $q(h_j \mid \mathbf{v})$ für alle $j \neq i$ festhalten, dann kann das optimale $q(h_i \mid \mathbf{v})$ durch Normalisieren der nicht normalisierten Wahrscheinlichkeitsverteilung

$$\tilde{q}(h_i \mid \mathbf{v}) = \exp\left(\mathbb{E}_{\mathbf{h}_{-i} \sim q(\mathbf{h}_{-i} \mid \mathbf{v})} \log \tilde{p}(\mathbf{v}, \mathbf{h})\right) \quad (19.56)$$

bestimmt werden, sofern p keiner multivariaten Konfiguration von Variablen die Wahrscheinlichkeit 0 zuweist. Durch Ausführen des Erwartungswerts innerhalb der Gleichung ergibt sich die korrekte funktionale Form von $q(h_i \mid \mathbf{v})$. Die direkte Ableitung funktionaler Formen von q mittels Variationsrechnung ist nur erforderlich, wenn Sie eine neue Form des Variational Learnings entwickeln möchten; Gleichung 19.56 ergibt die Molekularfeld-Approximation für beliebige probabilistische Modelle.

Gleichung 19.56 ist eine Fixpunktgleichung, die iterativ wiederholt auf jeden Wert von i angewandt wird, bis die Konvergenz erreicht ist. Allerdings gibt sie noch mehr an. Sie gibt an, welche funktionale Form die optimale Lösung annehmen wird und ob wir mittels Fixpunktgleichungen dorthin gelangen oder nicht. Das bedeutet, dass wir der Gleichung die funktionale Form entnehmen können, aber einige der darin enthaltenen Werte als Parameter betrachten, die wir mit einem beliebigen Optimierungsalgorithmus optimieren können.

Ein Beispiel: Gegeben sei ein einfaches probabilistisches Modell mit latenten Variablen $\mathbf{h} \in \mathbb{R}^2$ und nur einer sichtbaren Variable v . Sei $p(\mathbf{h}) = \mathcal{N}(\mathbf{h}; \mathbf{0}, \mathbf{I})$ und $p(v \mid \mathbf{h}) = \mathcal{N}(v; \mathbf{w}^\top \mathbf{h}; 1)$. Wir könnten dieses Modell tatsächlich durch Ausintegrieren von \mathbf{h} vereinfachen; das Ergebnis ist

einfach eine Normalverteilung über v . Das Modell selbst ist nicht interessant – wir haben es nur konstruiert, um eine einfache Demonstration der Variationsrechnung für die probabilistische Modellierung zu geben.

Die wahre A-posteriori-Verteilung ergibt sich bis zu einer normalisierenden Konstante aus

$$p(\mathbf{h} \mid \mathbf{v}) \quad (19.57)$$

$$\propto p(\mathbf{h}, \mathbf{v}) \quad (19.58)$$

$$= p(h_1)p(h_2)p(\mathbf{v} \mid \mathbf{h}) \quad (19.59)$$

$$\propto \exp\left(-\frac{1}{2} [h_1^2 + h_2^2 + (v - h_1 w_1 - h_2 w_2)^2]\right) \quad (19.60)$$

$$= \exp\left(-\frac{1}{2} [h_1^2 + h_2^2 + v^2 + h_1^2 w_1^2 + h_2^2 w_2^2 - 2vh_1 w_1 - 2vh_2 w_2 + 2h_1 w_1 h_2 w_2]\right). \quad (19.61)$$

Da Terme vorhanden sind, die h_1 und h_2 als Faktoren enthalten, ist klar, dass die wahre A-posteriori-Verteilung nicht in die Faktoren h_1 und h_2 zerlegt werden kann.

Durch Anwenden von Gleichung 19.56 stellen wir fest, dass

$$\tilde{q}(h_1 \mid \mathbf{v}) \quad (19.62)$$

$$= \exp\left(\mathbb{E}_{h_2 \sim q(h_2 \mid \mathbf{v})} \log \tilde{p}(\mathbf{v}, \mathbf{h})\right) \quad (19.63)$$

$$= \exp\left(-\frac{1}{2} \mathbb{E}_{h_2 \sim q(h_2 \mid \mathbf{v})} [h_1^2 + h_2^2 + v^2 + h_1^2 w_1^2 + h_2^2 w_2^2 \quad (19.64)$$

$$- 2vh_1 w_1 - 2vh_2 w_2 + 2h_1 w_1 h_2 w_2]\right) \quad (19.65)$$

ist. Wir sehen also, dass es effektiv lediglich zwei Werte gibt, die wir aus $q(h_2 \mid \mathbf{v})$ ermitteln müssen: $\mathbb{E}_{h_2 \sim q(h_2 \mid \mathbf{v})}[h_2]$ und $\mathbb{E}_{h_2 \sim q(h_2 \mid \mathbf{v})}[h_2^2]$. Schreiben wir diese als $\langle h_2 \rangle$ und $\langle h_2^2 \rangle$, erhalten wir

$$\tilde{q}(h_1 \mid \mathbf{v}) = \exp\left(-\frac{1}{2} [h_1^2 + \langle h_2^2 \rangle + v^2 + h_1^2 w_1^2 + \langle h_2^2 \rangle w_2^2 \quad (19.66)$$

$$- 2vh_1 w_1 - 2v\langle h_2 \rangle w_2 + 2h_1 w_1 \langle h_2 \rangle w_2]\right). \quad (19.67)$$

Es zeigt sich, dass \tilde{q} die funktionale Form einer Normalverteilung hat. Wir können somit schließen, dass $q(\mathbf{h} \mid \mathbf{v}) = \mathcal{N}(\mathbf{h}; \boldsymbol{\mu}, \boldsymbol{\beta}^{-1})$ ist, wobei $\boldsymbol{\mu}$ und die Diagonalmatrix $\boldsymbol{\beta}$ Variationsparameter sind, die wir mit einer beliebigen Methode optimieren können. Es ist wichtig, daran zu denken, dass wir

niemals angenommen haben, dass q eine Normalverteilung ist – die Gauß-Form wurde automatisch mittels der Variationsrechnung zur Maximierung von q bezüglich \mathcal{L} abgeleitet. Derselbe Ansatz für ein anderes Modell könnte zu einer anderen funktionalen Form von q führen.

Bedenken Sie, dass es sich hier nur um einen konstruierten Beispielfall zur Verdeutlichung handelt. Beispiele für echte Anwendungen des Variational Learnings mit stetigen Variablen im Deep-Learning-Kontext finden Sie in *Goodfellow et al.* (2013d).

19.4.4 Interaktionen zwischen Lernen und Inferenz

Die approximative Inferenz als Teil eines Lernalgorithmus beeinflusst den Lernprozess, der wiederum die Genauigkeit des Inferenzalgorithmus beeinflusst.

Vor allem tendiert der Trainingsalgorithmus dazu, das Modell auf eine Weise anzupassen, die dafür sorgt, dass die Approximationsannahmen, die dem Algorithmus zur approximativen Inferenz zugrunde liegen, eher wahr werden. Beim Trainieren der Parameter erhöht das Variational Learning

$$\mathbb{E}_{\mathbf{h} \sim q} \log p(\mathbf{v}, \mathbf{h}). \quad (19.68)$$

Für ein bestimmtes \mathbf{v} nimmt $p(\mathbf{h} | \mathbf{v})$ für Werte von \mathbf{h} , die unter $q(\mathbf{h} | \mathbf{v})$ eine hohe Wahrscheinlichkeit haben, zu. $p(\mathbf{h} | \mathbf{v})$ nimmt für Werte von \mathbf{h} , die unter $q(\mathbf{h} | \mathbf{v})$ eine niedrige Wahrscheinlichkeit haben, ab.

Dieses Verhalten führt dazu, dass aus unseren Approximationsannahmen selbsterfüllende Prophezeiungen werden. Wenn wir ein Modell mit einer unimodalen approximativen A-posteriori-Verteilung trainieren, erhalten wir ein Modell mit einer wahren A-posteriori-Verteilung, die deutlich eher unimodal ist, als wenn wir das Modell mittels exakter Inferenz trainiert hätten.

Die Berechnung des tatsächlichen Schadens, der einem Modell durch die Variationsapproximation auferlegt wird, ist somit sehr schwierig. Es gibt mehrere Methoden zum Schätzen von $\log p(\mathbf{v})$. Wir schätzen $\log p(\mathbf{v}; \boldsymbol{\theta})$ häufig nach dem Trainieren des Modells und stellen fest, dass die Lücke zu $\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, q)$ klein ist. Daraus können wir schließen, dass unsere Variationsapproximation für den speziellen Wert von $\boldsymbol{\theta}$, den wir aus dem Lernprozess gewonnen haben, korrekt ist. Wir dürfen nicht schließen, dass diese Approximation im Allgemeinen korrekt ist oder dass sie dem Lernprozess nur wenig Schaden zugefügt hat. Um das wahre Maß des Schadens zu messen, der aus dieser Approximation entstanden ist, müssten wir $\boldsymbol{\theta}^* = \max_{\boldsymbol{\theta}} \log p(\mathbf{v}; \boldsymbol{\theta})$ kennen. Es ist möglich, dass $\mathcal{L}(\mathbf{v}, \boldsymbol{\theta}, q) \approx \log p(\mathbf{v}; \boldsymbol{\theta})$ und $\log p(\mathbf{v}; \boldsymbol{\theta}) \ll \log p(\mathbf{v}; \boldsymbol{\theta}^*)$

zeitgleich zutreffen. Ist $\max_q \mathcal{L}(\mathbf{v}, \boldsymbol{\theta}^*, q) \ll \log p(\mathbf{v}; \boldsymbol{\theta}^*)$, da $\boldsymbol{\theta}^*$ eine für unsere q -Familie nicht erfassbare, da zu komplizierte, A-posteriori-Verteilung induziert, nähert sich der Lernprozess niemals an $\boldsymbol{\theta}^*$ an. Ein solches Problem lässt sich kaum erkennen, da wir nur sicher wissen, dass es aufgetreten ist, wenn wir einen entsprechend leistungsstarken Lernalgorithmus nutzen, der zu Vergleichszwecken $\boldsymbol{\theta}^*$ ermitteln kann.

19.5 Erlernte approximative Inferenz

Wir haben gesehen, dass die Inferenz eine Art Optimierungsverfahren ist, das den Wert einer Funktion \mathcal{L} erhöht. Die explizite Optimierung über iterative Verfahren wie Fixpunktgleichungen oder die Optimierung auf Gradientenbasis sind häufig sehr aufwendig und zeitraubend. Viele Ansätze für die Inferenz vermeiden diesen Aufwand, indem das Verfahren der approximativen Inferenz erlernt wird. Wir können uns den Optimierungsprozess als Funktion f vorstellen, die eine Eingabe \mathbf{v} einer approximativen Verteilung $q^* = \arg \max_q \mathcal{L}(\mathbf{v}, q)$ zuordnet. Sobald wir uns den mehrstufigen iterativen Optimierungsprozess als einfache Funktion vorstellen, können wir ihn mit einem neuronalen Netz approximieren, das eine Approximation $\hat{f}(\mathbf{v}; \boldsymbol{\theta})$ vornimmt.

19.5.1 Wake-Sleep

Eine der Hauptschwierigkeiten beim Trainieren eines Modells zum Schließen auf \mathbf{h} aus \mathbf{v} besteht darin, dass wir keine überwachte Trainingsdatenmenge haben, mit der wir das Training durchführen können. Wenn wir \mathbf{v} kennen, wissen wir nichts über das passende \mathbf{h} . Die Zuordnung von \mathbf{v} zu \mathbf{h} ist von der Wahl der Modellfamilie abhängig und entwickelt sich im Rahmen des Lernprozesses mit verändertem $\boldsymbol{\theta}$. Der Wake-Sleep-Algorithmus (*Hinton et al., 1995b; Frey et al., 1996*) löst dieses Problem, indem Stichproben für \mathbf{h} und \mathbf{v} aus der Modellverteilung gezogen werden. In einem gerichteten Modell ist dies zum Beispiel recht einfach mittels Ancestral Samplings (auf Vorfahren/Abstammung basierendes Stichprobenverfahren) beginnend mit \mathbf{h} und mit \mathbf{v} endend möglich. Das Inferenznetz kann dann für eine umgekehrte Zuordnung (engl. *reverse mapping*) trainiert werden, also die Vorhersage, welches \mathbf{h} zum vorliegenden \mathbf{v} geführt hat. Der wesentliche Nachteil dabei ist, dass wir das Inferenznetz nur für Werte von \mathbf{v} trainieren können, die unter dem Modell eine hohe Wahrscheinlichkeit aufweisen. Zu Beginn des Lernprozesses ähnelt die Modellverteilung der Datenverteilung nicht, sodass

das Inferenznetz keine Möglichkeit hat, mithilfe von Stichproben zu lernen, die den Daten ähneln.

In Abschnitt 18.2 haben Sie erfahren, dass eine mögliche Erklärung für die Rolle des Traumschlafs bei Menschen und Tieren die ist, dass Träume Stichproben der negativen Phase liefern, mit denen Monte-Carlo-Trainingsalgorithmen den negativen Gradienten der Log-Partitionsfunktion ungerichteter Modelle approximieren. Eine andere mögliche Erklärung für das Träumen ist, dass es Stichproben aus $p(\mathbf{h}, \mathbf{v})$ liefert, mit denen ein Inferenznetz zur Vorhersage von \mathbf{h} aus \mathbf{v} trainiert werden kann. In gewisser Weise ist diese Erklärung zufriedenstellender als die der Partitionsfunktion. Monte-Carlo-Algorithmen funktionieren im Allgemeinen nicht gut, wenn sie zwischen einigen Schritten ausschließlich der positiven Phase des Gradienten im Wechsel mit einigen Schritten ausschließlich der negativen Phase des Gradienten ausgeführt werden. Menschen und Tiere sind normalerweise mehrere Stunden am Stück wach, bevor sie wiederum mehrere Stunden durchgängig schlafen. Es ist nicht offensichtlich, wie diese zeitliche Einteilung beim Monte-Carlo-Training eines ungerichteten Modells hilfreich sein könnte. Lernalgorithmen, die auf dem Maximieren von \mathcal{L} beruhen, können jedoch längere Zeit zum Verbessern von q und anschließend längere Zeit zum Verbessern von $\boldsymbol{\theta}$ laufen. Wenn biologische Träume die Aufgabe haben, Netze für die Vorhersage von q zu trainieren, erklärt dies, wie Tiere mehrere Stunden wach bleiben (je länger die Wachphase, desto größer die Lücke zwischen \mathcal{L} und $\log p(\mathbf{v})$, aber \mathcal{L} bleibt eine untere Schranke) bzw. mehrere Stunden schlafen können (das generative Modell selbst wird während des Schlafs nicht verändert), ohne ihre internen Modelle zu beschädigen. Natürlich sind diese Vorstellungen rein spekulativ und es gibt keine echten Beweise, die nahelegen, dass das Träumen überhaupt einem dieser Zwecke dient. Das Träumen kann auch dem Reinforcement Learning und nicht der probabilistischen Modellierung dienen, indem synthetische Erfahrungen aus dem Übergangsmodell des Tieres gezogen werden, mit denen die Policy des Tieres trainiert wird. Oder vielleicht dient der Schlaf einem ganz anderen Zweck, den die Forschungsgemeinde des Machine-Learnings noch nicht in Betracht gezogen hat.

19.5.2 Andere Formen erlernter Inferenz

Das Verfahren der erlernten approximativen Inferenz wurde auch auf andere Modelle übertragen. *Salakhutdinov und Larochelle* (2010) haben gezeigt, dass ein einzelner Durchlauf in einem erlernten Inferenznetz zu einer schnelleren Inferenz als die schrittweisen Molekularfeld-Fixpunktgleichungen in einer

DBM führen können. Das Trainingsverfahren basiert auf dem Ausführen des Inferenznetzes, woraufhin ein Molekularfeld-Schritt zum Verbessern der Schätzwerte ausgeführt und das Inferenznetz so trainiert wird, dass es diesen verfeinerten Schätzwert anstelle des ursprünglichen Schätzwerts ausgibt.

Wir haben in Abschnitt 14.8 bereits gesehen, dass das Modell mit prädiktiver dünnbesetzter Zerlegung ein flaches Encodernetz darauf trainiert, Sparse Coding für die Eingabe vorherzusagen. Das lässt sich als Hybrid zwischen Autoencoder und Sparse Coding betrachten. Es ist möglich, probabilistische Semantiken für das Modell zu bestimmen, unter denen der Encoder quasi eine erlernte approximative MAP-Inferenz ausführt. Aufgrund des flachen Encoders ist die PSD nicht in der Lage, die Art Wettstreit zwischen Einheiten zu implementieren, die wir bei der Molekularfeld-Inferenz gefunden haben. Allerdings lässt sich dieser Umstand durch Trainieren eines tiefen Encoders beheben, der die erlernte approximative Inferenz ausführt, ähnlich der ISTA-Technik (*Gregor und LeCun*, 2010b).

Die erlernte approximative Inferenz ist seit Kurzem einer der vorherrschenden Ansätze in der generativen Modellierung, nämlich in Form des VAEs (Variational Autoencoders) (*Kingma*, 2013; *Rezende et al.*, 2014). Dieser elegante Ansatz benötigt keine für das Inferenznetz konstruierten expliziten Ziele. Stattdessen wird das Inferenznetz nur dazu genutzt, \mathcal{L} zu definieren. Anschließend werden die Parameter des Inferenznetzes angepasst, um \mathcal{L} zu erhöhen. Dieses Modell wird in Abschnitt 20.10.3 genauer beschrieben.

Mithilfe der approximativen Inferenz lässt sich eine Vielzahl von Modelle trainieren und verwenden. Viele davon lernen Sie im nächsten Kapitel genauer kennen.

20

Tiefe generative Modelle

In diesem Kapitel stellen wir einige der besonderen generativen Modelle vor, die mithilfe der in den Kapiteln 16–19 dargestellten Verfahren erstellt und trainiert werden können. All diese Modelle stellen auf die ein oder andere Art Wahrscheinlichkeitsverteilungen über mehrere Variablen dar. Einige erlauben die explizite Berechnung der Wahrscheinlichkeitsverteilungsfunktion. Mit anderen ist dies nicht möglich, dafür unterstützen sie Operationen, die ein implizites Wissen über die Funktion voraussetzen, zum Beispiel für das Ziehen von Stichproben aus der Verteilung. Einige dieser Modelle sind strukturierte probabilistische Modelle, die mittels von Graphen und Faktoren abgebildet werden. Dafür wird die in Kapitel 16 vorgestellte Darstellungsform mit graphischen Modellen verwendet. Andere wiederum lassen sich mit Faktoren nur schwerlich beschreiben, stellen aber trotzdem Wahrscheinlichkeitsverteilungen dar.

20.1 Boltzmann-Maschinen

Boltzmann-Maschinen wurden ursprünglich als allgemeiner »konkurrenzistischer« Ansatz zum Erlernen beliebiger Wahrscheinlichkeitsverteilungen über binäre Vektoren vorgestellt (*Fahlman et al.*, 1983; *Ackley et al.*, 1985; *Hinton et al.*, 1984; *Hinton und Sejnowski*, 1986). Allerdings haben Varianten der Boltzmann-Maschine, die noch weitere Arten von Variablen enthalten, die Beliebtheit des Originals längst hinter sich gelassen. In diesem Abschnitt stellen wir die binäre Boltzmann-Maschine kurz vor und behandeln die Fragen, die beim Trainieren und Durchführen von Inferenz in dem Modell auftreten.

Wir definieren die Boltzmann-Maschine über einem d -dimensionalen binären Zufallsvektor $\mathbf{x} \in \{0, 1\}^d$. Die Boltzmann-Maschine ist ein energiebasiertes Modell (Abschnitt 16.2.4), d. h., wir definieren die multivariate Verteilung mithilfe einer Energiefunktion:

$$P(\mathbf{x}) = \frac{\exp(-E(\mathbf{x}))}{Z}, \quad (20.1)$$

wobei $E(\mathbf{x})$ die Energiefunktion ist und Z die Partitionsfunktion, die für $\sum_{\mathbf{x}} P(\mathbf{x}) = 1$ sorgt. Die Energiefunktion der Boltzmann-Maschine ergibt sich aus

$$E(\mathbf{x}) = -\mathbf{x}^\top \mathbf{U}\mathbf{x} - \mathbf{b}^\top \mathbf{x}, \quad (20.2)$$

wobei \mathbf{U} die »Gewichtungsmatrix« der Modellparameter und \mathbf{b} der Vektor der Verzerrungsparameter (engl. *bias parameters*) ist.

Im Fall von Boltzmann-Maschinen steht uns eine Menge mit Trainingsbeispielen zur Verfügung, die alle n -dimensional sind. Gleichung 20.1 beschreibt die multivariate Verteilung über die beobachteten Variablen. Zwar kann das in der Praxis leicht umgesetzt werden, aber die möglichen Interaktionen zwischen den beobachteten Variablen werden dabei auf jene eingeschränkt, die durch die Gewichtungsmatrix beschrieben werden. Vor allem bedeutet dies, dass die Wahrscheinlichkeit für die Aktivierung einer Einheit sich aus einem linearen Modell (logistische Regression) aufgrund der Werte der anderen Einheiten ergibt.

Die Boltzmann-Maschine ist leistungsstärker, wenn nicht alle Variablen beobachtet werden. In diesem Fall können die latenten Variablen ähnlich den verdeckten Einheiten in einem mehrschichtigen Perzepron agieren und Interaktionen höherer Ordnung zwischen den sichtbaren Einheiten modellieren. Die Umwandlung von logistischer Regression in ein MLP durch das Hinzufügen verdeckter Einheiten führt dazu, dass das MLP (mehrschichtiges Perzepron) zu einem universellen Funktionsapproximator wird. Genauso ist auch die Boltzmann-Maschine mit verdeckten Einheiten nicht länger auf das Modellieren linearer Beziehungen zwischen Variablen eingeschränkt. Stattdessen wird sie zu einem universellen Approximator der Wahrscheinlichkeitsfunktionen über diskrete Variablen (*Le Roux und Bengio, 2008*).

Formal zerlegen wir die Einheiten \mathbf{x} in zwei Teilmengen, nämlich die sichtbaren Einheiten \mathbf{v} und die latenten (oder verdeckten) Einheiten \mathbf{h} . Die Energiefunktion wird zu

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{v}^\top \mathbf{R}\mathbf{v} - \mathbf{v}^\top \mathbf{W}\mathbf{h} - \mathbf{h}^\top \mathbf{S}\mathbf{h} - \mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h}. \quad (20.3)$$

Boltzmann Machine Learning Lernalgorithmen für Boltzmann-Maschinen basieren normalerweise auf der Maximum Likelihood. Alle Boltzmann-Maschinen haben eine nicht effizient berechenbare (engl. *intractable*) Partitionsfunktion, sodass der Maximum-Likelihood-Gradient anhand der in Kapitel 18 beschriebenen Verfahren approximiert werden muss.

Eine interessante Eigenschaft der Boltzmann-Maschinen beim Trainieren anhand von Lernregeln auf Basis der Maximum Likelihood ist, dass die Anpassung für ein bestimmtes Gewicht, das zwei Einheiten miteinander verbindet, nur von den statistischen Größen dieser beiden Einheiten abhängig ist, die unter unterschiedlichen Verteilungen erfasst wurden: $P_{\text{model}}(\mathbf{v})$ und $\hat{P}_{\text{data}}(\mathbf{v})P_{\text{model}}(\mathbf{h} \mid \mathbf{v})$. Der Rest des Netzes ist daran beteiligt, diese statistischen Größen zu formen, aber das Gewicht kann ohne Kenntnis über den Rest des Netzes oder die Herkunft der statistischen Größen angepasst werden. Die Lernregel ist also »lokal«, wodurch das Boltzmann Machine Learning eine gewisse biologische Begründung erhält. Wenn jedes Neuron eine Zufallsvariable in einer Boltzmann-Maschine wäre, ist es vorstellbar, dass die Axone und Dendriten, die zwei Zufallsvariablen miteinander verbinden, rein durch Beobachten der Feuerungsmuster der Zellen, die sie tatsächlich berühren, lernen könnten. So wird die Verbindung von zwei Einheiten, die häufig gemeinsam aktiviert werden, in der positiven Phase verstärkt. Dies ist ein Beispiel für eine hebbische Lernregel (Hebb, 1949), die im Englischen oft mit dem Merksatz »fire together, wire together« zusammengefasst wird. Hebbische Lernregeln gehören zu den ältesten hypothetisch gebildeten Erklärungen für das Lernen in biologischen Systemen und sie sind auch heute noch relevant (Giudice et al., 2009).

Andere Lernalgorithmen, die mehr Informationen als lokale statistische Größen nutzen, scheinen von uns zu erfordern, dass wir von der Existenz deutlich komplexerer Mechanismen ausgehen. Damit das Gehirn die Backpropagation in einem mehrschichtigen Perceptron implementieren kann, scheint es beispielsweise notwendig zu sein, dass es ein zweites Kommunikationsnetz unterhält, über das die Gradienteninformationen rückwärts durch das Netz übermittelt werden. Es wurden Vorschläge für biologisch plausible Implementierungen (und Approximationen) der Backpropagation unterbreitet (Hinton, 2007a; Bengio, 2015), deren Beweis aber noch aussteht. Bengio (2015) verknüpft die Backpropagation von Gradienten mit der Inferenz in energiebasierten Modellen ähnlich der Boltzmann-Maschine (aber mit stetigen latenten Variablen).

Die negative Phase beim Boltzmann Machine Learning lässt sich aus biologischer Sicht schwieriger erklären. Wie wir in Abschnitt 18.2 argumentieren, könnte der Traumschlaf eine Art Stichprobenentnahme der negativen

Phase (engl. *negative phase sampling*) darstellen. Dieses Konzept ist jedoch stark spekulativ.

20.2 Restricted Boltzmann Machines

Ursprünglich unter der Bezeichnung **Harmonium** (*Smolensky*, 1986) erfunden, gehören Restricted Boltzmann Machines (RBMs, eingeschränkte Boltzmann-Maschinen) zu den Standardbausteinen tiefer probabilistischer Modelle. Eine kurze Beschreibung der RBMs finden Sie in Abschnitt 16.7.1. Wir greifen hier die wesentlichen Punkte daraus auf und gehen auf Einzelheiten ein. RBMs sind ungerichtete probabilistische graphische Modelle, die eine Schicht mit beobachtbaren Variablen und eine einzige Schicht mit latenten Variablen enthalten. RBMs können (übereinander) gestapelt werden, um tiefere Modelle zu erhalten. Abbildung 20.1 zeigt einige Beispiele. Abbildung 20.1a zeigt die Graphenstruktur der RBM selbst. Es handelt sich um einen zweiteiligen Graphen, in dem keine Verbindungen zwischen Variablen in der beobachteten Schicht oder zwischen Einheiten in der latenten Schicht erlaubt sind.

Wir beginnen mit der binären Version der RBM; später sehen Sie noch, dass es Erweiterungen für andere Arten sichtbarer und verdeckter Einheiten gibt.

Formal betrachtet besteht die beobachtete Schicht aus einer Menge mit n_v binären Zufallsvariablen, die wir in ihrer Gesamtheit mit dem Vektor \mathbf{v} bezeichnen. Die latente (oder verdeckte) Schicht mit n_h binären Zufallsvariablen bezeichnen wir als \mathbf{h} .

Wie die allgemeine Boltzmann-Maschine ist auch die RBM ein energiebasiertes Modell mit einer durch seine Energiefunktion spezifizierten multivariaten Verteilung:

$$P(\mathbf{v} = \mathbf{v}, \mathbf{h} = \mathbf{h}) = \frac{1}{Z} \exp(-E(\mathbf{v}, \mathbf{h})). \quad (20.4)$$

Die Energiefunktion einer RBM ergibt sich aus

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h} - \mathbf{v}^\top \mathbf{W} \mathbf{h}, \quad (20.5)$$

und Z ist die normalisierende Konstante, auch Partitionsfunktion genannt:

$$Z = \sum_{\mathbf{v}} \sum_{\mathbf{h}} \exp \{-E(\mathbf{v}, \mathbf{h})\}. \quad (20.6)$$

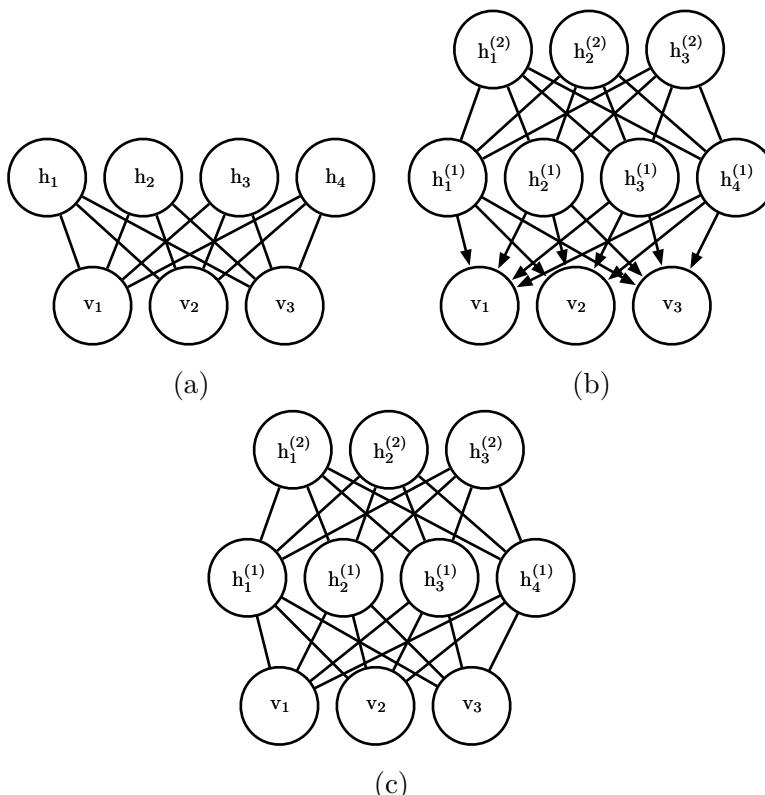


Abbildung 20.1: Beispiele für Modelle, die mit RBMs konstruiert werden können. (a) Die RBM selbst ist ein ungerichtetes graphisches Modell auf Basis eines zweiteiligen Graphen, der in einem Teil sichtbare und im anderen Teil verdeckte Einheiten aufweist. Es gibt keine Verbindungen zwischen den sichtbaren Einheiten und auch keine Verbindungen zwischen den verdeckten Einheiten. Normalerweise ist jede sichtbare Einheit mit jeder verdeckten Einheit verbunden, aber es ist auch möglich, RBMs zu konstruieren, die geringfügig miteinander verbunden sind – zum Beispiel gefaltete RBMs. (b) Ein Deep-Belief-Netz (DBN) ist ein graphisches Hybrid-Modell, das gerichtete und ungerichtete Verbindungen enthält. Wie eine RBM gibt es innerhalb der Schichten keine Verbindungen. Allerdings weist ein DBN mehrere verdeckte Schichten und somit Verbindungen zwischen verdeckten Einheiten in unterschiedlichen Schichten auf. Alle für das Deep-Belief-Netz erforderlichen lokalen bedingten Wahrscheinlichkeitsverteilungen werden direkt aus den lokalen bedingten Wahrscheinlichkeitsverteilungen der RBMs, aus denen es besteht, kopiert. Wir könnten das Deep-Belief-Netz auch als vollständig ungerichteten Graphen darstellen, der dann allerdings Verbindungen innerhalb der Schichten aufweisen müsste, um die Abhängigkeiten zwischen den Eltern zu erfassen. (c) Eine DBM (Deep Boltzmann Machine) ist ein ungerichtetes graphisches Modell mit mehreren Schichten latenter Variablen. Wie RBMs und DBNs enthalten auch DBMs keine Verbindungen innerhalb einer Schicht. DBMs sind nicht so eng wie DBNs mit RBMs verwandt. Beim Initialisieren einer DBM aus einem RBM-Stapel müssen die RBM-Parameter geringfügig angepasst werden. Einige Arten von DBMs können ohne vorheriges Training einer Reihe von RBMs trainiert werden.

Aus der Definition der Partitionsfunktion Z wird klar, dass die naive Methode zur Berechnung von Z (abschließendes Aufsummieren über alle Zustände) rechnerisch nicht effizient durchführbar sein könnte, es sei denn, ein intelligent konzipierter Algorithmus könnte die Regelmäßigkeiten in der Wahrscheinlichkeitsverteilung nutzen, um Z schneller zu bestimmen. Für den Fall der RBM haben *Long und Servedio* (2010) formal bewiesen, dass die Partitionsfunktion Z nicht effizient berechenbar ist. Die nicht effizient berechenbare Partitionsfunktion Z impliziert, dass die normalisierte multivariate Verteilung $P(\mathbf{v})$ ebenfalls nicht effizient berechnet werden kann.

20.2.1 Bedingte Verteilungen

Obwohl $P(\mathbf{v})$ nicht effizient berechenbar ist, weist die zweiteilige Graphenstruktur der RBM für die bedingten Verteilungen $P(\mathbf{h} \mid \mathbf{v})$ und $P(\mathbf{v} \mid \mathbf{h})$ eine Besonderheit auf: Sie sind faktoriell und relativ einfach hinsichtlich Berechnung und Stichprobenentnahme.

Die Ableitung der bedingten Verteilungen aus der multivariaten Verteilung ist ganz einfach:

$$P(\mathbf{h} \mid \mathbf{v}) = \frac{P(\mathbf{h}, \mathbf{v})}{P(\mathbf{v})} \quad (20.7)$$

$$= \frac{1}{P(\mathbf{v})} \frac{1}{Z} \exp \left\{ \mathbf{b}^\top \mathbf{v} + \mathbf{c}^\top \mathbf{h} + \mathbf{v}^\top \mathbf{W} \mathbf{h} \right\} \quad (20.8)$$

$$= \frac{1}{Z'} \exp \left\{ \mathbf{c}^\top \mathbf{h} + \mathbf{v}^\top \mathbf{W} \mathbf{h} \right\} \quad (20.9)$$

$$= \frac{1}{Z'} \exp \left\{ \sum_{j=1}^{n_h} c_j h_j + \sum_{j=1}^{n_h} \mathbf{v}^\top \mathbf{W}_{:,j} \mathbf{h}_j \right\} \quad (20.10)$$

$$= \frac{1}{Z'} \prod_{j=1}^{n_h} \exp \left\{ c_j h_j + \mathbf{v}^\top \mathbf{W}_{:,j} \mathbf{h}_j \right\}. \quad (20.11)$$

Da wir eine Konditionierung für die sichtbaren Einheiten \mathbf{v} vornehmen, können wir diese bezüglich der Verteilung $P(\mathbf{h} \mid \mathbf{v})$ als konstant betrachten. Der faktorielle Charakter der bedingten Verteilung $P(\mathbf{h} \mid \mathbf{v})$ folgt direkt aus unserer Fähigkeit, die multivariate Verteilung über den Vektor \mathbf{h} als Produkt der (nicht normalisierten) Verteilungen über die einzelnen Elemente, h_j , auszudrücken. Wir müssen nun nur noch die Verteilungen über die einzelnen binären h_j normalisieren.

$$P(h_j = 1 \mid \mathbf{v}) = \frac{\tilde{P}(h_j = 1 \mid \mathbf{v})}{\tilde{P}(h_j = 0 \mid \mathbf{v}) + \tilde{P}(h_j = 1 \mid \mathbf{v})} \quad (20.12)$$

$$= \frac{\exp\{c_j + \mathbf{v}^\top \mathbf{W}_{:,j}\}}{\exp\{0\} + \exp\{c_j + \mathbf{v}^\top \mathbf{W}_{:,j}\}} \quad (20.13)$$

$$= \sigma(c_j + \mathbf{v}^\top \mathbf{W}_{:,j}). \quad (20.14)$$

Jetzt können wir die vollständige bedingte Verteilung über die verdeckte Schicht als faktorielle Verteilung ausdrücken:

$$P(\mathbf{h} | \mathbf{v}) = \prod_{j=1}^{n_h} \sigma((2\mathbf{h} - 1) \odot (\mathbf{c} + \mathbf{W}^\top \mathbf{v}))_j. \quad (20.15)$$

Eine ähnliche Herleitung zeigt, dass die andere relevante Verteilung $P(\mathbf{v} | \mathbf{h})$ ebenfalls eine faktorielle Verteilung ist:

$$P(\mathbf{v} | \mathbf{h}) = \prod_{i=1}^{n_v} \sigma((2\mathbf{v} - 1) \odot (\mathbf{b} + \mathbf{W}\mathbf{h}))_i. \quad (20.16)$$

20.2.2 Trainieren von RBMs

Da die RBM eine effiziente Berechnung und Differentiation von $\tilde{P}(\mathbf{v})$ sowie ein effizientes MCMC-Stichprobenverfahren in Form eines Block-Gibbs-Samplings zulässt, kann sie problemlos mit einer der Methoden aus Kapitel 18 zum Trainieren von Modellen mit nicht effizient berechenbaren Partitionsfunktionen trainiert werden. Dazu gehören kontrastive Divergenz, SML (persistente kontrastive Divergenz), Ratio Matching usw. Gegenüber anderen ungerichteten Modellen im Deep Learning ist das Trainieren der RBM relativ einfach, da wir $P(\mathbf{h} | \mathbf{v})$ in geschlossener Form exakt berechnen können. Einige andere tiefe Modelle wie die DBM kombinieren die Schwierigkeit einer nicht effizient berechenbaren Partitionsfunktion mit der Schwierigkeit einer nicht effizient durchführbaren Inferenz.

20.3 Deep-Belief-Netze

Deep-Belief-Netze (DBNs) gehörten zu den ersten Modellen ohne Faltung, mit denen tiefe Architekturen erfolgreich trainiert werden konnten (*Hinton et al.*, 2006; *Hinton*, 2007b). Die Einführung der Deep-Belief-Netze in 2006 führte zur aktuellen Renaissance des Deep Learnings. Vor dem Aufkommen der Deep-Belief-Netze galt die Optimierung tiefer Modelle als zu schwierig. Kernel-Maschinen mit konvexen Zielfunktionen dominierten die Forschungslandschaft. Deep-Belief-Netze haben gezeigt, dass tiefe Architekturen erfolgreich sein können. Der Beweis war erbracht, als diese die

Support Vector Machines (SVMs) auf Kernel-Basis beim MNIST-Datensatz übertrafen (*Hinton et al.*, 2006). Heute werden Deep-Belief-Netze kaum noch verwendet – das gilt sogar im Vergleich zu anderen unüberwachten oder generativen Lernalgorithmen. Aber sie werden weiterhin für ihre wichtige Rolle in der Geschichte des Deep Learnings geschätzt.

Deep-Belief-Netze sind generative Modelle mit mehreren Schichten von latenten Variablen. Die latenten Variablen sind üblicherweise binär, die sichtbaren Einheiten können dagegen binär oder reell sein. Es gibt keine Verbindungen innerhalb einer Schicht. Normalerweise ist jede Einheit einer Schicht mit allen Einheiten in jeder benachbarten Schicht verbunden, aber auch dünner besetzte verbundene DBNs sind denkbar. Die Verbindung zwischen den obersten beiden Schichten sind ungerichtet. Die Verbindungen zwischen allen anderen Schichten sind gerichtet, wobei die Pfeile zu der Schicht weisen, die den Daten näher ist. Abbildung 20.1b enthält ein Beispiel.

Ein DBN mit l verdeckten Schichten enthält l Gewichtungsmatrizen: $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(l)}$. Es enthält auch $l + 1$ Verzerrungsvektoren $\mathbf{b}^{(0)}, \dots, \mathbf{b}^{(l)}$, wobei $\mathbf{b}^{(0)}$ die Verzerrungen für die sichtbare Schicht angibt. Die durch das DBN repräsentierte Wahrscheinlichkeitsverteilung ergibt sich aus

$$P(\mathbf{h}^{(l)}, \mathbf{h}^{(l-1)}) \propto \exp\left(\mathbf{b}^{(l)\top} \mathbf{h}^{(l)} + \mathbf{b}^{(l-1)\top} \mathbf{h}^{(l-1)} + \mathbf{h}^{(l-1)\top} \mathbf{W}^{(l)} \mathbf{h}^{(l)}\right), \quad (20.17)$$

$$P(h_i^{(k)} = 1 \mid \mathbf{h}^{(k+1)}) = \sigma\left(b_i^{(k)} + \mathbf{W}_{:,i}^{(k+1)\top} \mathbf{h}^{(k+1)}\right) \forall i, \forall k \in 1, \dots, l-2, \quad (20.18)$$

$$P(v_i = 1 \mid \mathbf{h}^{(1)}) = \sigma\left(b_i^{(0)} + \mathbf{W}_{:,i}^{(1)\top} \mathbf{h}^{(1)}\right) \forall i. \quad (20.19)$$

Für reellwertige sichtbare Einheiten setzen wir für

$$\mathbf{v} \sim \mathcal{N}\left(\mathbf{v}; \mathbf{b}^{(0)} + \mathbf{W}^{(1)\top} \mathbf{h}^{(1)}, \boldsymbol{\beta}^{-1}\right) \quad (20.20)$$

aus Gründen der effizienten Berechenbarkeit $\boldsymbol{\beta}$ als Diagonalmatrix ein. Generalisierungen für andere sichtbare Einheiten der Familien von Exponentialfunktionen sind, zumindest theoretisch, recht einfach. Bei einem DBN mit nur einer verdeckten Schicht handelt es sich um eine RBM.

Um Stichproben aus einem DBN zu generieren, führen wir zunächst einige Schritte lang ein Gibbs-Sampling für die obersten beiden verdeckten Schichten durch. Diese Phase zieht im Wesentlichen eine Stichprobe aus der RBM, die durch die obersten beiden verdeckten Schichten definiert wird. Anschließend können wir das Ancestral Sampling in einem Durchgang für den Rest des Modells verwenden, um eine Stichprobe aus den sichtbaren Einheiten zu ziehen.

Deep-Belief-Netze weisen viele der Probleme auf, die auch in gerichteten und ungerichteten Modellen zu finden sind.

Die Inferenz in einem Deep-Belief-Netz ist aufgrund des Explaining-Away-Effekts in jeder gerichteten Schicht und aufgrund der Interaktion zwischen den beiden verdeckten Schichten mit ungerichteten Verbindungen nicht effizient durchführbar. Die Bestimmung oder Maximierung der normalen ELBO (Evidence Lower Bound) für die Log-Likelihood ist ebenfalls nicht effizient möglich, da die ELBO den Erwartungswert der Cliques nutzt, deren Größe gleich der Breite des Netzes ist.

Die Bestimmung oder Maximierung der Log-Likelihood erfordert nicht nur die Lösung des Problems der nicht effizient durchführbaren Inferenz zum Entfernen der latenten Variablen, sondern auch des Problems einer nicht effizient berechenbaren Partitionsfunktion innerhalb des ungerichteten Modells der oberen beiden Schichten.

Um ein Deep-Belief-Netz zu trainieren, trainieren wir zunächst eine RBM, um $\mathbb{E}_{\mathbf{v} \sim p_{\text{data}}} \log p(\mathbf{v})$ mittels kontrastiver Divergenz oder stochastischer Maximum Likelihood (SML) zu maximieren. Die Parameter der RBM definieren dann die Parameter für die erste Schicht des DBN. Dann wird eine zweite RBM trainiert, um

$$\mathbb{E}_{\mathbf{v} \sim p_{\text{data}}} \mathbb{E}_{\mathbf{h}^{(1)} \sim p^{(1)}(\mathbf{h}^{(1)} | \mathbf{v})} \log p^{(2)}(\mathbf{h}^{(1)}) \quad (20.21)$$

näherungsweise zu maximieren, wobei $p^{(1)}$ die durch die erste RBM repräsentierte Wahrscheinlichkeitsverteilung ist und $p^{(2)}$ die durch die zweite RBM repräsentierte Wahrscheinlichkeitsverteilung. Anders ausgedrückt: Die zweite RBM wird darauf trainiert, die Verteilung zu modellieren, die definiert ist durch die Auswahl der verdeckten Einheiten der ersten RBM (die erste RBM ist datengesteuert). Dieses Verfahren kann unendlich oft wiederholt werden, um beliebig viele Schichten zum DBN hinzuzufügen; dabei modelliert jede neue RBM die Stichproben der vorherigen. Jede RBM definiert eine weitere DBN-Schicht. Dieses Verfahren lässt sich als Erhöhen einer ELBO der Log-Likelihood für die Daten unter dem DBN erklären (*Hinton et al., 2006*).

In den meisten Anwendungen wird kein Versuch unternommen, das DBN als Ganzes zu trainieren, nachdem das schichtweise Verfahren mit Greedy-Algorithmen abgeschlossen ist. Allerdings ist es möglich, über den Wake-Sleep-Algorithmus eine generative Feinabstimmung durchzuführen.

Das trainierte DBN kann direkt als generatives Modell verwendet werden. Aber DBNs sind in erster Linie interessant, weil sie die Möglichkeit bieten,

Klassifizierungsmodelle zu verbessern. Wir können mit den Gewichten des DBNs ein mehrschichtiges Perzeptron definieren:

$$\mathbf{h}^{(1)} = \sigma \left(b^{(1)} + \mathbf{v}^\top \mathbf{W}^{(1)} \right), \quad (20.22)$$

$$\mathbf{h}^{(l)} = \sigma \left(b_i^{(l)} + \mathbf{h}^{(l-1)\top} \mathbf{W}^{(l)} \right) \forall l \in 2, \dots, m. \quad (20.23)$$

Nachdem wir dieses mehrschichtige Perzeptron mit den Gewichten und Verzerrungen initialisiert haben, die im Rahmen des generativen DBN-Trainings erlernt wurden, können wir es für eine Klassifizierungsaufgabe trainieren. Dieses zusätzliche Training des mehrschichtigen Perzeptrons ist ein Beispiel für eine diskriminative Feinabstimmung.

Diese spezielle Wahl des mehrschichtigen Perzeptrons ist auf eine gewisse Weise willkürlich, verglichen mit vielen der Inferenzgleichungen aus Kapitel 19, die von den Grundprinzipien abgeleitet sind. Dieses mehrschichtige Perzeptron ist eine heuristische Wahl, die in der Praxis gute Ergebnisse zu liefern scheint und in der Literatur häufig erwähnt wird. Viele approximative Inferenzverfahren bieten unter bestimmten Bedingungen die Möglichkeit, eine maximal *enge* ELBO für die Log-Likelihood zu finden. Man kann eine ELBO für die Log-Likelihood mithilfe der Erwartungswerte verdeckter Einheiten, die vom mehrschichtigen Perzeptron des DBNs definiert werden, konstruieren, aber das gilt für *jede* Wahrscheinlichkeitsverteilung über die verdeckten Einheiten, sodass es keinen Grund dafür gibt anzunehmen, dass dieses mehrschichtige Perzeptron eine besonders enge Schranke bietet. So ignoriert das mehrschichtige Perzeptron viele wichtige Interaktionen im graphischen Modell des DBNs. Das mehrschichtige Perzeptron propagiert Informationen von den sichtbaren Einheiten aufwärts zu den tiefsten verdeckten Einheiten, aber es propagiert keine Informationen abwärts oder seitwärts. Das graphische Modell des DBNs weist Explaining-Away-Interaktionen zwischen allen verdeckten Einheiten in derselben Schicht sowie in den Top-Down-Interaktionen zwischen Schichten auf.

Zwar ist die Log-Likelihood eines DBNs nicht effizient berechenbar, aber die Approximation mittels Annealed Importance Samplings (AIS) ist möglich (*Salakhutdinov und Murray, 2008*). So kann ihre Qualität als generatives Modell beurteilt werden.

Der Begriff »Deep-Belief-Netz« wird häufig falsch verwendet und beinhaltet dann alle Arten tiefer neuronaler Netze, sogar solche ohne die Semantik latenter Variablen. Er sollte aber ausschließlich für Modelle verwendet werden, die in der tiefsten Schicht ungerichtete Verbindungen und in allen anderen aufeinanderfolgenden Schichtpaaren abwärtsweisende gerichtete Verbindungen aufweisen.

Der Begriff kann auch Verwirrung stiften, da rein gerichtete Modelle manchmal ebenfalls als »Belief-Netz« bezeichnet werden, wohingegen Deep-Belief-Netze eine ungerichtete Schicht enthalten. Das Akronym DBN für Deep-Belief-Netze wird außerdem auch für dynamische Bayes-Netze verwendet (*Dean und Kanazawa, 1989*), also Bayes-Netze zur Darstellung von Markow-Ketten.

20.4 Deep Boltzmann Machines

Eine **Deep Boltzmann Machine** (DBM, dt. *tiefe Boltzmann-Maschine*) (*Salakhutdinov und Hinton, 2009a*) ist ebenfalls ein tiefes generatives Modell. Anders als beim Deep-Belief-Netz (DBN) handelt es sich dabei um ein gänzlich ungerichtetes Modell. Anders als die RBM weist die DBM mehrere Schichten mit latenten Variablen auf (bei der RBM ist es nur eine). Aber wie bei der RBM sind alle Variablen innerhalb einer Schicht untereinander unabhängig und durch die Variablen in den benachbarten Schichten bestimmt. Abbildung 20.2 zeigt die Graphenstruktur. DBMs wurden für viele Aufgaben eingesetzt, darunter die Dokumentenmodellierung (*Srivastava et al., 2013*).

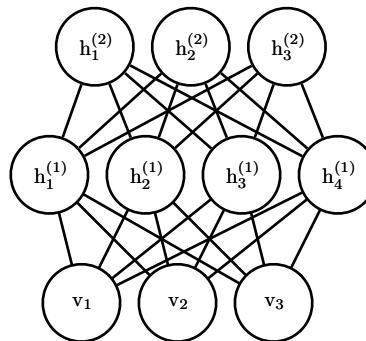


Abbildung 20.2: Das graphische Modell einer DBM mit einer sichtbaren Schicht (unten) und zwei verdeckten Schichten. Es gibt nur Verbindungen zwischen Einheiten in benachbarten Schichten. In einer Schicht selbst gibt es keine Verbindungen.

Wie RBMs und DBNs enthalten auch DBMs meist nur binäre Einheiten – wie wir aus Gründen der Einfachheit für unsere Darstellung des Modells annehmen –, aber es ist kein Problem, reellwertige sichtbare Einheiten einzubinden.

Eine DBM ist ein energiebasiertes Modell, d. h. die multivariate Verteilung über die Modellvariablen wird über eine Energiefunktion E parametri-

siert. Im Fall einer DBM mit einer sichtbaren Schicht, \mathbf{v} , und drei verdeckten Schichten, $\mathbf{h}^{(1)}$, $\mathbf{h}^{(2)}$ und $\mathbf{h}^{(3)}$, ergibt sich die multivariate Verteilung aus

$$P(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{h}^{(3)}) = \frac{1}{Z(\boldsymbol{\theta})} \exp(-E(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{h}^{(3)}; \boldsymbol{\theta})). \quad (20.24)$$

Um die Darstellung zu vereinfachen, lassen wir die Verzerrungsparameter unten weg. Die Energiefunktion für die DBM wird somit wie folgt definiert:

$$E(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \mathbf{h}^{(3)}; \boldsymbol{\theta}) = -\mathbf{v}^\top \mathbf{W}^{(1)} \mathbf{h}^{(1)} - \mathbf{h}^{(1)\top} \mathbf{W}^{(2)} \mathbf{h}^{(2)} - \mathbf{h}^{(2)\top} \mathbf{W}^{(3)} \mathbf{h}^{(3)}. \quad (20.25)$$

Im Gegensatz zur Energiefunktion für die RBM (Gleichung 20.5) enthält die Energiefunktion für die DBM Verbindungen zwischen den verdeckten Einheiten (latente Variablen) in Form der Gewichtungsmatrizen ($\mathbf{W}^{(2)}$ und $\mathbf{W}^{(3)}$). Wie wir noch sehen werden, haben diese Verbindungen erhebliche Folgen für das Modellverhalten und das Durchführen von Inferenz im Modell.

Im Vergleich zu vollständig verbundenen Boltzmann-Maschinen (engl. *fully connected Boltzmann machines*), in denen jede Einheit mit allen anderen Einheiten verbunden ist, bietet die DBM einige Vorteile, die denen der RBM ähneln. Vor allem können die Schichten der DBM, wie in Abbildung 20.3 dargestellt, zu einem zweiteiligen Graphen angeordnet werden, in dem die ungeraden Schichten auf der einen und die geraden Schichten auf der anderen Seite stehen. Damit wird direkt angezeigt, dass die Variablen in den ungeraden Schichten bei der Konditionierung anhand der Variablen in der geraden Schicht bedingt unabhängig werden. Das gilt umgekehrt genauso, d. h. beim Abstimmen anhand der Variablen in den ungeraden Schichten werden die Variablen in den geraden Schichten bedingt unabhängig.

Die zweiteilige Struktur der DBM bedeutet, dass wir dieselben Gleichungen, die wir bereits für die bedingten Verteilungen einer RBM genutzt haben, auch zum Bestimmen der bedingten Verteilungen einer DBM einsetzen können. Die Einheiten innerhalb einer Schicht sind für Werte der benachbarten Schichten bedingt unabhängig voneinander, sodass die Verteilungen über die binären Variablen vollständig durch die Bernoulli-Parameter beschrieben werden können, und zwar anhand der Wahrscheinlichkeit dafür, dass die jeweilige Einheit aktiviert ist. In unserem Beispiel mit zwei verdeckten Schichten betragen die Aktivierungswahrscheinlichkeiten

$$P(v_i = 1 | \mathbf{h}^{(1)}) = \sigma(\mathbf{W}_{i,:}^{(1)} \mathbf{h}^{(1)}), \quad (20.26)$$

$$P(h_i^{(1)} = 1 | \mathbf{v}, \mathbf{h}^{(2)}) = \sigma(\mathbf{v}^\top \mathbf{W}_{:,i}^{(1)} + \mathbf{W}_{i,:}^{(2)} \mathbf{h}^{(2)}), \quad (20.27)$$

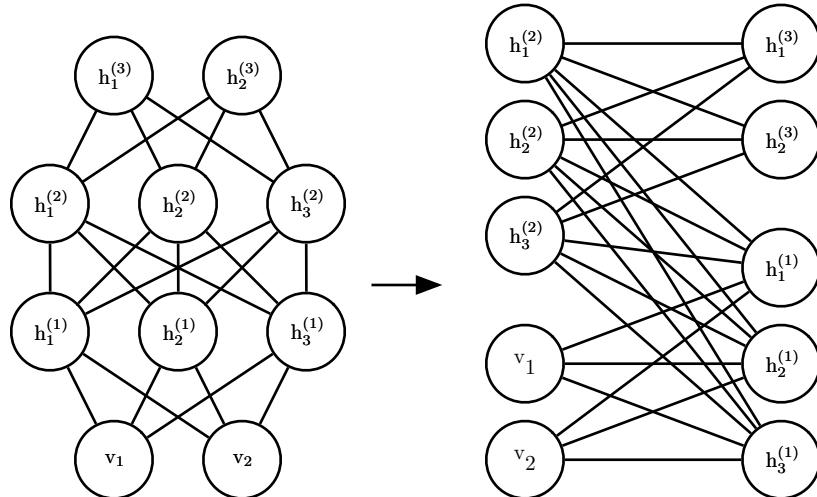


Abbildung 20.3: Eine DBM wird neu angeordnet, um die zweiteilige Graphenstruktur zu veranschaulichen.

und

$$P(h_k^{(2)} = 1 \mid \mathbf{h}^{(1)}) = \sigma(\mathbf{h}^{(1)\top} \mathbf{W}_{:,k}^{(2)}). \quad (20.28)$$

Die zweiteilige Struktur macht das Gibbs-Sampling für eine DBM effizient. Der naive Ansatz für das Gibbs-Sampling besteht darin, zu jedem Zeitpunkt nur eine Variable zu aktualisieren. RBMs ermöglichen das Aktualisieren aller sichtbaren Einheiten als Block und aller verdeckten Einheiten in einem zweiten Block. Man könnte naiverweise annehmen, dass eine DBM mit l Schichten $l+1$ Aktualisierungen erfordert – wobei jede Iteration zur Aktualisierung eines Blocks aus einer Schicht mit Einheiten besteht. Es ist jedoch möglich, alle Einheiten in nur zwei Iterationen zu aktualisieren. Das Gibbs-Sampling kann in zwei Blöcke zur Aktualisierung aufgeteilt werden: einer für alle geraden Schichten (einschließlich der sichtbaren Schicht) und einer für alle ungeraden Schichten. Aufgrund des zweiteiligen DBM-Verbindungsmusters mit vorgegebenen geraden Schichten ist die Verteilung über die ungeraden Schichten faktoriell und somit können Stichproben daraus zeitgleich und unabhängig als Block gezogen werden. Ebenso können mit vorgegebenen ungeraden Schichten aus den geraden Schichten zeitgleich und unabhängig als Block Stichproben gezogen werden. Eine effiziente Stichprobenentnahme ist besonders wichtig, wenn der Algorithmus der stochastischen Maximum Likelihood für das Training eingesetzt wird.

20.4.1 Interessante Eigenschaften

DBMs haben viele interessante Eigenschaften.

Sie wurden nach den DBNs entwickelt. Im Gegensatz zu DBNs ist die A-posteriori-Verteilung $P(\mathbf{h} | \mathbf{v})$ bei DBMs viel einfacher. Anders als vielleicht erwartet, ermöglicht die Einfachheit dieser A-posteriori-Verteilung umfassendere Approximationen der Verteilung. In einem DBN führen wir die Klassifizierung mit einem heuristisch motivierten approximativen Inferenzverfahren durch, bei dem wir annehmen, dass ein angemessener Wert für den Molekularfeld-Erwartungswert der verdeckten Einheiten bereitgestellt werden kann, und zwar durch einen nach oben erfolgenden Durchgang durch das Netz in einem mehrschichtigen Perzeptron, das sigmoide Aktivierungsfunktionen und dieselben Gewichte wie das ursprüngliche DBN einsetzt. Jede Verteilung $Q(\mathbf{h})$ kann verwendet werden, um eine ELBO für die Log-Likelihood zu ermitteln. Dieses heuristische Verfahren ermöglicht somit das Bestimmen einer solchen Schranke. Da die Schranke überhaupt nicht explizit optimiert ist, ist sie möglicherweise alles andere als eng. So ignoriert der heuristische Schätzwert für Q die Interaktionen zwischen verdeckten Einheiten in derselben Schicht sowie den Einfluss eines Top-Down-Feedbacks der verdeckten Einheiten in tieferen Schichten auf verdeckte Einheiten, die der Eingabe näher liegen. Da das heuristische Inferenzverfahren auf MLP-Basis im DBN diese Interaktionen nicht berücksichtigen kann, ist das resultierende Q vermutlich alles andere als optimal. In DBMs sind alle verdeckten Einheiten innerhalb einer Schicht bedingt unabhängig in Bezug auf die anderen Schichten. Diese fehlende Interaktion innerhalb einer Schicht macht es möglich, Fixpunktgleichungen zum Optimieren der ELBO einzusetzen und die echten optimalen Molekularfeld-Erwartungswerte zu bestimmen (auf eine gewisse rechnerische Toleranz genau).

Die Verwendung des korrekten Molekularfelds ermöglicht es dem approximativen Inferenzverfahren für DBMs, den Einfluss der Top-Down-Feedback-Interaktionen zu erfassen. Dadurch werden DBMs aus neurowissenschaftlicher Sicht interessant, denn das menschliche Gehirn nutzt bekanntermaßen viele Top-Down-Feedback-Verbindungen. Aufgrund dieser Eigenschaft wurden DBMs als Berechnungsmodelle für echte neurowissenschaftliche Phänomene eingesetzt (Series et al., 2010; Reichert et al., 2011).

Eine ungünstige Eigenschaft von DBMs ist, dass es recht schwierig ist, aus ihnen Stichproben zu ziehen. DBNs müssen das MCMC-Stichprobenverfahren nur in den oberen beiden Schichten anwenden. Die anderen Schichten werden erst am Ende der Stichprobenentnahme im Rahmen eines effizienten Ancestral-Sampling-Durchgangs einbezogen. Um eine Stichprobe aus einer

DBM zu generieren, muss eine MCMC für sämtliche Schichten eingesetzt werden, wobei jede Schicht im Modell an jedem Markow-Ketten-Übergang beteiligt ist.

20.4.2 DBM-Molekularfeld-Inferenz

Die bedingte Verteilung über eine DBM-Schicht anhand der benachbarten Schichten ist faktoriell. Im Beispiel der DBM mit zwei verdeckten Schichten handelt es sich um die Verteilungen $P(\mathbf{v} | \mathbf{h}^{(1)})$, $P(\mathbf{h}^{(1)} | \mathbf{v}, \mathbf{h}^{(2)})$ und $P(\mathbf{h}^{(2)} | \mathbf{h}^{(1)})$. Die Verteilung über alle verdeckten Schichten kann normalerweise aufgrund der Interaktionen zwischen den Schichten nicht in Faktoren zerlegt werden. Im Beispiel mit den zwei verdeckten Schichten, faktorisiert $P(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v})$ aufgrund der Interaktionsgewichte $\mathbf{W}^{(2)}$ zwischen $\mathbf{h}^{(1)}$ und $\mathbf{h}^{(2)}$, die diese Variablen gegenseitig abhängig darstellen, nicht.

Wie beim DBN müssen wir nach Methoden suchen, um die A-posteriori-Verteilung der DBM zu approximieren. Anders als beim DBN lässt sich die A-posteriori-Verteilung der DBM über die verdeckten Einheiten jedoch – obschon kompliziert – leicht mithilfe einer Variationsapproximation approximieren (siehe Abschnitt 19.4), insbesondere mit einer Molekularfeld-Approximation. Die Molekularfeld-Approximation ist eine einfache Form der Variational Inference, in der die approximierende Verteilung auf vollständige faktorielle Verteilungen eingeschränkt wird. Im Zusammenhang mit DBMs erfassen die Molekularfeld-Gleichungen die bidirektionale Interaktionen zwischen Schichten. In diesem Abschnitt leiten wir das iterative approximative Inferenzverfahren ab, das ursprünglich von *Salakhutdinov und Hinton* (2009a) eingeführt wurde.

In Variationsapproximationen der Inferenz nähern wir uns der Approximation einer bestimmten Zielverteilung – in unserem Fall der A-posteriori-Verteilung über die verdeckten Einheiten für die sichtbaren Einheiten – mit einer hinreichend einfachen Familie von Verteilungen an. Im Fall der Molekularfeld-Approximation ist die Approximationsfamilie die Menge der Verteilungen, in denen die verdeckten Einheiten bedingt unabhängig sind.

Wir entwickeln nun den Molekularfeld-Ansatz für das Beispiel mit zwei verdeckten Schichten. $Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v})$ sei die Approximation von $P(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v})$. Laut Molekularfeld-Annahme gilt

$$Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v}) = \prod_j Q(h_j^{(1)} | \mathbf{v}) \prod_k Q(h_k^{(2)} | \mathbf{v}). \quad (20.29)$$

Die Molekularfeld-Approximation versucht, ein Element dieser Verteilungsfamilie zu finden, das am besten zur wahren A-posteriori-Verteilung

$P(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v})$ passt. Es ist wichtig, den Inferenzprozess für jeden neuen Wert von \mathbf{v} zu wiederholen, um eine andere Verteilung Q zu finden.

Wie gut $Q(\mathbf{h} | \mathbf{v})$ zu $P(\mathbf{h} | \mathbf{v})$ passt, lässt sich auf viele Arten herausfinden. Den Molekularfeld-Ansatz verwenden wir für die Minimierung von

$$\text{KL}(Q \| P) = \sum_{\mathbf{h}} Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v}) \log \left(\frac{Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v})}{P(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v})} \right). \quad (20.30)$$

Grundsätzlich müssen wir keine parametrische Form der approximierenden Verteilung über die erzwungenen Annahmen der Unabhängigkeit hinaus angeben. Die Variationsapproximation ist im Allgemeinen in der Lage, eine funktionale Form der approximativen Verteilung wiederherzustellen. Allerdings gibt es im Falle einer Molekularfeld-Annahme für binäre verdeckte Einheiten (diesen Fall entwickeln wir hier) keine Einschränkung der Allgemeinheit durch eine vorherige Parametrisierung des Modells.

Wir parametrisieren Q als Produkt der Bernoulli-Verteilungen, verknüpfen also die Wahrscheinlichkeit für jedes Element aus $\mathbf{h}^{(1)}$ mit einem Parameter. Für jedes j , $\hat{h}_j^{(1)} = Q(h_j^{(1)} = 1 | \mathbf{v})$, mit $\hat{h}_j^{(1)} \in [0, 1]$, und für jedes k , $\hat{h}_k^{(2)} = Q(h_k^{(2)} = 1 | \mathbf{v})$, mit $\hat{h}_k^{(2)} \in [0, 1]$. Wir erhalten so die folgende Approximation an die A-posteriori-Verteilung:

$$Q(\mathbf{h}^{(1)}, \mathbf{h}^{(2)} | \mathbf{v}) = \prod_j Q(h_j^{(1)} | \mathbf{v}) \prod_k Q(h_k^{(2)} | \mathbf{v}) \quad (20.31)$$

$$= \prod_j (\hat{h}_j^{(1)})^{h_j^{(1)}} (1 - \hat{h}_j^{(1)})^{(1-h_j^{(1)})} \times \prod_k (\hat{h}_k^{(2)})^{h_k^{(2)}} (1 - \hat{h}_k^{(2)})^{(1-h_k^{(2)})} \quad (20.32)$$

Natürlich kann die Parametrisierung der approximativen A-posteriori-Verteilung für DBMs mit mehr Schichten auf die übliche Weise erweitert werden, wobei die zweiteilige Struktur des Graphen verwendet wird, um alle geraden Schichten und im Anschluss alle ungeraden Schichten zeitgleich zu aktualisieren, wie beim Gibbs-Sampling.

Nachdem wir unsere Familie von approximierenden Verteilungen Q spezifiziert haben, müssen wir noch ein Verfahren zur Wahl des Elements dieser Familie festlegen, das am besten zu P passt. Die einfachste Methode nutzt die Molekularfeld-Gleichungen aus Gleichung 19.56. Diese Gleichungen wurden durch Lösen der Stellen hergeleitet, an denen die Ableitungen der ELBO Null sind. Sie beschreiben auf abstrakte Weise, wie die ELBO für jedes beliebige Modell optimiert werden kann, einfach durch Verwenden der Erwartungswerte bezüglich Q .

Mit diesem allgemeinen Gleichungen erhalten wir die Update-Regel (wiederum ohne Berücksichtigung von Verzerrungstermen):

$$\hat{h}_j^{(1)} = \sigma \left(\sum_i v_i W_{i,j}^{(1)} + \sum_{k'} W_{j,k'}^{(2)} \hat{h}_{k'}^{(2)} \right), \quad \forall j, \quad (20.33)$$

$$\hat{h}_k^{(2)} = \sigma \left(\sum_{j'} W_{j',k}^{(2)} \hat{h}_{j'}^{(1)} \right), \quad \forall k. \quad (20.34)$$

In einem unveränderlichen Punkt dieses Gleichungssystems gibt es ein lokales Maximum der ELBO $\mathcal{L}(Q)$. Somit definieren die Fixpunktgleichungen für die Aktualisierung einen iterativen Algorithmus, in dem wir zwischen den Aktualisierungen von $\hat{h}_j^{(1)}$ (mittels Gleichung 20.33) und den Aktualisierungen von $\hat{h}_k^{(2)}$ (mittels Gleichung 20.34) abwechseln. Bei kleinen Problemen wie der MNIST reichen vielleicht schon zehn Iterationen aus, um einen approximativen Gradienten der positiven Phase für das Lernen zu finden. Fünfzig sind für gewöhnlich ausreichend, um eine hochwertige Repräsentation eines einzelnen spezifischen Beispiels für eine hochgenaue Klassifizierung zu ermitteln. Das Erweitern der approximative Variational Inference auf tiefere DBMs ist kein Problem.

20.4.3 Parameterlernen in DBMs

Lernen in der DBM muss sich den Herausforderungen der nicht effizient berechenbaren Partitionsfunktion (mithilfe der Methoden aus Kapitel 18) und der nicht effizient berechenbaren A-posteriori-Verteilung (mithilfe der Methoden aus Kapitel 19) stellen.

Wie in Abschnitt 20.4.2 beschrieben, ermöglicht die Variational Inference das Konstruieren einer Verteilung $Q(\mathbf{h} | \mathbf{v})$, die die nicht effizient berechenbare Verteilung $P(\mathbf{h} | \mathbf{v})$ approximiert. Anschließend wird das Lernen durch Maximieren von $\mathcal{L}(\mathbf{v}, Q, \boldsymbol{\theta})$, der ELBO der nicht effizient berechenbaren Log-Likelihood, $\log P(\mathbf{v}; \boldsymbol{\theta})$, fortgesetzt.

Für eine DBM mit zwei verdeckten Schichten ergibt sich \mathcal{L} aus

$$\mathcal{L}(Q, \boldsymbol{\theta}) = \sum_i \sum_{j'} v_i W_{i,j'}^{(1)} \hat{h}_{j'}^{(1)} + \sum_{j'} \sum_{k'} \hat{h}_{j'}^{(1)} W_{j',k'}^{(2)} \hat{h}_{k'}^{(2)} - \log Z(\boldsymbol{\theta}) + \mathcal{H}(Q). \quad (20.35)$$

Dieser Ausdruck enthält noch immer die Log-Partitionsfunktion $\log Z(\boldsymbol{\theta})$. Da eine DBM RBMs als Komponenten enthält, gelten die Schweregrade (engl. *hardness results*) für die Berechnung der Partitionsfunktion und die Stichprobenentnahme für RBMs ebenso für DBMs. Die Bewertung

der Wahrscheinlichkeitsfunktionen einer Boltzmann-Maschine setzt somit approximative Verfahren wie das Annealed Importance Sampling (AIS) voraus. Ebenso werden zum Trainieren des Modells Approximationen an den Gradienten der Log-Partitionsfunktion benötigt. Kapitel 18 enthält eine allgemeine Beschreibung dieser Verfahren. DBMs werden meist mit der stochastischen Maximum Likelihood trainiert. Viele der anderen in Kapitel 18 beschriebenen Verfahren sind nicht anwendbar. Verfahren wie die Pseudo-Likelihood setzen die Fähigkeit voraus, die nicht normalisierten Wahrscheinlichkeiten zu bestimmen, statt lediglich eine ELBO dafür zu ermitteln. Die kontrastive Divergenz ist in DBMs langsam, da sie keine effiziente Stichprobenentnahme der verdeckten Einheiten anhand der sichtbaren Einheiten ermöglicht – stattdessen müsste für die kontrastive Divergenz für jede neue Stichprobe der negativen Phase ein Burn-In einer Markow-Kette erfolgen.

Die allgemeine Version des Algorithmus zur stochastischen Maximum Likelihood wird in Abschnitt 18.2 behandelt. Die variationale stochastische Maximum Likelihood, wie sie für die DBM eingesetzt wird, finden Sie in Algorithmus 20.1. Denken Sie daran, dass wir eine vereinfachte Variante der DBM beschreiben, der die Verzerrungsparameter fehlen. Diese einzubeziehen ist trivial.

20.4.4 Schichtweises Pretraining

Leider endet das Trainieren einer DBM mittels stochastischer Maximum Likelihood (wie oben beschrieben) mit einer Zufallsinitialisierung meist in einem Fehlschlag. Manchmal kann das Modell eine adäquate Repräsentation der Verteilung nicht erlernen. Und manchmal stellt die DBM vielleicht die Verteilung gut dar, aber nicht mit einer höheren Likelihood als eine RBM. Eine DBM mit sehr kleinen Gewichten in allen bis auf der ersten Schicht repräsentiert näherungsweise dieselbe Verteilung wie eine RBM.

Diverse Verfahren für ein übergreifendes Training wurden entwickelt; diese werden in Abschnitt 20.4.5 beschrieben. Die erste und beliebteste Methode hinsichtlich des Problems eines übergreifenden Trainings von DBMs ist und bleibt jedoch das schichtweise Pretraining mit Greedy-Algorithmen. Dabei wird jede Schicht der DBM für sich allein als RBM trainiert. Die erste Schicht wird für das Modellieren der Eingangsdaten trainiert. Jede weitere RBM wird darauf trainiert, Stichproben aus der A-posteriori-Verteilung der vorherigen RBM zu modellieren. Nachdem auf diese Weise alle RBMs trainiert wurden, können sie zu einer DBM kombiniert werden. Diese kann anschließend mit persistenter kontrastiver Divergenz trainiert werden. Das

Algorithmus 20.1 Der Algorithmus der variationalen stochastischen Maximum Likelihood zum Trainieren einer DBM mit zwei verdeckten Schichten

Setze ϵ , die Schrittweite, auf eine kleine positive Zahl.

Setze k , die Anzahl der Gibbs-Schritte, auf einen Wert, der hoch genug ist, damit ein Burn-In einer Markow-Kette aus $p(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}; \boldsymbol{\theta} + \epsilon\Delta_{\boldsymbol{\theta}})$ erfolgen kann, angefangen mit Stichproben aus $p(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}; \boldsymbol{\theta})$.

Initialisiere drei Matrizen $\tilde{\mathbf{V}}$, $\tilde{\mathbf{H}}^{(1)}$ und $\tilde{\mathbf{H}}^{(2)}$ mit jeweils m Zeilen, die auf zufällige Werte gesetzt sind (z.B. aus Bernoulli-Verteilungen, möglicherweise mit Randverteilungen, die zu den Randverteilungen des Modells passen).

while nicht konvergiert (Lernschleife) **do**

Ziehen von Stichproben aus einem Mini-Batch mit m Beispielen aus den Trainingsdaten und Anordnen in den Zeilen einer Entwurfsmatrix \mathbf{V} .

Initialisieren der Matrizen $\hat{\mathbf{H}}^{(1)}$ und $\hat{\mathbf{H}}^{(2)}$, möglicherweise für die Randwahrscheinlichkeiten des Modells.

while nicht konvergiert (Molekularfeld-Inferenzschleife) **do**

$$\hat{\mathbf{H}}^{(1)} \leftarrow \sigma \left(\mathbf{V} \mathbf{W}^{(1)} + \hat{\mathbf{H}}^{(2)} \mathbf{W}^{(2)\top} \right).$$

$$\hat{\mathbf{H}}^{(2)} \leftarrow \sigma \left(\hat{\mathbf{H}}^{(1)} \mathbf{W}^{(2)} \right).$$

end while

$$\Delta_{\mathbf{W}^{(1)}} \leftarrow \frac{1}{m} \mathbf{V}^\top \hat{\mathbf{H}}^{(1)}$$

$$\Delta_{\mathbf{W}^{(2)}} \leftarrow \frac{1}{m} \hat{\mathbf{H}}^{(1)\top} \hat{\mathbf{H}}^{(2)}$$

for $l = 1$ to k (Gibbs-Sampling) **do**

Gibbs-Block 1:

$$\forall i, j, \tilde{V}_{i,j} \text{ gezogen aus } P(\tilde{V}_{i,j} = 1) = \sigma \left(\mathbf{W}_{j,:}^{(1)} \left(\tilde{\mathbf{H}}_{i,:}^{(1)} \right)^\top \right).$$

$$\forall i, j, \tilde{H}_{i,j}^{(2)} \text{ gezogen aus } P(\tilde{H}_{i,j}^{(2)} = 1) = \sigma \left(\tilde{\mathbf{H}}_{i,:}^{(1)} \mathbf{W}_{:,j}^{(2)} \right).$$

Gibbs-Block 2:

$$\forall i, j, \tilde{H}_{i,j}^{(1)} \text{ gezogen aus } P(\tilde{H}_{i,j}^{(1)} = 1) = \sigma \left(\tilde{\mathbf{V}}_{i,:} \mathbf{W}_{:,j}^{(1)} + \tilde{\mathbf{H}}_{i,:}^{(2)} \mathbf{W}_{j,:}^{(2)\top} \right).$$

end for

$$\Delta_{\mathbf{W}^{(1)}} \leftarrow \Delta_{\mathbf{W}^{(1)}} - \frac{1}{m} \mathbf{V}^\top \tilde{\mathbf{H}}^{(1)}$$

$$\Delta_{\mathbf{W}^{(2)}} \leftarrow \Delta_{\mathbf{W}^{(2)}} - \frac{1}{m} \tilde{\mathbf{H}}^{(1)\top} \tilde{\mathbf{H}}^{(2)}$$

$\mathbf{W}^{(1)} \leftarrow \mathbf{W}^{(1)} + \epsilon \Delta_{\mathbf{W}^{(1)}}$ (Dies ist eine vereinfachte Darstellung. In der Praxis sollten Sie einen effektiveren Algorithmus nutzen, zum Beispiel den Momentum-Algorithmus mit einer Verringerung der Lernrate.)

$$\mathbf{W}^{(2)} \leftarrow \mathbf{W}^{(2)} + \epsilon \Delta_{\mathbf{W}^{(2)}}$$

end while

Training mit persistenter kontrastiver Divergenz führt meist nur zu einer kleinen Änderung der Parameter und Leistung des Modells (gemessen durch die Log-Likelihood, die den Daten zugewiesen wird, bzw. seine Fähigkeit, Eingaben zu klassifizieren). Abbildung 20.4 stellt das Trainingsverfahren dar.

Dieses schichtweise Trainingsverfahren mit Greedy-Algorithmen ist mehr als ein Coordinate Ascent (Koordinatenanstieg). Es ähnelt entfernt dem Coordinate Ascent, da wir in jedem Schritt eine Teilmenge der Parameter optimieren. Die beiden Verfahren unterscheiden sich, da das eingesetzte schichtweise Trainingsverfahren mit Greedy-Algorithmen für jeden Schritt eine andere Zielfunktion verwendet.

Das schichtweise Pretraining mit Greedy-Algorithmen einer DBM unterscheidet sich vom schichtweisen Pretraining mit Greedy-Algorithmen eines DBNs. Die Parameter jeder einzelnen RBM können direkt in das entsprechende DBN kopiert werden. Im Fall der DBM müssen die Parameter der RBM vor der Einbindung in die DBM geändert werden. Eine Schicht in der Mitte des RBM-Stapels wird nur mit Bottom-Up-Eingabe trainiert, aber nachdem der Stapel zu einer DBM kombiniert wurde, weist die Schicht sowohl Bottom-Up- als auch Top-Down-Eingaben auf. Um diesem Umstand Rechnung zu tragen, fordern *Salakhutdinov und Hinton* (2009a) dazu auf, die Gewichte aller RBMs mit Ausnahme der obersten und untersten zu halbieren, bevor sie in die DBM eingefügt werden. Außerdem muss die unterste RBM trainiert werden, indem zwei »Kopien« jeder sichtbaren Einheit verwendet werden, wobei die Gewichte zwischen den beiden Kopien gleich bleiben müssen. Das führt dazu, dass die Gewichte letztlich während des Aufwärtsdurchgangs verdoppelt werden. Ebenso soll die oberste RBM mit zwei Exemplaren der obersten Schicht trainiert werden.

Um mit der DBM Ergebnisse zu erhalten, die dem Stand der Technik entsprechen, muss der normale SML-Algorithmus geändert werden, sodass ein kleiner Anteil des Molekularfelds während der negativen Phase des übergreifenden PCD-Trainingsschritts verwendet wird (*Salakhutdinov und Hinton*, 2009a). Insbesondere muss der Erwartungswert des Energiegradienten bezüglich der Molekularfeld-Verteilung berechnet werden, in der alle Einheiten voneinander unabhängig sind. Die Parameter dieser Molekularfeld-Verteilung müssen dann anhand der Molekularfeld-Fixpunktgleichungen für nur einen Schritt ermittelt werden. *Goodfellow et al.* (2013b) vergleichen die Leistung zentrierter DBMs mit und ohne partielles Molekularfeld in der negativen Phase.

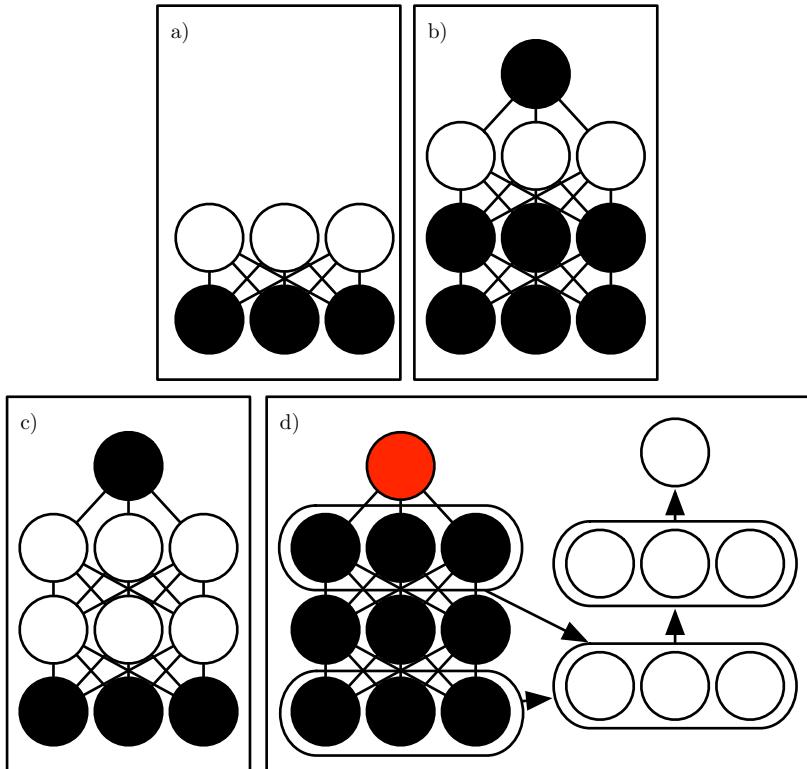


Abbildung 20.4: Das Trainingsverfahren für die DBMs für die Klassifizierung des MNIST-Datensatzes (*Salakhutdinov und Hinton, 2009a; Srivastava et al., 2014*). (a) Trainieren einer RBM mittels kontrastiver Divergenz zur approximativen Maximierung von $\log P(\mathbf{v})$. (b) Trainieren einer zweiten RBM, die $\mathbf{h}^{(1)}$ und die Zielklasse y mittels CD- k zur approximativen Maximierung von $\log P(\mathbf{h}^{(1)}, y)$ modelliert, wobei $\mathbf{h}^{(1)}$ aus der A-posteriori-Verteilung der ersten RBM (konditioniert für die Daten) gezogen wird. k wird während des Lernens von 1 bis 20 erhöht. (c) Kombinieren der beiden RBMs zu einer DBM. Trainieren der DBM zur approximativen Maximierung von $\log P(\mathbf{v}, y)$ mittels stochastischer Maximum Likelihood mit $k = 5$. (d) Löschen von y aus dem Modell. Definieren einer neuen Menge von Merkmalen $\mathbf{h}^{(1)}$ und $\mathbf{h}^{(2)}$, die mittels Molekularfeld-Inferenz im Modell ohne y ermittelt werden. Verwenden dieser Merkmale als Eingabe für ein mehrschichtiges Perzeptron, dessen Struktur einem weiteren Molekularfeld-Durchgang entspricht, mit einer zusätzlichen Ausgabeschicht für den Schätzwert von y . Initialisieren der MLP-Gewichtungen, sodass diese den DBM-Gewichtungen gleich sind. Trainieren des mehrschichtigen Perzeptrons zur approximativen Maximierung von $\log P(y | \mathbf{v})$ mittels stochastischem Gradientenabstiegsverfahren (engl. *stochastic gradient descent*, SGD) und Dropout. Abbildung aus *Goodfellow et al. (2013b)*.

20.4.5 Übergreifendes Trainieren von DBMs

Klassische DBMs benötigen ein unüberwachtes Pretraining mit Greedy-Algorithmen und daher für eine gute Klassifizierungsleistung zusätzlich zu den verdeckten Merkmalen, die sie extrahieren, einen separaten Klassifikator auf MLP-Basis. Das führt zu einigen unerwünschten Eigenschaften. Es ist kompliziert, die Leistung während des Trainings im Blick zu behalten, da wir die Eigenschaften der vollständigen DBM während des Trainings der ersten RBM nicht bewerten können. Daher ist es schwer zu sagen, wie gut unsere Hyperparameter funktionieren – das ist erst recht spät im Trainingsverfahren möglich. Softwareimplementierungen von DBMs müssen viele unterschiedliche Komponenten für das CD-Training einzelner RBMs, das PCD-Training der gesamten DBM und das Training auf Basis der Backpropagation durch das mehrschichtige Perzeptron einsetzen. Schließlich verliert ein auf die Boltzmann-Maschine »aufgepfropftes« mehrschichtiges Perzeptron viele Vorteile des probabilistischen Modells der Boltzmann-Maschine, zum Beispiel, dass das sich Inferenz trotz einiger fehlender Eingabewerte durchführen lässt.

Es gibt zwei wesentliche Optionen zum Lösen des Problems des übergreifenden Trainings von DBMs. Die erste wird **centered DBM** (zentrierte DBM) (*Montavon und Muller, 2012*) genannt und parametrisiert das Modell neu, um die Hesse-Matrix der Kostenfunktion zu Beginn des Lernprozesses besser abzustimmen. Das führt zu einem Modell, das sich ohne eine Phase des schichtweisen Pretrainings mit Greedy-Algorithmen trainieren lässt. Das resultierende Modell erzielt eine hervorragende Log-Likelihood für die Testdatenmenge und erzeugt hochwertige Stichproben. Leider kann es sich bei der Klassifizierung nicht mit einem angemessen regularisierten mehrschichtigen Perzeptron messen. Die zweite Möglichkeit für das übergreifende Training einer DBM ist die **Multi-Prediction DBM** (DBM mit mehreren Vorhersagen) (*Goodfellow et al., 2013b*). Dieses Modell nutzt ein alternatives Trainingskriterium, bei dem Probleme mit MCMC-Schätzwerten des Gradienten durch Verwendung des Backpropagation-Algorithmus umgangen werden. Leider führt das neue Kriterium weder zu einer guten Likelihood noch zu guten Stichproben. Aber, anders als beim MCMC-Ansatz, resultiert es in einer überragenden Klassifizierungsleistung und der Fähigkeit, gut auf fehlende Eingaben zu schließen.

Die Zentrierung für Boltzmann-Maschinen lässt sich am besten beschreiben, wenn wir uns die Boltzmann-Maschine als Gebilde aus einer Menge von

Einheiten \mathbf{x} mit einer Gewichtungsmatrix \mathbf{U} und Verzerrungen \mathbf{b} vorstellen. Aus Gleichung 20.2 wissen Sie, dass sich die Energiefunktion ergibt aus

$$E(\mathbf{x}) = -\mathbf{x}^\top \mathbf{U} \mathbf{x} - \mathbf{b}^\top \mathbf{x}. \quad (20.36)$$

Mittels diverser Muster der dünnen Besetzung in der Gewichtungsmatrix \mathbf{U} können wir Strukturen von Boltzmann-Maschinen implementieren, zum Beispiel RBMs oder DBMs mit unterschiedlich vielen Schichten. Das geht durch Aufteilen von \mathbf{x} in sichtbare und verdeckte Einheiten sowie Eliminieren von Elementen aus \mathbf{U} für Einheiten, die nicht interagieren. Die zentrierte Boltzmann-Maschine führt einen Vektor $\boldsymbol{\mu}$ ein, der von allen Zuständen subtrahiert wird:

$$E'(\mathbf{x}; \mathbf{U}, \mathbf{b}) = -(\mathbf{x} - \boldsymbol{\mu})^\top \mathbf{U} (\mathbf{x} - \boldsymbol{\mu}) - (\mathbf{x} - \boldsymbol{\mu})^\top \mathbf{b}. \quad (20.37)$$

Meist ist $\boldsymbol{\mu}$ ein Hyperparameter, der bei Trainingsbeginn festgelegt wird. Es wird so gewählt, dass $\mathbf{x} - \boldsymbol{\mu} \approx \mathbf{0}$ bei Initialisierung des Modells ist. Diese Reparametrisierung ändert nicht die Menge der Wahrscheinlichkeitsverteilungen, die das Modell darstellen kann, aber sie ändert die Dynamik des stochastischen Gradientenabstiegs der Likelihood. In vielen Fällen führt diese Reparametrisierung zu einer besser darauf abgestimmten Hesse-Matrix. *Melchior et al.* (2013) haben experimentell bestätigt, dass die Konditionierung der Hesse-Matrix dabei verbessert wird, und beobachtet, dass die Zentrierung das Äquivalent einer weiteren Boltzmann-Machine-Learning-Methode ist, dem sogenannten **enhanced Gradient** (erweiterter Gradient) (*Cho et al.*, 2011). Die verbesserte Form der Hesse-Matrix ermöglicht den Lernerfolg auch in schwierigen Fällen wie beim Trainieren einer DBM mit vielen Schichten.

Der andere Ansatz für das übergreifende Trainieren von DBMs ist die Multi-Prediction Deep Boltzmann Machine (MP-DBM, dt. *DBM mit mehreren Vorhersagen*); darin werden die Molekularfeld-Gleichungen als Definition einer Familie von RNNs betrachtet, die eine approximative Lösung jedes möglichen Inferenzproblems ermöglichen (*Goodfellow et al.*, 2013b). Statt das Modell für eine Maximierung der Likelihood zu trainieren, wird es so trainiert, dass jedes RNN eine korrekte Antwort auf das entsprechende Inferenzproblem liefert. Das Trainingsverfahren ist in Abbildung 20.5 dargestellt. Es besteht aus dem zufälligen Ziehen eines Trainingsbeispiels, dem zufälligen Ziehen einer Teilmenge der Eingaben für das Inferenznetz und dem anschließenden Training des Inferenznetzes zum Vorhersagen der Werte der restlichen Einheiten.

Dieses allgemeine Prinzip der Backpropagation durch den Berechnungsgraphen für die approximative Inferenz wurde auf viele andere Modelle

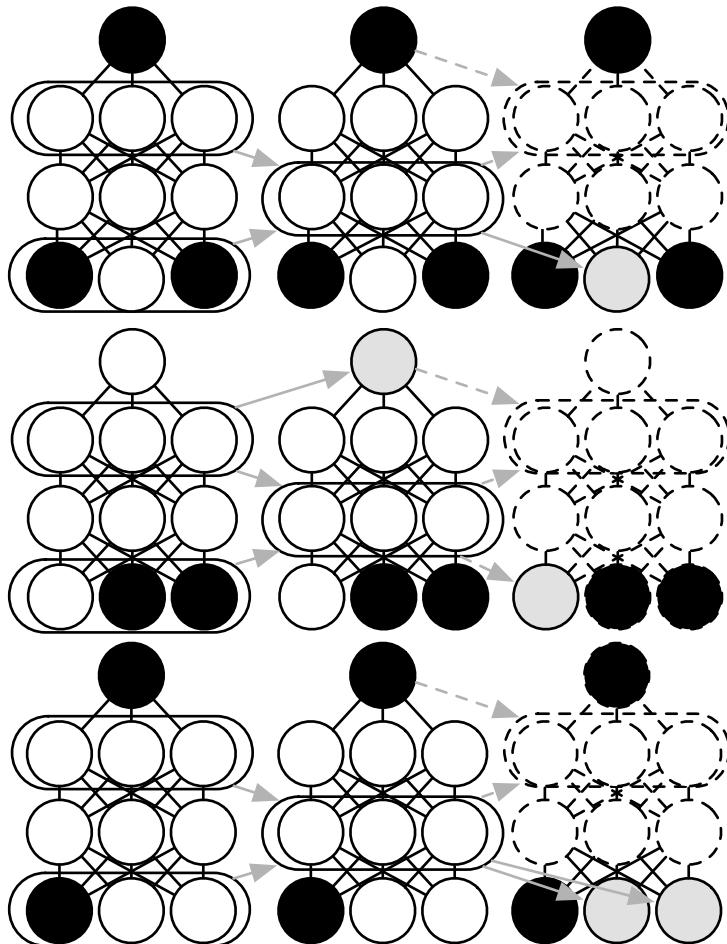


Abbildung 20.5: Eine Darstellung des Trainingsverfahrens mit mehreren Vorhersagen für eine DBM. Jede Zeile steht für ein anderes Beispiel innerhalb eines Mini-Batches für denselben Trainingsschritt. Jede Spalte steht für einen Zeitschritt im Molekularfeld-Inferenzprozess. Für jedes Beispiel ziehen wir eine Teilmenge der Datenvariablen als Eingabe für den Inferenzprozess. Diese Variablen sind schwarz eingefärbt, um dies anzuzeigen. Anschließend wird der Molekularfeld-Inferenzprozess ausgeführt; die Pfeile geben an, welche Variablen im Prozess welche anderen Variablen beeinflussen. In praktischen Anwendungen wickeln wir das Molekularfeld über mehrere Schritte ab. In dieser Abbildung sind es nur zwei Schritte. Gestrichelte Pfeile geben an, wie der Prozess in weiteren Schritten abgewickelt werden könnte. Die Datenvariablen, die nicht als Eingaben für den Inferenzprozess dienten, werden zu Zielwerten (grau eingefärbt). Wir können den Inferenzprozess für jedes Beispiel als RNN betrachten. Wir verwenden Gradientenabstiegsverfahren und Backpropagation zum Trainieren dieser RNNs für die Erzeugung der korrekten Zielwerte auf Grundlage der Eingaben. So wird der Molekularfeld-Prozess für die MP-DBM so trainiert, dass korrekte Schätzwerte erzeugt werden. Auf Basis einer Abbildung aus Goodfellow et al. (2013b).

angewandt (*Stoyanov et al.*, 2011; *Brakel et al.*, 2013). In diesen Modellen und in der MP-DBM ist der letzte Verlust nicht die untere Schranke der Likelihood. Stattdessen beruht der letzte Verlust meist auf der approximativen bedingten Verteilung, die das approximative Inferenznetz den fehlenden Werten auferlegt. Das Training dieser Modelle ist also auf gewisse Weise heuristisch motiviert. Wenn wir das durch die von der MP-DBM erlernte Boltzmann-Maschine repräsentierte $p(\mathbf{v})$ untersuchen, ist häufig eine Art Defekt zu erkennen, d. h., das Gibbs-Sampling führt zu schlechten Stichproben.

Die Backpropagation durch den Inferenzgraphen hat zwei große Vorteile. Erstens wird das Modell so trainiert, wie es tatsächlich verwendet wird – mit approximativer Inferenz. Das bedeutet, dass die approximative Inferenz zum Beispiel fehlende Eingaben ergänzt oder eine Klassifizierung trotz fehlender Eingangsdaten durchführt und dabei in der MP-DBM exakter als in der ursprünglichen DBM ist. Die ursprüngliche DBM ist selbst kein genauer Klassifikator; die besten Klassifizierungsergebnisse mit der ursprünglichen DBM basierten auf dem Trainieren eines separaten Klassifikators zur Verwendung von Merkmalen, die von der DBM extrahiert wurden – nicht auf dem Einsatz von Inferenz in der DBM zum Berechnen der Verteilung über die Klassen-Label. Die Molekularfeld-Inferenz in der MP-DBM funktioniert ohne besondere Anpassungen gut als Klassifikator. Der andere Vorteil der Backpropagation durch die approximative Inferenz ist, dass damit der exakte Gradient des Verlusts berechnet wird. Dies ist für die Optimierung vorteilhafter als die approximativen Gradienten aus dem SML-Training, die mit Verzerrung und Varianz behaftet sind. Das erklärt vermutlich, warum MP-DBMs übergreifend trainiert werden können, wohingegen für DBMs ein schichtweises Pretraining mit Greedy-Algorithmen notwendig ist. Der Nachteil der Backpropagation durch den approximativen Inferenzgraphen besteht darin, dass sich dabei keine Möglichkeit zur Optimierung der Log-Likelihood ergibt, sondern vielmehr eine heuristische Approximation der generalisierten Pseudo-Likelihood.

Die MP-DBM diente als Inspiration für die Erweiterung NADE- k (*Raiko et al.*, 2014) für das NADE-Framework, das in Abschnitt 20.10.10 beschrieben wird.

Die MP-DBM weist einige Verbindungen zu Dropout auf. Dropout nutzt dieselben Parameter für viele unterschiedliche Berechnungsgraphen, wobei der Unterschied zwischen den einzelnen Graphen darin besteht, ob jede Einheit ein- oder ausgeschlossen wird. Die MP-DBM nutzt Parameter ebenfalls für viele Berechnungsgraphen. Im Falle der MP-DBM besteht der Unterschied zwischen den Graphen darin, ob jede Eingabeeinheit beobachtet

wird oder nicht. Wenn eine Einheit nicht beobachtet wird, löscht die MP-DBM diese nicht gänzlich, wie es beim Dropout der Fall ist. Stattdessen behandelt die MP-DBM sie als latente Variable, auf die es zu schließen gilt. Es ist vorstellbar, Dropout für eine MP-DBM zu verwenden, indem zusätzlich einige Einheiten entfernt und nicht latent gemacht werden.

20.5 Boltzmann-Maschinen für reellwertige Daten

Boltzmann-Maschinen wurden ursprünglich für den Einsatz mit binären Daten entwickelt, doch viele Anwendungen wie das Modellieren von Bildern und Klängen scheinen die Fähigkeit zu erfordern, Wahrscheinlichkeitsverteilungen über reelle Werte darstellen zu können. In einigen Fällen können reellwertige Daten im Intervall $[0, 1]$ als Repräsentation des Erwartungswerts einer binären Variable behandelt werden. Zum Beispiel behandelt Hinton (2000) Graustufenbilder in der Trainingsdatenmenge so, als ob sie $[0,1]$ Wahrscheinlichkeitswerte definierten. Jedes Pixel definiert die Wahrscheinlichkeit, mit der ein Binärwert 1 ist, und die binären Pixel werden alle unabhängig voneinander gezogen. Dies ist ein übliches Vorgehen zur Bewertung binärer Modelle für Datensätze mit Graustufenbildern. Nichtsdestotrotz ist es aus theoretischer Sicht kein zufriedenstellender Ansatz und auf diese Weise unabhängig voneinander gezogenen Binärbilder sind von Rauschen geprägt. In diesem Abschnitt stellen wir Boltzmann-Maschinen vor, die eine Wahrscheinlichkeitsdichte über reellwertige Daten definieren.

20.5.1 Gauß-Bernoulli-RBMs

RBM kann für viele Familien von Exponentialfunktionen von bedingten Verteilungen entwickelt werden (Welling et al., 2005). Die häufigste davon ist die RBM mit binären verdeckten Einheiten und reellwertigen sichtbaren Einheiten, bei der die bedingte Verteilung über die sichtbaren Einheiten eine Normalverteilung ist, deren Mittelwert eine Funktion der verdeckten Einheiten ist.

Es gibt viele Möglichkeiten zur Parametrisierung von Gauß-Bernoulli-RBMs. So müssen Sie zwischen einer Kovarianzmatrix und einer Präzisionsmatrix für die Normalverteilung wählen. Hier arbeiten wir mit der Präzisionsmatrix. Die Anpassungen bei Verwendung der Kovarianzmatrix sind recht einfach. Gewünscht ist die bedingte Verteilung

$$p(\mathbf{v} \mid \mathbf{h}) = \mathcal{N}(\mathbf{v}; \mathbf{Wh}, \boldsymbol{\beta}^{-1}). \quad (20.38)$$

Wir können die benötigten Terme, die zur Energiefunktion addiert werden müssen, durch Erweitern der nicht normalisierten bedingten Log-Verteilung ermitteln:

$$\log \mathcal{N}(\mathbf{v}; \mathbf{W}\mathbf{h}, \boldsymbol{\beta}^{-1}) = -\frac{1}{2} (\mathbf{v} - \mathbf{W}\mathbf{h})^\top \boldsymbol{\beta} (\mathbf{v} - \mathbf{W}\mathbf{h}) + f(\boldsymbol{\beta}). \quad (20.39)$$

Hier fasst f sämtliche Terme zusammen, die eine Funktion nur der Parameter und nicht der Zufallsvariablen im Modell sind. Wir können f verwerfen, da es nur der Normalisierung der Verteilung dient und diese Aufgabe von der Partitionsfunktion der von uns gewählten Energiefunktion übernommen wird.

Wenn wir sämtliche Terme (mit umgekehrtem Vorzeichen), die \mathbf{v} aus Gleichung 20.39 enthalten, in unsere Energiefunktion einschließen und keine weiteren Terme, die \mathbf{v} beinhalten, hinzufügen, repräsentiert unsere Energiefunktion die gesuchte bedingte Verteilung $p(\mathbf{v} | \mathbf{h})$.

Wir haben eine gewisse Freiheit hinsichtlich der anderen bedingten Verteilung $p(\mathbf{h} | \mathbf{v})$. Beachten Sie, dass Gleichung 20.39 den folgenden Term enthält:

$$\frac{1}{2} \mathbf{h}^\top \mathbf{W}^\top \boldsymbol{\beta} \mathbf{W} \mathbf{h}. \quad (20.40)$$

Dieser kann nicht in seiner Gesamtheit einbezogen werden, da er $h_i h_j$ -Terme enthält. Diese entsprechen den Kanten zwischen den verdeckten Einheiten. Würden wir diese Terme einbeziehen, hätten wir am Ende ein lineares Faktorenmodell anstelle einer RBM. Beim Konzipieren unserer Boltzmann-Maschine lassen wir diese $h_i h_j$ -Kreuzterme einfach weg. Das ändert die bedingte Verteilung $p(\mathbf{v} | \mathbf{h})$ nicht und Gleichung 20.39 trifft nach wie vor zu. Wir haben allerdings noch immer die Möglichkeit, die Terme einzubeziehen, die nur ein einzelnes h_i enthalten. Wenn wir von einer Diagonal-Präzisionsmatrix ausgehen, stellen wir fest, dass es für jede verdeckte Einheit h_i den folgenden Term gibt:

$$\frac{1}{2} h_i \sum_j \beta_j W_{j,i}^2. \quad (20.41)$$

Oben haben wir $h_i^2 = h_i$ genutzt, da $h_i \in \{0, 1\}$ ist. Wenn wir diesen Term (mit umgekehrtem Vorzeichen) in die Energiefunktion aufnehmen, wird h_i auf natürliche Weise verzerrt und deaktiviert, wenn die Gewichte der Einheit groß und mit sichtbaren Einheiten mit hoher Präzision verbunden sind. Die Wahl für oder gegen das Einbeziehen dieses Verzerrungsterms beeinflusst nicht die Familie der Verteilungen, die das Modell darstellen kann (vorausgesetzt, wir schließen die Verzerrungsparameter für die verdeckten Einheiten

ein), sehr wohl aber die Lerndynamik des Modells. Das Einbeziehen des Terms kann die Aktivierungen der verdeckten Einheiten auch dann in einem angemessenen Rahmen halten, wenn die Gewichte rasant wachsen.

Eine Möglichkeit zum Definieren der Energiefunktion für eine Gauß-Bernoulli-RBM ist somit diese:

$$E(\mathbf{v}, \mathbf{h}) = \frac{1}{2} \mathbf{v}^\top (\boldsymbol{\beta} \odot \mathbf{v}) - (\mathbf{v} \odot \boldsymbol{\beta})^\top \mathbf{W} \mathbf{h} - \mathbf{b}^\top \mathbf{h}, \quad (20.42)$$

aber wir können auch zusätzliche Terme hinzufügen oder die Energie als Varianz statt als Präzision parametrisieren, wenn dies gewünscht ist.

In dieser Herleitung haben wir keinen Verzerrungsterm für die sichtbaren Einheiten verwendet, aber das wäre kein Problem. Eine abschließende Quelle der Variabilität in der Parametrisierung einer Gauß-Bernoulli-RBM ist die Entscheidung hinsichtlich des Umgangs mit der Präzisionsmatrix. Sie kann entweder als Konstante (die zum Beispiel anhand der Randpräzision der Daten geschätzt wird) definiert oder erlernt werden. Es kann sich auch um ein skalares Vielfaches der Einheitsmatrix oder um eine Diagonalmatrix handeln. Üblicherweise lassen wir in diesem Kontext nicht zu, dass die Präzisionsmatrix nicht-diagonal ist, da einige Operationen für die Normalverteilung ein Umkehren der Matrix erfordern und dies bei einer Diagonalmatrix problemlos möglich ist. In den folgenden Abschnitten zeigen wir, dass andere Formen der Boltzmann-Maschine das Modellieren der Kovarianzstruktur mit diversen Verfahren ermöglichen, die ein Umkehren der Präzisionsmatrix umgehen.

20.5.2 Ungerichtete Modelle der bedingten Kovarianz

Obwohl die gaußsche RBM das Energiemodell für reellwertige Daten schlechthin ist, argumentieren Ranzato *et al.* (2010a), dass die induktive Verzerrung der gaußschen RBM nicht gut für die statistischen Variationen in einigen Arten reellwertiger Daten geeignet ist, insbesondere nicht für natürliche Bilder. Das Problem liegt darin, dass ein großer Teil des Informationsgehalts in natürlichen Bildern in die Kovarianz zwischen den Pixeln eingebettet ist, und nicht in die reinen Pixelwerte. Anders ausgedrückt stecken die meisten nützlichen Informationen über Bilder in der Beziehung zwischen den Pixeln, und nicht etwa in ihren Absolutbeträgen. Da die gaußsche RBM nur den bedingten Mittelwert der Eingabe aufgrund der verdeckten Einheiten modelliert, kann sie keine Informationen zur bedingten Kovarianz erfassen. Als Antwort auf diese Kritik wurden alternative Modelle vorgeschlagen, in denen die Kovarianz reellwertiger Daten besser

berücksichtigt werden soll. Dazu gehören die Mean and Covariance RBMs (mcRBMs¹, dt. *RBM mit Mittelwert und Kovarianz*), das Mean Product of Student *t*-Distribution (dt. *Mittelwertproduktmodell der studentischen t-Verteilung*, kurz mPoT) und die Spike and Slab RBM (ssRBM).

Mean and Covariance RBM Die mcRBM nutzt ihre verdeckten Einheiten, um den bedingten Mittelwert und die Kovarianz aller beobachteten Einheiten unabhängig zu codieren. Die verdeckte Schicht der mcRBM ist in zwei Einheitengruppen unterteilt: Mittelwerteinheiten und Kovarianzeinheiten. Die Gruppe, die den bedingten Mittelwert modelliert, ist einfach eine gaußsche RBM. Die andere Hälfte ist eine Kovarianz-RBM (*Ranzato et al., 2010a*), kurz cRBM, deren Komponenten die bedingte Kovarianzstruktur wie unten beschrieben modellieren.

Für binäre Mittelwerteinheiten $\mathbf{h}^{(m)}$ und binäre Kovarianzeinheiten $\mathbf{h}^{(c)}$ ist das mcRBM-Modell als Kombination von zwei Energiefunktionen definiert:

$$E_{\text{mc}}(\mathbf{x}, \mathbf{h}^{(m)}, \mathbf{h}^{(c)}) = E_{\text{m}}(\mathbf{x}, \mathbf{h}^{(m)}) + E_{\text{c}}(\mathbf{x}, \mathbf{h}^{(c)}), \quad (20.43)$$

wobei E_{m} die normale Energiefunktion der Gauß-Bernoulli-RBM²

$$E_{\text{m}}(\mathbf{x}, \mathbf{h}^{(m)}) = \frac{1}{2} \mathbf{x}^\top \mathbf{x} - \sum_j \mathbf{x}^\top \mathbf{W}_{:,j} h_j^{(m)} - \sum_j b_j^{(m)} h_j^{(m)} \quad (20.44)$$

ist und E_{c} die Energiefunktion der cRBM, die die bedingten Kovarianzinformationen modelliert:

$$E_{\text{c}}(\mathbf{x}, \mathbf{h}^{(c)}) = \frac{1}{2} \sum_j h_j^{(c)} (\mathbf{x}^\top \mathbf{r}^{(j)})^2 - \sum_j b_j^{(c)} h_j^{(c)}. \quad (20.45)$$

Der Parameter $\mathbf{r}^{(j)}$ entspricht dem Gewichtungsvektor der Kovarianz für $h_j^{(c)}$, und $\mathbf{b}^{(c)}$ ist ein Vektor der Kovarianz der Offsets. Die kombinierte Energiefunktion definiert eine multivariate Verteilung

$$p_{\text{mc}}(\mathbf{x}, \mathbf{h}^{(m)}, \mathbf{h}^{(c)}) = \frac{1}{Z} \exp \left\{ -E_{\text{mc}}(\mathbf{x}, \mathbf{h}^{(m)}, \mathbf{h}^{(c)}) \right\}, \quad (20.46)$$

¹ Der Begriff »mcRBM« wird M-C-R-B-M gesprochen und beginnt nicht mit dem Kürzel der bekannten Fast-Food-Kette.

² Diese Version der Energiefunktion der Gauß-Bernoulli-RBM geht davon aus, dass der Mittelwert der Bilddaten pro Pixel gleich Null ist. Pixelversatz (engl. *pixel offset*) können problemlos zum Modell addiert werden, um von Null verschiedene Pixelmittelwerte zu berücksichtigen.

und eine zugehörige bedingte Verteilung über die Beobachtungen für $\mathbf{h}^{(m)}$ und $\mathbf{h}^{(c)}$ als multivariate Normalverteilung:

$$p_{\text{mc}}(\mathbf{x} \mid \mathbf{h}^{(m)}, \mathbf{h}^{(c)}) = \mathcal{N} \left(\mathbf{x}; \mathbf{C}_{\mathbf{x} \mid \mathbf{h}}^{\text{mc}} \left(\sum_j \mathbf{W}_{:,j} h_j^{(m)} \right), \mathbf{C}_{\mathbf{x} \mid \mathbf{h}}^{\text{mc}} \right). \quad (20.47)$$

Beachten Sie, dass die Kovarianzmatrix $\mathbf{C}_{\mathbf{x} \mid \mathbf{h}}^{\text{mc}} = \left(\sum_j h_j^{(c)} \mathbf{r}^{(j)} \mathbf{r}^{(j)\top} + \mathbf{I} \right)^{-1}$ nicht-diagonal ist und dass \mathbf{W} die Gewichtungsmatrix für die gaußsche RBM zur Modellierung der bedingten Mittelwerte ist. Es ist schwierig, die mcRBM mittels kontrastiver Divergenz oder persistenter kontrastiver Divergenz zu trainieren, da die Struktur der bedingten Kovarianz nicht-diagonal ist. Kontrastive Divergenz und persistente kontrastive Divergenz erfordern eine Stichprobenentnahme aus der multivariaten Verteilung von $\mathbf{x}, \mathbf{h}^{(m)}, \mathbf{h}^{(c)}$, das – in einer normalen RBM – mittels Gibbs-Sampling über die bedingten Verteilungen erfolgt. Allerdings muss in der mcRBM für die Stichprobenentnahme aus $p_{\text{mc}}(\mathbf{x} \mid \mathbf{h}^{(m)}, \mathbf{h}^{(c)})$ in jeder Lerniteration $(\mathbf{C}^{\text{mc}})^{-1}$ berechnet werden. Das kann bei größeren Beobachtungen einen nicht praktikablen Rechenaufwand bedeuten. *Ranzato und Hinton* (2010) vermeiden die direkte Stichprobenentnahme aus der bedingten Verteilung $p_{\text{mc}}(\mathbf{x} \mid \mathbf{h}^{(m)}, \mathbf{h}^{(c)})$ durch Ziehen aus der Randverteilung $p(\mathbf{x})$ mittels hamiltonschem (Hybrid-)Monte-Carlo-Sampling (*Neal*, 1993) der freien Energie der mcRBM.

Mittelwertprodukt der studentschen t -Verteilungen Das Mittelwertproduktmodell der studentschen t -Verteilung (mPoT) (*Ranzato et al.*, 2010b) erweitert das PoT-Modell (*Welling et al.*, 2003a) auf ähnliche Weise, wie die mcRBM die cRBM erweitert. Das gelingt durch Einbeziehen der von Null verschiedenen Mittelwerte der Normalverteilung mittels Addition von verdeckten Einheiten, die einer gaußschen RBM ähneln. Wie die mcRBM ist auch die bedingte PoT-Verteilung über die Beobachtung eine multivariate Normalverteilung (mit nicht-diagonaler Kovarianz). Allerdings ergibt sich die komplementäre bedingte Verteilung über die verdeckten Variablen anders als die mcRBM aus den bedingt unabhängigen Gamma-Verteilungen. Die Gamma-Verteilung $\mathcal{G}(k, \theta)$ ist eine Wahrscheinlichkeitsverteilung über positive reelle Zahlen mit dem Mittelwert $k\theta$. Es ist nicht notwendig, die Gamma-Verteilung genauer zu verstehen, um die Grundlagen hinter dem mPoT-Modell zu verstehen.

Die mPoT-Energiefunktion lautet

$$E_{\text{mPoT}}(\mathbf{x}, \mathbf{h}^{(m)}, \mathbf{h}^{(c)}) \quad (20.48)$$

$$= E_m(\mathbf{x}, \mathbf{h}^{(m)}) + \sum_j \left(h_j^{(c)} \left(1 + \frac{1}{2} \left(\mathbf{r}^{(j)\top} \mathbf{x} \right)^2 \right) + (1 - \gamma_j) \log h_j^{(c)} \right), \quad (20.49)$$

wobei $\mathbf{r}^{(j)}$ der Gewichtungsvektor der Kovarianz der Einheit $h_j^{(c)}$ ist und $E_m(\mathbf{x}, \mathbf{h}^{(m)})$ der Definition aus Gleichung 20.44 folgt.

Wie bei der mcRBM spezifiziert die Energiefunktion des mPoT-Modells eine multivariate Normalverteilung mit einer bedingten Verteilung über \mathbf{x} mit nicht-diagonaler Kovarianz.

Das Lernen im mPoT-Modell wird – wie bei der mcRBM – dadurch verkompliziert, dass es nicht möglich ist, Stichproben aus der nicht-diagonal bedingten Normalverteilung $p_{\text{mPoT}}(\mathbf{x} | \mathbf{h}^{(m)}, \mathbf{h}^{(c)})$ zu ziehen, sodass Ranzato et al. (2010b) außerdem für eine direktes Stichprobenentnahme von $p(\mathbf{x})$ mittels hamiltonschem (Hybrid)-Monte-Carlo-Sampling plädieren.

Spike and Slab Restricted Boltzmann Machine Spike and Slab Restricted Boltzmann Machines (eingeschränkte Spike-and-Slab-Boltzmann-Maschinen) (Courville et al., 2011) oder kurz ssRBMs sind eine weitere Möglichkeit zum Modellieren der Kovarianzstruktur reellwertiger Daten. Im Gegensatz zu mcRBMs bieten ssRBMs den Vorteil, dass weder Matrixinversion noch hamiltonsche Monte-Carlo-Verfahren zum Einsatz kommen. Wie bei mcRBM und mPoT codieren die binären verdeckten Einheiten der ssRBM die bedingte Kovarianz über Pixeln durch Verwendung reellwertiger Hilfsvariablen.

Die ssRBM weist zwei verdeckte Einheiten auf: binäre **Spike**-Einheiten \mathbf{h} und reellwertige **Slab**-Einheiten \mathbf{s} . Der Mittelwert der durch die verdeckten Einheiten bestimmten sichtbaren Einheiten ergibt sich aus $(\mathbf{h} \odot \mathbf{s})\mathbf{W}^\top$. Anders ausgedrückt: Jede Spalte $\mathbf{W}_{:,i}$ definiert eine Komponente, die in der Eingabe erscheinen kann, wenn $h_i = 1$ ist. Die entsprechende Spike-Variable h_i bestimmt, ob diese Komponente überhaupt vorliegt. Die entsprechende Slab-Variable s_i bestimmt die Intensität der Komponente sofern sie vorliegt. Ist eine Spike-Variable aktiv, fügt die zugehörige Slab-Variable Varianz zur Eingabe entlang der von $\mathbf{W}_{:,i}$ definierten Achse hinzu. So können wir die Kovarianz der Eingaben modellieren. Zum Glück können kontrastive Divergenz und persistente kontrastive Divergenz mit Gibbs-Sampling weiterhin angewandt werden. Eine Matrixinversion ist nicht erforderlich.

Formal wird das ssRBM-Modell über seine Energiefunktion definiert:

$$E_{\text{ss}}(\mathbf{x}, \mathbf{s}, \mathbf{h}) = - \sum_i \mathbf{x}^\top \mathbf{W}_{:,i} s_i h_i + \frac{1}{2} \mathbf{x}^\top \left(\mathbf{\Lambda} + \sum_i \mathbf{\Phi}_i h_i \right) \mathbf{x} \quad (20.50)$$

$$+ \frac{1}{2} \sum_i \alpha_i s_i^2 - \sum_i \alpha_i \mu_i s_i h_i - \sum_i b_i h_i + \sum_i \alpha_i \mu_i^2 h_i, \quad (20.51)$$

wobei b_i der Offset des Spikes h_i und Λ eine Diagonal-Präzisionsmatrix der Beobachtung \mathbf{x} ist. Der Parameter $\alpha_i > 0$ ist ein skalarer Präzisionsparameter für die reellwertige Slab-Variable s_i . Der Parameter Φ_i ist eine nicht-negative Diagonalmatrix, die einen \mathbf{h} -modulierten quadratischen Strafterm für \mathbf{x} definiert. Jedes μ_i ist ein Mittelwertparameter für die Slab-Variable s_i .

Mit der über die Energiefunktion definierten multivariaten Verteilung ist die Ableitung der bedingten ssRBM-Verteilungen recht problemlos. Ein Beispiel: Durch Entfernen der Slab-Variablen \mathbf{s} ergibt sich die bedingte Verteilung über die Beobachtungen für die binären Spike-Variablen \mathbf{h} aus

$$p_{ss}(\mathbf{x} | \mathbf{h}) = \frac{1}{P(\mathbf{h})} \frac{1}{Z} \int \exp \{-E(\mathbf{x}, \mathbf{s}, \mathbf{h})\} d\mathbf{s} \quad (20.52)$$

$$= \mathcal{N} \left(\mathbf{x}; \mathbf{C}_{\mathbf{x}|\mathbf{h}}^{\text{ss}} \sum_i \mathbf{W}_{:,i} \mu_i h_i, \mathbf{C}_{\mathbf{x}|\mathbf{h}}^{\text{ss}} \right) \quad (20.53)$$

mit $\mathbf{C}_{\mathbf{x}|\mathbf{h}}^{\text{ss}} = (\Lambda + \sum_i \Phi_i h_i - \sum_i \alpha_i^{-1} h_i \mathbf{W}_{:,i} \mathbf{W}_{:,i}^\top)^{-1}$. Die letzte Gleichung gilt nur, wenn die Kovarianzmatrix $\mathbf{C}_{\mathbf{x}|\mathbf{h}}^{\text{ss}}$ positiv definit ist.

Der Einsatz der Spike-Variablen als Gate bedeutet, dass die wahre Randverteilung über $\mathbf{h} \odot \mathbf{s}$ dünnbesetzt ist. Das unterscheidet sich von Sparse Coding, bei dem Stichproben aus dem Modell »fast nie« (im maßtheoretischen Sinne) Nullen im Code enthalten, wobei die MAP-Inferenz die dünne Besetzung auferlegen muss.

Ein Vergleich der ssRBM mit den mcRBM- und mPoT-Modellen zeigt, dass die ssRBM die bedingte Kovarianz der Beobachtung auf fundamental andere Weise parametrisiert. mcRBM und mPoT modellieren die Kovarianzstruktur der Beobachtung als $(\sum_j h_j^{(c)} \mathbf{r}^{(j)} \mathbf{r}^{(j)\top} + \mathbf{I})^{-1}$, wobei die Aktivierung der verdeckten Einheiten $h_j > 0$ genutzt wird, um Bedingungen für die bedingte Kovarianz in Richtung $\mathbf{r}^{(j)}$ zu erzwingen. Dagegen spezifiziert die ssRBM die bedingte Kovarianz der Beobachtungen mittels der verdeckten Spike-Aktivierungen $h_i = 1$, um die Präzisionsmatrix in der durch den zugehörigen Gewichtungsvektor angegebenen Richtung zu drücken. Die bedingte ssRBM-Kovarianz ähnelt der eines anderen Modells: dem Produkt der probabilistischen Hauptkomponentenanalyse (engl. *product of probabilistic principal component analysis*, PoPPCA) (Williams und Agakov, 2002). Im übervollständigen (engl. *overcomplete*) Fall erlauben schwache Aktivierungen mit der ssRBM-Parametrisierung nur in den durch schwach

aktiviertes h_i ausgewählten Richtungen eine nennenswerte Varianz (über die nominale Varianz als Ergebnis von Λ^{-1} hinaus). In den mcRBM- oder mPoT-Modellen würde eine übervollständige Repräsentation dazu führen, dass zum Erfassen der Variation in einer bestimmten Richtung im Beobachtungsraum potenziell alle Bedingungen mit positiver Projektion in dieser Richtung entfernt werden müssten. Das würde darauf hindeuten, dass diese Modelle weniger gut für den übervollständigen Fall geeignet sind.

Der größte Nachteil der ssRBM ist, dass einige Parametereinstellungen einer nicht positiv definiten Kovarianzmatrix entsprechen können. Eine solche Kovarianzmatrix führt zu einer stärker nicht normalisierten Wahrscheinlichkeit für Werte, die weiter entfernt vom Mittelwert liegen, sodass das Integral über allen möglichen Ergebnissen divergiert. Normalerweise lässt sich das mit einfachen heuristischen Tricks vermeiden. Es gibt bisher keine theoretisch zufriedenstellende Lösung. Der Einsatz der Optimierung unter Nebenbedingungen zur expliziten Vermeidung von Bereichen mit nicht definierter Wahrscheinlichkeit ist schwierig, wenn wir nicht extrem konservativ vorgehen möchten. Außerdem kann dies dazu führen, dass das Modell hochperformante Bereiche des Parameterraums nicht aufsucht.

Qualitativ erstellen gefaltete ssRBM-Varianten hervorragende Stichproben natürlicher. Einige Beispiele finden Sie in Abbildung 16.1.

Die ssRBM lässt sich auf unterschiedliche Weise erweitern. Mit Interaktionen höherer Ordnung und Average Pooling der Slab-Variablen (*Courville et al.*, 2014) kann das Modell exzellente Merkmale für einen Klassifikator erlernen, wenn nur wenige mit Labels gekennzeichnete Daten verfügbar sind. Durch Ergänzen eines Terms zur Energiefunktion, der eine nicht definierte Partitionsfunktion verhindert, erhalten wir ein Sparse-Coding-Modell, des Spike and Slab Sparse Codings (*Goodfellow et al.*, 2013d), kurz S3C.

20.6 Gefaltete Boltzmann-Maschinen

In Kapitel 9 haben wir gezeigt, dass extrem hochdimensionale Eingaben wie Bilder einen hohen Berechnungsaufwand, einen hohen Speicherbedarf und besondere statistische Anforderungen für Machine-Learning-Modelle bedeuten. Der Einsatz der diskreten Faltung mit einem kleinen Kernel anstelle der Matrizenmultiplikation löst diese Probleme normalerweise für Eingaben, die eine verschiebungsinvariante räumliche oder temporale Struktur aufweisen. *Desjardins und Bengio* (2008) haben gezeigt, dass dieser Ansatz auch bei RBMs gut funktioniert.

Für tiefe CNNs wird für gewöhnlich eine Pooling-Operation benötigt, damit die räumliche Größe der aufeinanderfolgenden Schichten zunimmt. Feedforward-CNNs nutzen häufig eine Pooling-Funktion wie den Maximumwert der zusammenfassenden Elemente. Es ist nicht klar, wie sich dies für energiebasierte Modelle generalisieren lässt. Wir könnten eine binäre Pooling-Einheit p über n binären Erkennungseinheiten (engl. *detector units*) d einführen und $p = \max_i d_i$ erzwingen, indem wir die Energiefunktion bei Verstößen gegen die Bedingung auf ∞ setzen. Dieses Verfahren skaliert jedoch nicht gut, da zum Berechnen der Normalisierungskonstante 2^n unterschiedliche Energiekonfigurationen evaluiert werden müssen. Für einen kleinen Pooling-Bereich von 3×3 müssen also für jede Pooling-Einheit $2^9 = 512$ Evaluationen der Energiefunktion erfolgen!

Lee et al. (2009) haben eine Lösung für dieses Problem entwickelt, das **probabilistische Max-Pooling** (nicht zu verwechseln mit dem »stochastischen Pooling«, bei dem Ensembles von Feedforward-CNNs implizit konstruiert werden). Das Verfahren hinter dem probabilistischen Max-Pooling besteht darin, die Erkennungseinheiten so einzuschränken, dass maximal eine davon zu einem bestimmten Zeitpunkt aktiv sein kann. Damit gibt es insgesamt nur $n+1$ Zustände (einen für jede der n möglichen Erkennungseinheiten und einen zusätzlichen Zustand, wenn keine der Erkennungseinheiten aktiv ist). Die Pooling-Einheit ist genau dann aktiv, wenn eine der Erkennungseinheiten aktiv ist. Sind alle Einheiten inaktiv, weist der Zustand die Energie Null auf. Wir können uns dies als Modell mit einer einzelnen Variable vorstellen, die $n+1$ Zustände annehmen kann – oder als Modell, das über $n+1$ Variablen verfügt, und die Energie ∞ allen bis auf $n+1$ übergreifenden Zuordnungen der Variablen zuweist.

Das probabilistische Max-Pooling ist zwar effizient, aber es zwingt Erkennungseinheiten, einander auszuschließen, was in einigen Fällen eine nützliche regularisierende Einschränkung oder in anderen Fällen ein gefährliches Limit der Modellkapazität darstellen kann. Auch werden keine Pooling-Bereiche mit Überschneidung unterstützt. Solche Pooling-Bereiche werden normalerweise benötigt, um die beste Leistung bei Feedforward-CNNs zu erzielen – diese Bedingung reduziert die Leistung der gefalteten Boltzmann-Maschinen daher wahrscheinlich beträchtlich.

Lee et al. (2009) haben gezeigt, dass probabilistisches Max-Pooling zum Konstruieren gefalteter DBMs eingesetzt werden könnte.³ Dieses Modell

³ Die Veröffentlichung beschreibt das Modell als »Deep-Belief-Netz«, aber da es als rein ungerichtetes Modell mit effizient berechenbaren schichtweisen Aktualisierung des Molekularfeld-Fixpunktes beschrieben werden kann, passt es besser zur Definition einer DBM.

kann Operationen wie das Ergänzen fehlender Eingabedaten durchführen. Obwohl dies theoretisch durchaus ansprechend ist, gibt es Herausforderungen bei der Umsetzung des Modells. Außerdem schneidet es als Klassifikator nicht so gut ab wie klassische CNNs, die mittels überwachtem Lernen trainiert wurden.

Viele Modelle mit Faltung funktionieren gleich gut mit Eingaben unterschiedlicher räumlicher Größe. Bei Boltzmann-Maschinen stellt das Ändern der Eingabegröße aus diversen Gründen eine Herausforderung dar. Die Partitionsfunktion ändert sich bei Änderungen der Eingabegröße. Außerdem erreichen viele CNNs eine Invarianz gegenüber der Größe, indem die Größe der Pooling-Bereiche proportional zur Größe der Eingabe skaliert wird – aber das Skalieren von Pooling-Bereichen für Boltzmann-Maschinen ist unhandlich. Klassische CNNs können eine feste Anzahl von Pooling-Einheiten nutzen und die Größe ihrer Pooling-Bereiche dynamisch erhöhen, um eine einheitlich große Repräsentation einer Eingabe mit variabler Größe zu erzielen. Bei Boltzmann-Maschinen sind große Pooling-Bereiche für den naiven Ansatz zu aufwendig. Der Ansatz von *Lee et al.* (2009), bei dem die Erkennungseinheiten im selben Pooling-Bereich einander ausschließen, löst zwar die Berechnungsprobleme, aber ermöglicht keine Pooling-Bereiche variabler Größe. Angenommen, wir erlernen ein Modell mit probabilistischem 2×2 -Max-Pooling über Erkennungseinheiten, die Kantendetektoren erlernen. Damit wird die Bedingung, dass nur eine dieser Kanten pro 2×2 -Bereich erscheinen darf, erzwungen. Wenn wir die Größe des Eingabebildes in jeder Richtung um 50 Prozent erhöhen, sollte sich auch die Anzahl der Kanten entsprechend erhöhen. Wenn wir stattdessen die Größe der Pooling-Bereiche um 50 Prozent in jeder Richtung auf 3×3 erhöhen, geben sich gegenseitig ausschließende Bedingungen an, dass jede dieser Kanten nur einmal in einem Bereich der Größe 3×3 auftauchen darf. Wenn wir das Eingabebild des Modells auf diese Weise vergrößern, erzeugt es Kanten mit geringerer Dichte. Natürlich kommt es nur zu diesen Problemen, wenn das Modell das Pooling variieren muss, um einen Ausgabevektor zu erzeugen, der stets denselben Betrag aufweist. Modelle mit probabilistischem Max-Pooling können problemlos Eingabebilder mit unterschiedlicher Größe annehmen, wenn die Ausgabe des Modells eine Merkmalskarte ist, die proportional zum Eingabebild skaliert werden kann.

Pixel an der Bildgrenze stellen ebenfalls ein gewisses Problem dar, das durch die symmetrische Eigenschaft der Verbindungen in einer Boltzmann-Maschine noch verstärkt wird. Wenn wir kein implizites Zero Padding der Eingabe durchführen, gibt es weniger verdeckte als sichtbare Einheiten, und die sichtbaren Einheiten an der Bildgrenze werden nicht gut modelliert, da

sie im rezeptiven Feld einer geringeren Anzahl verdeckter Einheiten liegen. Wenn wir allerdings ein implizites Zero Padding der Eingabe durchführen, werden die verdeckten Einheiten an der Grenze von weniger Eingabepixeln angetrieben, sodass ihre Aktivierung im Bedarfsfall möglicherweise fehlschlägt.

20.7 Boltzmann-Maschinen für strukturierte und sequenzielle Ausgaben

Im Falle der strukturierten Ausgabe möchten wir ein Modell trainieren, das eine Eingabe \mathbf{x} einer Ausgabe \mathbf{y} zuordnen kann, wobei die unterschiedlichen Einträge von \mathbf{y} in einer Beziehung zueinander stehen und außerdem gewisse Bedingungen erfüllen müssen. Zum Beispiel ist \mathbf{y} bei der Sprachsynthese eine Wellenform; die gesamte Wellenform muss sich wie eine kohärente Äußerung anhören.

Eine natürliche Möglichkeit zur Darstellung der Beziehungen zwischen den Einträgen in \mathbf{y} besteht in der Verwendung einer Wahrscheinlichkeitsverteilung $p(\mathbf{y} | \mathbf{x})$. Boltzmann-Maschinen, die für das Modellieren bedingter Verteilungen erweitert wurden, können dieses probabilistische Modell bereitstellen.

Ebenso kann die bedingte Modellierung mit einer Boltzmann-Maschine nicht nur für strukturierte Ausgabeaufgaben, sondern auch für die Sequenzmodellierung genutzt werden. Im letzteren Fall muss das Modell keine Eingabe \mathbf{x} einer Ausgabe \mathbf{y} zuordnen, sondern eine Wahrscheinlichkeitsverteilung über eine Sequenz von Variablen schätzen: $p(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)})$. Bedingte Boltzmann-Maschinen können Faktoren der Form $p(\mathbf{x}^{(t)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t-1)})$ repräsentieren, um diese Aufgabe zu erfüllen.

Eine wichtige Aufgabe der Sequenzmodellierung in der Videospiel- und Filmbranche ist das Modellieren von Sequenzen der Gelenke für die Darstellung von 3-D-Figuren. Diese Sequenzen werden häufig mithilfe von Motion-Capture-Systemen erfasst, die Bewegungen von Schauspielern aufnehmen. Ein probabilistisches Modell der Bewegungen einer Figur ermöglicht das Erzeugen neuer bisher ungesiehener Animationen auf realistische Weise. Zum Lösen dieser Sequenzmodellierungsaufgabe haben *Taylor et al. (2007)* eine bedingte RBM-Modellierung $p(\mathbf{x}^{(t)} | \mathbf{x}^{(t-1)}, \dots, \mathbf{x}^{(t-m)})$ für kleine m eingesetzt. Das Modell ist eine RBM über $p(\mathbf{x}^{(t)})$, deren Verzerrungsparameter eine lineare Funktion der vorherigen m -Werte von \mathbf{x} sind. Wenn wir für andere Werte von $\mathbf{x}^{(t-1)}$ und frühere Variablen abstimmen, erhalten wir eine neue RBM über \mathbf{x} . Die Gewichte in der RBM über \mathbf{x} ändern sich nie, aber

durch Konditionierung anhand unterschiedlicher vergangener Werte können wir die Wahrscheinlichkeit der Aktivierung der unterschiedlichen verdeckten Einheiten in der RBM ändern. Durch das Aktivieren und Deaktivieren unterschiedlicher Teilmengen mit verdeckten Einheiten können wir große Änderungen an der für \mathbf{x} induzierten Wahrscheinlichkeitsverteilung vornehmen. Andere Varianten der bedingten RBM (*Mnih et al.*, 2011) und andere Varianten der Sequenzmodellierung mithilfe bedingter RBMs sind möglich (*Taylor und Hinton*, 2009; *Sutskever et al.*, 2009; *Boulanger-Lewandowski et al.*, 2012).

Eine weitere Aufgabe der Sequenzmodellierung ist das Modellieren der Verteilung über Sequenzen von Noten, die zum Komponieren verwendet werden. *Boulanger-Lewandowski et al.* (2012) haben das **RNN-RBM**-Sequenzmodell vorgestellt und für diese Aufgabe genutzt. Die RNN-RBM ist ein generatives Modell einer Sequenz der Frames $\mathbf{x}^{(t)}$, bestehend aus einem RNN, das die RBM-Parameter für jeden Zeitschritt ausgibt. Anders als die bisherigen Ansätze, in denen nur der Verzerrungsparameter der RBM zwischen zwei Zeitschritten variierte, nutzt die RNN-RBM das RNN, um sämtliche Parameter der RBM auszugeben, einschließlich der Gewichte. Zum Trainieren des Modells müssen wir den Gradienten der Verlustfunktion durch das RNN backpropagieren. Die Verlustfunktion wird nicht direkt auf die RNN-Ausgaben angewandt. Stattdessen wird sie auf die RBM angewandt. Wir müssen daher den Verlust bezüglich der RBM-Parameter mittels kontrastiver Divergenz oder eines verwandten Algorithmus näherungsweise differenzieren. Dieser approximative Gradient kann dann durch das RNN backpropagiert werden – dazu wird der übliche BPTT-Algorithmus (Backpropagation Through Time) verwendet.

20.8 Weitere Boltzmann-Maschinen

Es gibt viele weitere Varianten von Boltzmann-Maschinen.

Boltzmann-Maschinen können mit unterschiedlichen Trainingskriterien erweitert werden. Wir haben uns auf Boltzmann-Maschinen konzentriert, die darauf trainiert werden, das generative Kriterium $\log p(\mathbf{v})$ näherungsweise zu maximieren. Es ist auch möglich, diskriminative RBMs zu trainieren, die versuchen, stattdessen $\log p(y \mid \mathbf{v})$ zu maximieren (*Larochelle und Bengio*, 2008). Dieser Ansatz funktioniert oft am besten, wenn eine Linearkombination aus generativen und diskriminativen Kriterien eingesetzt wird. Leider scheinen RBMs keine ebenso leistungsstarken überwachten Klassifikato-

ren wie MLPs zu sein, zumindest unter Berücksichtigung der bestehenden Methodologie.

Die meisten in der Praxis genutzten Boltzmann-Maschinen weisen lediglich Interaktionen zweiter Ordnung in ihren Energiefunktionen auf, sodass diese Energiefunktionen die Summe vieler Terme darstellen, von denen jeder einzelne nur das Produkt zwischen zwei Zufallsvariablen enthält. Ein Beispiel für einen solchen Term ist $v_i W_{i,j} h_j$. Es ist auch möglich, Boltzmann-Maschinen höherer Ordnung zu trainieren (*Sejnowski, 1987*), deren Energiefunktionsterme die Produkte zwischen vielen Variablen umfassen. Drei-Wege-Interaktionen zwischen einer verdeckten Einheit und zwei unterschiedlichen Bildern können räumliche Transformationen von einem Einzelbild eines Videos zum nächsten modellieren (*Memisevic und Hinton, 2007, 2010*). Durch Multiplikation mit einer One-hot-Klassenvariable kann die Beziehung zwischen sichtbaren und verdeckten Einheiten abhängig von der vorliegenden Klasse geändert werden (*Nair und Hinton, 2009*). Ein neueres Beispiel für den Einsatz von Interaktionen höherer Ordnung ist eine Boltzmann-Maschine mit zwei Gruppen verdeckter Einheiten: Die eine Gruppe interagiert sowohl mit den sichtbaren Einheiten v und den Klassen-Labels y , die andere Gruppe interagiert nur mit den v -Eingabewerten (*Luo et al., 2011*). Dies kann so interpretiert werden, dass einige verdeckte Einheiten dazu angeregt werden, zu lernen, die Eingabe anhand der Merkmale zu modellieren, die für die Klasse relevant sind. Gleichzeitig erlernen sie weitere verdeckte Einheiten, die unwichtige Details erklären, die aber erforderlich sind, damit die Stichproben von v realistisch sind, ohne dass die Klasse des Beispiels bestimmt werden muss. Eine weitere Verwendung von Interaktionen höherer Ordnung ist die Überwachung bestimmter Merkmale. *Sohn et al. (2013)* haben eine Boltzmann-Maschine mit Interaktionen dritter Ordnung und binären Maskierungsvariablen für alle sichtbaren Einheiten vorgestellt. Wenn diese Maskierungsvariablen auf Null gesetzt werden, entfernen sie den Einfluss einer sichtbaren Einheit auf die verdeckten Einheiten. So können sichtbare Einheiten, die für das Klassifizierungsproblem keine Relevanz haben, aus dem Pfad der Inferenz entfernt werden, der die Klasse schätzt.

Allgemeiner gesehen ist das Framework der Boltzmann-Maschine ein sehr großer Modellraum, der viel mehr Modellstrukturen zulässt, als bisher untersucht wurden. Das Entwickeln einer neuen Form von Boltzmann-Maschine erfordert etwas mehr Sorgfalt und Kreativität als das Entwickeln einer neuen Schicht eines neuronalen Netzes, da es oft Probleme bereitet, eine Energiefunktion zu finden, die die effiziente Berechenbarkeit all der unterschiedlichen bedingten Verteilungen beibehält, die für die Nutzung der

Boltzmann-Maschine benötigt werden. Trotz dieses Aufwands ist das Feld weiterhin offen für Innovationen.

20.9 Backpropagation durch Zufallsoperationen

Klassische neuronale Netze implementieren eine deterministische Transformation einiger Eingangsvariablen \mathbf{x} . Beim Entwickeln generativer Modelle möchten wir neuronale Netze häufig für die Implementierung stochastischer Transformationen von \mathbf{x} erweitern. Eine einfache Möglichkeit, dies zu tun, besteht darin, das neuronale Netz mit zusätzlichen Eingaben \mathbf{z} zu erweitern, die aus einer einfachen Wahrscheinlichkeitsverteilung gezogen werden, zum Beispiel aus einer Gleichverteilung oder einer Normalverteilung. Das neuronale Netz kann dann intern mit der deterministischen Berechnung fortfahren, während die Funktion $f(\mathbf{x}, \mathbf{z})$ auf einen Beobachter, der keinen Zugang zu \mathbf{z} hat, stochastisch wirkt. Sofern f stetig und differenzierbar ist, können wir die für das Training mittels Backpropagation notwendigen Gradienten wie gewohnt berechnen.

Ein Beispiel: Es werden Stichproben y aus einer Normalverteilung mit dem Mittelwert μ und der Varianz σ^2 gezogen:

$$y \sim \mathcal{N}(\mu, \sigma^2). \quad (20.54)$$

Da eine einzelne Stichprobe von y nicht durch eine Funktion, sondern durch eine Stichprobenentnahme erzeugt wird, deren Ausgabe sich bei jedem Aufruf ändert, erscheint es vielleicht gegen die eigene Intuition zu sein, die Ableitungen von y bezüglich der Parameter der eigenen Verteilung, μ und σ^2 , zu nehmen. Allerdings können wir die Stichprobenentnahme als Transformation eines zugrunde liegenden Wertes $z \sim \mathcal{N}(z; 0, 1)$ zur Ermittlung einer Stichprobe aus der gewünschten Verteilung neu schreiben:

$$y = \mu + \sigma z. \quad (20.55)$$

Nun ist die Backpropagation durch die Stichprobenentnahme möglich, indem wir sie als deterministische Operation mit einer zusätzlichen Eingabe z betrachten. Es ist wichtig, dass die zusätzliche Eingabe eine Zufallsvariable ist, deren Verteilung keine Funktion einer der anderen Variablen ist, deren Ableitungen wir berechnen möchten. Das Ergebnis zeigt, wie eine infinitesimalen Änderung von μ oder σ die Ausgabe verändert würde, wenn wir die Stichprobenentnahme nochmals mit demselben Wert für z wiederholen könnten.

Die Möglichkeit zur Backpropagation durch diese Stichprobenentnahme erlaubt es uns, diese in einen größeren Graphen zu integrieren. Wir können Elemente des Graphen aufbauend auf der Ausgabe der Stichprobenverteilung erstellen. Zum Beispiel können wir die Ableitungen einer Verlustfunktion $J(y)$ berechnen. Wir können auch Elemente des Graphen erstellen, deren Ausgaben als Eingaben oder Parameter der Stichprobenentnahme dienen. Zum Beispiel könnten wir einen größeren Graphen mit $\mu = f(\mathbf{x}; \boldsymbol{\theta})$ und $\sigma = g(\mathbf{x}; \boldsymbol{\theta})$ erstellen. In diesem erweiterten Graphen können wir die Backpropagation durch diese Funktionen zur Ableitung von $\nabla_{\boldsymbol{\theta}} J(y)$ anwenden.

Das Prinzip, das in diesem Beispiel für eine Stichprobenentnahme aus einer Normalverteilung angewendet wurde, lässt sich auch generalisieren. Wir können jede Wahrscheinlichkeitsverteilung der Form $p(y; \boldsymbol{\theta})$ oder $p(y | \mathbf{x}; \boldsymbol{\theta})$ als $p(y | \boldsymbol{\omega})$ ausdrücken, wobei $\boldsymbol{\omega}$ eine Variable ist, die die beiden Parameter $\boldsymbol{\theta}$ und gegebenenfalls die Eingaben \mathbf{x} enthält. Für einen Wert y , der aus der Verteilung $p(y | \boldsymbol{\omega})$ gezogen wurde, wobei $\boldsymbol{\omega}$ wiederum eine Funktion anderer Variablen sein kann, können wir

$$\mathbf{y} \sim p(\mathbf{y} | \boldsymbol{\omega}) \quad (20.56)$$

umformen zu

$$\mathbf{y} = f(\mathbf{z}; \boldsymbol{\omega}), \quad (20.57)$$

wobei \mathbf{z} eine Quelle der Zufälligkeit ist. Wir können dann die Ableitungen von \mathbf{y} bezüglich $\boldsymbol{\omega}$ mit den üblichen Tools wie dem Backpropagation-Algorithmus für f berechnen, sofern f fast überall stetig und differenzierbar ist. Es ist entscheidend, dass $\boldsymbol{\omega}$ keine Funktion von \mathbf{z} ist und \mathbf{z} keine Funktion von $\boldsymbol{\omega}$. Dieses Verfahren wird häufig als **Reparametrisierung** (engl. *reparametrization trick*), **stochastische Backpropagation** oder **Störungsanalyse** (engl. *perturbation analysis*) bezeichnet.

Die Bedingung, dass f stetig und differenzierbar sein muss, erfordert natürlich auch, dass \mathbf{y} stetig ist. Wenn eine Backpropagation durch Stichprobenentnahme durchgeführt werden soll, die diskrete Stichproben erzeugt, ist es vielleicht immer noch möglich, einen Gradienten auf $\boldsymbol{\omega}$ zu schätzen, indem Algorithmen für Reinforcement Learning (dt. *bestärkendes* oder *verstärkendes Lernen*) eingesetzt werden – zum Beispiel mittels Varianten des REINFORCE-Algorithmus (Williams, 1992), der in Abschnitt 20.9.1 behandelt wird.

In Anwendungen mit neuronalen Netzen entscheiden wir uns üblicherweise dafür, \mathbf{z} aus einer einfachen Verteilung zu ziehen, zum Beispiel einer Standard-Gleichverteilung oder einer Standardnormalverteilung; komplexere Verteilungen erreichen wir, indem wir zulassen, dass der deterministische Teil des Netzes seine Eingabe umformt.

Die Idee der Gradientenpropagation oder der Optimierung durch stochastische Operationen kam bereits Mitte des zwanzigsten Jahrhunderts auf (Price, 1958; Bonnet, 1964) und wurde erstmals im Rahmen des Reinforcement Learnings für das Machine Learning eingesetzt (Williams, 1992). Erst kürzlich wurde sie für Variationsapproximationen (Opper und Archambeau, 2009) sowie stochastische und generative neuronale Netze verwendet (Bengio et al., 2013b; Kingma, 2013; Kingma und Welling, 2014b,a; Rezende et al., 2014; Goodfellow et al., 2014c). Viele Netze, wie Denoising Autoencoder oder mit Dropout regularisierte Netze, sind auf natürliche Weise dafür entworfen, Rauschen als Eingabe entgegenzunehmen, ohne dass eine besondere Reparametrisierung erforderlich wäre, die das Rauschen vom Modell entkoppelt.

20.9.1 Backpropagation mittels diskreter stochastischer Operationen

Wenn ein Modell eine diskrete Variable \mathbf{y} ausgibt, kann die Reparametrisierung nicht genutzt werden. Angenommen, das Modell verfügt über die Eingaben \mathbf{x} und die Parameter $\boldsymbol{\theta}$, beide zusammengefasst im Vektor $\boldsymbol{\omega}$, und kombiniert diese mit Zufallsrauschen \mathbf{z} zum Erzeugen von \mathbf{y} :

$$\mathbf{y} = f(\mathbf{z}; \boldsymbol{\omega}). \quad (20.58)$$

Da \mathbf{y} diskret ist, muss f eine Treppenfunktion sein. Die Ableitungen einer Treppenfunktion sind in keinem Punkt nützlich. Direkt an der Grenze jeder Stufe sind die Ableitungen nicht definiert, aber das ist nur ein kleines Problem. Das viel größere Problem ist, dass die Ableitungen fast überall in den Bereichen zwischen den Grenzen der Stufen Null sind. Die Ableitungen jeder Kostenfunktion $J(\mathbf{y})$ enthalten daher keine Informationen darüber, wie die Modellparameter $\boldsymbol{\theta}$ aktualisiert werden können.

Der REINFORCE-Algorithmus (**R**Eward **I**ncrement = **N**onnegative **F**actor \times **O**ffset **R**einforcement \times **C**haracteristic **E**ligibility) bietet ein Framework, das eine Familie einfacher, aber leistungsstarker Lösungen definiert (Williams, 1992). Die Grundidee ist, dass – obwohl es sich bei $J(f(\mathbf{z}; \boldsymbol{\omega}))$ um eine Treppenfunktion mit unbrauchbaren Ableitungen handelt – der erwartete Aufwand $\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} J(f(\mathbf{z}; \boldsymbol{\omega}))$ häufig eine glatte Funktion ist, die sich für das Gradientenabstiegsverfahren gut eignet. Obwohl dieser Erwartungswert üblicherweise für hochdimensionale \mathbf{y} nicht effizient berechenbar ist (oder das Ergebnis einer Komposition vieler diskreter stochastischer Entscheidungen ist), kann er ohne Verzerrung mittels Monte-Carlo-Durchschnittswert geschätzt werden. Der stochastische Schätzwert des Gradienten

kann mit SGD oder anderen stochastischen Optimierungsverfahren auf Gradientenbasis verwendet werden.

Die einfachste Version von REINFORCE erhalten wir durch einfache Differentiation des erwarteten Aufwands:

$$\mathbb{E}_z[J(\mathbf{y})] = \sum_{\mathbf{y}} J(\mathbf{y}) p(\mathbf{y}), \quad (20.59)$$

$$\frac{\partial \mathbb{E}[J(\mathbf{y})]}{\partial \omega} = \sum_{\mathbf{y}} J(\mathbf{y}) \frac{\partial p(\mathbf{y})}{\partial \omega} \quad (20.60)$$

$$= \sum_{\mathbf{y}} J(\mathbf{y}) p(\mathbf{y}) \frac{\partial \log p(\mathbf{y})}{\partial \omega} \quad (20.61)$$

$$\approx \frac{1}{m} \sum_{\mathbf{y}^{(i)} \sim p(\mathbf{y}), i=1}^m J(\mathbf{y}^{(i)}) \frac{\partial \log p(\mathbf{y}^{(i)})}{\partial \omega}. \quad (20.62)$$

Gleichung 20.60 setzt voraus, dass J keine direkte Referenz auf ω enthält. Es ist trivial, den Ansatz so zu erweitern, dass diese Voraussetzungen nicht erfüllt werden müssen. Gleichung 20.61 nutzt die Ableitungsregel für den Logarithmus, $\frac{\partial \log p(\mathbf{y})}{\partial \omega} = \frac{1}{p(\mathbf{y})} \frac{\partial p(\mathbf{y})}{\partial \omega}$. Gleichung 20.62 stellt einen erwartungstreuen Monte-Carlo-Schätzer des Gradienten bereit.

Überall dort, wo wir in diesem Abschnitt $p(\mathbf{y})$ schreiben, können wir ebenso gut $p(\mathbf{y} | \mathbf{x})$ schreiben. Das liegt daran, dass $p(\mathbf{y})$ durch ω parametrisiert wird und ω sowohl θ als auch \mathbf{x} enthält, wenn \mathbf{x} vorliegt.

Ein Problem des einfachen REINFORCE-Schätzers ist seine sehr hohe Varianz, sodass viele Stichproben aus \mathbf{y} gezogen werden müssen, um einen guten Schätzer des Gradienten zu erhalten; äquivalent gilt, dass beim Ziehen nur einer Stichprobe das SGD sehr langsam konvergiert und eine kleinere Lernrate voraussetzt. Es ist möglich, die Varianz dieses Schätzers mithilfe von Verfahren zur **Varianzreduktion** deutlich zu reduzieren (Wilson, 1984; L'Ecuyer, 1994). Dabei wird der Schätzer so verändert, dass sein Erwartungswert unverändert bleibt, während die Varianz reduziert wird. Für REINFORCE umfassen die empfohlenen Verfahren zur Varianzreduktion die Berechnung einer **Baseline**, mit der ein Offset von $J(\mathbf{y})$ erfolgt. Jedes Offset $b(\omega)$, das nicht von \mathbf{y} abhängig ist, führt nicht zu einer Änderung des Erwartungswerts des geschätzten Gradienten, da

$$E_{p(\mathbf{y})} \left[\frac{\partial \log p(\mathbf{y})}{\partial \omega} \right] = \sum_{\mathbf{y}} p(\mathbf{y}) \frac{\partial \log p(\mathbf{y})}{\partial \omega} \quad (20.63)$$

$$= \sum_{\mathbf{y}} \frac{\partial p(\mathbf{y})}{\partial \omega} \quad (20.64)$$

$$= \frac{\partial}{\partial \omega} \sum_{\mathbf{y}} p(\mathbf{y}) = \frac{\partial}{\partial \omega} 1 = 0 \quad (20.65)$$

ist, woraus folgt, dass

$$E_{p(\mathbf{y})} \left[(J(\mathbf{y}) - b(\omega)) \frac{\partial \log p(\mathbf{y})}{\partial \omega} \right] \quad (20.66)$$

$$= E_{p(\mathbf{y})} \left[J(\mathbf{y}) \frac{\partial \log p(\mathbf{y})}{\partial \omega} \right] - b(\omega) E_{p(\mathbf{y})} \left[\frac{\partial \log p(\mathbf{y})}{\partial \omega} \right] \quad (20.66)$$

$$= E_{p(\mathbf{y})} \left[J(\mathbf{y}) \frac{\partial \log p(\mathbf{y})}{\partial \omega} \right] \quad (20.67)$$

ist. Weiterhin können wir das optimale $b(\omega)$ durch Berechnung der Varianz von $(J(\mathbf{y}) - b(\omega)) \frac{\partial \log p(\mathbf{y})}{\partial \omega}$ unter $p(\mathbf{y})$ und Minimieren bezüglich $b(\omega)$ bestimmen. Wir stellen fest, dass diese optimale Baseline $b^*(\omega)_i$ für jedes Element ω_i des Vektors ω eine andere ist:

$$b^*(\omega)_i = \frac{E_{p(\mathbf{y})} \left[J(\mathbf{y}) \frac{\partial \log p(\mathbf{y})^2}{\partial \omega_i} \right]}{E_{p(\mathbf{y})} \left[\frac{\partial \log p(\mathbf{y})^2}{\partial \omega_i} \right]}. \quad (20.68)$$

Der Gradientenschätzer bezüglich ω_i wird dann

$$(J(\mathbf{y}) - b(\omega)_i) \frac{\partial \log p(\mathbf{y})}{\partial \omega_i}, \quad (20.69)$$

wobei $b(\omega)_i$ das obige $b^*(\omega)_i$ schätzt. Der Schätzwert b wird meist durch Hinzufügen zusätzlicher Ausgaben zum neuronalen Netz und Trainieren der neuen Ausgaben zur Schätzung von $E_{p(\mathbf{y})} [J(\mathbf{y}) \frac{\partial \log p(\mathbf{y})^2}{\partial \omega_i}]$ und $E_{p(\mathbf{y})} \left[\frac{\partial \log p(\mathbf{y})^2}{\partial \omega_i} \right]$ für jedes Element von ω ermittelt. Diese zusätzlichen Ausgaben können mit der Zielfunktion des mittleren quadratischen Fehlers trainiert werden, wobei $J(\mathbf{y}) \frac{\partial \log p(\mathbf{y})^2}{\partial \omega_i}$ bzw. $\frac{\partial \log p(\mathbf{y})^2}{\partial \omega_i}$ als Zielwert verwendet wird, wenn \mathbf{y} aus $p(\mathbf{y})$ für ein gegebenes ω gezogen wird. Der Schätzwert b kann dann durch Einsetzen dieser Schätzwerte in Gleichung 20.68 wiederhergestellt werden. *Mnih und Gregor (2014)* präferieren eine einzelne gemeinsame Ausgabe (für alle Elemente i aus ω), die mit der Zielfunktion $J(\mathbf{y})$ anhand einer Baseline $b(\omega) \approx E_{p(\mathbf{y})} [J(\mathbf{y})]$ trainiert wird.

Verfahren zur Varianzreduktion wurden im Bereich des Reinforcement Learnings eingeführt (*Sutton et al., 2000; Weaver und Tao, 2001*), die frühere Arbeiten zur Binary Reward (binäre Belohnung) von *Dayan (1990)* generalisiert haben. Beispiele für moderne Anwendungen des REINFORCE-

Algorithmus mit reduzierter Varianz im Deep-Learning-Kontext finden Sie in *Bengio et al.* (2013b), *Mnih und Gregor* (2014), *Ba et al.* (2014), *Mnih et al.* (2014) sowie *Xu et al.* (2015). Neben dem Nutzen einer von der Eingabe abhängigen Baseline $b(\omega)$ haben *Mnih und Gregor* (2014) herausgefunden, dass der Maßstab von $(J(\mathbf{y}) - b(\omega))$ während des Trainings angepasst werden kann, und zwar durch Division durch seine Standardabweichung, geschätzt durch einen gleitenden Mittelwert, ähnlich einer adaptiven Lernrate, um den Auswirkungen der maßgeblichen Variationen entgegenzuwirken, die während des Trainings in der entsprechenden Größenordnung auftreten. *Mnih und Gregor* (2014) nennen dies eine heuristische **Normalisierung der Varianz** (engl. *variance normalization*).

REINFORCE-basierte Schätzer stellen eine Art Schätzung des Gradienten durch korrelierende Wahl von \mathbf{y} mit entsprechenden Werten für $J(\mathbf{y})$ dar. Wenn ein guter Wert für \mathbf{y} für die gegebene Parametrisierung unwahrscheinlich ist, dauert es möglicherweise sehr lange, ihn zufällig zu ermitteln und das erforderliche Signal zu erhalten, dass diese Konfiguration bestärkt werden soll..

20.10 Gerichtete generative Netze (Directed Generative Nets)

In Kapitel 16 haben wir gezeigt, dass gerichtete graphische Modelle eine wichtige Klasse graphischer Modelle darstellen. Obwohl gerichtete graphische Modelle in der größeren Machine-Learning-Forschungsgemeinde sehr beliebt sind, wurden sie in der kleineren Deep-Learning-Forschungsgemeinde bis etwa 2013 von ungerichteten Modellen wie der RBM in den Schatten gestellt.

In diesem Abschnitt betrachten wir einige der gängigen gerichteten graphischen Modelle, die typischerweise mit der Deep-Learning-Forschungsgemeinde in Verbindung gebracht werden.

Wir haben bereits Deep-Belief-Netze beschrieben; diese sind ein partiell gerichtetes Modell. Wir haben auch Sparse-Coding-Modelle behandelt, die eine Art flaches gerichtetes generatives Modell darstellen. Häufig werden sie im Deep-Learning-Kontext als Klassifikatoren für Merkmale eingesetzt, obwohl sie bei der Stichprobengenerierung und der Dichteschätzung eher schlecht abschneiden. Nun widmen wir uns einer Reihe tiefer vollständig gerichteter Modelle.

20.10.1 Sigmoid-Belief-Netze

Sigmoid-Belief-Netze (Neal, 1990) sind eine Form des einfachen gerichteten graphischen Modells mit einer besonderen Art bedingter Wahrscheinlichkeitsverteilung. Grundsätzlich können wir uns vorstellen, dass ein Sigmoid-Belief-Netz einen Vektor aus binären Zuständen s aufweist, in dem jedes Zustandselement durch seine Vorgänger beeinflusst wird:

$$p(s_i) = \sigma \left(\sum_{j < i} W_{j,i} s_j + b_i \right). \quad (20.70)$$

Die häufigste Struktur des Sigmoid-Belief-Netzes ist eine, die in viele Schichten unterteilt ist, wobei ein Ancestral Sampling eine Reihe mit vielen verdeckten Schichten durchläuft und schließlich die sichtbare Schicht erzeugt. Diese Struktur ähnelt stark dem Deep-Belief-Netz; allerdings sind die Einheiten zu Beginn der Stichprobenentnahme unabhängig voneinander und werden nicht aus einer RBM gezogen. Eine solche Struktur ist aus mehreren Gründen interessant. Einer davon ist, dass die Struktur insofern ein universeller Approximator für Wahrscheinlichkeitsverteilungen über die sichtbaren Einheiten ist, als sie jegliche Wahrscheinlichkeitsverteilung über binäre Variablen beliebig gut approximieren kann, sofern die Tiefe hinreichend groß ist, und zwar auch dann, wenn die Breite der einzelnen Schichten auf die Dimensionalität der sichtbaren Schicht eingeschränkt ist (Sutskever und Hinton, 2008).

Die Stichprobengenerierung der sichtbaren Einheiten ist in einem Sigmoid-Belief-Netz sehr effizient, aber das gilt nicht für die meisten anderen Operationen. Inferenz über den verdeckten Einheiten auf Grundlage der sichtbaren Einheiten ist nicht effizient durchführbar. Auch die Molekularfeld-Inferenz ist nicht effizient durchführbar, da für die ELBO Erwartungswerte für Cliques genommen werden müssen, die ganze Schichten umfassen. Dieses Problem ist nach wie vor schwierig genug, um die Beliebtheit gerichteter diskreter Netze einzuschränken.

Ein Ansatz für das Durchführen von Inferenz in einem Sigmoid-Belief-Netz besteht darin, eine andere untere Schranke zu konstruieren, die auf Sigmoid-Belief-Netze spezialisiert ist (Saul et al., 1996). Dieser Ansatz wurde und wird nur für sehr kleine Netze genutzt. Ein weiterer Ansatz besteht darin, die erlernten Inferenzmechanismen wie in Abschnitt 19.5 beschrieben zu nutzen. Die Helmholtz-Maschine (Dayan et al., 1995; Dayan und Hinton, 1996) ist ein Sigmoid-Belief-Netz in Kombination mit einem Inferenznetz, das die Parameter der Molekularfeld-Verteilung über die verdeckten Einheiten

vorhersagt. Moderne Ansätze für Sigmoid-Belief-Netze (*Gregor et al.*, 2014; *Mnih und Gregor*, 2014) nutzen den Ansatz mit Inferenznetz noch immer. Diese Verfahren bleiben aufgrund der diskreten Natur der latenten Variablen schwierig. Eine Backpropagation durch die Ausgabe des Inferenznetzes ist nicht so einfach möglich. Stattdessen ist ein relativ unzuverlässiges System für die Backpropagation durch diskrete Stichprobenverfahren erforderlich (vgl. Abschnitt 20.9.1). Neuere Ansätze basieren auf Importance Sampling, neu gewichteten Wake-Sleep- (*Bornschein und Bengio*, 2015) und bidirektionalen Helmholtz-Maschinen (*Bornschein et al.*, 2015). Sie ermöglichen ein schnelles Training von Sigmoid-Belief-Netzen und erreichen zeitgemäße Leistungskennzahlen in Benchmark-Aufgaben.

Ein Sonderfall der Sigmoid-Belief-Netze liegt vor, wenn es keine latenten Variablen gibt. Das Lernen ist in diesem Fall effizient, da keine latenten Variablen aus der Likelihood entfernt werden müssen. Eine Modellfamilie, die *autoregressiven Netze*, generalisiert dieses vollständig sichtbare Belief-Netz für andere Arten von Variablen außer binären Variablen und andere Strukturen bedingter Verteilungen außer logarithmisch-linearen Beziehungen. Autoregressive Netze werden in Abschnitt 20.10.7 behandelt.

20.10.2 Differenzierbare Generator-Netze (Differentiable Generator Networks)

Viele generative Modelle beruhen auf der Idee, ein differenzierbares **Generator-Netz** einzusetzen. Das Modell transformiert Stichproben von latenten Variablen \mathbf{z} zu Stichproben \mathbf{x} oder Verteilungen über Stichproben \mathbf{x} mithilfe einer differenzierbaren Funktion $g(\mathbf{z}; \theta^{(g)})$, die meist durch ein neuronales Netz repräsentiert wird. Dieses Modell umfasst VAEs (Variational Autoencoder), die das Generator-Netz mit einem Inferenznetz koppeln, Generative-Adversarial-Netze, die das Generator-Netz mit einem Diskriminatoren-Netz (engl. *discriminator network*) koppeln, und Verfahren, die Generator-Netze isoliert trainieren.

Generator-Netze sind im Wesentlichen lediglich parametrisierte Rechenverfahren zur Stichprobengenerierung, in denen die Architektur die Familie möglicher Verteilungen bereitstellt, aus denen Stichproben gezogen werden können; die Parameter wählen eine Verteilung aus der Familie aus.

Ein Beispiel: Das übliche Verfahren zur Stichprobenentnahme aus einer Normalverteilung mit Mittelwert μ und Kovarianz Σ besteht darin, Stichproben \mathbf{z} aus einer Normalverteilung mit Mittelwert gleich Null und

Kovarianz gleich Eins in ein sehr einfaches Generator-Netz einzuspeisen. Dieses Generator-Netz enthält nur eine affine Schicht:

$$\mathbf{x} = g(\mathbf{z}) = \mu + \mathbf{L}\mathbf{z}, \quad (20.71)$$

wobei sich \mathbf{L} aus der Cholesky-Zerlegung von Σ ergibt.

Pseudozufallszahlengeneratoren können ebenfalls nichtlineare Transformationen einfacher Verteilungen verwenden. Zum Beispiel verwendet die **Inversionsmethode** (engl. *inverse transform sampling*) (Devroye, 2013) einen Skalar z aus $U(0, 1)$ und wendet eine nichtlineare Transformation auf einen Skalar x an. In diesem Fall ergibt sich $g(z)$ aus der Inversen der kumulativen Verteilungsfunktion $F(x) = \int_{-\infty}^x p(v)dv$. Wenn wir $p(x)$ spezifizieren, über x integrieren und die resultierende Funktion invertieren können, können wir das Ziehen von Stichproben aus $p(x)$ ganz ohne Machine Learning vornehmen.

Um Stichproben aus komplexeren Verteilungen zu generieren, die Probleme bei direkter Spezifizierung, Integration oder Invertierung der resultierenden Integrale bereiten, verwenden wir ein Feedforward-Netz zur Repräsentation einer parametrischen Familie nichtlinearer Funktionen g und nutzen Trainingsdaten zum Schließen auf die Parameter zur Wahl der gewünschten Funktion.

Wir können uns vorstellen, dass g eine nichtlineare Änderung der Variablen bereitstellt, die die Verteilung über \mathbf{z} in die gesuchte Verteilung über \mathbf{x} transformiert.

Aus Gleichung 3.47 wissen Sie, dass für invertierbares differenzierbares stetiges g gilt:

$$p_z(\mathbf{z}) = p_x(g(\mathbf{z})) \left| \det\left(\frac{\partial g}{\partial \mathbf{z}}\right) \right|. \quad (20.72)$$

Das führt implizit zu einer Wahrscheinlichkeitsverteilung über \mathbf{x} :

$$p_x(\mathbf{x}) = \frac{p_z(g^{-1}(\mathbf{x}))}{\left| \det\left(\frac{\partial g}{\partial \mathbf{z}}\right) \right|}. \quad (20.73)$$

Natürlich ist diese Formel möglicherweise schwer auszuwerten – je nach Wahl von g . Daher nutzen wir oft indirekte Mittel zum Erlernen von g und versuchen gar nicht erst, $\log p(\mathbf{x})$ direkt zu maximieren.

Manchmal verwenden wir g nicht, um direkt eine Stichprobe aus \mathbf{x} bereitzustellen, sondern nutzen g , um eine bedingte Verteilung über \mathbf{x} zu definieren. Wir könnten zum Beispiel ein Generator-Netz verwenden, dessen

letzte Schicht aus sigmoiden Ausgaben zur Angabe der Mittelwertparameter von Bernoulli-Verteilungen besteht:

$$p(x_i = 1 \mid \mathbf{z}) = g(\mathbf{z})_i. \quad (20.74)$$

In diesem Fall, also bei Verwendung von g zur Definition von $p(\mathbf{x} \mid \mathbf{z})$, erzwingen wir eine Verteilung über \mathbf{x} durch Marginalisieren von \mathbf{z} :

$$p(\mathbf{x}) = \mathbb{E}_{\mathbf{z}} p(\mathbf{x} \mid \mathbf{z}). \quad (20.75)$$

Beide Ansätze definieren eine Verteilung $p_g(\mathbf{x})$ und erlauben es uns, verschiedene Kriterien von p_g mithilfe der Reparametrisierung aus Abschnitt 20.9 zu trainieren.

Die beiden unterschiedlichen Ansätze zur Bildung von Generator-Netzen – Ausgeben der Parameter einer bedingten Verteilung und direktes Ausgeben von Stichproben – haben komplementäre Stärken und Schwächen. Wenn das Generator-Netz eine bedingte Verteilung über \mathbf{x} definiert, kann es diskrete und stetige Daten gleichermaßen gut erzeugen. Wenn das Generator-Netz Stichproben direkt ausgibt, kann es nur stetige Daten erzeugen (wir könnten eine Diskretisierung in der Forward-Propagation einführen, aber dann ließe sich das Modell nicht mehr mittels Backpropagation trainieren). Der Vorteil der direkten Stichprobenentnahme ist der, dass wir nicht länger bedingte Verteilungen verwenden müssen, deren Form problemlos von einem Menschen notiert und algebraisch verändert werden kann.

Auf differenzierbaren Generator-Netzen basierende Ansätze sind durch den Erfolg des Gradientenabstiegsverfahrens in differenzierbaren Feedforward-Netzen für Klassifizierungsaufgaben motiviert. Im Kontext des überwachten Lernens scheinen auf Gradientenbasis trainierte tiefe Feedforward-Netze praktisch garantiert Erfolg zu haben, sofern es hinreichend verdeckte Einheiten und genügend Trainingsdaten gibt. Lässt sich dieses Erfolgsrezept auch auf die generative Modellierung übertragen?

Die generative Modellierung erscheint schwieriger als die Klassifizierung oder Regression, da im Lernprozess nicht effizient berechenbare Kriterien optimiert werden müssen. In Zusammenhang mit differenzierbaren Generator-Netzen sind die Kriterien nicht effizient berechenbar, da die Daten nicht gleichzeitig Eingaben \mathbf{z} und Ausgaben \mathbf{x} des Generator-Netzes spezifizieren. Im Fall des überwachten Lernens waren Eingaben \mathbf{x} und Ausgaben \mathbf{y} bekannt; das Optimierungsverfahren muss lediglich lernen, wie die spezifizierte Zuordnung erzeugt werden kann. Im Fall der generativen Modellierung muss das Lernverfahren bestimmen, wie der \mathbf{z} -Raum auf nützliche Weise angeordnet wird und wie die Zuordnung von \mathbf{z} zu \mathbf{x} erfolgt.

Dosovitskiy et al. (2015) haben ein vereinfachtes Problem untersucht, bei dem die Korrespondenz zwischen z und x gegeben ist. Dabei dienten vom Computer erzeugte Bilder von Stühlen als Trainingsdaten. Die latenten Variablen z sind Parameter, die dem Rendering-Algorithmus übergeben wurden und die Wahl des Stuhlmodells, die Position des Stuhls und weitere Konfigurationsdetails, die das Rendering beeinflussen, beschreiben. Anhand dieser synthetisch erzeugten Daten kann ein CNN lernen, z Beschreibungen des Bildinhalts den x Approximationen der gerenderten Bilder zuzuordnen. Es deutet sich somit an, dass aktuelle differenzierbare Generator-Netze eine hinreichende Modellkapazität aufweisen, um als gute generative Modelle dienen zu können, und dass die aktuellen Optimierungsalgorithmen die dafür erforderliche Anpassungsfähigkeit liefern. Die Schwierigkeit liegt darin, zu bestimmen, wie Generator-Netze trainiert werden, wenn der Wert von z für jedes x nicht fest und im Voraus bekannt ist.

Die folgenden Abschnitte beschreiben diverse Ansätze für das Training von differenzierbaren Generator-Netzen allein auf Basis der Trainingsstichproben von x .

20.10.3 Variational Autoencoder (VAE)

Der **Variational Autoencoder** oder VAE (*Kingma, 2013; Rezende et al., 2014*) ist ein gerichtetes Modell, das die erlernte approximative Inferenz nutzt und allein mit Verfahren auf Gradientenbasis trainiert werden kann.

Um eine Stichprobe aus dem Modell zu generieren, zieht der VAE zunächst eine Stichprobe z aus der Code-Verteilung $p_{\text{model}}(z)$. Die Stichprobe wird dann in einem differenzierbaren Generator-Netz $g(z)$ verarbeitet. Zuletzt wird x aus einer Verteilung $p_{\text{model}}(x; g(z)) = p_{\text{model}}(x | z)$ gezogen. Während des Trainings wird das approximative Inferenznetz (oder der Encoder) $q(z | x)$ allerdings zum Bestimmen von z genutzt; in diesem Fall wird $p_{\text{model}}(x | z)$ als Decoder-Netz betrachtet.

Die wesentliche Erkenntnis beim VAE ist, dass diese Encoder durch Maximieren der ELBO $\mathcal{L}(q)$ für den Datenpunkt x trainiert werden können:

$$\mathcal{L}(q) = \mathbb{E}_{z \sim q(z|x)} \log p_{\text{model}}(z, x) + \mathcal{H}(q(z | x)) \quad (20.76)$$

$$= \mathbb{E}_{z \sim q(z|x)} \log p_{\text{model}}(x | z) - D_{\text{KL}}(q(z | x) || p_{\text{model}}(z)) \quad (20.77)$$

$$\leq \log p_{\text{model}}(x). \quad (20.78)$$

In Gleichung 20.76 sehen wir, dass der erste Term die gemeinsame Log-Likelihood der sichtbaren und verdeckten Variablen unter der approximativen A-posteriori-Verteilung über die latenten Variablen darstellt (wie

bei der EM, allerdings wird eine approximative anstelle der exakten Verteilung genutzt). Der zweite Term ist die Entropie der approximativen A-posteriori-Verteilung. Wenn q als Normalverteilung gewählt wird und Rauschen zu einem vorhergesagten Mittelwert addiert wird, animiert das Maximieren dieses Entropieterms zu einer Zunahme der Standardabweichung dieses Rauschens. Allgemein betrachtet animiert dieser Entropieterm die Variations-A-posteriori-Verteilung dazu, vielen z -Werten, die x hätten erzeugen können, eine hohe Wahrscheinlichkeit(smasse) aufzuerlegen, anstatt in einem einzelnen Punktschätzwert des wahrscheinlichsten Wertes zusammenzufallen. In Gleichung 20.77 haben wir den ersten Term als Log-Likelihood der Rekonstruktion aus anderen Autoencodern wiedererkannt. Der zweite Term versucht, die approximative A-posteriori-Verteilung $q(z | x)$ und die A-priori-Verteilung des Modells, $p_{\text{model}}(z)$, einander anzunähern.

Klassische Ansätze zur Variational Inference und zum Variational Learning erschließen q mittels Optimierungsalgorithmus, meist mit Iteration von Fixpunktgleichungen (Abschnitt 19.4). Diese Ansätze sind langsam und setzen häufig voraus, dass $\mathbb{E}_{z \sim q} \log p_{\text{model}}(z, x)$ in geschlossener Form berechnet werden kann. Die Grundidee hinter dem VAE ist es, einen parametrischen Encoder (auch als Inferenznetz oder Erkennungsmodell bezeichnet) zu trainieren, der die Parameter von q erzeugt. Sofern z eine stetige Variable ist, können wir anschließend die Backpropagation mit Stichproben aus z , die aus $q(z | x) = q(z; f(x; \theta))$ gezogen wurden, vornehmen, um einen Gradienten bezüglich θ zu ermitteln. Das Lernen besteht dann einzig aus dem Maximieren von \mathcal{L} bezüglich der Parameter von Encoder und Decoder. Alle Erwartungswerte in \mathcal{L} lassen sich mittels Monte-Carlo-Stichprobenverfahren approximieren.

Der Ansatz des VAEs ist elegant, theoretisch zufriedenstellend und einfach zu implementieren. Er erzielt auch exzellente Ergebnisse und gehört zu den modernsten Ansätzen für die generative Modellierung. Sein wesentlicher Nachteil ist, dass Stichproben von mit Bildern trainierten VAEs häufig etwas unscharf sind. Die Ursachen dafür sind noch nicht bekannt. Eine Möglichkeit wäre, dass diese Unschärfe ein intrinsischer Effekt der Maximum Likelihood ist, die $D_{\text{KL}}(p_{\text{data}} \| p_{\text{model}})$ minimiert. Wie Abbildung 3.6 zeigt, weist das Modell dadurch Punkten, die in der Trainingsdatenmenge vorliegen, aber möglicherweise auch weiteren Punkten, eine hohe Wahrscheinlichkeit zu. Diese weiteren Punkte könnten unscharfe Bilder enthalten. Mit ein Grund für das Auferlegen einer Wahrscheinlichkeit(smasse) auf unscharfe Bilder (und nicht auf einen anderen Teil des Raums) durch das Modell ist, dass die in der Praxis eingesetzten VAEs meist eine Normalverteilung für $p_{\text{model}}(x; g(z))$ nutzen. Das Maximieren einer unteren Schranke für die Likelihood einer

solchen Verteilung ähnelt dem Trainieren eines klassischen Autoencoders mittels mittleren quadratischen Fehlers in dem Sinne, dass die Tendenz besteht, Merkmale der Eingabe zu ignorieren, die wenige Pixel belegen oder nur zu einer geringfügigen Veränderung der Helligkeit der von ihnen belegten Pixel führen. Das Problem betrifft nicht nur VAEs, sondern auch generative Modelle, die eine Log-Likelihood optimieren oder auch $D_{\text{KL}}(p_{\text{data}} \parallel p_{\text{model}})$, wie von *Theis et al.* (2015) und *Huszar* (2015) erörtert. Ein weiteres störendes Problem aktueller VAE-Modelle ist ihre Neigung dazu, nur eine kleine Teilmenge der Dimensionen von \mathbf{z} zu nutzen, als wäre der Encoder nicht in der Lage, genug der lokalen Richtungen im Eingaberaum in einen Raum zu transformieren, in dem die Randverteilung der faktorisierten A-priori-Verteilung entspricht.

Das VAE-Framework lässt sich problemlos auf eine Vielzahl von Modellarchitekturen erweitern. Dies ist ein wesentlicher Vorteil gegenüber Boltzmann-Maschinen, die nur mit äußerst sorgfältigen Modellentwürfen effizient berechenbar bleiben. VAEs funktionieren sehr gut mit diversen differenzierbaren Operatoren. Ein besonders ausgereifter VAE ist das **DRAW-Modell** (Deep Recurrent Attention Writer) (*Gregor et al.*, 2015). DRAW nutzt einen rekurrenten Encoder und rekurrenten Decoder in Kombination mit einem Aufmerksamkeitsmechanismus (engl. *attention mechanism*). Der Entwicklungsprozess für das DRAW-Modell besteht aus sequenziellem Aufsuchen unterschiedlicher kleiner Bildbereiche und Zeichnen der Werte der Pixel an diesen Punkten. VAEs können auch so erweitert werden, dass Sequenzen durch Definieren von Variational RNNs (*Chung et al.*, 2015b) mittels rekurrentem Encoder und Decoder innerhalb des VAE-Frameworks erzeugt werden. Für die Generierung einer Stichprobe aus einem klassischen RNN sind nur nichtdeterministische Operationen im Ausgaberaum nötig. Variational RNNs weisen eine Zufallsvariabilität auf der potenziell abstrakteren Ebene auf, die von den latenten Variablen des VAEs erfasst wird.

Das VAE-Framework wurde so erweitert, dass nicht nur die übliche ELBO maximiert wird, sondern auch die Zielfunktion des **Autoencoders mit Gewichten nach Wichtigkeit** (*Burda et al.*, 2015):

$$\mathcal{L}_k(\mathbf{x}, q) = \mathbb{E}_{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(k)} \sim q(\mathbf{z} | \mathbf{x})} \left[\log \frac{1}{k} \sum_{i=1}^k \frac{p_{\text{model}}(\mathbf{x}, \mathbf{z}^{(i)})}{q(\mathbf{z}^{(i)} | \mathbf{x})} \right]. \quad (20.79)$$

Diese neue Zielfunktion ist für $k = 1$ äquivalent zur traditionellen unteren Schranke \mathcal{L} . Allerdings ist auch die Interpretation einer Schätzwertbildung des wahren $\log p_{\text{model}}(\mathbf{x})$ mittels Importance Samplings von \mathbf{z} aus der Proposal-Verteilung $q(\mathbf{z} | \mathbf{x})$ möglich. Die Zielfunktion des Autoencoders

mit Gewichten nach Wichtigkeit ist außerdem eine untere Schranke für $\log p_{\text{model}}(\mathbf{x})$ und wird mit zunehmendem k enger.

VAEs weisen einige interessante Verbindungen zu MP-DBMs und weiteren Ansätzen mit Backpropagation durch den approximativen Inferenzgraphen auf (*Goodfellow et al.*, 2013b; *Stoyanov et al.*, 2011; *Brakel et al.*, 2013). Diese bisherigen Ansätze erfordern ein Inferenzverfahren wie die Molekularfeld-Fixpunktgleichungen zum Bereitstellen des Berechnungsgraphen. Der VAE ist für beliebige Berechnungsgraphen definiert, sodass er auf eine größere Bandbreite von probabilistischen Modellfamilien angewendet werden kann, da die Wahl der Modelle nicht auf solche mit effizient lösbarer Molekularfeld-Fixpunktgleichungen eingeschränkt wird. Der VAE hat auch den Vorteil, dass eine Schranke für die Log-Likelihood des Modells erhöht wird, während die Kriterien für die MP-DBM und ähnliche Modelle stärker heuristisch geprägt sind und kaum probabilistische Interpretationen über die Genauigkeit der Ergebnisse der approximativen Inferenz hinaus aufweisen. Ein Nachteil des VAEs ist, dass er ein Inferenznetz für lediglich ein Problem erlernt, also aus einem \mathbf{z} auf ein \mathbf{x} schließt. Die bisherigen Verfahren können die approximative Inferenz für eine beliebige Teilmenge von Variablen anhand einer beliebigen anderen Teilmenge von Variablen vornehmen, da die Molekularfeld-Fixpunktgleichungen vorgeben, wie das Parameter Sharing zwischen den Berechnungsgraphen all dieser unterschiedlichen Probleme erfolgt.

Eine sehr vorteilhafte Eigenschaft des VAEs ist, dass ein zeitgleiches Training eines parametrischen Encoders in Verbindung mit dem Generator-Netz das Modell dazu zwingt, ein vorhersagbares Koordinatensystem zu erlernen, das vom Encoder erfasst werden kann. Das stellt einen hervorragenden Manifold-Learning-Algorithmus dar. Abbildung 20.6 zeigt Beispiele für geringdimensionale Mannigfaltigkeiten, die vom VAE erlernt wurden. In einem der dort dargestellten Fälle hat der Algorithmus zwei unabhängige Faktoren der Variation in Bildern von Gesichtern entdeckt: Rotationswinkel und Gefühlsausdruck.

20.10.4 Generative-Adversarial-Netze (Generative Adversarial Networks)

Generative-Adversarial-Netze oder kurz GANs (*Goodfellow et al.*, 2014c) sind ein weiterer Ansatz zur generativen Modellierung auf Basis differenzierbarer Generator-Netze.

Generative-Adversarial-Netze basieren auf einem spieltheoretischen Szenario, in dem das Generator-Netz gegen einen Gegner antreten muss. Das

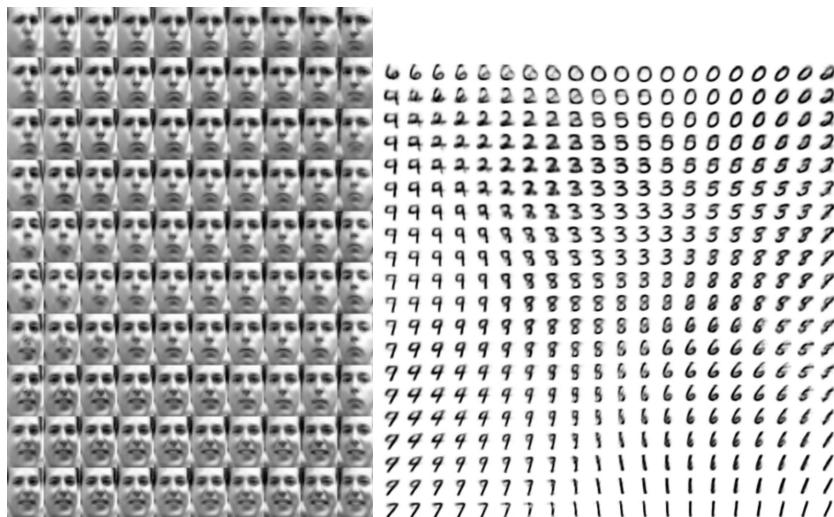


Abbildung 20.6: Beispiele für 2-D-Koordinatensysteme für hochdimensionale Mannigfaltigkeiten, die von einem VAE erlernt wurden (*Kingma und Welling, 2014a*). Zwei Dimensionen können auf einem einfachen Blatt Papier dargestellt werden, sodass Sie ein Verständnis für die Funktionsweise des Modells erhalten, indem Sie ein Modell mit einem latenten 2-D-Code trainieren, obwohl wir der Überzeugung sind, dass die intrinsische Dimensionalität der Datenmannigfaltigkeit viel höher ist. Die Bilder sind keine Beispiele aus der Trainingsdatenmenge, sondern Bilder \mathbf{x} , die tatsächlich vom Modell $p(\mathbf{x} | \mathbf{z})$ ganz einfach durch Änderungen am 2-D-»Code« \mathbf{z} erzeugt wurden (jedes Bild steht für eine andere Auswahl des »Codes« \mathbf{z} auf einem gleichmäßigen 2-D-Raster). (*Links*) Die 2-D-Karte der Gesichter aus der Frey-Mannigfaltigkeit. Eine der entdeckten Dimensionen (horizontal) entspricht in erster Linie einer Drehung des Gesichts, die andere (vertikal) dagegen dem Gefühlsausdruck. (*Rechts*) Die 2-D-Karte der MNIST-Mannigfaltigkeit.

Generator-Netz erzeugt direkt die Stichproben $\mathbf{x} = g(\mathbf{z}; \boldsymbol{\theta}^{(g)})$. Sein Gegner, das **Diskriminator-Netz**, versucht, zwischen den aus den Trainingsdaten gezogenen Stichproben und den aus dem Generator-Netz gezogenen Stichproben zu unterscheiden. Der Diskriminator gibt einen Wahrscheinlichkeitswert $d(\mathbf{x}; \boldsymbol{\theta}^{(d)})$ aus, der angibt, mit welcher Wahrscheinlichkeit \mathbf{x} tatsächlich ein Element aus den vorhandenen Trainingsdaten und keine »gefälschte«, aus dem Modell gezogene, Stichprobe ist.

Die einfachste Methode zum Formulieren des Lernprozesses in Generative-Adversarial-Netzen ist ein Nullsummenspiel, in dem eine Funktion $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ den Payoff (Gewinn oder Nutzen) für den Diskriminator bestimmt. Der Generator erhält $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ als Payoff. Während des Lernens

versuchen beide Spieler, ihren Payoff zu maximieren; bei Konvergenz gilt also

$$g^* = \arg \min_g \max_d v(g, d). \quad (20.80)$$

Die Standardwahl für v ist

$$v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)}) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \log d(\mathbf{x}) + \mathbb{E}_{\mathbf{x} \sim p_{\text{model}}} \log (1 - d(\mathbf{x})). \quad (20.81)$$

Das führt dazu, dass der Diskriminatator versucht, die korrekte Klassifizierung von Stichproben als echt oder falsch zu erlernen. Gleichzeitig versucht der Generator, den Klassifikator hinter Licht zu führen und ihm vorzugaukeln, dass die von ihm erzeugten Stichproben echt sind. Bei Konvergenz sind die Stichproben des Generators von echten Daten nicht mehr unterscheidbar und der Diskriminatator gibt überall $\frac{1}{2}$ aus. An diesem Punkt wird der Diskriminatator nicht mehr benötigt.

Der wesentliche Beweggrund für die Entwicklung von GANs ist, dass der Lernprozess weder approximative Inferenz noch die Approximation eines Gradienten einer Partitionsfunktion erfordert. Wenn $\max_d v(g, d)$ in $\boldsymbol{\theta}^{(g)}$ konvex ist (z. B. wenn die Optimierung direkt im Rahmen der Wahrscheinlichkeitsdichtefunktionen erfolgt), konvergiert das Verfahren garantiert und ist asymptotisch konsistent.

Leider kann das Lernen in GANs praktische Schwierigkeiten mit sich bringen, wenn g und d durch neuronale Netze repräsentiert werden und $\max_d v(g, d)$ nicht konvex ist. Goodfellow (2014) hat die Nichtkonvergenz als Problem erkannt, das bei GANs zu einer Unteranpassung führen kann. Allgemein ist nicht garantiert, dass der zeitgleiche Gradientenabstieg der Kosten beider Spieler in einem Gleichgewicht mündet. Betrachten Sie zum Beispiel die Wertfunktion $v(a, b) = ab$, in der ein Spieler a steuert und die Kosten ab verursacht, während der andere Spieler b steuert und die Kosten $-ab$ betragen. Wenn wir beide Spieler so modellieren, dass sie infinitesimal kleine Gradientenschritte machen, reduziert jeder Spieler die eigenen Kosten zulasten des anderen Spielers und a und b gehen in einen stabilen kreisförmigen »Orbit« über, statt irgendwann den Gleichgewichtspunkt im Ursprung zu erreichen. Beachten Sie, dass die Gleichgewichte eines Minimax-Spiels nicht die lokalen Minima von v sind. Stattdessen handelt es sich um Punkte, die gleichzeitig Minima der Kosten beider Spieler sind. Das bedeutet, dass es sich um Sattelpunkte von v handelt, die lokale Minima bezüglich der Parameter des ersten Spielers und lokale Maxima bezüglich der Parameter des zweiten Spielers sind. Es ist möglich, dass die beiden Spieler durch ihre abwechselnden Züge v in alle Ewigkeit steigern und anschließend reduzieren, ohne jemals exakt den Sattelpunkt zu erreichen, in dem keiner der Spieler

seine Kosten mehr reduzieren kann. Es ist nicht bekannt, inwiefern GANs diesem Problem der Nichtkonvergenz unterliegen.

Goodfellow (2014) hat einen alternativen Ansatz der Payoffs entdeckt, bei dem das Spiel kein Nullsummenspiel mehr ist und denselben erwarteten Gradienten wie beim Maximum Likelihood Learning aufweist, sofern der Diskriminator optimal arbeitet. Da das Maximum-Likelihood-Training konvergiert, sollte auch dieser neue Ansatz des GAN-Spiels konvergieren, sofern hinreichend viele Stichproben vorliegen. Leider scheint dieser alternative Ansatz die Konvergenz in der Praxis nicht zu verbessern – möglicherweise aufgrund der Suboptimalität des Diskriminators oder einer hohen Varianz im Bereich des erwarteten Gradienten.

In realistischen Experimenten schneidet ein anderer Ansatz des GAN-Spiels am besten ab, der weder ein Nullsummenspiel noch um ein Äquivalent zur Maximum Likelihood ist und von *Goodfellow et al.* (2014c) aus einer heuristischen Motivation heraus vorgestellt wurde. In diesem leistungsstarken Ansatz versucht der Generator, die Log-Wahrscheinlichkeit, mit der der Diskriminator einen Fehler begeht, zu erhöhen, statt die Log-Wahrscheinlichkeit, mit der der Diskriminator die korrekte Vorhersage macht, zu verringern. Dieser neue Ansatz beruht rein auf der Beobachtung, dass dadurch die Ableitung der Kostenfunktion des Generators bezüglich der Logits des Diskriminators auch dann groß bleibt, wenn der Diskriminator souverän alle Stichproben des Generators zurückweist.

Die Stabilisierung des Lernens bei GANs stellt ein ungelöstes Problem dar. Zum Glück funktioniert das Lernen bei GANs gut, wenn die Modellarchitektur und die Hyperparameter sorgfältig ausgewählt wurden. *Radford et al.* (2015) haben ein tiefes gefaltetes GAN (engl. *deep convolutional GAN*, DCGAN) konstruiert, das bei der Bildsynthese sehr gut abschneidet, und gezeigt, dass dessen latenter Darstellungsraum wichtige Faktoren der Variation erfasst (vgl. Abbildung 15.9). Abbildung 20.7 zeigt Beispiele von Bildern, die mit einem DCGAN-Generator erzeugt wurden.

Das Problem des Lernens bei GANs lässt sich vereinfachen, indem der Erzeugungsprozess in viele Detailstufen aufgeteilt wird. Es ist möglich, bedingte GANs zu trainieren (*Mirza und Osindero*, 2014), die lernen, aus einer Verteilung $p(\mathbf{x} | \mathbf{y})$ Stichproben zu ziehen, anstatt lediglich Stichproben aus einer Randverteilung $p(\mathbf{x})$ zu ziehen. *Denton et al.* (2015) haben gezeigt, dass eine Reihe bedingter GANs so trainiert werden kann, das zunächst eine sehr niedrig auflösende Version eines Bildes erzeugt wird, bevor nach und nach Details zum Bild hinzugefügt werden. Diese Technik wird als Laplacian-Generative-Adversarial-Network-Modell (LAPGAN) bezeichnet, da

eine Laplace-Pyramide eingesetzt wird, um die Bilder mit unterschiedlichen Detailstufen zu erzeugen. LAPGAN-Generatoren können nicht nur Diskriminator-Netze in die Irre führen, sondern sogar menschliche Beobachter: Versuchspersonen haben bis zu 40 Prozent der Ausgaben des Netzes als echt eingestuft. Abbildung 20.7 zeigt Beispielbilder, die mit einem LAPGAN-Generator erzeugt wurden.



Abbildung 20.7: Von GANs erzeugte Bilder, die mit dem LSUN-Datensatz trainiert wurden. (*Links*) Schlafzimmer, die von einem DCGAN-Modell erzeugt wurden (Wiedergabe mit freundlicher Genehmigung von *Radford et al. (2015)*) (*Rechts*) Kirchen, die von einem LAPGAN-Modell erzeugt wurden (Wiedergabe mit freundlicher Genehmigung von *Denton et al. (2015)*)

Eine ungewöhnliche Fähigkeit des GAN-Trainingsverfahrens ist, dass damit Wahrscheinlichkeitsverteilungen angepasst werden können, die Trainingspunkten eine Wahrscheinlichkeit von Null zuweisen. Statt die Log-Wahrscheinlichkeit bestimmter Punkte zu maximieren, lernt das Generator-Netz, eine Mannigfaltigkeit zu finden, deren Punkte den Trainingspunkten irgendwie ähneln. Paradoxe Weise bedeutet dies, dass das Modell einer Testdatenmenge möglicherweise eine negativ unendliche Log-Likelihood zuweist, aber dennoch eine Mannigfaltigkeit darstellt, die ein menschlicher Beobachter für gut genug hält, um die Kernaussage der Aufgabe zu erfassen. Das ist weder eindeutig ein Vor- noch ein Nachteil. Es lässt sich auch sicherstellen, dass das Generator-Netz allen Punkten von Null verschiedene Wahrscheinlichkeiten zuweist, indem die letzte Schicht des Generator-Netzes allen erzeugten Werten normalverteiltes Rauschen hinzufügt. Generator-Netze, die auf diese Weise normalverteiltes Rauschen einbringen, ziehen Stichproben aus derselben Verteilung, die sich ergibt, wenn das Generator-Netz zur Parametrisierung des Mittelwerts einer bedingten Normalverteilung genutzt wird.

Dropout scheint im Diskriminator-Netz von Bedeutung zu sein. Vor allem sollten Einheiten stochastisch entfernt werden, während der Gradient berechnet wird, dem das Generator-Netz folgen soll. Das Auffinden des Gradienten der deterministischen Version des Diskriminators mit durch zwei geteilten Gewichtungen scheint nicht so wirkungsvoll zu sein. Ebenso führt der völlige Verzicht auf Dropout zu schlechten Ergebnissen.

Das GAN-Framework wurde zwar für differenzierbare Generator-Netze designt, aber ähnliche Prinzipien lassen sich auch zum Trainieren anderer Modelle nutzen. Zum Beispiel kann **Self-Supervised Boosting** zum Trainieren eines RBM-Generators eingesetzt werden, um einen Diskriminator mit logistischer Regression zu täuschen (*Welling et al.*, 2002).

20.10.5 Generative-Moment-Matching-Netze (Generative Moment Matching Networks)

Generative-Moment-Matching-Netze (*Li et al.*, 2015; *Dziugaite et al.*, 2015) sind eine weitere Form generativer Modelle auf Basis differenzierbarer Generator-Netze. Anders als VAEs und GANs müssen sie das Generator-Netz nicht mit einem anderen Netz koppeln – weder mit einem Inferenznetz wie bei VAEs noch mit einem Diskriminator-Netz wie bei GANs.

Generative-Moment-Matching-Netze werden mit einer **Moment Matching** genannten Methode trainiert. Dabei wird der Generator so trainiert, dass viele der statistischen Größen der vom Modell generierten Stichproben den statistischen Größen der Beispiele in der Trainingsdatenmenge möglichst ähnlich sind. In diesem Zusammenhang bezeichnet **Moment** einen Erwartungswert für unterschiedliche Potenzen einer Zufallsvariable. So ist der erste Moment der Mittelwert, der zweite Moment der Mittelwert der quadrierten Werte usw. In mehreren Dimensionen kann jedes Element des Zufallsvektors unterschiedlich potenziert werden, sodass ein Moment eine beliebige Größe der Form

$$\mathbb{E}_{\mathbf{x}} \prod_i x_i^{n_i} \quad (20.82)$$

darstellen kann, wobei $\mathbf{n} = [n_1, n_2, \dots, n_d]^\top$ ein Vektor nicht-negativer ganzer Zahlen ist.

Auf den ersten Blick erscheint dieser Ansatz rechnerisch undurchführbar. Wenn wir beispielsweise ein Moment Matching für alle Momente der Form $x_i x_j$ durchführen möchten, müssen wir die Differenz zwischen einer Anzahl von Werten minimieren, die quadratisch von der Dimension von \mathbf{x} abhängt. Außerdem würde selbst ein Matching aller ersten und zweiten Momente

lediglich dazu reichen, eine multivariate Normalverteilung anzupassen, die nur lineare Beziehungen zwischen Werten erfasst. Unser Ziel ist es, mit neuronalen Netzen komplexe nichtlineare Beziehungen zu erfassen – und dafür werden viel mehr Momente benötigt. GANs umgehen dieses Problem des vollständigen Spezifizierens aller Momente mittels eines dynamisch aktualisierten Diskriminators, der seine Aufmerksamkeit automatisch auf die statistische Größe richtet, für die das Matching vom Generator-Netz am wenigsten effektiv ist.

Stattdessen können Generative-Moment-Matching-Netze durch Minimieren einer Kostenfunktion namens **maximale Mittelwertabweichung** (engl. *maximum mean discrepancy*, MMD) trainiert werden (*Schölkopf und Smola*, 2002; *Gretton et al.*, 2012). Diese Kostenfunktion misst den Fehler in den ersten Momenten eines unendlich-dimensionalen Raums mithilfe einer impliziten Zuordnung zum Merkmalsraum, der durch eine Kernel-Funktion definiert ist, um die Berechnungen auf unendlich-dimensionalen Vektoren effizient durchführbar zu machen. Die Kosten der MMD sind genau dann Null, wenn die beiden miteinander verglichenen Verteilungen gleich sind.

Optisch sind die Stichproben der Generative-Moment-Matching-Netze etwas enttäuschend. Zum Glück können sie durch eine Kombination des Generator-Netzes mit einem Autoencoder verbessert werden. Zunächst wird ein Autoencoder darauf trainiert, die Trainingsdatenmenge wiederherzustellen. Dann transformiert der Encoder des Autoencoders die gesamte Trainingsdatenmenge in einen Code-Raum. Das Generator-Netz wird anschließend so trainiert, dass es Code-Beispiele erzeugt, die mittels Decoder optisch zufriedenstellenden Stichproben zugeordnet werden können.

Im Gegensatz zu GANs ist die Kostenfunktion nur bezüglich eines Batches mit Beispielen sowohl aus der Trainingsdatenmenge als auch dem Generator-Netz definiert. Es ist nicht möglich, ein Trainingsupdate als Funktion nur eines Trainingsbeispiels oder nur einer Stichprobe aus dem Generator-Netz vorzunehmen. Das liegt daran, dass die Momente als empirischer Durchschnittswert vieler Stichproben berechnet werden müssen. Wenn die Batch-Größe zu klein ist, kann die MMD das tatsächliche Ausmaß der Variation in den Verteilungen, aus denen Stichproben gezogen werden, unterschätzen. Keine endliche Batch-Größe ist hinreichend groß, um dieses Problem gänzlich zu eliminieren, aber größere Batches reduzieren das Maß der Unterschätzung. Wenn die Batch-Größe zu groß ist, wird das Trainingsverfahren undurchführbar langsam, da viele Beispiele verarbeitet werden müssen, um einen einzelnen kleinen Gradientenschritt zu berechnen.

Wie bei GANs ist es möglich, ein Generator-Netz auch dann mittels MMD zu trainieren, wenn das Generator-Netz den Trainingspunkten die Wahrscheinlichkeit Null zuweist.

20.10.6 Gefaltete generative Netze (Convolutional Generative Networks)

Beim Erzeugen von Bildern ist es oft hilfreich, ein Generator-Netz einzusetzen, das eine gefaltete Struktur aufweist (vgl. z. B. *Goodfellow et al.*, 2014c; *Dosovitskiy et al.*, 2015). Dazu verwenden wir die »Transponierte« des Faltungsoperators (siehe Abschnitt 9.5). Dieser Ansatz führt oft zu realistischeren Bildern und benötigt dabei weniger Parameter als die Verwendung vollständig verbundener Schichten (engl. *fully connected layers*) ohne Parameter Sharing.

CNNs für Erkennungsaufgaben weisen einen Informationsfluss vom Bild zu einer Zusammenfassungsschicht oben im Netz auf, häufig zu einem Klassen-Label. Während das Bild nach oben durch das Netz »fließt«, werden mit zunehmender Invarianz des Bildes gegenüber störenden Transformationen Informationen verworfen. In einem Generator-Netz gilt das Gegenteil. Umfangreiche Details müssen hinzugefügt werden, während die Repräsentation des zu erzeugenden Bildes durch das Netz propagiert; das führt schließlich zu der endgültigen Repräsentation des Bildes, nämlich dem Bild selbst mit all seinen feinen Details, Objektpositionen, Haltungen, Texturen und Lichtstimmungen. Der primäre Mechanismus zum Verwerfen von Informationen in einem gefalteten Erkennungsnetz ist die Pooling-Schicht. Das Generator-Netz scheint darauf angewiesen zu sein, Informationen hinzuzufügen. Es ist nicht möglich, die Inverse einer Pooling-Schicht in das Generator-Netz einzubauen, da die meisten Pooling-Funktionen unumkehrbar sind. Eine einfachere Operation ist das simple Erhöhen der räumlichen Dimension der Repräsentation. Ein Ansatz, der gut zu funktionieren scheint, ist das von *Dosovitskiy et al.* (2015) vorgestellte »Un-Pooling«. Diese Schicht entspricht der Inversen der Max-Pooling-Operation unter bestimmten vereinfachenden Bedingungen. Erstens ist die Bedingung an die Schrittweite der Max-Pooling-Operation, dass sie gleich der Breite des Pooling-Bereichs ist. Zweitens wird vorausgesetzt, dass die maximale Eingabe in jedem Pooling-Bereich die Eingabe in der oberen linken Ecke ist. Schließlich wird vorausgesetzt, dass alle nicht-maximalen Eingaben in jedem Pooling-Bereich Null sind. Das sind zwar sehr starke und unrealistische Voraussetzungen, aber sie ermöglichen die Umkehr des Max-Pooling-Operators. Die inverse Un-Pooling-Operation weist einen Tensor aus Nullen zu und kopiert dann jeden Wert

aus der räumlichen Koordinate i der Eingabe in die räumliche Koordinate $i \times k$ der Ausgabe. Der ganzzahlige Wert k definiert die Größe des Pooling-Bereichs. Obwohl die Voraussetzungen für die Definition des Un-Pooling-Operators unrealistisch sind, können die folgenden Schichten lernen, seine ungewöhnliche Ausgabe zu kompensieren, sodass die vom Modell generierten Stichproben insgesamt optisch ansprechend sind.

20.10.7 Autoregressive Netze (Auto-Regressive Networks)

Autoregressive Netze sind gerichtete probabilistische Modelle ohne latente Zufallsvariablen. Die bedingten Wahrscheinlichkeitsverteilungen in diesen Modellen werden durch neuronale Netze repräsentiert (manchmal extrem simple neuronale Netze wie die logistische Regression). Die Graphenstruktur dieser Modelle ist der vollständige Graph. Sie zerlegen eine multivariate Verteilung über die beobachteten Variablen anhand der Produktregel für Wahrscheinlichkeiten, um ein Produkt der bedingten Verteilungen der Form $P(x_d | x_{d-1}, \dots, x_1)$ zu erhalten. Derartige Modelle wurden **vollständig sichtbare Bayes-Netze** (engl. *fully-visible Bayes networks*, FVBMs) genannt und in unterschiedlicher Form erfolgreich eingesetzt, zuerst mit logistischer Regression für jede bedingte Verteilung (Frey, 1998), später dann mit neuronalen Netzen mit verdeckten Einheiten (Bengio und Bengio, 2000b; Larochelle und Murray, 2011). Bei einigen Formen von autoregressiven Netzen wie dem NADE (Larochelle und Murray, 2011), das in Abschnitt 20.10.10 beschrieben wird, können wir eine Art Parameter Sharing einbringen, das sowohl einen statistischen Vorteil (weniger einmalige Parameter) als auch einen rechnerischen Vorteil (weniger Berechnungen) bietet. Das ist ein weiterer Fall für das wiederkehrende Deep-Learning-Motiv der *Wiederwendung von Merkmalen*.

20.10.8 Lineare autoregressive Netze (Linear Auto-Regressive Networks)

Die einfachste Form eines autoregressiven Netzes weist keine verdeckten Einheiten und kein Sharing von Parametern oder Merkmalen auf. Jedes $P(x_i | x_{i-1}, \dots, x_1)$ wird als lineares Modell parametrisiert (lineare Regression für reellwertige Daten, logistische Regression für binäre Daten, softmax-Regression für diskrete Daten). Dieses Modell wurde von Frey (1998) vorgestellt und weist $O(d^2)$ Parameter bei d zu modellierenden Variablen auf. Es ist in Abbildung 20.8 dargestellt.

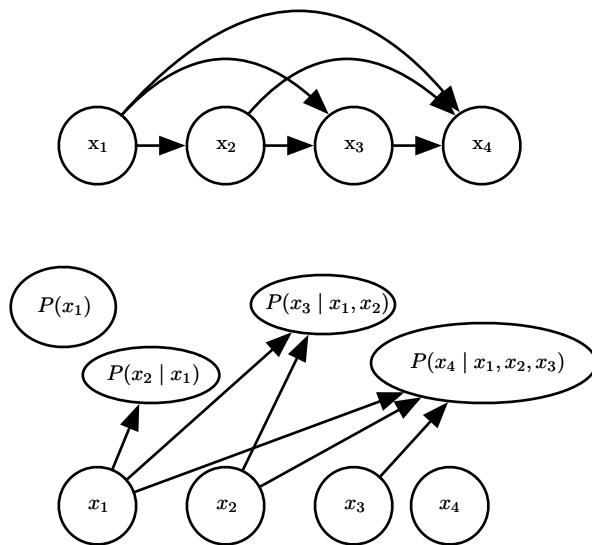


Abbildung 20.8: Ein vollständig sichtbares Belief-Netz sagt die i -te Variable anhand der $i - 1$ vorangegangenen Variablen voraus. (Oben) Das gerichtete graphische Modell eines FVBNs. (Unten) Der Berechnungsgraph für das logistische FVBN, in dem jede Vorhersage durch einen linearen Prädiktor erfolgt.

Wenn die Variablen stetig sind, ist ein lineares autoregressives Modell (engl. *linear auto-regressive model*) lediglich eine andere Möglichkeit zur Bildung einer multivariaten Normalverteilung für die Erfassung linearer paarweiser Interaktionen zwischen den beobachteten Variablen.

Lineare autoregressive Netze sind im Wesentlichen die Generalisierung der linearen Klassifizierungsverfahren für die generative Modellierung. Sie haben daher dieselben Vor- und Nachteile wie lineare Klassifikatoren. Wie diese können sie mithilfe konvexer Verlustfunktionen trainiert werden und lassen manchmal geschlossene Lösungen (engl. *closed form solutions*) zu (wie im Fall einer Normalverteilung). Wie lineare Klassifikatoren bietet das Modell selbst keinerlei Möglichkeit zum Erhöhen seiner Kapazität, sodass diese mithilfe von Verfahren wie Basisexpansionen der Eingabe oder Kernel-Trick erhöht werden muss.

20.10.9 Neuronale autoregressive Netze (Neural Auto-Regressive Networks)

Neuronale autoregressive Netze (Bengio und Bengio, 2000a,b) weisen das-selbe von links nach rechts verlaufende graphische Modell wie logistische autoregressive Netze auf (Abbildung 20.8), nutzen aber eine andere Para-

metrisierung der bedingten Verteilungen innerhalb der Struktur des grafischen Modells. Die neue Parametrisierung ist insofern mächtiger, als ihre Kapazität beliebig erhöht werden kann, was die Approximation jeder beliebigen multivariaten Verteilung ermöglicht. Die neue Parametrisierung kann außerdem die Generalisierung durch Einführen von Parameter Sharing und Feature Sharing (Teilen von Merkmalen) verbessern, wie es im Deep Learning allgemein bekannt ist. Die Modelle entstanden, um dem Fluch der Dimensionalität zu entgehen, der bei klassischen tabellarischen grafischen Modellen auftritt; sie teilen die Struktur aus Abbildung 20.8. In tabellarischen diskreten probabilistischen Modellen wird jede bedingte Verteilung als Tabelle mit Wahrscheinlichkeiten dargestellt; für jede mögliche Konfiguration der beteiligten Variablen gibt es je einen Eintrag und einen Parameter. Verwendet man stattdessen ein neuronales Netz, ergeben sich hier zwei Vorteile:

1. Die Parametrisierung jedes $P(x_i | x_{i-1}, \dots, x_1)$ durch ein neuronales Netz mit $(i-1) \times k$ Eingaben und k Ausgaben (sofern die Variablen diskret sind und k One-hot-codierte Werte annehmen) ermöglicht die Schätzung der bedingten Wahrscheinlichkeit ohne eine exponentielle Anzahl von Parametern (und Beispielen); dennoch können damit Abhängigkeiten hoher Ordnung zwischen den Zufallsvariablen erfasst werden.
2. Statt mehrere neuronale Netze für die Vorhersage jedes x_i zu nutzen, ermöglicht die Konnektivität von *links nach rechts* (vgl. Abbildung 20.9) das Vereinen sämtlicher neuronaler Netze zu einem Netz. Ebenso können die Merkmale der verdeckten Schichten, die für die Vorhersage von x_i berechnet wurden, für die Vorhersage von x_{i+k} ($k > 0$) wiederverwendet werden. Die verdeckten Einheiten sind somit in *Gruppen* organisiert, die die Besonderheit aufweisen, dass sämtliche Einheiten in der i -ten Gruppe ausschließlich von den Eingabewerten x_1, \dots, x_i abhängig sind. Die zur Berechnung dieser verdeckten Einheiten verwendeten Parameter werden übergreifend optimiert, um die Vorhersage aller Variablen in der Sequenz zu verbessern. Dies ist ein Beispiel für das *Wiederverwendungsprinzip*, das sich wie ein roter Faden durch Deep-Learning-Szenarios von rekurrenten und gefalteten Netzarchitekturen bis hin zum Multitask Learning und Transfer Learning zieht.

Jedes $P(x_i | x_{i-1}, \dots, x_1)$ kann eine bedingte Verteilung darstellen, indem Ausgaben des neuronalen Netzes *Parameter* der bedingten Verteilung

von x_i vorhersagen (vgl. Abschnitt 6.2.1.1). Obwohl die ursprünglichen neuronalen autoregressiven Netze im Kontext von rein diskreten multivariaten Daten evaluiert wurden (mit einer sigmoiden Ausgabe für eine Bernoulli-Variable oder einer softmax-Ausgabe für eine Multinoulli-Variable), ist es üblich, derartige Modelle auf stetige Variablen oder multivariate Verteilungen zu erweitern, die sowohl diskrete als auch stetige Variablen enthalten.

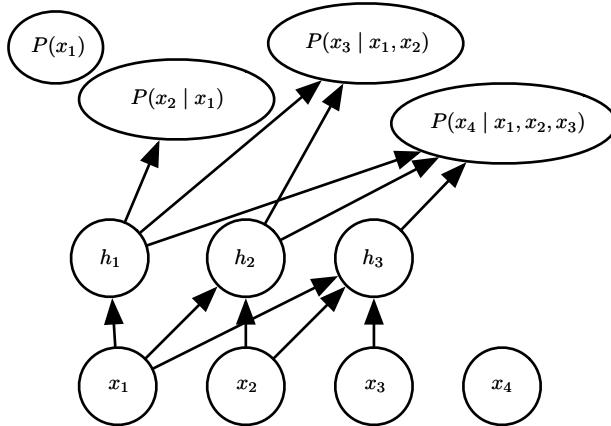


Abbildung 20.9: Ein neuronales autoregressives Netz sagt die i -te Variable x_i anhand der $i - 1$ vorangegangenen Variablen voraus, ist aber so parametrisiert, dass Merkmale (Gruppen verdeckter Einheiten, die mit h_i bezeichnet sind), die Funktionen von x_1, \dots, x_i sind, für die Vorhersage aller folgenden Variablen $x_{i+1}, x_{i+2}, \dots, x_d$ wiederverwendet werden können.

20.10.10 NADE

Der **Neural Auto-Regressive Density Estimator** (NADE, dt. *neuronaler autoregressiver Dichteschätzer*) ist ein sehr erfolgreiches neuronales autoregressives Netz aus der jüngeren Vergangenheit (*Larochelle und Murray, 2011*). Die Konnektivität entspricht dem ursprünglichen neuronalen autoregressiven Netz aus *Bengio und Bengio* (2000b), aber NADE führt ein zusätzliches Parameter Sharing ein, das in Abbildung 20.10 dargestellt ist. Die Parameter der verdeckten Einheiten einzelner Gruppen j werden gemeinsam genutzt.

Die Gewichte $W'_{j,k,i}$ zwischen der i -ten Eingabe x_i und dem k -ten Element der j -ten Gruppe der verdeckten Einheit $h_k^{(j)}$ ($j \geq i$) werden zwischen den Gruppen geteilt:

$$W'_{j,k,i} = W_{k,i}. \quad (20.83)$$

Die verbleibenden Gewichte mit $j < i$ sind Null.

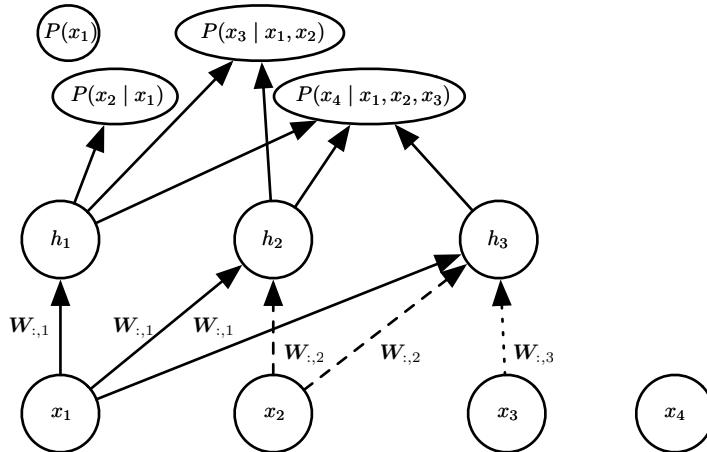


Abbildung 20.10: Eine Darstellung des Neural Auto-Regressive Density Estimators (NADE). Die verdeckten Einheiten sind so in Gruppen $\mathbf{h}^{(j)}$ unterteilt, dass für $j > i$ nur die Eingaben x_1, \dots, x_i an der Berechnung von $\mathbf{h}^{(i)}$ und der Vorhersage von $P(x_j | x_{j-1}, \dots, x_1)$ partizipieren. NADE unterscheidet sich von früheren neuronalen autoregressiven Netzen durch den Einsatz einer besonderen Form des Teilen von Gewichten (engl. *weight sharing*): $W'_{j,k,i} = W_{k,i}$ wird für alle Gewichte, die aus x_i stammen und in die k -te Einheit einer beliebigen Gruppe $j \geq i$ einfließen, geteilt (in der Abbildung durch denselben Linienstil für alle Instanzen eines replizierten Gewichts dargestellt). Wichtig: Der Vektor $(W_{1,i}, W_{2,i}, \dots, W_{n,i})$ wird geschrieben als $\mathbf{W}_{:,i}$.

Larochelle und Murray (2011) haben dieses Verfahren gewählt, damit die Forward-Propagation in einem NADE-Modell grob den Berechnungen ähnelt, die im Rahmen der Molekularfeld-Inferenz durchgeführt werden, um fehlende Eingaben in einer RBM zu ergänzen. Diese Molekularfeld-Inferenz entspricht der Arbeit eines RNNs mit gemeinsam genutzten Gewichten und der erste Schritt dieser Inferenz ist im NADE identisch. Der einzige Unterschied im NADE ist, dass die Ausgabegewichte, die die verdeckten Einheiten mit der Ausgabe verbinden, unabhängig von den Gewichten parametrisiert werden, die die Eingabeeinheiten mit den verdeckten Einheiten verbinden. In der RBM sind die Verdeckt-zu-Ausgabe-Gewichten die Transponierten der Eingabe-zu-Verdeckt-Gewichten. Die NADE-Architektur kann so erweitert werden, dass nicht nur ein Zeitschritt der rekurrenten Molekularfeld-Inferenz nachgebildet wird, sondern k Schritte. Dieser Ansatz nennt sich NADE- k (*Raiko et al., 2014*).

Wie bereits erwähnt, können autoregressive Netze für die Verarbeitung von stetigwertigen Daten erweitert werden. Eine besonders mächtige und generische Möglichkeit zur Parametrisierung einer stetigen Dichte ist eine gaußsche Mischverteilung (siehe Abschnitt 3.9.6) mit Gewichtungen der Mischverteilungen α_i (der Koeffizient oder die A-priori-Wahrscheinlichkeit der Komponente i), bedingtem Mittelwert μ_i für jede Komponente und bedingter Varianz σ_i^2 für jede Komponente. Ein Modell namens RNADE (*Uria et al.*, 2013) nutzt diese Parametrisierung, um NADE für reelle Werte zu erweitern. Wie bei anderen Mixture-Density-Netzen sind die Parameter dieser Verteilung Ausgaben des Netzes, wobei die Wahrscheinlichkeiten der Gewichtungen der Mischverteilungen von einer softmax-Einheit erzeugt werden und die Varianz so parametrisiert wird, dass sie positiv ist. Das stochastische Gradientenabstiegsverfahren kann aufgrund der Interaktionen zwischen den bedingten Mittelwerten μ_i und den bedingten Varianzen σ_i^2 rechnerische Probleme aufwerfen. Um diese Schwierigkeit zu reduzieren, nutzen *Uria et al.* (2013) in der Backpropagation-Phase einen Pseudogradienten, der den Gradienten des Mittelwerts ersetzt.

Eine weitere sehr interessante Erweiterung der neuronalen autoregressiven Architekturen eliminiert das Erfordernis, eine willkürliche Reihenfolge für die beobachteten Variablen zu bestimmen (*Murray und Larochelle*, 2014). In autoregressiven Netzen soll das Netz so trainiert werden, dass es mit beliebigen Anordnungen zurechtkommt, und zwar durch eine zufällige Reihenfolge der Stichproben und Bereitstellen der Informationen für verdeckte Einheiten, die angeben, welche der Eingaben beobachtet (auf der rechten Seite des Bedingungszeichens) und welche vorhergesagt werden sollen und somit als fehlend gelten (auf der linken Seite des Bedingungszeichens). Das ist positiv, da es uns erlaubt, einen trainierten autoregressiven Netz zu verwenden, um extrem effizient *jede beliebige Inferenzaufgabe durchzuführen* (d. h. Vorhersagen oder Stichprobenentnahme aus der Wahrscheinlichkeitsverteilung über eine beliebige Teilmenge von Variablen für eine beliebige Teilmenge). Schließlich können wir, da viele Anordnungen von Variablen möglich sind ($n!$ für n Variablen) und jede Anordnung o der Variablen zu einem anderen $p(\mathbf{x} \mid o)$ führt, ein Modellensemble für viele Werte von o bilden:

$$p_{\text{ensemble}}(\mathbf{x}) = \frac{1}{k} \sum_{i=1}^k p(\mathbf{x} \mid o^{(i)}). \quad (20.84)$$

Dieses Ensemblemodell generalisiert meist besser und weist der Testdatenset eine höhere Wahrscheinlichkeit zu als ein einzelnes Modell, das durch eine einzelne Anordnung definiert ist.

In derselben Veröffentlichung schlagen die Autoren tiefe Versionen der Architektur vor, aber leider macht das die Berechnung sofort wieder ebenso aufwendig wie im ursprünglichen neuronalen autoregressiven Netz (*Bengio und Bengio*, 2000b). Die erste Schicht und die Ausgabeschicht können noch mit $O(nh)$ Multiply-Add-Operationen berechnet werden – wie im normalen NADE, wo h die Anzahl der verdeckten Einheiten ist (die Größe der Gruppen h_i , in Abbildungen 20.10 und 20.9), wohingegen es in *Bengio und Bengio* (2000b) $O(n^2h)$ ist. Für die anderen verdeckten Schichten ist die Berechnung jedoch $O(n^2h^2)$, wenn jede »vorangegangene« Gruppe in der Schicht l an der Vorhersage der »folgenden« Gruppe in der Schicht $l+1$ beteiligt ist, vorausgesetzt, es gibt n Gruppen mit h verdeckten Einheiten in jeder Schicht. Das Erstellen der i -ten Gruppe in der Schicht $l+1$ ist nur von der i -ten Gruppe abhängig (wie in *Murray und Larochelle*, 2014); in der Schicht l erhalten wir $O(nh^2)$ – und das ist noch immer h -mal schlechter als beim normalen NADE.

20.11 Ziehen von Stichproben aus Autoencodern

In Kapitel 14 haben wir gezeigt, dass viele Arten von Autoencodern die Datenverteilung erlernen. Es gibt enge Verbindungen zwischen Score Matching, Denoising Autoencodern und Contractive Autoencodern. Diese Verbindungen zeigen, dass bestimmte Arten von Autoencodern die Datenverteilung auf eine bestimmte Weise erlernen. Wir haben noch nicht gezeigt, wie Stichproben aus solchen Modellen gezogen werden.

Einige Arten von Autoencodern wie der VAE repräsentieren explizit eine Wahrscheinlichkeitsverteilung und erlauben das direkte Ancestral Sampling. Die meisten anderen Arten erfordern ein MCMC-Stichprobenverfahren.

Contractive Autoencoder sind so konzipiert, dass ein Schätzwert der Tangentialebene der Datenmannigfaltigkeit wiederhergestellt wird. Durch das wiederholte Codieren und Decodieren mit hinzugefügtem Rauschen entsteht daher ein Random Walk auf der Oberfläche der Mannigfaltigkeit (*Rifai et al.*, 2012; *Mesnil et al.*, 2012). Dieses Diffusionsverfahren für Mannigfaltigkeit ist eine Art von Markow-Kette.

Des Weiteren gibt es eine allgemeinere Markow-Kette, die Stichproben aus beliebigen Denoising Autoencodern ziehen kann.

20.11.1 Markow-Kette für beliebige Denoising Autoencoder

Die obige Diskussion hat die Fragen offen gelassen, welches Rauschen eingespeist werden soll und wie die Markow-Kette zum Generieren aus der vom Autoencoder geschätzten Verteilung ermittelt werden soll. *Bengio et al.* (2013c) haben gezeigt, wie eine solche Markow-Kette für **generalisierte Denoising Autoencoder** konstruiert werden kann. Generalisierte Denoising Autoencoder zeichnen sich durch eine Denoising-Verteilung aus, die Stichproben eines Schätzwerts der einwandfreien Eingabe auf Grundlage der beschädigten Eingabe erzeugt.

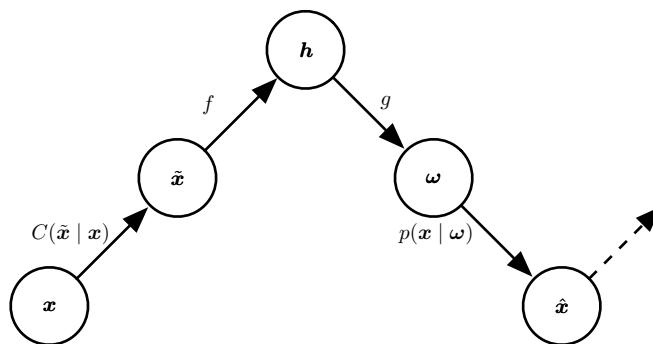


Abbildung 20.11: Jeder Schritt der Markow-Kette, die für einen trainierten Denoising Autoencoder genutzt wird, womit Stichproben aus dem implizit durch das Kriterium der Denoising Log-Likelihood trainierten probabilistischen Modell generiert werden. Jeder Schritt besteht aus diesen Teilschritten: (a) Hinzufügen von Rauschen im Rahmen eines Beschädigungsprozesses C im Zustand x ; es ergibt sich \tilde{x} . (b) Codieren mit der Funktion f ; es ergibt sich $h = f(\tilde{x})$. (c) Decodieren des Ergebnisses mit der Funktion g ; es ergeben sich die Parameter ω für die Rekonstruktionsverteilung. (d) Verwenden von ω für das Ziehen eines neuen Zustands aus der Rekonstruktionsverteilung $p(x | \omega = g(f(\tilde{x})))$. Im Normalfall eines quadratischen Rekonstruktionsfehlers, $g(h) = \hat{x}$, der $\mathbb{E}[x | \hat{x}]$ schätzt, besteht die Beschädigung aus dem Hinzufügen von normalverteiltem Rauschen; das Stichprobenentnahmen aus $p(x | \omega)$ besteht aus dem nochmaligen Hinzufügen von normalverteiltem Rauschen in die Rekonstruktion \hat{x} . Der letzte Rauschpegel sollte dem mittleren quadratischen Fehler der Rekonstruktionen entsprechen, wohingegen das hinzugefügte Rauschen ein Hyperparameter ist, der die Mischungsgeschwindigkeit sowie das Ausmaß, in dem der Schätzer die empirische Verteilung glättet, steuert (*Vincent*, 2011). Im gezeigten Beispiel sind nur die bedingten Verteilungen C und p stochastische Schritte (f und g sind deterministische Berechnungen), obwohl das Rauschen auch innerhalb des Autoencoders hinzugefügt werden kann, wie in generativen stochastischen Netzen (*Bengio et al.*, 2014).

Jeder Schritt der Markow-Kette, die aus der geschätzten Verteilung erzeugt wird, besteht aus den folgenden Teilschritten, die in Abbildung 20.11 dargestellt sind:

1. Beginnend mit dem vorherigen Zustand \mathbf{x} wird Rauschen eingespeist; $\tilde{\mathbf{x}}$ wird aus $C(\tilde{\mathbf{x}} \mid \mathbf{x})$ gezogen.
2. $\tilde{\mathbf{x}}$ wird in $\mathbf{h} = f(\tilde{\mathbf{x}})$ codiert.
3. \mathbf{h} wird decodiert, um die Parameter $\boldsymbol{\omega} = g(\mathbf{h})$ aus $p(\mathbf{x} \mid \boldsymbol{\omega} = g(\mathbf{h})) = p(\mathbf{x} \mid \tilde{\mathbf{x}})$ zu bestimmen.
4. Der nächste Zustand \mathbf{x} wird aus $p(\mathbf{x} \mid \boldsymbol{\omega} = g(\mathbf{h})) = p(\mathbf{x} \mid \tilde{\mathbf{x}})$ gezogen.

Bengio et al. (2014) haben gezeigt, dass die stationäre Verteilung der obigen Markow-Kette einen stetigen Schätzer (wenn auch einen impliziten) der datengenerierenden Verteilung von \mathbf{x} bildet, wenn ein Autoencoder $p(\mathbf{x} \mid \tilde{\mathbf{x}})$ einen stetigen Schätzer der entsprechenden wahren bedingten Verteilung bildet.

20.11.2 Clamping und bedingte Stichprobenentnahme

Ähnlich wie Boltzmann-Maschinen können Denoising Autoencoder und ihre Generalisierungen (wie die unten beschriebenen GSNs) verwendet werden, um Stichproben aus einer bedingten Verteilung $p(\mathbf{x}_f \mid \mathbf{x}_o)$ zu ziehen, einfach indem ein sogenanntes Clamping⁴ der *beobachteten* Einheiten \mathbf{x}_f durchgeführt wird und eine Stichprobewiederholung (engl. *resampling*) lediglich der *freien* Einheiten \mathbf{x}_o erfolgt, wenn \mathbf{x}_f und die gezogenen latenten Variablen (sofern vorhanden) gegeben sind. Zum Beispiel lassen sich MP-DBMs als eine Art Denoising Autoencoder betrachten; sie sind in der Lage, fehlende Eingaben als Stichproben zu ziehen. GSNs haben später einige der Konzepte von MP-DBMs generalisiert, um dieselbe Operation auszuführen (*Bengio et al.*, 2014). *Alain et al.* (2015) haben eine fehlende Bedingung in der 1. These aus *Bengio et al.* (2014) entdeckt, nämlich, dass der Übergangsoperator (der durch stochastische Zuordnung zwischen zwei Zuständen der Kette definiert wird) eine Eigenschaft namens **detailliertes Gleichgewicht** (engl. *detailed balance*) erfüllen muss, die angibt, dass eine Markow-Kette im Gleichgewicht unabhängig davon, ob der Übergangsoperator sich vorwärts oder rückwärts bewegt, in diesem Gleichgewichtszustand verbleibt.

⁴ Anm. zur Übersetzung: »to clamp« steht für einklemmen, einspannen, halten oder festhalten.

Ein Experiment, bei dem Clamping auf die Hälfte der Pixel (rechter Teil der Abbildung) angewandt, und für die andere Hälfte eine Markow-Kette ausgeführt wird, finden Sie in Abbildung 20.12.



Abbildung 20.12: Clamping in der rechten Hälfte des Bildes und Ausführen einer Markow-Kette durch Stichprobewiederholung der linken Hälfte in jedem Schritt. Diese Stichproben stammen aus einem GSN, das für die Rekonstruktion von MNIST-Ziffern in jedem Zeitschritt mittels Walk-Back-Verfahrens trainiert wurde.

20.11.3 Walk-Back-Trainingsverfahren

Das Walk-Back-Trainingsverfahren wurde von *Bengio et al.* (2013c) als Möglichkeit vorgeschlagen, die Konvergenz des generativen Trainings von Denoising Autoencodern zu beschleunigen. Statt eine Codier-Decodier-Rekonstruktion mit einem Schritt vorzunehmen, wechselt dieses Verfahren zwischen mehreren stochastischen Codier-Decodier-Schritten ab (wie in der generativen Markow-Kette), die an einem Trainingsbeispiel initialisiert werden (wie im CD-Algorithmus, siehe Abschnitt 18.2) und die letzten probabilistischen Rekonstruktionen bestraft (oder alle Rekonstruktionen auf dem Weg).

Das Training mit k Schritten ist äquivalent (im Sinne des Erreichens derselben stationären Verteilung) zu dem Training mit einem Schritt, bietet aber praktisch den Vorteil, dass Störmodi in größerer Entfernung von den Daten effizienter entfernt werden können.

20.12 Generative stochastische Netze

Generative stochastische Netze (engl. *generative stochastic networks*, GSNs) (*Bengio et al.*, 2014) sind Generalisierungen von Denoising Autoencodern, die neben den sichtbaren Variablen (meist als \mathbf{x} bezeichnet) latente Variablen \mathbf{h} in der generativen Markow-Kette enthalten.

Ein GSN wird durch zwei bedingte Wahrscheinlichkeitsverteilungen parametrisiert, die einen Schritt der Markow-Kette spezifizieren:

1. $p(\mathbf{x}^{(k)} | \mathbf{h}^{(k)})$ gibt an, wie die nächste sichtbare Variable auf Grundlage des aktuellen latenten Zustands erzeugt werden soll. Eine solche »Rekonstruktionsverteilung« gibt es auch in Denoising Autoencodern, RBMs, DBNs und DBMs.
2. $p(\mathbf{h}^{(k)} | \mathbf{h}^{(k-1)}, \mathbf{x}^{(k-1)})$ gibt an, wie die latente Zustandsvariable anhand der vorherigen latenten Zustandsvariable und der sichtbaren Variable aktualisiert werden soll.

Denoising Autoencoder und GSNs unterscheiden sich von klassischen probabilistischen Modellen (gerichtet oder ungerichtet) insofern, als sie den generativen Prozess selbst parametrisieren, nicht die mathematische Spezifikation der multivariaten Verteilung der sichtbaren und latenten Variablen. Die Letztere wird vielmehr *implizit* als stationäre Verteilung der generativen Markow-Kette definiert, *sofern es sie gibt*. Die Bedingungen für das Vorhandensein der stationären Verteilung sind schwach und entsprechen den Bedingungen für die üblichen MCMC-Verfahren (siehe Abschnitt 17.3). Diese Bedingungen sind notwendig, damit ein Mischen der Kette garantiert ist, aber durch eine bestimmte Wahl der Übergangsverteilungen kann dagegen verstößen werden (zum Beispiel wenn diese deterministisch sind).

Man kann sich unterschiedliche Trainingskriterien für GSNs vorstellen. Das von *Bengio et al.* (2014) vorgeschlagene und evaluierte Kriterium ist einfach eine Rekonstruktion der Log-Wahrscheinlichkeit der sichtbaren Einheiten wie bei Denoising Autoencodern. Dazu erfolgt ein Clamping von $\mathbf{x}^{(0)} = \mathbf{x}$ auf das beobachtete Beispiel und eine Maximierung der Wahrscheinlichkeit für die Erzeugung von \mathbf{x} in einem der folgenden Zeitschritte, also eine Maximierung von $\log p(\mathbf{x}^{(k)} = \mathbf{x} | \mathbf{h}^{(k)})$, wobei $\mathbf{h}^{(k)}$ als Stichprobe aus der Kette gezogen wird (für $\mathbf{x}^{(0)} = \mathbf{x}$). Um den Gradienten von $\log p(\mathbf{x}^{(k)} = \mathbf{x} | \mathbf{h}^{(k)})$ bezüglich der anderen Teile des Modells zu schätzen, setzen *Bengio et al.* (2014) auf die Reparametrisierung aus Abschnitt 20.9.

Das Walk-Back-Trainingsverfahren (siehe Abschnitt 20.11.3) wurde von (*Bengio et al.*, 2014) genutzt, um die Trainingskonvergenz von GSNs zu verbessern.

Der ursprüngliche GSN-Ansatz (*Bengio et al.*, 2014) war für unüberwachtes Lernen und implizite Modellierung von $p(\mathbf{x})$ für beobachtete Daten \mathbf{x} gedacht, aber es ist dennoch möglich, das Framework zu modifizieren, um $p(\mathbf{y} | \mathbf{x})$ zu optimieren.

Zum Beispiel generalisieren *Zhou und Troyanskaya* (2014) GSNs durch einfache Backpropagation der Rekonstruktion der Log-Wahrscheinlichkeit über die Ausgabevervariablen, wobei die Eingangsvariablen fest sind. Sie haben dies erfolgreich auf Modellsequenzen (sekundäre Proteinstruktur) angewandt und eine (eindimensionale) gefaltete Struktur in den Übergangsoperator der Markow-Kette eingeführt. Es ist wichtig, zu bedenken, dass für jeden Schritt der Markow-Kette eine neue Sequenz für jede Schicht erzeugt wird und dass diese Sequenz als Eingabe zur Berechnung weiterer Schichtwerte (z. B. dem darunter- und dem darüberliegenden) im nächsten Zeitschritt dient.

Deshalb arbeitet die Markow-Kette mit den Ausgabevervariablen (und zugehörigen verdeckten Schichten höherer Stufen) und die Eingabesequenz dient nur der Konditionierung dieser Kette, wobei die Backpropagation es ermöglicht zu lernen, wie die Eingabesequenz die Ausgabeverteilung, die implizit durch die Markow-Kette repräsentiert wird, konditionieren kann. Es handelt sich somit um den Einsatz des GSNs im Rahmen strukturierter Ausgaben.

Zöhrer und Pernkopf (2014) haben ein Hybridmodell vorgestellt, das eine überwachte Zielfunktion (wie in obigem Text) und eine unüberwachte Zielfunktion (wie in der ursprünglichen Abhandlung zu GSNs) kombiniert, einfach durch Addieren (mit unterschiedlicher Gewichtung) der überwachten und unüberwachten Kosten, also der Rekonstruktion der Log-Wahrscheinlichkeiten von \mathbf{y} bzw. \mathbf{x} . Ein solches Hybirdkriterium wurde bereits zuvor von *Larochelle und Bengio* (2008) für RBMs eingeführt. Sie weisen mit diesem Konzept eine verbesserte Klassifizierungsleistung auf.

20.13 Andere Generierungskonzepte

Die bisher beschriebenen Verfahren nutzen entweder das MCMC-Stichprobenverfahren, Ancestral Sampling oder eine Mischung der beiden zur Stichprobengenerierung. Dabei handelt es sich zwar um die beliebtesten Ansätze für die generative Modellierung, aber es sind nicht die einzigen.

Sohl-Dickstein et al. (2015) haben ein **Diffusion-Inversion**-Trainingsverfahren zum Erlernen eines generativen Modells auf Basis der Nichtgleichgewichtsthermodynamik entwickelt. Der Ansatz beruht auf dem Konzept, dass die Wahrscheinlichkeitsverteilungen, aus denen wir Stichproben ziehen möchten, eine Struktur aufweisen. Diese Struktur kann nach und nach durch einen Diffusionsprozess zerstört werden, der die Wahrscheinlichkeitsverteilung inkrementell so ändert, dass sie eine höhere Entropie aufweist. Um ein generatives Modell zu bilden, können wir den Prozess in umgekehrter Richtung ablaufen lassen und ein Modell trainieren, das die Struktur einer unstrukturierten Verteilung nach und nach wieder herstellt. Durch iterative Anwendung eines Prozesses, der eine Verteilung der Zielverteilung näherbringt, können wir uns der Zielverteilung langsam annähern. Dieser Ansatz erinnert an MCMC-Verfahren, da er viele Iterationen zum Erzeugen einer Stichprobe nutzt. Allerdings wird das Modell als die Wahrscheinlichkeitsverteilung definiert, die im letzten Schritt der Kette entsteht. In diesem Sinne wird durch das iterative Verfahren keine Approximation hervorgerufen. Der von *Sohl-Dickstein et al.* (2015) vorgestellte Ansatz ähnelt auch stark der generativen Interpretation des Denoising Autoencoders (Abschnitt 20.11.1). Wie beim Denoising Autoencoder trainiert die Diffusion-Inversion einen Übergangsoperator, der versucht, den Effekt des Hinzufügens von Rauschen probabilistisch zu negieren. Der Unterschied besteht darin, dass für die Diffusion-Inversion nur ein Schritt des Diffusionsprozesses rückgängig gemacht und nicht der gesamte Pfad bis zu einem sauberen Datenpunkt zurückverfolgt werden muss. Das berücksichtigt das folgende Dilemma der gewöhnlichen Zielfunktion für die Log-Likelihood der Rekonstruktion bei Denoising Autoencodern: Für geringe Rauschpegel sieht der Klassifikator nur Konfigurationen in der Nähe der Datenpunkte während er für hohe Rauschpegel eine fast unmögliche Aufgabe erledigen soll (da die Denoising-Verteilung sehr komplex und multimodal ist). Mit der Zielfunktion der Diffusion-Inversion kann der Klassifikator die Form der Dichte um die Datenpunkte herum exakter erlernen und dabei Störmodi entfernen, die sich weit abseits der Datenpunkte zeigen könnten.

Ein weiterer Ansatz zur Stichprobengenerierung ist das ABC-Framework. Die Bezeichnung stammt von der **approximativen baysschen Berechnung**, im Englischen *Approximate Bayesian Computation*. Bei diesem Ansatz werden Stichproben verworfen oder verändert, damit die Momente ausgewählter Funktionen der Stichproben zu denen der gewünschten Verteilung passen. Diese Idee nutzt zwar wie bei dem Moment Matching die Momente der Stichproben, unterscheidet sich aber vom Moment Matching, da die Stichproben selbst verändert werden und nicht das Modell so trainiert

wird, dass es automatisch Stichproben mit den passenden Momenten ausgibt. *Bachman und Precup* (2015) haben gezeigt, wie Konzepte aus dem ABC-Framework im Deep-Learning-Kontext eingesetzt werden können, indem sie MCMC-Trajektorien von GSNs mittels ABC geformt haben.

Wir glauben, dass noch viele weitere Ansätze für die generative Modellierung auf ihre Entdeckung warten.

20.14 Bewerten von generativen Modellen

Forscher im Bereich generativer Modelle müssen oft zwei generative Modelle miteinander vergleichen, um zu zeigen, dass ein neu entwickeltes generatives Modell eine bestimmte Verteilung besser als bereits vorhandene Modelle erfasst.

Das ist eine komplexe und delikate Angelegenheit. Häufig können wir die Log-Wahrscheinlichkeit der Daten mit dem Modell nicht wirklich bewerten, sondern nur eine Approximation davon. In diesen Fällen müssen wir uns ganz klar darüber werden (und dies auch angeben!), was genau gemessen wird. Ein Beispiel: Wir können einen stochastischen Schätzwert der Log-Likelihood für Modell A und eine deterministische untere Schranke der Log-Likelihood für Modell B bewerten. Der Score für Modell A ist höher als der für Modell B. Welches Modell ist besser? Wenn es uns darum geht, welches Modell die bessere interne Repräsentation der Verteilung bietet, können wir diese Frage nicht beantworten, da wir nicht wissen, wie lose die Schranke für Modell B ist. Wenn wir allerdings wissen möchten, wie gut das Modell in der Praxis einsetzbar ist – zum Beispiel für die Anomalie-Erkennung –, dann können wir zuversichtlich behaupten, dass ein Modell angesichts eines Kriteriums für die jeweilige Aufgabe zu bevorzugen ist, zum Beispiel auf Grundlage der Rangfolge von Testbeispielen und Rangfolgekriterien wie Genauigkeit und Trefferquote.

Eine weitere Schwierigkeit bei der Bewertung generativer Modelle besteht darin, dass die Kriterien für die Bewertung selbst häufig komplizierte Forschungsprobleme sind. Es kann sehr schwierig sein, Modelle objektiv miteinander zu vergleichen. Ein Beispiel: Wir verwenden das Annealed Importance Sampling (AIS) zum Bewerten von $\log Z$, um $\log \tilde{p}(\mathbf{x}) - \log Z$ für ein von uns neu entwickeltes Modell zu berechnen. Eine rechnerisch ökonomische Implementierung des AIS findet möglicherweise nicht alle Modi der Modellverteilung und unterschätzt Z , sodass wir $\log p(\mathbf{x})$ überschätzen. Es kann somit schwierig sein zu erkennen, ob ein hoher Likelihood-

Schätzwert das Ergebnis eines guten Modells oder einer schlechten AIS-Implementierung ist.

Andere Bereiche des Machine Learnings weisen für gewöhnlich einen gewissen Spielraum bei der Vorverarbeitung der Daten auf. Zum Beispiel ist es beim Vergleich der Korrektklassifikationsrate von Algorithmen zur Objekterkennung normalerweise akzeptabel, die Eingabebilder in der Vorverarbeitung für die einzelnen Algorithmen leicht unterschiedlich zu handhaben – je nach Voraussetzungen für die Eingabe. Anders bei der generativen Modellierung, denn hier sind selbst kleinste und subtile Änderungen in der Vorverarbeitung nicht akzeptabel. Jede Änderung der Eingangsdaten verändert die Verteilung, die erfasst werden soll, und führt zu einer dramatischen Änderung der Aufgabe. Wird die Eingabe beispielsweise mit 0,1 multipliziert, erhöht sich die Likelihood künstlich um den Faktor 10.

Probleme mit der Vorverarbeitung treten häufig auf, wenn generative Modelle anhand des MNIST-Datensatzes (einem der beliebten Benchmarks für die generative Modellierung) bewertet werden. Der MNIST-Datensatz enthält Graustufenbilder. Einige Modelle handhaben MNIST-Bilder als Punkte in einem reellen Vektorraum, andere als binäre Daten. Wieder andere behandeln die Graustufenwerte als Wahrscheinlichkeiten für binäre Stichproben. Reellwertige Modelle dürfen nur mit anderen reellwertigen Modellen verglichen werden, ebenso dürfen binärwertige Modelle nur mit anderen binärwertigen Modellen verglichen werden. Ansonsten befinden sich die Likelihoods, die gemessen werden, nicht im selben Raum. Bei binärwertigen Modellen kann die Log-Likelihood höchstens Null betragen, bei reellwertigen Modellen kann sie dagegen beliebig hoch sein, da sie ein Maß einer Dichte darstellt. Wenn es um binäre Modelle geht, dürfen nur Modelle verglichen werden, die exakt dieselbe Binarisierung nutzen. Ein Beispiel: Wir können ein graues Pixel als 0 oder 1 mit Schwellenwertbildung 0,5 binarisieren oder wir binarisieren es durch Ziehen einer Zufallsstichprobe, deren Wahrscheinlichkeit, 1 zu sein, sich aus der Intensität des grauen Pixels ergibt. Bei der Zufallsbinarisierung können wir den gesamten Datensatz einmalig binarisieren oder wir binarisieren in jedem Trainingsschritt ein anderes zufälliges Beispiel und ziehen dann mehrere Stichproben für die Bewertung. Alle drei Schemata führen zu stark unterschiedlichen Likelihood-Werten. Beim Vergleich unterschiedlicher Modelle ist es wichtig, dass beide dasselbe Binarisierungsschema für das Training und die Bewertung nutzen. Tatsächlich veröffentlichen Forscher, die einen einzelnen zufälligen Binarisierungsschritt nutzen, eine Datei, die die Ergebnisse der Zufallsbinarisierung enthält, damit es keine Unterschiede bei Ergebnissen aufgrund unterschiedlicher Resultate des Binarisierungsschritts gibt.

Da die Generierung realistischer Stichproben aus der Datenverteilung zu den Zielen eines generativen Modells gehört, werden generative Modelle in der Praxis häufig durch eine visuelle Prüfung der Stichproben bewertet. Im besten Fall übernehmen Versuchspersonen diesen Schritt, die die Quelle der Stichproben nicht kennen (*Denton et al.*, 2015). Leider kann auch ein sehr schlechtes probabilistisches Modell sehr gute Stichproben generieren. Die Kontrolle, ob das Modell nur einige der Trainingsbeispiele kopiert, läuft häufig wie in Abbildung 16.1 dargestellt ab. Das Konzept besteht darin, für einige der generierten Stichproben den jeweiligen Nearest-Neighbor in der Trainingsdatenmenge zu zeigen (gemäß euklidischem Abstand im Raum von \mathbf{x}). Dieser Test soll erkennen, ob das Modell eine Überanpassung der Trainingsdatenmenge aufweist und lediglich Trainingsinstanzen reproduziert. Es ist sogar möglich, gleichzeitig eine Überanpassung und eine Unteranpassung aufzuweisen und dennoch für sich genommen gut wirkende Stichproben zu erzeugen. Stellen Sie sich ein generatives Modell vor, das mit Bildern von Hunden und Katzen trainiert wurde, und lediglich lernt, die Trainingsbilder von Hunden zu reproduzieren. Bei einem solchen Modell liegt ganz klar eine Überanpassung vor, denn es produziert keine Bilder, die nicht in der Trainingsdatenmenge enthalten waren. Ebenso liegt aber eine Unteranpassung vor, denn es weist den Trainingsbildern mit Katzen keine Wahrscheinlichkeit zu. Dennoch würde ein Mensch jedem einzelnen Hundebild eine hohe Qualität zuschreiben. In diesem einfachen Beispiel wäre es natürlich für einen Menschen, der mehrere Stichproben untersucht, ein Leichtes, festzustellen, dass die Katzenbilder fehlen. In realistischeren Szenarios könnte ein generatives Modell, das mit Daten mit Zehntausenden von Modi trainiert wurde, eine kleine Anzahl von Modi ignorieren, ohne dass dies einem Menschen auffallen würde, der einfach nicht genügend Bilder untersuchen oder im Kopf behalten kann.

Da die visuelle Qualität der Stichproben keine zuverlässige Richtschnur darstellt, bewerten wir häufig auch die Log-Likelihood, die das Modell den Testdaten zuweist, sofern dies rechnerisch durchführbar ist. Leider scheint die Likelihood in einigen Fällen keines der für uns interessanten Attributte des Modells zu bestimmen. Zum Beispiel können reellwertige Modelle der MNIST-Sammlung eine beliebig hohe Likelihood erzielen, indem den sich niemals ändernden Hintergrundpixeln eine beliebig niedrige Varianz zugewiesen wird. Modelle und Algorithmen, die diese konstanten Merkmale erkennen, können unbegrenzte Belohnungen einstreichen, obwohl daraus kein besonderer Nutzen erwächst. Das Potential, Kosten, die gegen negativ unendlich gehen, zu erreichen, ist bei vielen Arten von Maximum-Likelihood-Aufgaben mit reellen Werten gegeben, aber sie ist besonders problematisch

bei generativen Modellen der MNIST-Sammlung, weil so viele der Ausgabewerte einfach vorherzusagen sind. Es ist also wichtig, andere Möglichkeiten zur Bewertung generativer Modelle zu entwickeln.

Theis et al. (2015) haben viele der Probleme im Zusammenhang mit der Bewertung generativer Modelle untersucht, darunter auch viele der vorgenannten Konzepte. Sie haben betont, dass es viele Einsatzmöglichkeiten für generative Modelle gibt und dass die Wahl des Kriteriums zur beabsichtigten Verwendung des Modells passen muss. Zum Beispiel sind einige generative Modelle besser darin, realistischeren Punkten eine hohe Wahrscheinlichkeit zuzuweisen, andere dagegen sind besser darin, unrealistischen Punkten nur selten eine hohe Wahrscheinlichkeit zuzuweisen. Diese Unterschiede können davon abhängen, ob ein generatives Modell entwickelt wurde, um $D_{\text{KL}}(p_{\text{data}} \parallel p_{\text{model}})$ oder $D_{\text{KL}}(p_{\text{model}} \parallel p_{\text{data}})$ zu minimieren (vgl. Abbildung 3.6). Leider weisen alle aktuell verwendeten Kriterien weiterhin ernste Schwächen auf, selbst wenn wir die Nutzung der einzelnen Kriterien auf die Aufgaben beschränken, für die sie sich am besten eignen. Eines der wichtigsten Forschungsthemen im Bereich der generativen Modellierung ist daher nicht nur die Verbesserung generativer Modelle, sondern vor allem die Entwicklung neuer Techniken, mit denen wir unsere Fortschritte messen können.

20.15 Schlussbemerkungen

Das Trainieren generativer Modelle mit verdeckten Einheiten ist eine hervorragende Möglichkeit, um Modellen ein Verständnis der Welt nahezubringen, die durch die Trainingsdaten repräsentiert wird. Durch das Erlernen eines Modells $p_{\text{model}}(\mathbf{x})$ und einer Repräsentation $p_{\text{model}}(\mathbf{h} \mid \mathbf{x})$ kann ein generatives Modell Antworten auf viele Inferenzprobleme über die Beziehungen zwischen Eingangsvariablen in \mathbf{x} liefern und viele unterschiedliche Arten zur Darstellung von \mathbf{x} anhand der Erwartungswerte von \mathbf{h} in verschiedenen Schichten der Hierarchie bieten. Generative Modelle werden KI-Systemen eines Tages ein Framework für alle zahlreichen verschiedenen intuitiven Konzepte zur Verfügung zu stellen, die sie zum Verstehen benötigen und sie in die Lage versetzen, in Angesicht der Ungewissheit Schlussfolgerungen hinsichtlich dieser Konzepte zu ziehen. Wir hoffen, dass unsere Leser neue Möglichkeiten finden, mit denen diese Ansätze leistungsfähiger werden und weiterhin den Weg hin zum Verständnis der zugrunde liegenden Prinzipien des Lernens und die Intelligenz bereiten.

Literaturverzeichnis

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., und Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software unter tensorflow.org.

Ackley, D. H., Hinton, G. E., und Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, **9**, 147–169.

Alain, G. und Bengio, Y. (2013). What regularized auto-encoders learn from the data generating distribution. In *ICLR'2013*, arXiv:1211.4246.

Alain, G., Bengio, Y., Yao, L., Éric Thibodeau-Laufer, Yosinski, J., und Vincent, P. (2015). GSNs: Generative stochastic networks. arXiv:1503.05571.

Allen, R. B. (1987). Several studies on natural language and back-propagation. In *IEEE First International Conference on Neural Networks*, Band 2, Seiten 335–341, San Diego. http://boballen.info/RBA/PAPERS/NL-BP/nl-bp.pdf.

Anderson, E. (1935). The Irises of the Gaspé Peninsula. *Bulletin of the American Iris Society*, **59**, 2–5.

Ba, J., Mnih, V., und Kavukcuoglu, K. (2014). Multiple object recognition with visual attention. arXiv:1412.7755.

Bachman, P. und Precup, D. (2015). Variational generative stochastic networks with collaborative shaping. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, Frankreich, 6–11. Juli 2015*, Seiten 1964–1972.

Bacon, P.-L., Bengio, E., Pineau, J., und Precup, D. (2015). Conditional computation in neural networks using a decision-theoretic approach. In *2nd Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM 2015)*.

- Bagnell, J. A. und Bradley, D. M. (2009). Differentiable sparse coding. In D. Koller, D. Schuurmans, Y. Bengio, und L. Bottou, Hrsg., *Advances in Neural Information Processing Systems 21 (NIPS'08)*, Seiten 113–120.
- Bahdanau, D., Cho, K., und Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *ICLR'2015, arXiv:1409.0473*.
- Bahl, L. R., Brown, P., de Souza, P. V., und Mercer, R. L. (1987). Speech recognition with continuous-parameter hidden Markov models. *Computer, Speech and Language*, **2**, 219–234.
- Baldi, P. und Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks*, **2**, 53–58.
- Baldi, P., Brunak, S., Frasconi, P., Soda, G., und Pollastri, G. (1999). Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, **15**(11), 937–946.
- Baldi, P., Sadowski, P., und Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, **5**.
- Ballard, D. H., Hinton, G. E., und Sejnowski, T. J. (1983). Parallel vision computation. *Nature*.
- Barlow, H. B. (1989). Unsupervised learning. *Neural Computation*, **1**, 295–311.
- Barron, A. E. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Trans. on Information Theory*, **39**, 930–945.
- Bartholomew, D. J. (1987). *Latent variable models and factor analysis*. Oxford University Press.
- Basilevsky, A. (1994). *Statistical Factor Analysis and Related Methods: Theory and Applications*. Wiley.
- Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., und Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.
- Basu, S. und Christensen, J. (2013). Teaching classification boundaries to humans. In *AAAI'2013*.
- Baxter, J. (1995). Learning internal representations. In *Proceedings of the 8th International Conference on Computational Learning Theory (COLT'95)*, Seiten 311–320, Santa Cruz, Kalifornien. ACM Press.
- Bayer, J. und Osendorfer, C. (2014). Learning stochastic recurrent networks. *ArXiv e-prints*.

- Becker, S. und Hinton, G. (1992). A self-organizing neural network that discovers surfaces in random-dot stereograms. *Nature*, **355**, 161–163.
- Behnke, S. (2001). Learning iterative image reconstruction in the neural abstraction pyramid. *Int. J. Computational Intelligence and Applications*, **1**(4), 427–438.
- Beiú, V., Quintana, J. M., und Avedillo, M. J. (2003). VLSI implementations of threshold logic-a comprehensive survey. *Neural Networks, IEEE Transactions on*, **14**(5), 1217–1243.
- Belkin, M. und Niyogi, P. (2002). Laplacian eigenmaps and spectral techniques for embedding and clustering. In T. Dietterich, S. Becker, und Z. Ghahramani, Hrsg., *Advances in Neural Information Processing Systems 14 (NIPS'01)*, Cambridge, MA. MIT Press.
- Belkin, M. und Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, **15**(6), 1373–1396.
- Bengio, E., Bacon, P.-L., Pineau, J., und Precup, D. (2015a). Conditional computation in neural networks for faster models. arXiv:1511.06297.
- Bengio, S. und Bengio, Y. (2000a). Taking on the curse of dimensionality in joint distributions using neural networks. *IEEE Transactions on Neural Networks, Sonderausgabe Data Mining and Knowledge Discovery*, **11**(3), 550–557.
- Bengio, S., Vinyals, O., Jaitly, N., und Shazeer, N. (2015b). Scheduled sampling for sequence prediction with recurrent neural networks. Technischer Bericht, arXiv:1506.03099.
- Bengio, Y. (1991). *Artificial Neural Networks and their Application to Sequence Recognition*. Doktorarbeit, McGill University, (Computer Science), Montreal, Kanada.
- Bengio, Y. (2000). Gradient-based optimization of hyperparameters. *Neural Computation*, **12**(8), 1889–1900.
- Bengio, Y. (2002). New distributed probabilistic language models. Technischer Bericht 1215, Dept. IRO, Université de Montréal.
- Bengio, Y. (2009). *Learning deep architectures for AI*. Now Publishers.
- Bengio, Y. (2013). Deep learning of representations: looking forward. In *Statistical Language and Speech Processing*, Band 7978 der *Lecture Notes in Computer Science*, Seiten 1–37. Springer, auch in arXiv unter <http://arxiv.org/abs/1305.0445>.
- Bengio, Y. (2015). Early inference in energy-based models approximates back-propagation. Technischer Bericht arXiv:1510.02777, Universite de Montreal.

- Bengio, Y. und Bengio, S. (2000b). Modeling high-dimensional discrete data with multi-layer neural networks. In *NIPS 12*, Seiten 400–406. MIT Press.
- Bengio, Y. und Delalleau, O. (2009). Justifying and generalizing contrastive divergence. *Neural Computation*, **21**(6), 1601–1621.
- Bengio, Y. und Grandvalet, Y. (2004). No unbiased estimator of the variance of k-fold cross-validation. In S. Thrun, L. Saul, und B. Schölkopf, Hrsg., *Advances in Neural Information Processing Systems 16 (NIPS'03)*, Cambridge, MA. MIT Press, Cambridge.
- Bengio, Y. und LeCun, Y. (2007). Scaling learning algorithms towards AI. In *Large Scale Kernel Machines*.
- Bengio, Y. und Monperrus, M. (2005). Non-local manifold tangent learning. In L. Saul, Y. Weiss, und L. Bottou, Hrsg., *Advances in Neural Information Processing Systems 17 (NIPS'04)*, Seiten 129–136. MIT Press.
- Bengio, Y. und Sénécal, J.-S. (2003). Quick training of probabilistic neural nets by importance sampling. In *Proceedings of AISTATS 2003*.
- Bengio, Y. und Sénécal, J.-S. (2008). Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Trans. Neural Networks*, **19**(4), 713–722.
- Bengio, Y., De Mori, R., Flammia, G., und Kompe, R. (1991). Phonetically motivated acoustic parameters for continuous speech recognition using artificial neural networks. In *Proceedings of EuroSpeech'91*.
- Bengio, Y., De Mori, R., Flammia, G., und Kompe, R. (1992). Neural network-Gaussian mixture hybrid for speech recognition or density estimation. In *NIPS 4*, Seiten 175–182. Morgan Kaufmann.
- Bengio, Y., Frasconi, P., und Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, Seiten 1183–1195, San Francisco. IEEE Press. (Invited Paper).
- Bengio, Y., Simard, P., und Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Tr. Neural Nets*.
- Bengio, Y., Latendresse, S., und Dugas, C. (1999). Gradient-based learning of hyper-parameters. Learning Conference, Snowbird.
- Bengio, Y., Ducharme, R., und Vincent, P. (2001). A neural probabilistic language model. In T. K. Leen, T. G. Dietterich, und V. Tresp, Hrsg., *NIPS'2000*, Seiten 932–938. MIT Press.
- Bengio, Y., Ducharme, R., Vincent, P., und Jauvin, C. (2003). A neural probabilistic language model. *JMLR*, **3**, 1137–1155.

- Bengio, Y., Le Roux, N., Vincent, P., Delalleau, O., und Marcotte, P. (2006a). Convex neural networks. In *NIPS'2005*, Seiten 123–130.
- Bengio, Y., Delalleau, O., und Le Roux, N. (2006b). The curse of highly variable functions for local kernel machines. In *NIPS'2005*.
- Bengio, Y., Larochelle, H., und Vincent, P. (2006c). Non-local manifold Parzen windows. In *NIPS'2005*. MIT Press.
- Bengio, Y., Lamblin, P., Popovici, D., und Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *NIPS'2006*.
- Bengio, Y., Louradour, J., Collobert, R., und Weston, J. (2009). Curriculum learning. In *ICML'09*.
- Bengio, Y., Mesnil, G., Dauphin, Y., und Rifai, S. (2013a). Better mixing via deep representations. In *ICML'2013*.
- Bengio, Y., Léonard, N., und Courville, A. (2013b). Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv:1308.3432.
- Bengio, Y., Yao, L., Alain, G., und Vincent, P. (2013c). Generalized denoising auto-encoders as generative models. In *NIPS'2013*.
- Bengio, Y., Courville, A., und Vincent, P. (2013d). Representation learning: A review and new perspectives. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, **35**(8), 1798–1828.
- Bengio, Y., Thibodeau-Laufer, E., Alain, G., und Yosinski, J. (2014). Deep generative stochastic networks trainable by backprop. In *ICML'2014*.
- Bennett, C. (1976). Efficient estimation of free energy differences from Monte Carlo data. *Journal of Computational Physics*, **22**(2), 245–268.
- Bennett, J. und Lanning, S. (2007). The Netflix prize.
- Berger, A. L., Della Pietra, V. J., und Della Pietra, S. A. (1996). A maximum entropy approach to natural language processing. *Computational Linguistics*, **22**, 39–71.
- Berglund, M. und Raiko, T. (2013). Stochastic gradient estimate variance in contrastive divergence and persistent contrastive divergence. *CoRR*, **abs/1312.6002**.
- Bergstra, J. (2011). *Incorporating Complex Cells into Neural Networks for Pattern Classification*. Doktorarbeit, Université de Montréal.
- Bergstra, J. und Bengio, Y. (2009). Slow, decorrelated features for pretraining complex cell-like networks. In *NIPS'2009*.

- Bergstra, J. und Bengio, Y. (2012). Random search for hyper-parameter optimization. *J. Machine Learning Res.*, **13**, 281–305.
- Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., und Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proc. SciPy*.
- Bergstra, J., Bardenet, R., Bengio, Y., und Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *NIPS'2011*.
- Berkes, P. und Wiskott, L. (2005). Slow feature analysis yields a rich repertoire of complex cell properties. *Journal of Vision*, **5**(6), 579–602.
- Bertsekas, D. P. und Tsitsiklis, J. (1996). *Neuro-Dynamic Programming*. Athena Scientific.
- Besag, J. (1975). Statistical analysis of non-lattice data. *The Statistician*, **24**(3), 179–195.
- Bishop, C. M. (1994). Mixture density networks.
- Bishop, C. M. (1995a). Regularization and complexity control in feed-forward networks. In *Proceedings International Conference on Artificial Neural Networks ICANN'95*, Band 1, Seite 141–148.
- Bishop, C. M. (1995b). Training with noise is equivalent to Tikhonov regularization. *Neural Computation*, **7**(1), 108–116.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
- Blum, A. L. und Rivest, R. L. (1992). Training a 3-node neural network is NP-complete.
- Blumer, A., Ehrenfeucht, A., Haussler, D., und Warmuth, M. K. (1989). Learnability and the Vapnik–Chervonenkis dimension. *Journal of the ACM*, **36**(4), 929—865.
- Bonnet, G. (1964). Transformations des signaux aléatoires à travers les systèmes non linéaires sans mémoire. *Annales des Télécommunications*, **19**(9–10), 203–220.
- Bordes, A., Weston, J., Collobert, R., und Bengio, Y. (2011). Learning structured embeddings of knowledge bases. In *AAAI 2011*.
- Bordes, A., Glorot, X., Weston, J., und Bengio, Y. (2012). Joint learning of words and meaning representations for open-text semantic parsing. *AISTATS'2012*.
- Bordes, A., Glorot, X., Weston, J., und Bengio, Y. (2013a). A semantic matching energy function for learning with multi-relational data. *Machine Learning: Special Issue on Learning Semantics*.

- Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., und Yakhnenko, O. (2013b). Translating embeddings for modeling multi-relational data. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, und K. Weinberger, Hrsg., *Advances in Neural Information Processing Systems 26*, Seiten 2787–2795. Curran Associates, Inc.
- Bornschein, J. und Bengio, Y. (2015). Reweighted wake-sleep. In *ICLR'2015*, arXiv:1406.2751.
- Bornschein, J., Shabanian, S., Fischer, A., und Bengio, Y. (2015). Training bidirectional Helmholtz machines. Technischer Bericht, arXiv:1506.03877.
- Boser, B. E., Guyon, I. M., und Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, Seiten 144–152, New York, NY, USA. ACM.
- Bottou, L. (1998). Online algorithms and stochastic approximations. In D. Saad, Hrsg., *Online Learning in Neural Networks*. Cambridge University Press, Cambridge, UK.
- Bottou, L. (2011). From machine learning to machine reasoning. Technischer Bericht, arXiv:1102.1808.
- Bottou, L. (2015). Multilayer neural networks. Deep Learning Summer School.
- Bottou, L. und Bousquet, O. (2008). The tradeoffs of large scale learning. In *NIPS'2008*.
- Boulanger-Lewandowski, N., Bengio, Y., und Vincent, P. (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *ICML'12*.
- Boureau, Y., Ponce, J., und LeCun, Y. (2010). A theoretical analysis of feature pooling in vision algorithms. In *Proc. International Conference on Machine learning (ICML'10)*.
- Boureau, Y., Le Roux, N., Bach, F., Ponce, J., und LeCun, Y. (2011). Ask the locals: multi-way local pooling for image recognition. In *Proc. International Conference on Computer Vision (ICCV'11)*. IEEE.
- Bourlard, H. und Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, **59**, 291–294.
- Bourlard, H. und Wellekens, C. (1989). Speech pattern discrimination and multi-layered perceptrons. *Computer Speech and Language*, **3**, 1–19.
- Boyd, S. und Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA.

- Brady, M. L., Raghavan, R., und Slawny, J. (1989). Back-propagation fails to separate where perceptrons succeed. *IEEE Transactions on Circuits and Systems*, **36**, 665–674.
- Brakel, P., Stroobandt, D., und Schrauwen, B. (2013). Training energy-based models for time-series imputation. *Journal of Machine Learning Research*, **14**, 2771–2797.
- Brand, M. (2003). Charting a manifold. In *NIPS'2002*, Seiten 961–968. MIT Press.
- Breiman, L. (1994). Bagging predictors. *Machine Learning*, **24**(2), 123–140.
- Breiman, L., Friedman, J. H., Olshen, R. A., und Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA.
- Bridle, J. S. (1990). Alphanets: a recurrent ‘neural’ network architecture with a hidden Markov model interpretation. *Speech Communication*, **9**(1), 83–92.
- Briggman, K., Denk, W., Seung, S., Helmstaedter, M. N., und Turaga, S. C. (2009). Maximin affinity learning of image segmentation. In *NIPS'2009*, Seiten 1865–1873.
- Brown, P. F., Cocke, J., Pietra, S. A. D., Pietra, V. J. D., Jelinek, F., Lafferty, J. D., Mercer, R. L., und Roossin, P. S. (1990). A statistical approach to machine translation. *Computational linguistics*, **16**(2), 79–85.
- Brown, P. F., Pietra, V. J. D., DeSouza, P. V., Lai, J. C., und Mercer, R. L. (1992). Class-based n -gram models of natural language. *Computational Linguistics*, **18**, 467–479.
- Bryson, A. und Ho, Y. (1969). *Applied optimal control: optimization, estimation, and control*. Blaisdell Pub. Co.
- Bryson, Jr., A. E. und Denham, W. F. (1961). A steepest-ascent method for solving optimum programming problems. Technischer Bericht BR-1303, Raytheon Company, Missle and Space Division.
- Buciluă, C., Caruana, R., und Niculescu-Mizil, A. (2006). Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, Seiten 535–541. ACM.
- Burda, Y., Grosse, R., und Salakhutdinov, R. (2015). Importance weighted autoencoders. *arXiv Vorabdruck arXiv:1509.00519*.
- Cai, M., Shi, Y., und Liu, J. (2013). Deep maxout neural networks for speech recognition. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, Seiten 291–296. IEEE.

- Carreira-Perpiñan, M. A. und Hinton, G. E. (2005). On contrastive divergence learning. In R. G. Cowell und Z. Ghahramani, Hrsg., *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS'05)*, Seiten 33–40. Society for Artificial Intelligence and Statistics.
- Caruana, R. (1993). Multitask connectionist learning. In *Proc. 1993 Connectionist Models Summer School*, Seiten 372–379.
- Cauchy, A. (1847). Méthode générale pour la résolution de systèmes d'équations simultanées. In *Compte rendu des séances de l'académie des sciences*, Seiten 536–538.
- Cayton, L. (2005). Algorithms for manifold learning. Technischer Bericht CS2008-0923, UCSD.
- Chandola, V., Banerjee, A., und Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, **41**(3), 15.
- Chapelle, O., Weston, J., und Schölkopf, B. (2003). Cluster kernels for semi-supervised learning. In S. Becker, S. Thrun, und K. Obermayer, Hrsg., *Advances in Neural Information Processing Systems 15 (NIPS'02)*, Seiten 585–592, Cambridge, MA. MIT Press.
- Chapelle, O., Schölkopf, B., und Zien, A., Hrsg. (2006). *Semi-Supervised Learning*. MIT Press, Cambridge, MA.
- Chellapilla, K., Puri, S., und Simard, P. (2006). High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, Hrsg., *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (Frankreich). Université de Rennes 1, Suvisoft. <http://www.suvisoft.com>.
- Chen, B., Ting, J.-A., Marlin, B. M., und de Freitas, N. (2010). Deep learning of invariant spatio-temporal features from video. NIPS*2010 Deep Learning and Unsupervised Feature Learning Workshop.
- Chen, S. F. und Goodman, J. T. (1999). An empirical study of smoothing techniques for language modeling. *Computer, Speech and Language*, **13**(4), 359–393.
- Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., und Temam, O. (2014a). DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, Seiten 269–284. ACM.
- Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., und Zhang, Z. (2015). MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv Vorabdruck arXiv:1512.01274*.

- Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., et al. (2014b). DaDianNao: A machine-learning supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, Seiten 609–622. IEEE.
- Chilimbi, T., Suzue, Y., Apacible, J., und Kalyanaraman, K. (2014). Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.
- Cho, K., Raiko, T., und Ilin, A. (2010). Parallel tempering is efficient for learning restricted Boltzmann machines. In *IJCNN'2010*.
- Cho, K., Raiko, T., und Ilin, A. (2011). Enhanced gradient and adaptive learning rate for training restricted Boltzmann machines. In *ICML'2011*, Seiten 105–112.
- Cho, K., van Merriënboer, B., Gulcehre, C., Bougares, F., Schwenk, H., und Bengio, Y. (2014a). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP 2014)*.
- Cho, K., Van Merriënboer, B., Bahdanau, D., und Bengio, Y. (2014b). On the properties of neural machine translation: Encoder-decoder approaches. *ArXiv e-prints*, **abs/1409.1259**.
- Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., und LeCun, Y. (2014). The loss surface of multilayer networks.
- Chorowski, J., Bahdanau, D., Cho, K., und Bengio, Y. (2014). End-to-end continuous speech recognition using attention-based recurrent NN: First results. arXiv:1412.1602.
- Chrisman, L. (1991). Learning recursive distributed representations for holistic computation. *Connection Science*, **3**(4), 345–366. <http://repository.cmu.edu/cgi/viewcontent.cgi?article=3061&context=compsci>.
- Christianson, B. (1992). Automatic Hessians by reverse accumulation. *IMA Journal of Numerical Analysis*, **12**(2), 135–150.
- Chrupala, G., Kadar, A., und Alishahi, A. (2015). Learning language through pictures. arXiv 1506.03694.
- Chung, J., Gulcehre, C., Cho, K., und Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *NIPS'2014 Deep Learning workshop*, arXiv 1412.3555.
- Chung, J., Gülgür, C., Cho, K., und Bengio, Y. (2015a). Gated feedback recurrent neural networks. In *ICML'15*.

- Chung, J., Kastner, K., Dinh, L., Goel, K., Courville, A., und Bengio, Y. (2015b). A recurrent latent variable model for sequential data. In *NIPS'2015*.
- Ciresan, D., Meier, U., Masci, J., und Schmidhuber, J. (2012). Multi-column deep neural network for traffic sign classification. *Neural Networks*, **32**, 333–338.
- Ciresan, D. C., Meier, U., Gambardella, L. M., und Schmidhuber, J. (2010). Deep big simple neural nets for handwritten digit recognition. *Neural Computation*, **22**, 1–14.
- Coates, A. und Ng, A. Y. (2011). The importance of encoding versus training with sparse coding and vector quantization. In *ICML'2011*.
- Coates, A., Lee, H., und Ng, A. Y. (2011). An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*.
- Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., und Andrew, N. (2013). Deep learning with COTS HPC systems. In S. Dasgupta und D. McAllester, Hrsg., *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, Band 28 (3), Seiten 1337–1345. JMLR Workshop and Conference Proceedings.
- Cohen, N., Sharir, O., und Shashua, A. (2015). On the expressive power of deep learning: A tensor analysis. arXiv:1509.05009.
- Collobert, R. (2004). *Large Scale Machine Learning*. Doktorarbeit, Université de Paris VI, LIP6.
- Collobert, R. (2011). Deep learning for efficient discriminative parsing. In *AISTATS'2011*.
- Collobert, R. und Weston, J. (2008a). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML'2008*.
- Collobert, R. und Weston, J. (2008b). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML'2008*.
- Collobert, R., Bengio, S., und Bengio, Y. (2001). A parallel mixture of SVMs for very large scale problems. Technischer Bericht IDIAP-RR-01-12, IDIAP.
- Collobert, R., Bengio, S., und Bengio, Y. (2002). Parallel mixture of SVMs for very large scale problems. *Neural Computation*, **14**(5), 1105–1114.
- Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., und Kuksa, P. (2011a). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, **12**, 2493–2537.
- Collobert, R., Kavukcuoglu, K., und Farabet, C. (2011b). Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

- Comon, P. (1994). Independent component analysis - a new concept? *Signal Processing*, **36**, 287–314.
- Cortes, C. und Vapnik, V. (1995). Support vector networks. *Machine Learning*, **20**, 273–297.
- Couprise, C., Farabet, C., Najman, L., und LeCun, Y. (2013). Indoor semantic segmentation using depth information. In *International Conference on Learning Representations (ICLR2013)*.
- Courbariaux, M., Bengio, Y., und David, J.-P. (2015). Low precision arithmetic for deep learning. In *Arxiv:1412.7024, ICLR'2015 Workshop*.
- Courville, A., Bergstra, J., und Bengio, Y. (2011). Unsupervised models of images by spike-and-slab RBMs. In *ICML'11*.
- Courville, A., Desjardins, G., Bergstra, J., und Bengio, Y. (2014). The spike-and-slab RBM and extensions to discrete and sparse data distributions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **36**(9), 1874–1887.
- Cover, T. M. und Thomas, J. A. (2006). *Elements of Information Theory, 2. Auflage*. Wiley-Interscience.
- Cox, D. und Pinto, N. (2011). Beyond simple features: A large-scale feature search approach to unconstrained face recognition. In *Automatic Face & Gesture Recognition and Workshops (FG 2011), 2011 IEEE International Conference on*, Seiten 8–15. IEEE.
- Cramér, H. (1946). *Mathematical methods of statistics*. Princeton University Press.
- Crick, F. H. C. und Mitchison, G. (1983). The function of dream sleep. *Nature*, **304**, 111–114.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, **2**, 303–314.
- Dahl, G. E., Ranzato, M., Mohamed, A., und Hinton, G. E. (2010). Phone recognition with the mean-covariance restricted Boltzmann machine. In *NIPS'2010*.
- Dahl, G. E., Yu, D., Deng, L., und Acero, A. (2012). Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, **20**(1), 33–42.
- Dahl, G. E., Sainath, T. N., und Hinton, G. E. (2013). Improving deep neural networks for LVCSR using rectified linear units and dropout. In *ICASSP'2013*.
- Dahl, G. E., Jaitly, N., und Salakhutdinov, R. (2014). Multi-task neural networks for QSAR predictions. arXiv:1406.1231.

- Dauphin, Y. und Bengio, Y. (2013). Stochastic ratio matching of RBMs for sparse high-dimensional inputs. In *NIPS'26*. NIPS Foundation.
- Dauphin, Y., Glorot, X., und Bengio, Y. (2011). Large-scale learning of embeddings with reconstruction sampling. In *ICML'2011*.
- Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., und Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS'2014*.
- Davis, A., Rubinstein, M., Wadhwa, N., Mysore, G., Durand, F., und Freeman, W. T. (2014). The visual microphone: Passive recovery of sound from video. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, **33**(4), 79:1–79:10.
- Dayan, P. (1990). Reinforcement comparison. In *Connectionist Models: Proceedings of the 1990 Connectionist Summer School*, San Mateo, CA.
- Dayan, P. und Hinton, G. E. (1996). Varieties of Helmholtz machine. *Neural Networks*, **9**(8), 1385–1403.
- Dayan, P., Hinton, G. E., Neal, R. M., und Zemel, R. S. (1995). The Helmholtz machine. *Neural computation*, **7**(5), 889–904.
- Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., und Ng, A. Y. (2012). Large scale distributed deep networks. In *NIPS'2012*.
- Dean, T. und Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, **5**(3), 142–150.
- Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., und Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, **41**(6), 391–407.
- Delalleau, O. und Bengio, Y. (2011). Shallow vs. deep sum-product networks. In *NIPS*.
- Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., und Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.
- Deng, J., Berg, A. C., Li, K., und Fei-Fei, L. (2010a). What does classifying more than 10,000 image categories tell us? In *Proceedings of the 11th European Conference on Computer Vision: Part V*, ECCV'10, Seiten 71–84, Berlin, Heidelberg. Springer-Verlag.
- Deng, L. und Yu, D. (2014). Deep learning – methods and applications. *Foundations and Trends in Signal Processing*.
- Deng, L., Seltzer, M., Yu, D., Acero, A., Mohamed, A., und Hinton, G. (2010b). Binary coding of speech spectrograms using a deep auto-encoder. In *Interspeech 2010*, Makuhari, Chiba, Japan.

- Denil, M., Bazzani, L., Larochelle, H., und de Freitas, N. (2012). Learning where to attend with deep architectures for image tracking. *Neural Computation*, **24**(8), 2151–2184.
- Denton, E., Chintala, S., Szlam, A., und Fergus, R. (2015). Deep generative image models using a Laplacian pyramid of adversarial networks. *NIPS*.
- Desjardins, G. und Bengio, Y. (2008). Empirical evaluation of convolutional RBMs for vision. Technischer Bericht 1327, Département d’Informatique et de Recherche Opérationnelle, Université de Montréal.
- Desjardins, G., Courville, A. C., Bengio, Y., Vincent, P., und Delalleau, O. (2010). Tempered Markov chain Monte Carlo for training of restricted Boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, Seiten 145–152.
- Desjardins, G., Courville, A., und Bengio, Y. (2011). On tracking the partition function. In *NIPS’2011*.
- Desjardins, G., Simonyan, K., Pascanu, R., et al. (2015). Natural neural networks. In *Advances in Neural Information Processing Systems*, Seiten 2062–2070.
- Devlin, J., Zbib, R., Huang, Z., Lamar, T., Schwartz, R., und Makhoul, J. (2014). Fast and robust neural network joint models for statistical machine translation. In *Proc. ACL’2014*.
- Devroye, L. (2013). *Non-Uniform Random Variate Generation*. SpringerLink: Bücher. Springer New York.
- DiCarlo, J. J. (2013). Mechanisms underlying visual object recognition: Humans vs. neurons vs. machines. NIPS Tutorial.
- Dinh, L., Krueger, D., und Bengio, Y. (2014). NICE: Non-linear independent components estimation. arXiv:1410.8516.
- Donahue, J., Hendricks, L. A., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., und Darrell, T. (2014). Long-term recurrent convolutional networks for visual recognition and description. arXiv:1411.4389.
- Donoho, D. L. und Grimes, C. (2003). Hessian eigenmaps: new locally linear embedding techniques for high-dimensional data. Technischer Bericht 2003-08, Dept. Statistics, Stanford University.
- Dosovitskiy, A., Springenberg, J. T., und Brox, T. (2015). Learning to generate chairs with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Seiten 1538–1546.
- Doya, K. (1993). Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on Neural Networks*, **1**, 75–80.

- Dreyfus, S. E. (1962). The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, **5**(1), 30–45.
- Dreyfus, S. E. (1973). The computational solution of optimal control problems with time lag. *IEEE Transactions on Automatic Control*, **18**(4), 383–385.
- Drucker, H. und LeCun, Y. (1992). Improving generalisation performance using double back-propagation. *IEEE Transactions on Neural Networks*, **3**(6), 991–997.
- Duchi, J., Hazan, E., und Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*.
- Dudik, M., Langford, J., und Li, L. (2011). Doubly robust policy evaluation and learning. In *Proceedings of the 28th International Conference on Machine learning*, ICML '11.
- Dugas, C., Bengio, Y., Bélisle, F., und Nadeau, C. (2001). Incorporating second-order functional knowledge for better option pricing. In T. Leen, T. Dietterich, und V. Tresp, Hrsg., *Advances in Neural Information Processing Systems 13 (NIPS'00)*, Seiten 472–478. MIT Press.
- Dziugaite, G. K., Roy, D. M., und Ghahramani, Z. (2015). Training generative neural networks via maximum mean discrepancy optimization. *arXiv Vorabdruck arXiv:1505.03906*.
- El Hihi, S. und Bengio, Y. (1996). Hierarchical recurrent neural networks for long-term dependencies. In *NIPS'1995*.
- Elkahky, A. M., Song, Y., und He, X. (2015). A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *Proceedings of the 24th International Conference on World Wide Web*, Seiten 278–288.
- Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition*, **48**, 781–799.
- Erhan, D., Manzagol, P.-A., Bengio, Y., Bengio, S., und Vincent, P. (2009). The difficulty of training deep architectures and the effect of unsupervised pre-training. In *Proceedings of AISTATS'2009*.
- Erhan, D., Bengio, Y., Courville, A., Manzagol, P., Vincent, P., und Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *J. Machine Learning Res.*
- Fahlman, S. E., Hinton, G. E., und Sejnowski, T. J. (1983). Massively parallel architectures for AI: NETL, thistle, and Boltzmann machines. In *Proceedings of the National Conference on Artificial Intelligence AAAI-83*.

- Fang, H., Gupta, S., Iandola, F., Srivastava, R., Deng, L., Dollár, P., Gao, J., He, X., Mitchell, M., Platt, J. C., Zitnick, C. L., und Zweig, G. (2015). From captions to visual concepts and back. arXiv:1411.4952.
- Farabet, C., LeCun, Y., Kavukcuoglu, K., Culurciello, E., Martini, B., Akselrod, P., und Talay, S. (2011). Large-scale FPGA-based convolutional networks. In R. Bekkerman, M. Bilenko, und J. Langford, Hrsg., *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press.
- Farabet, C., Couprie, C., Najman, L., und LeCun, Y. (2013). Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **35**(8), 1915–1929.
- Fei-Fei, L., Fergus, R., und Perona, P. (2006). One-shot learning of object categories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **28**(4), 594–611.
- Finn, C., Tan, X. Y., Duan, Y., Darrell, T., Levine, S., und Abbeel, P. (2015). Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. *arXiv Vorabdruck arXiv:1509.06113*.
- Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, 179–188.
- Földiák, P. (1989). Adaptive network for optimal linear feature extraction. In *International Joint Conference on Neural Networks (IJCNN)*, Band 1, Seiten 401–405, Washington 1989. IEEE, New York.
- Forcada, M., und Ñeco, R. (1997). Recursive hetero-associative memories for translation. In *Biological and Artificial Computation: From Neuroscience to Technology*, Seiten 453–462.
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.1968>.
- Franzius, M., Sprikeler, H., und Wiskott, L. (2007). Slowness and sparseness lead to place, head-direction, and spatial-view cells.
- Franzius, M., Wilbert, N., und Wiskott, L. (2008). Invariant object recognition with slow feature analysis. In *Artificial Neural Networks-ICANN 2008*, Seiten 961–970. Springer.
- Frasconi, P., Gori, M., und Sperduti, A. (1997). On the efficient classification of data structures by neural networks. In *Proc. Int. Joint Conf. on Artificial Intelligence*.
- Frasconi, P., Gori, M., und Sperduti, A. (1998). A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, **9**(5), 768–786.

- Freund, Y. und Schapire, R. E. (1996a). Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of Thirteenth International Conference*, Seiten 148–156, USA. ACM.
- Freund, Y. und Schapire, R. E. (1996b). Game theory, on-line prediction and boosting. In *Proceedings of the Ninth Annual Conference on Computational Learning Theory*, Seiten 325–332.
- Frey, B. J. (1998). *Graphical models for machine learning and digital communication*. MIT Press.
- Frey, B. J., Hinton, G. E., und Dayan, P. (1996). Does the wake-sleep algorithm learn good density estimators? In D. Touretzky, M. Mozer, und M. Hasselmo, Hrsg., *Advances in Neural Information Processing Systems 8 (NIPS'95)*, Seiten 661–670. MIT Press, Cambridge, MA.
- Frobenius, G. (1908). Über Matrizen aus positiven Elementen, s. *B. Preuss. Akad. Wiss. Berlin*.
- Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, **20**, 121–136.
- Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, **36**, 193–202.
- Gal, Y. und Ghahramani, Z. (2015). Bayesian convolutional neural networks with Bernoulli approximate variational inference. *arXiv Vorabdruck arXiv:1506.02158*.
- Gallinari, P., LeCun, Y., Thiria, S., und Fogelman-Soulie, F. (1987). Memoires associatives distribuees. In *Proceedings of COGNITIVA 87*, Paris, La Villette.
- Garcia-Duran, A., Bordes, A., Usunier, N., und Grandvalet, Y. (2015). Combining two and three-way embeddings models for link prediction in knowledge bases. *arXiv Vorabdruck arXiv:1506.00999*.
- Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., und Pallett, D. S. (1993). Darpa timit acoustic-phonetic continuous speech corpus cd-rom. nist speech disc 1-1.1. *NASA STI/Recon Technical Report N*, **93**, 27403.
- Garson, J. (1900). The metric system of identification of criminals, as used in Great Britain and Ireland. *The Journal of the Anthropological Institute of Great Britain and Ireland*, (2), 177–227.
- Gers, F. A., Schmidhuber, J., und Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural computation*, **12**(10), 2451–2471.

- Ghahramani, Z. und Hinton, G. E. (1996). The EM algorithm for mixtures of factor analyzers. Technischer Bericht CRG-TR-96-1, Dpt. of Comp. Sci., Univ. of Toronto.
- Gillick, D., Brunk, C., Vinyals, O., und Subramanya, A. (2015). Multilingual language processing from bytes. *arXiv Vorabdruck arXiv:1512.00103*.
- Girshick, R., Donahue, J., Darrell, T., und Malik, J. (2015). Region-based convolutional networks for accurate object detection and segmentation.
- Giudice, M. D., Manera, V., und Keysers, C. (2009). Programmed to learn? The ontogeny of mirror neurons. *Dev. Sci.*, **12**(2), 350–363.
- Glorot, X. und Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *AISTATS'2010*.
- Glorot, X., Bordes, A., und Bengio, Y. (2011a). Deep sparse rectifier neural networks. In *AISTATS'2011*.
- Glorot, X., Bordes, A., und Bengio, Y. (2011b). Domain adaptation for large-scale sentiment classification: A deep learning approach. In *ICML'2011*.
- Goldberger, J., Roweis, S., Hinton, G. E., und Salakhutdinov, R. (2005). Neighbourhood components analysis. In L. Saul, Y. Weiss, und L. Bottou, Hrsg., *Advances in Neural Information Processing Systems 17 (NIPS'04)*. MIT Press.
- Gong, S., McKenna, S., und Psarrou, A. (2000). *Dynamic Vision: From Images to Face Recognition*. Imperial College Press.
- Goodfellow, I., Le, Q., Saxe, A., und Ng, A. (2009). Measuring invariances in deep networks. In *NIPS'2009*, Seiten 646–654.
- Goodfellow, I., Koenig, N., Muja, M., Pantofaru, C., Sorokin, A., und Takayama, L. (2010). Help me help you: Interfaces for personal robots. In *Proc. of Human Robot Interaction (HRI)*, Osaka, Japan. ACM Press, ACM Press.
- Goodfellow, I. J. (2010). Technischer Bericht: Multidimensional, downsampled convolution for autoencoders. Technischer Bericht, Université de Montréal.
- Goodfellow, I. J. (2014). On distinguishability criteria for estimating generative models. In *International Conference on Learning Representations, Workshops Track*.
- Goodfellow, I. J., Courville, A., und Bengio, Y. (2011). Spike-and-slab sparse coding for unsupervised feature discovery. In *NIPS Workshop on Challenges in Learning Hierarchical Models*.
- Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., und Bengio, Y. (2013a). Maxout networks. In S. Dasgupta und D. McAllester, Hrsg., *ICML'13*, Seiten 1319–1327.

- Goodfellow, I. J., Mirza, M., Courville, A., und Bengio, Y. (2013b). Multi-prediction deep Boltzmann machines. In *NIPS'26*. NIPS Foundation.
- Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., und Bengio, Y. (2013c). PyLearn2: a machine learning research library. *arXiv Vorabdruck arXiv:1308.4214*.
- Goodfellow, I. J., Courville, A., und Bengio, Y. (2013d). Scaling up spike-and-slab models for unsupervised feature learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **35**(8), 1902–1914.
- Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., und Bengio, Y. (2014a). An empirical investigation of catastrophic forgetting in gradient-based neural networks. In *ICLR'2014*.
- Goodfellow, I. J., Shlens, J., und Szegedy, C. (2014b). Explaining and harnessing adversarial examples. *CoRR*, **abs/1412.6572**.
- Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., und Bengio, Y. (2014c). Generative adversarial networks. In *NIPS'2014*.
- Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., und Shet, V. (2014d). Multi-digit number recognition from Street View imagery using deep convolutional neural networks. In *International Conference on Learning Representations*.
- Goodfellow, I. J., Vinyals, O., und Saxe, A. M. (2015). Qualitatively characterizing neural network optimization problems. In *International Conference on Learning Representations*.
- Goodman, J. (2001). Classes for fast maximum entropy training. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Utah.
- Gori, M. und Tesi, A. (1992). On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-14**(1), 76–86.
- Gosset, W. S. (1908). The probable error of a mean. *Biometrika*, **6**(1), 1–25.
Ursprünglich unter dem Pseudonym »Student« veröffentlicht.
- Gouws, S., Bengio, Y., und Corrado, G. (2014). BilBOWA: Fast bilingual distributed representations without word alignments. Technischer Bericht, *arXiv:1410.2455*.
- Graf, H. P. und Jackel, L. D. (1989). Analog electronic neural network circuits. *Circuits and Devices Magazine, IEEE*, **5**(4), 44–49.
- Graves, A. (2011). Practical variational inference for neural networks. In *NIPS'2011*.

- Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence. Springer.
- Graves, A. (2013). Generating sequences with recurrent neural networks. Technischer Bericht, arXiv:1308.0850.
- Graves, A. und Jaitly, N. (2014). Towards end-to-end speech recognition with recurrent neural networks. In *ICML'2014*.
- Graves, A. und Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, **18**(5), 602–610.
- Graves, A. und Schmidhuber, J. (2009). Offline handwriting recognition with multidimensional recurrent neural networks. In D. Koller, D. Schuurmans, Y. Bengio, und L. Bottou, Hrsg., *NIPS'2008*, Seiten 545–552.
- Graves, A., Fernández, S., Gomez, F., und Schmidhuber, J. (2006). Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *ICML'2006*, Seiten 369–376, Pittsburgh, USA.
- Graves, A., Liwicki, M., Bunke, H., Schmidhuber, J., und Fernández, S. (2008). Unconstrained on-line handwriting recognition with recurrent neural networks. In J. Platt, D. Koller, Y. Singer, und S. Roweis, Hrsg., *NIPS'2007*, Seiten 577–584.
- Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H., und Schmidhuber, J. (2009). A novel connectionist system for unconstrained handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **31**(5), 855–868.
- Graves, A., Mohamed, A., und Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *ICASSP'2013*, Seiten 6645–6649.
- Graves, A., Wayne, G., und Danihelka, I. (2014a). Neural Turing machines. arXiv:1410.5401.
- Graves, A., Wayne, G., und Danihelka, I. (2014b). Neural Turing machines. *arXiv Vorabdruck arXiv:1410.5401*.
- Grefenstette, E., Hermann, K. M., Suleyman, M., und Blunsom, P. (2015). Learning to transduce with unbounded memory. In *NIPS'2015*.
- Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., und Schmidhuber, J. (2015). LSTM: a search space odyssey. *arXiv Vorabdruck arXiv:1503.04069*.
- Gregor, K. und LeCun, Y. (2010a). Emergence of complex-like cells in a temporal product network with local receptive fields. Technischer Bericht, arXiv:1006.0448.

- Gregor, K. und LeCun, Y. (2010b). Learning fast approximations of sparse coding. In L. Bottou und M. Littman, Hrsg., *Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML-10)*. ACM.
- Gregor, K., Danihelka, I., Mnih, A., Blundell, C., und Wierstra, D. (2014). Deep autoregressive networks. In *International Conference on Machine Learning (ICML'2014)*.
- Gregor, K., Danihelka, I., Graves, A., und Wierstra, D. (2015). DRAW: A recurrent neural network for image generation. *arXiv Vorabdruck arXiv:1502.04623*.
- Gretton, A., Borgwardt, K. M., Rasch, M. J., Schölkopf, B., und Smola, A. (2012). A kernel two-sample test. *The Journal of Machine Learning Research*, **13**(1), 723–773.
- Gülgeyre, C. und Bengio, Y. (2013). Knowledge matters: Importance of prior information for optimization. In *International Conference on Learning Representations (ICLR'2013)*.
- Guo, H. und Gelfand, S. B. (1992). Classification trees with neural network feature extraction. *Neural Networks, IEEE Transactions on*, **3**(6), 923–933.
- Gupta, S., Agrawal, A., Gopalakrishnan, K., und Narayanan, P. (2015). Deep learning with limited numerical precision. *CoRR*, **abs/1502.02551**.
- Gutmann, M. und Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10)*.
- Hadsell, R., Sermanet, P., Ben, J., Erkan, A., Han, J., Muller, U., und LeCun, Y. (2007). Online learning for offroad robots: Spatial label propagation to learn long-range traversability. In *Proceedings of Robotics: Science and Systems*, Atlanta, GA, USA.
- Hajnal, A., Maass, W., Pudlak, P., Szegedy, M., und Turan, G. (1993). Threshold circuits of bounded depth. *J. Comput. System. Sci.*, **46**, 129–154.
- Håstad, J. (1986). Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th annual ACM Symposium on Theory of Computing*, Seiten 6–20, Berkeley, California. ACM Press.
- Håstad, J. und Goldmann, M. (1991). On the power of small-depth threshold circuits. *Computational Complexity*, **1**, 113–129.
- Hastie, T., Tibshirani, R., und Friedman, J. (2001). *The elements of statistical learning: data mining, inference and prediction*. Springer Series in Statistics. Springer Verlag.

- He, K., Zhang, X., Ren, S., und Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. *arXiv Vorabdruck arXiv:1502.01852*.
- Hebb, D. O. (1949). *The Organization of Behavior*. Wiley, New York.
- Henaff, M., Jarrett, K., Kavukcuoglu, K., und LeCun, Y. (2011). Unsupervised learning of sparse features for scalable audio classification. In *ISMIR'11*.
- Henderson, J. (2003). Inducing history representations for broad coverage statistical parsing. In *HLT-NAACL*, Seiten 103–110.
- Henderson, J. (2004). Discriminative training of a neural network statistical parser. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, Seite 95.
- Henniges, M., Puertas, G., Bornschein, J., Eggert, J., und Lücke, J. (2010). Binary sparse coding. In *Latent Variable Analysis and Signal Separation*, Seiten 450–457. Springer.
- Herault, J. und Ans, B. (1984). Circuits neuronaux à synapses modifiables: Décodage de messages composites par apprentissage non supervisé. *Comptes Rendus de l'Académie des Sciences*, **299(III-13)**, 525—528.
- Hinton, G. (2012). Neural networks for machine learning. Coursera, Videovorträge.
- Hinton, G., Deng, L., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., und Kingsbury, B. (2012a). Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, **29**(6), 82–97.
- Hinton, G., Vinyals, O., und Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv Vorabdruck arXiv:1503.02531*.
- Hinton, G. E. (1989). Connectionist learning procedures. *Artificial Intelligence*, **40**, 185–234.
- Hinton, G. E. (1990). Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, **46**(1), 47–75.
- Hinton, G. E. (1999). Products of experts. In *ICANN'1999*.
- Hinton, G. E. (2000). Training products of experts by minimizing contrastive divergence. Technischer Bericht GCNU TR 2000-004, Gatsby Unit, University College London.
- Hinton, G. E. (2006). To recognize shapes, first learn to generate images. Technischer Bericht UTML TR 2006-003, University of Toronto.

- Hinton, G. E. (2007a). How to do backpropagation in a brain. Invited Talk beim NIPS'2007 Deep Learning Workshop.
- Hinton, G. E. (2007b). Learning multiple layers of representation. *Trends in cognitive sciences*, **11**(10), 428–434.
- Hinton, G. E. (2010). A practical guide to training restricted Boltzmann machines. Technischer Bericht UTML TR 2010-003, Department of Computer Science, University of Toronto.
- Hinton, G. E. und Ghahramani, Z. (1997). Generative models for discovering sparse distributed representations. *Philosophical Transactions of the Royal Society of London*.
- Hinton, G. E. und McClelland, J. L. (1988). Learning representations by recirculation. In *NIPS'1987*, Seiten 358–366.
- Hinton, G. E. und Roweis, S. (2003). Stochastic neighbor embedding. In *NIPS'2002*.
- Hinton, G. E. und Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, **313**(5786), 504–507.
- Hinton, G. E. und Sejnowski, T. J. (1986). Learning and relearning in Boltzmann machines. In D. E. Rumelhart und J. L. McClelland, Hrsg., *Parallel Distributed Processing*, Band 1, Kapitel 7, Seiten 282–317. MIT Press, Cambridge.
- Hinton, G. E. und Sejnowski, T. J. (1999). *Unsupervised learning: foundations of neural computation*. MIT press.
- Hinton, G. E. und Shallice, T. (1991). Lesioning an attractor network: investigations of acquired dyslexia. *Psychological review*, **98**(1), 74.
- Hinton, G. E. und Zemel, R. S. (1994). Autoencoders, minimum description length, and Helmholtz free energy. In *NIPS'1993*.
- Hinton, G. E., Sejnowski, T. J., und Ackley, D. H. (1984). Boltzmann machines: Constraint satisfaction networks that learn. Technischer Bericht TR-CMU-CS-84-119, Carnegie-Mellon University, Dept. of Computer Science.
- Hinton, G. E., McClelland, J., und Rumelhart, D. (1986). Distributed representations. In D. E. Rumelhart und J. L. McClelland, Hrsg., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Band 1, Seiten 77–109. MIT Press, Cambridge.
- Hinton, G. E., Revow, M., und Dayan, P. (1995a). Recognizing handwritten digits using mixtures of linear models. In G. Tesauro, D. Touretzky, und T. Leen, Hrsg., *Advances in Neural Information Processing Systems 7 (NIPS'94)*, Seiten 1015–1022. MIT Press, Cambridge, MA.

- Hinton, G. E., Dayan, P., Frey, B. J., und Neal, R. M. (1995b). The wake-sleep algorithm for unsupervised neural networks. *Science*, **268**, 1558–1161.
- Hinton, G. E., Dayan, P., und Revow, M. (1997). Modelling the manifolds of images of handwritten digits. *IEEE Transactions on Neural Networks*, **8**, 65–74.
- Hinton, G. E., Welling, M., Teh, Y. W., und Osindero, S. (2001). A new view of ICA. In *Proceedings of 3rd International Conference on Independent Component Analysis and Blind Signal Separation (ICA '01)*, Seiten 746–751, San Diego, CA.
- Hinton, G. E., Osindero, S., und Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, **18**, 1527–1554.
- Hinton, G. E., Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., und Kingsbury, B. (2012b). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.*, **29**(6), 82–97.
- Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., und Salakhutdinov, R. (2012c). Improving neural networks by preventing co-adaptation of feature detectors. Technischer Bericht, arXiv:1207.0580.
- Hinton, G. E., Vinyals, O., und Dean, J. (2014). Dark knowledge. Invited Talk beim BayLearn Bay Area Machine Learning Symposium.
- Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diplomarbeit, TU München.
- Hochreiter, S. und Schmidhuber, J. (1995). Simplifying neural nets by discovering flat minima. In *Advances in Neural Information Processing Systems 7*, Seiten 529–536. MIT Press.
- Hochreiter, S. und Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, **9**(8), 1735–1780.
- Hochreiter, S., Bengio, Y., und Frasconi, P. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In J. Kolen und S. Kremer, Hrsg., *Field Guide to Dynamical Recurrent Networks*. IEEE Press.
- Holt, J. L. und Hwang, J.-N. (1993). Finite precision error analysis of neural network hardware implementations. *Computers, IEEE Transactions on*, **42**(3), 281–290.
- Hornik, K., Stinchcombe, M., und White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, **2**, 359–366.

- Hornik, K., Stinchcombe, M., und White, H. (1990). Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, **3**(5), 551–560.
- Hsu, F.-H. (2002). *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA.
- Huang, F. und Ogata, Y. (2002). Generalized pseudo-likelihood estimates for Markov random fields on lattice. *Annals of the Institute of Statistical Mathematics*, **54**(1), 1–18.
- Huang, P.-S., He, X., Gao, J., Deng, L., Acero, A., und Heck, L. (2013). Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, Seiten 2333–2338. ACM.
- Hubel, D. und Wiesel, T. (1968). Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, **195**, 215–243.
- Hubel, D. H. und Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, **148**, 574–591.
- Hubel, D. H. und Wiesel, T. N. (1962). Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology (London)*, **160**, 106–154.
- Huszar, F. (2015). How (not) to train your generative model: schedule sampling, likelihood, adversary? *arXiv:1511.05101*.
- Hutter, F., Hoos, H., und Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *LION-5*. Extended version as UBC Tech report TR-2010-10.
- Hyötyläinen, H. (1996). Turing machines are recurrent neural networks. In *StEP'96*, Seiten 13–24.
- Hyvärinen, A. (1999). Survey on independent component analysis. *Neural Computing Surveys*, **2**, 94–128.
- Hyvärinen, A. (2005). Estimation of non-normalized statistical models using score matching. *Journal of Machine Learning Research*, **6**, 695–709.
- Hyvärinen, A. (2007a). Connections between score matching, contrastive divergence, and pseudolikelihood for continuous-valued variables. *IEEE Transactions on Neural Networks*, **18**, 1529–1531.
- Hyvärinen, A. (2007b). Some extensions of score matching. *Computational Statistics and Data Analysis*, **51**, 2499–2512.

- Hyvärinen, A. und Hoyer, P. O. (1999). Emergence of topography and complex cell properties from natural images using extensions of ica. In *NIPS*, Seiten 827–833.
- Hyvärinen, A. und Pajunen, P. (1999). Nonlinear independent component analysis: Existence and uniqueness results. *Neural Networks*, **12**(3), 429–439.
- Hyvärinen, A., Karhunen, J., und Oja, E. (2001a). *Independent Component Analysis*. Wiley-Interscience.
- Hyvärinen, A., Hoyer, P. O., und Inki, M. O. (2001b). Topographic independent component analysis. *Neural Computation*, **13**(7), 1527–1558.
- Hyvärinen, A., Hurri, J., und Hoyer, P. O. (2009). *Natural Image Statistics: A probabilistic approach to early computational vision*. Springer-Verlag.
- Iba, Y. (2001). Extended ensemble Monte Carlo. *International Journal of Modern Physics*, **C12**, 623–656.
- Inayoshi, H. und Kurita, T. (2005). Improved generalization by adding both auto-association and hidden-layer noise to neural-network-based-classifiers. *IEEE Workshop on Machine Learning for Signal Processing*, Seiten 141–146.
- Ioffe, S. und Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.
- Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural networks*, **1**(4), 295–307.
- Jacobs, R. A., Jordan, M. I., Nowlan, S. J., und Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, **3**, 79–87.
- Jaeger, H. (2003). Adaptive nonlinear system identification with echo state networks. In *Advances in Neural Information Processing Systems 15*.
- Jaeger, H. (2007a). Discovering multiscale dynamical features with hierarchical echo state networks. Technischer Bericht, Jacobs University.
- Jaeger, H. (2007b). Echo state network. *Scholarpedia*, **2**(9), 2330.
- Jaeger, H. (2012). Long short-term memory in echo state networks: Details of a simulation study. Technischer Bericht, Technischer Bericht, Jacobs University Bremen.
- Jaeger, H. und Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, **304**(5667), 78–80.
- Jaeger, H., Lukosevicius, M., Popovici, D. und Siewert, U. (2007). Optimization and applications of echo state networks with leaky- integrator neurons. *Neural Networks*, **20**(3), 335–352.

- Jain, V., Murray, J. F., Roth, F., Turaga, S., Zhigulin, V., Briggman, K. L., Helmstaedter, M. N., Denk, W., und Seung, H. S. (2007). Supervised learning of image restoration with convolutional networks. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, Seiten 1–8. IEEE.
- Jaitly, N. und Hinton, G. (2011). Learning a better representation of speech soundwaves using restricted Boltzmann machines. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, Seiten 5884–5887. IEEE.
- Jaitly, N. und Hinton, G. E. (2013). Vocal tract length perturbation (VTLP) improves speech recognition. In *ICML'2013*.
- Jarrett, K., Kavukcuoglu, K., Ranzato, M., und LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *ICCV'09*.
- Jarzynski, C. (1997). Nonequilibrium equality for free energy differences. *Phys. Rev. Lett.*, **78**, 2690–2693.
- Jaynes, E. T. (2003). *Probability Theory: The Logic of Science*. Cambridge University Press.
- Jean, S., Cho, K., Memisevic, R., und Bengio, Y. (2014). On using very large target vocabulary for neural machine translation. arXiv:1412.2007.
- Jelinek, F. und Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In E. S. Gelsema und L. N. Kanal, Hrsg., *Pattern Recognition in Practice*. Nord, Amsterdam.
- Jia, Y. (2013). Caffe: An open source convolutional architecture for fast feature embedding. <http://caffe.berkeleyvision.org/>.
- Jia, Y., Huang, C., und Darrell, T. (2012). Beyond spatial pyramids: Receptive field learning for pooled image features. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, Seiten 3370–3377. IEEE.
- Jim, K.-C., Giles, C. L., und Horne, B. G. (1996). An analysis of noise in recurrent neural networks: convergence and generalization. *IEEE Transactions on Neural Networks*, **7**(6), 1424–1438.
- Jordan, M. I. (1998). *Learning in Graphical Models*. Kluwer, Dordrecht, Niederlande.
- Joulin, A. und Mikolov, T. (2015). Inferring algorithmic patterns with stack-augmented recurrent nets. *arXiv Vorabdruck arXiv:1503.01007*.
- Jozefowicz, R., Zaremba, W., und Sutskever, I. (2015). An empirical evaluation of recurrent network architectures. In *ICML'2015*.

- Judd, J. S. (1989). *Neural Network Design and the Complexity of Learning*. MIT press.
- Jutten, C. und Herault, J. (1991). Blind separation of sources, part i: an adaptive algorithm based on neuromimetic architecture. *Signal Processing*, **24**, 1–10.
- Kahou, S. E., Pal, C., Bouthillier, X., Froumenty, P., Gürçehre, c., Memisevic, R., Vincent, P., Courville, A., Bengio, Y., Ferrari, R. C., Mirza, M., Jean, S., Carrier, P. L., Dauphin, Y., Boulanger-Lewandowski, N., Aggarwal, A., Zumer, J., Lamblin, P., Raymond, J.-P., Desjardins, G., Pascanu, R., Warde-Farley, D., Torabi, A., Sharma, A., Bengio, E., Côté, M., Konda, K. R., und Wu, Z. (2013). Combining modality specific deep neural networks for emotion recognition in video. In *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*.
- Kalchbrenner, N. und Blunsom, P. (2013). Recurrent continuous translation models. In *EMNLP'2013*.
- Kalchbrenner, N., Danihelka, I., und Graves, A. (2015). Grid long short-term memory. *arXiv Vorabdruck arXiv:1507.01526*.
- Kamyshanska, H. und Memisevic, R. (2015). The potential energy of an autoencoder. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- Karpathy, A. und Li, F.-F. (2015). Deep visual-semantic alignments for generating image descriptions. In *CVPR'2015*. arXiv:1412.2306.
- Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., und Fei-Fei, L. (2014). Large-scale video classification with convolutional neural networks. In *CVPR*.
- Karush, W. (1939). *Minima of Functions of Several Variables with Inequalities as Side Constraints*. Masterarbeit, Dept. of Mathematics, Univ. of Chicago.
- Katz, S. M. (1987). Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **ASSP-35**(3), 400–401.
- Kavukcuoglu, K., Ranzato, M., und LeCun, Y. (2008). Fast inference in sparse coding algorithms with applications to object recognition. Technischer Bericht, Computational and Biological Learning Lab, Courant Institute, NYU. Technischer Bericht CBLL-TR-2008-12-01.
- Kavukcuoglu, K., Ranzato, M.-A., Fergus, R., und LeCun, Y. (2009). Learning invariant features through topographic filter maps. In *CVPR'2009*.
- Kavukcuoglu, K., Sermanet, P., Boureau, Y.-L., Gregor, K., Mathieu, M., und LeCun, Y. (2010). Learning convolutional feature hierarchies for visual recognition. In *NIPS'2010*.

- Kelley, H. J. (1960). Gradient theory of optimal flight paths. *ARS Journal*, **30**(10), 947–954.
- Khan, F., Zhu, X., und Mutlu, B. (2011). How do humans teach: On curriculum learning and teaching dimension. In *Advances in Neural Information Processing Systems 24 (NIPS'11)*, Seiten 1449–1457.
- Kim, S. K., McAfee, L. C., McMahon, P. L., und Olukotun, K. (2009). A highly scalable restricted Boltzmann machine FPGA implementation. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Seiten 367–372. IEEE.
- Kindermann, R. (1980). *Markov Random Fields and Their Applications (Contemporary Mathematics ; V. 1)*. American Mathematical Society.
- Kingma, D. und Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv Vorabdruck arXiv:1412.6980*.
- Kingma, D. und LeCun, Y. (2010). Regularized estimation of image statistics by score matching. In *NIPS'2010*.
- Kingma, D., Rezende, D., Mohamed, S., und Welling, M. (2014). Semi-supervised learning with deep generative models. In *NIPS'2014*.
- Kingma, D. P. (2013). Fast gradient-based inference with continuous latent variable models in auxiliary form. Technischer Bericht, arxiv:1306.0733.
- Kingma, D. P. und Welling, M. (2014a). Auto-encoding variational bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
- Kingma, D. P. und Welling, M. (2014b). Efficient gradient-based inference through transformations between bayes nets and neural nets. Technischer Bericht, arxiv:1402.0480.
- Kirkpatrick, S., Jr., C. D. G., und Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, **220**, 671–680.
- Kiros, R., Salakhutdinov, R., und Zemel, R. (2014a). Multimodal neural language models. In *ICML'2014*.
- Kiros, R., Salakhutdinov, R., und Zemel, R. (2014b). Unifying visual-semantic embeddings with multimodal neural language models. *arXiv:1411.2539 [cs.LG]*.
- Klementiev, A., Titov, I., und Bhattachari, B. (2012). Inducing crosslingual distributed representations of words. In *Proceedings of COLING 2012*.
- Knowles-Barley, S., Jones, T. R., Morgan, J., Lee, D., Kasthuri, N., Lichtman, J. W., und Pfister, H. (2014). Deep learning for the connectome. *GPU Technology Conference*.

- Koller, D. und Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
- Konig, Y., Bourlard, H., und Morgan, N. (1996). REMAP: Recursive estimation and maximization of a posteriori probabilities – application to transition-based connectionist speech recognition. In D. Touretzky, M. Mozer, und M. Hasselmo, Hrsg., *Advances in Neural Information Processing Systems 8 (NIPS'95)*. MIT Press, Cambridge, MA.
- Koren, Y. (2009). The BellKor solution to the Netflix grand prize.
- Kotzias, D., Denil, M., de Freitas, N., und Smyth, P. (2015). From group to individual labels using deep features. In *ACM SIGKDD*.
- Koutnik, J., Greff, K., Gomez, F., und Schmidhuber, J. (2014). A clockwork RNN. In *ICML'2014*.
- Kočiský, T., Hermann, K. M., und Blunsom, P. (2014). Learning Bilingual Word Representations by Marginalizing Alignments. In *Proceedings of ACL*.
- Krause, O., Fischer, A., Glasmachers, T., und Igel, C. (2013). Approximation properties of DBNs with binary hidden units and real-valued visible units. In *ICML'2013*.
- Krizhevsky, A. (2010). Convolutional deep belief networks on CIFAR-10. Technischer Bericht, University of Toronto. Unveröffentlichtes Manuskript: <http://www.cs.utoronto.ca/~kriz/conv-cifar10-aug2010.pdf>.
- Krizhevsky, A. und Hinton, G. (2009). Learning multiple layers of features from tiny images. Technischer Bericht, University of Toronto.
- Krizhevsky, A. und Hinton, G. E. (2011). Using very deep autoencoders for content-based image retrieval. In *ESANN*.
- Krizhevsky, A., Sutskever, I., und Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. In *NIPS'2012*.
- Krueger, K. A. und Dayan, P. (2009). Flexible shaping: how learning in small steps helps. *Cognition*, **110**, 380–394.
- Kuhn, H. W. und Tucker, A. W. (1951). Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, Seiten 481–492, Berkeley, Kalifornien. University of California Press.
- Kumar, A., Irsoy, O., Su, J., Bradbury, J., English, R., Pierce, B., Ondruska, P., Iyyer, M., Gulrajani, I., und Socher, R. (2015). Ask me anything: Dynamic memory networks for natural language processing. *arXiv:1506.07285*.
- Kumar, M. P., Packer, B., und Koller, D. (2010). Self-paced learning for latent variable models. In *NIPS'2010*.

- Lang, K. J. und Hinton, G. E. (1988). The development of the time-delay neural network architecture for speech recognition. Technischer Bericht CMU-CS-88-152, Carnegie-Mellon University.
- Lang, K. J., Waibel, A. H., und Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural networks*, **3**(1), 23–43.
- Langford, J. und Zhang, T. (2008). The epoch-greedy algorithm for contextual multi-armed bandits. In *NIPS'2008*, Seiten 1096–1103.
- Lappalainen, H., Giannakopoulos, X., Honkela, A., und Karhunen, J. (2000). Nonlinear independent component analysis using ensemble learning: Experiments and discussion. In *Proc. ICA*. Citeseer.
- Larochelle, H. und Bengio, Y. (2008). Classification using discriminative restricted Boltzmann machines. In *ICML'2008*.
- Larochelle, H. und Hinton, G. E. (2010). Learning to combine foveal glimpses with a third-order Boltzmann machine. In *Advances in Neural Information Processing Systems 23*, Seiten 1243–1251.
- Larochelle, H. und Murray, I. (2011). The Neural Autoregressive Distribution Estimator. In *AISTATS'2011*.
- Larochelle, H., Erhan, D., und Bengio, Y. (2008). Zero-data learning of new tasks. In *AAAI Conference on Artificial Intelligence*.
- Larochelle, H., Bengio, Y., Louradour, J., und Lamblin, P. (2009). Exploring strategies for training deep neural networks. *Journal of Machine Learning Research*, **10**, 1–40.
- Lasserre, J. A., Bishop, C. M., und Minka, T. P. (2006). Principled hybrids of generative and discriminative models. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'06)*, Seiten 87–94, Washington, DC, USA. IEEE Computer Society.
- Le, Q., Ngiam, J., Chen, Z., hao Chia, D. J., Koh, P. W., und Ng, A. (2010). Tiled convolutional neural networks. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. Zemel, und A. Culotta, Hrsg., *Advances in Neural Information Processing Systems 23 (NIPS'10)*, Seiten 1279–1287.
- Le, Q., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., und Ng, A. (2011). On optimization methods for deep learning. In *Proc. ICML'2011*. ACM.
- Le, Q., Ranzato, M., Monga, R., Devin, M., Corrado, G., Chen, K., Dean, J., und Ng, A. (2012). Building high-level features using large scale unsupervised learning. In *ICML'2012*.

- Le Roux, N. und Bengio, Y. (2008). Representational power of restricted Boltzmann machines and deep belief networks. *Neural Computation*, **20**(6), 1631–1649.
- Le Roux, N. und Bengio, Y. (2010). Deep belief networks are compact universal approximators. *Neural Computation*, **22**(8), 2192–2207.
- LeCun, Y. (1985). Une procédure d'apprentissage pour Réseau à seuil assymétrique. In *Cognitiva 85: A la Frontière de l'Intelligence Artificielle, des Sciences de la Connaissance et des Neurosciences*, Seiten 599–604, Paris 1985. CESTA, Paris.
- LeCun, Y. (1986). Learning processes in an asymmetric threshold network. In F. Fogelman-Soulie, E. Bienenstock, und G. Weisbuch, Hrsg., *Disordered Systems and Biological Organization*, Seiten 233–240. Springer-Verlag, Les Houches, Frankreich.
- LeCun, Y. (1987). *Modèles connexionnistes de l'apprentissage*. Doktorarbeit, Université de Paris VI.
- LeCun, Y. (1989). Generalization and network design strategies. Technischer Bericht CRG-TR-89-4, University of Toronto.
- LeCun, Y., Jackel, L. D., Boser, B., Denker, J. S., Graf, H. P., Guyon, I., Henderson, D., Howard, R. E., und Hubbard, W. (1989). Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, **27**(11), 41–46.
- LeCun, Y., Bottou, L., Orr, G. B., und Müller, K.-R. (1998a). Efficient backprop. In *Neural Networks, Tricks of the Trade*, Vorlesungsnotizen in Computer Science LNCS 1524. Springer Verlag.
- LeCun, Y., Bottou, L., Bengio, Y., und Haffner, P. (1998b). Gradient based learning applied to document recognition. *Proc. IEEE*.
- LeCun, Y., Kavukcuoglu, K., und Farabet, C. (2010). Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, Seiten 253–256. IEEE.
- L'Ecuyer, P. (1994). Efficiency improvement and variance reduction. In *Proceedings of the 1994 Winter Simulation Conference*, Seiten 122—132.
- Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z., und Tu, Z. (2014). Deeply-supervised nets. *arXiv Vorabdruck arXiv:1409.5185*.
- Lee, H., Battle, A., Raina, R., und Ng, A. (2007). Efficient sparse coding algorithms. In B. Schölkopf, J. Platt, und T. Hoffman, Hrsg., *Advances in Neural Information Processing Systems 19 (NIPS'06)*, Seiten 801–808. MIT Press.

- Lee, H., Ekanadham, C., und Ng, A. (2008). Sparse deep belief net model for visual area V2. In *NIPS'07*.
- Lee, H., Grosse, R., Ranganath, R., und Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In L. Bottou und M. Littman, Hrsg., *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*. ACM, Montreal, Kanada.
- Lee, Y. J. und Grauman, K. (2011). Learning the easy things first: self-paced visual category discovery. In *CVPR'2011*.
- Leibniz, G. W. (1676). Memoir using the chain rule. (Cited in TMME 7:2&3 S. 321–332, 2010).
- Lenat, D. B. und Guha, R. V. (1989). *Building large knowledge-based systems; representation and inference in the Cyc project*. Addison-Wesley Longman Publishing Co., Inc.
- Leshno, M., Lin, V. Y., Pinkus, A., und Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, **6**, 861–867.
- Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathematics*, **II**(2), 164–168.
- L'Hôpital, G. F. A. (1696). *Analyse des infiniment petits, pour l'intelligence des lignes courbes*. Paris: L'Imprimerie Royale.
- Li, Y., Swersky, K., und Zemel, R. S. (2015). Generative moment matching networks. *CoRR*, **abs/1502.02761**.
- Lin, T., Horne, B. G., Tino, P., und Giles, C. L. (1996). Learning long-term dependencies is not as difficult with NARX recurrent neural networks. *IEEE Transactions on Neural Networks*, **7**(6), 1329–1338.
- Lin, Y., Liu, Z., Sun, M., Liu, Y., und Zhu, X. (2015). Learning entity and relation embeddings for knowledge graph completion. In *Proc. AAAI'15*.
- Linde, N. (1992). The machine that changed the world, Folge 3. Mini-Doku.
- Lindsey, C. und Lindblad, T. (1994). Review of hardware neural networks: a user's perspective. In *Proc. Third Workshop on Neural Networks: From Biology to High Energy Physics*, Seiten 195–202, Isola d'Elba, Italien.
- Linnainmaa, S. (1976). Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, **16**(2), 146–160.
- LISA (2008). Deep learning tutorials: Restricted Boltzmann machines. Technischer Bericht, LISA Lab, Université de Montréal.

- Long, P. M. und Servedio, R. A. (2010). Restricted Boltzmann machines are hard to approximately evaluate or simulate. In *Proceedings of the 27th International Conference on Machine Learning (ICML'10)*.
- Lotter, W., Kreiman, G., und Cox, D. (2015). Unsupervised learning of visual structure using predictive generative networks. *arXiv Vorabdruck arXiv:1511.06380*.
- Lovelace, A. (1842). Notes upon L. F. Menabrea's "Sketch of the Analytical Engine invented by Charles Babbage".
- Lu, L., Zhang, X., Cho, K., und Renals, S. (2015). A study of the recurrent neural network encoder-decoder for large vocabulary speech recognition. In *Proc. Interspeech*.
- Lu, T., Pál, D., und Pál, M. (2010). Contextual multi-armed bandits. In *International Conference on Artificial Intelligence and Statistics*, Seiten 485–492.
- Luenberger, D. G. (1984). *Linear and Nonlinear Programming*. Addison Wesley.
- Lukoševičius, M. und Jaeger, H. (2009). Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3), 127–149.
- Luo, H., Shen, R., Niu, C., und Ullrich, C. (2011). Learning class-relevant features and class-irrelevant features via a hybrid third-order RBM. In *International Conference on Artificial Intelligence and Statistics*, Seiten 470–478.
- Luo, H., Carrier, P. L., Courville, A., und Bengio, Y. (2013). Texture modeling with convolutional spike-and-slab RBMs and deep extensions. In *AISTATS'2013*.
- Lyu, S. (2009). Interpretation and generalization of score matching. In *Proceedings of the Twenty-fifth Conference in Uncertainty in Artificial Intelligence (UAI'09)*.
- Ma, J., Sheridan, R. P., Liaw, A., Dahl, G. E., und Svetnik, V. (2015). Deep neural nets as a method for quantitative structure – activity relationships. *J. Chemical information and modeling*.
- Maas, A. L., Hannun, A. Y., und Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing*.
- Maass, W. (1992). Bounds for the computational power and learning complexity of analog neural nets (extended abstract). In *Proc. of the 25th ACM Symp. Theory of Computing*, Seiten 335–344.
- Maass, W., Schnitger, G., und Sontag, E. D. (1994). A comparison of the computational power of sigmoid and Boolean threshold circuits. *Theoretical Advances in Neural Computation and Learning*, Seiten 127–151.

- Maass, W., Natschlaeger, T., und Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, **14**(11), 2531–2560.
- MacKay, D. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press.
- Maclaurin, D., Duvenaud, D., und Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. *arXiv Vorabdruck arXiv:1502.03492*.
- Mao, J., Xu, W., Yang, Y., Wang, J., Huang, Z., und Yuille, A. L. (2015). Deep captioning with multimodal recurrent neural networks. In *ICLR'2015*. arXiv:1410.1090.
- Marcotte, P. und Savard, G. (1992). Novel approaches to the discrimination problem. *Zeitschrift für Operations Research (Theory)*, **36**, 517–545.
- Marlin, B. und de Freitas, N. (2011). Asymptotic efficiency of deterministic estimators for discrete energy-based models: Ratio matching and pseudolikelihood. In *UAI'2011*.
- Marlin, B., Swersky, K., Chen, B., und de Freitas, N. (2010). Inductive principles for restricted Boltzmann machine learning. In *Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10)*, Band 9, Seiten 509–516.
- Marquardt, D. W. (1963). An algorithm for least-squares estimation of non-linear parameters. *Journal of the Society of Industrial and Applied Mathematics*, **11**(2), 431–441.
- Marr, D. und Poggio, T. (1976). Cooperative computation of stereo disparity. *Science*, **194**.
- Martens, J. (2010). Deep learning via Hessian-free optimization. In L. Bottou und M. Littman, Hrsg., *Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML-10)*, Seiten 735–742. ACM.
- Martens, J. und Medabalimi, V. (2014). On the expressive efficiency of sum product networks. *arXiv:1411.7717*.
- Martens, J. und Sutskever, I. (2011). Learning recurrent neural networks with Hessian-free optimization. In *Proc. ICML'2011*. ACM.
- Mase, S. (1995). Consistency of the maximum pseudo-likelihood estimator of continuous state space Gibbsian processes. *The Annals of Applied Probability*, **5**(3), Seiten 603–612.

- McClelland, J., Rumelhart, D., und Hinton, G. (1995). The appeal of parallel distributed processing. In *Computation & intelligence*, Seiten 305–341. American Association for Artificial Intelligence.
- McCulloch, W. S. und Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, **5**, 115–133.
- Mead, C. und Ismail, M. (2012). *Analog VLSI implementation of neural systems*, Band 80. Springer Science & Business Media.
- Melchior, J., Fischer, A., und Wiskott, L. (2013). How to center binary deep Boltzmann machines. *arXiv Vorabdruck arXiv:1311.1354*.
- Memisevic, R. und Hinton, G. E. (2007). Unsupervised learning of image transformations. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'07)*.
- Memisevic, R. und Hinton, G. E. (2010). Learning to represent spatial transformations with factored higher-order Boltzmann machines. *Neural Computation*, **22**(6), 1473–1492.
- Mesnil, G., Dauphin, Y., Glorot, X., Rifai, S., Bengio, Y., Goodfellow, I., Lavoie, E., Muller, X., Desjardins, G., Warde-Farley, D., Vincent, P., Courville, A., und Bergstra, J. (2011). Unsupervised and transfer learning challenge: a deep learning approach. In *JMLR W&CP: Proc. Unsupervised and Transfer Learning*, Band 7.
- Mesnil, G., Rifai, S., Dauphin, Y., Bengio, Y., und Vincent, P. (2012). Surfing on the manifold. Learning Workshop, Snowbird.
- Miikkulainen, R. und Dyer, M. G. (1991). Natural language processing with modular PDP networks and distributed lexicon. *Cognitive Science*, **15**, 343–399.
- Mikolov, T. (2012). *Statistical Language Models based on Neural Networks*. Doktorarbeit, Brno University of Technology.
- Mikolov, T., Deoras, A., Kombrink, S., Burget, L., und Cernocky, J. (2011a). Empirical evaluation and combination of advanced language modeling techniques. In *Proc. 12th annual conference of the international speech communication association (INTERSPEECH 2011)*.
- Mikolov, T., Deoras, A., Povey, D., Burget, L., und Cernocky, J. (2011b). Strategies for training large scale neural network language models. In *Proc. ASRU'2011*.
- Mikolov, T., Chen, K., Corrado, G., und Dean, J. (2013a). Efficient estimation of word representations in vector space. In *International Conference on Learning Representations: Workshops Track*.

- Mikolov, T., Le, Q. V., und Sutskever, I. (2013b). Exploiting similarities among languages for machine translation. Technischer Bericht, arXiv:1309.4168.
- Minka, T. (2005). Divergence measures and message passing. *Microsoft Research Cambridge UK Tech Rep MSRTR2005173*, **72**(TR-2005-173).
- Minsky, M. L. und Papert, S. A. (1969). *Perceptrons*. MIT Press, Cambridge.
- Mirza, M. und Osindero, S. (2014). Conditional generative adversarial nets. *arXiv Vorabdruck arXiv:1411.1784*.
- Mishkin, D. und Matas, J. (2015). All you need is a good init. *arXiv Vorabdruck arXiv:1511.06422*.
- Misra, J. und Saha, I. (2010). Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, **74**(1), 239–255.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York.
- Miyato, T., Maeda, S., Koyama, M., Nakae, K., und Ishii, S. (2015). Distributional smoothing with virtual adversarial training. In *ICLR*. Vorabdruck: arXiv:1507.00677.
- Mnih, A. und Gregor, K. (2014). Neural variational inference and learning in belief networks. In *ICML'2014*.
- Mnih, A. und Hinton, G. E. (2007). Three new graphical models for statistical language modelling. In Z. Ghahramani, Hrsg., *Proceedings of the Twenty-fourth International Conference on Machine Learning (ICML'07)*, Seiten 641–648. ACM.
- Mnih, A. und Hinton, G. E. (2009). A scalable hierarchical distributed language model. In D. Koller, D. Schuurmans, Y. Bengio, und L. Bottou, Hrsg., *Advances in Neural Information Processing Systems 21 (NIPS'08)*, Seiten 1081–1088.
- Mnih, A. und Kavukcuoglu, K. (2013). Learning word embeddings efficiently with noise-contrastive estimation. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, und K. Weinberger, Hrsg., *Advances in Neural Information Processing Systems 26*, Seiten 2265–2273. Curran Associates, Inc.
- Mnih, A. und Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. In *ICML'2012*, Seiten 1751–1758.
- Mnih, V. und Hinton, G. (2010). Learning to detect roads in high-resolution aerial images. In *Proceedings of the 11th European Conference on Computer Vision (ECCV)*.
- Mnih, V., Larochelle, H., und Hinton, G. (2011). Conditional restricted Boltzmann machines for structure output prediction. In *Proc. Conf. on Uncertainty in Artificial Intelligence (UAI)*.

- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., und Wierstra, D. (2013). Playing Atari with deep reinforcement learning. Technischer Bericht, arXiv:1312.5602.
- Mnih, V., Heess, N., Graves, A., und Kavukcuoglu, K. (2014). Recurrent models of visual attention. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, und K. Weinberger, Hrsg., *NIPS'2014*, Seiten 2204–2212.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., und Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, **518**, 529–533.
- Mobahi, H. und Fisher, III, J. W. (2015). A theoretical analysis of optimization by Gaussian continuation. In *AAAI'2015*.
- Mobahi, H., Collobert, R., und Weston, J. (2009). Deep learning from temporal coherence in video. In L. Bottou und M. Littman, Hrsg., *Proceedings of the 26th International Conference on Machine Learning*, Seiten 737–744, Montreal. Omnipress.
- Mohamed, A., Dahl, G., und Hinton, G. (2009). Deep belief networks for phone recognition.
- Mohamed, A., Sainath, T. N., Dahl, G., Ramabhadran, B., Hinton, G. E., und Picheny, M. A. (2011). Deep belief networks using discriminative features for phone recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, Seiten 5060–5063. IEEE.
- Mohamed, A., Dahl, G., und Hinton, G. (2012a). Acoustic modeling using deep belief networks. *IEEE Trans. on Audio, Speech and Language Processing*, **20**(1), 14–22.
- Mohamed, A., Hinton, G., und Penn, G. (2012b). Understanding how deep belief networks perform acoustic modelling. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, Seiten 4273–4276. IEEE.
- Moller, M. F. (1993). A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, **6**, 525–533.
- Montavon, G. und Muller, K.-R. (2012). Deep Boltzmann machines and the centering trick. In G. Montavon, G. Orr, und K.-R. Müller, Hrsg., *Neural Networks: Tricks of the Trade*, Band 7700 der *Lecture Notes in Computer Science*, Seiten 621–637. Vorabdruck: <http://arxiv.org/abs/1203.3783>.
- Montúfar, G. (2014). Universal approximation depth and errors of narrow belief networks with discrete units. *Neural Computation*, **26**.

- Montúfar, G. und Ay, N. (2011). Refinements of universal approximation results for deep belief networks and restricted Boltzmann machines. *Neural Computation*, **23**(5), 1306–1319.
- Montufar, G. F., Pascanu, R., Cho, K., und Bengio, Y. (2014). On the number of linear regions of deep neural networks. In *NIPS'2014*.
- Mor-Yosef, S., Samueloff, A., Modan, B., Navot, D., und Schenker, J. G. (1990). Ranking the risk factors for cesarean: logistic regression analysis of a nationwide study. *Obstet Gynecol*, **75**(6), 944–7.
- Morin, F. und Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *AISTATS'2005*.
- Mozer, M. C. (1992). The induction of multiscale temporal structure. In J. M. S. Hanson und R. Lippmann, Hrsg., *Advances in Neural Information Processing Systems 4 (NIPS'91)*, Seiten 275–282, San Mateo, CA. Morgan Kaufmann.
- Murphy, K. P. (2012). *Machine Learning: a Probabilistic Perspective*. MIT Press, Cambridge, MA, USA.
- Murray, B. U. I. und Larochelle, H. (2014). A deep and tractable density estimator. In *ICML'2014*.
- Nair, V. und Hinton, G. (2010). Rectified linear units improve restricted Boltzmann machines. In *ICML'2010*.
- Nair, V. und Hinton, G. E. (2009). 3d object recognition with deep belief nets. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, und A. Culotta, Hrsg., *Advances in Neural Information Processing Systems 22*, Seiten 1339–1347. Curran Associates, Inc.
- Narayanan, H. und Mitter, S. (2010). Sample complexity of testing the manifold hypothesis. In *NIPS'2010*.
- Naumann, U. (2008). Optimal Jacobian accumulation is NP-complete. *Mathematical Programming*, **112**(2), 427–441.
- Navigli, R. und Velardi, P. (2005). Structural semantic interconnections: a knowledge-based approach to word sense disambiguation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, **27**(7), 1075–1086.
- Neal, R. und Hinton, G. (1999). A view of the EM algorithm that justifies incremental, sparse, and other variants. In M. I. Jordan, Hrsg., *Learning in Graphical Models*. MIT Press, Cambridge, MA.
- Neal, R. M. (1990). Learning stochastic feedforward networks. Technischer Bericht.
- Neal, R. M. (1993). Probabilistic inference using Markov chain Monte-Carlo methods. Technischer Bericht CRG-TR-93-1, Dept. of Computer Science, University of Toronto.

- Neal, R. M. (1994). Sampling from multimodal distributions using tempered transitions. Technischer Bericht 9421, Dept. of Statistics, University of Toronto.
- Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Lecture Notes in Statistics. Springer.
- Neal, R. M. (2001). Annealed importance sampling. *Statistics and Computing*, **11**(2), 125–139.
- Neal, R. M. (2005). Estimating ratios of normalizing constants using linked importance sampling.
- Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate $O(1/k^2)$. *Soviet Mathematics Doklady*, **27**, 372–376.
- Nesterov, Y. (2004). *Introductory lectures on convex optimization : a basic course*. Applied optimization. Kluwer Academic Publ., Boston, Dordrecht, London.
- Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., und Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. Deep Learning and Unsupervised Feature Learning Workshop, NIPS.
- Ney, H. und Kneser, R. (1993). Improved clustering techniques for class-based statistical language modelling. In *European Conference on Speech Communication and Technology (Eurospeech)*, Seiten 973–976, Berlin.
- Ng, A. (2015). Advice for applying machine learning. <https://see.stanford.edu/materials/aimlcs229/ML-advice.pdf>.
- Niesler, T. R., Whittaker, E. W. D., und Woodland, P. C. (1998). Comparison of part-of-speech and automatically derived category-based language models for speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Seiten 177–180.
- Ning, F., Delhomme, D., LeCun, Y., Piano, F., Bottou, L., und Barbano, P. E. (2005). Toward automatic phenotyping of developing embryos from videos. *Image Processing, IEEE Transactions on*, **14**(9), 1360–1371.
- Nocedal, J. und Wright, S. (2006). *Numerical Optimization*. Springer.
- Norouzi, M. und Fleet, D. J. (2011). Minimal loss hashing for compact binary codes. In *ICML'2011*.
- Nowlan, S. J. (1990). Competing experts: An experimental investigation of associative mixture models. Technischer Bericht CRG-TR-90-5, University of Toronto.
- Nowlan, S. J. und Hinton, G. E. (1992). Simplifying neural networks by soft weight-sharing. *Neural Computation*, **4**(4), 473–493.

- Olshausen, B. und Field, D. J. (2005). How close are we to understanding V1? *Neural Computation*, **17**, 1665–1699.
- Olshausen, B. A. und Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, **381**, 607–609.
- Olshausen, B. A., Anderson, C. H., und Van Essen, D. C. (1993). A neurobiological model of visual attention and invariant pattern recognition based on dynamic routing of information. *J. Neurosci.*, **13**(11), 4700–4719.
- Opper, M. und Archambeau, C. (2009). The variational Gaussian approximation revisited. *Neural computation*, **21**(3), 786–792.
- Quab, M., Bottou, L., Laptev, I., und Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, Seiten 1717–1724. IEEE.
- Osindero, S. und Hinton, G. E. (2008). Modeling image patches with a directed hierarchy of Markov random fields. In J. Platt, D. Koller, Y. Singer, und S. Roweis, Hrsg., *Advances in Neural Information Processing Systems 20 (NIPS'07)*, Seiten 1121–1128, Cambridge, MA. MIT Press.
- Ovid und Martin, C. (2004). *Metamorphoses*. W.W. Norton.
- Paccanaro, A. und Hinton, G. E. (2000). Extracting distributed representations of concepts and relations from positive and negative propositions. In *International Joint Conference on Neural Networks (IJCNN)*, Como, Italien. IEEE, New York.
- Paine, T. L., Khorrami, P., Han, W., und Huang, T. S. (2014). An analysis of unsupervised pre-training in light of recent advances. *arXiv Vorabdruck arXiv:1412.6597*.
- Palatucci, M., Pomerleau, D., Hinton, G. E., und Mitchell, T. M. (2009). Zero-shot learning with semantic output codes. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, und A. Culotta, Hrsg., *Advances in Neural Information Processing Systems 22*, Seiten 1410–1418. Curran Associates, Inc.
- Parker, D. B. (1985). Learning-logic. Technischer Bericht TR-47, Center for Comp. Research in Economics and Management Sci., MIT.
- Pascanu, R., Mikolov, T., und Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *ICML'2013*.
- Pascanu, R., Gülcühre, Ç., Cho, K., und Bengio, Y. (2014a). How to construct deep recurrent neural networks. In *ICLR'2014*.
- Pascanu, R., Montufar, G., und Bengio, Y. (2014b). On the number of inference regions of deep feed forward networks with piece-wise linear activations. In *ICLR'2014*.

- Pati, Y., Rezaiifar, R., und Krishnaprasad, P. (1993). Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proceedings of the 27 th Annual Asilomar Conference on Signals, Systems, and Computers*, Seiten 40–44.
- Pearl, J. (1985). Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine*, Seiten 329–334.
- Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.
- Perron, O. (1907). Zur theorie der matrices. *Mathematische Annalen*, **64**(2), 248–263.
- Petersen, K. B. und Pedersen, M. S. (2006). The matrix cookbook. Version 20051003.
- Peterson, G. B. (2004). A day of great illumination: B. F. Skinner's discovery of shaping. *Journal of the Experimental Analysis of Behavior*, **82**(3), 317–328.
- Pham, D.-T., Garat, P., und Jutten, C. (1992). Separation of a mixture of independent sources through a maximum likelihood approach. In *EUSIPCO*, Seiten 771–774.
- Pham, P.-H., Jelaca, D., Farabet, C., Martini, B., LeCun, Y., und Culurciello, E. (2012). NeuFlow: dataflow vision processing system-on-a-chip. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, Seiten 1044–1047. IEEE.
- Pinheiro, P. H. O. und Collobert, R. (2014). Recurrent convolutional neural networks for scene labeling. In *ICML'2014*.
- Pinheiro, P. H. O. und Collobert, R. (2015). From image-level to pixel-level labeling with convolutional networks. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Pinto, N., Cox, D. D., und DiCarlo, J. J. (2008). Why is real-world visual object recognition hard? *PLoS Comput Biol*, **4**.
- Pinto, N., Stone, Z., Zickler, T., und Cox, D. (2011). Scaling up biologically-inspired computer vision: A case study in unconstrained face recognition on facebook. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, Seiten 35–42. IEEE.
- Pollack, J. B. (1990). Recursive distributed representations. *Artificial Intelligence*, **46**(1), 77–105.
- Polyak, B. und Juditsky, A. (1992). Acceleration of stochastic approximation by averaging. *SIAM J. Control and Optimization*, **30**(4), 838–855.

- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5), 1–17.
- Poole, B., Sohl-Dickstein, J., und Ganguli, S. (2014). Analyzing noise in autoencoders and deep networks. *CoRR*, **abs/1406.1831**.
- Poon, H. und Domingos, P. (2011). Sum-product networks: A new deep architecture. In *Proceedings of the Twenty-seventh Conference in Uncertainty in Artificial Intelligence (UAI)*, Barcelona, Spanien.
- Presley, R. K. und Haggard, R. L. (1994). A fixed point implementation of the backpropagation learning algorithm. In *Southeastcon'94. Creative Technology Transfer-A Global Affair., Proceedings of the 1994 IEEE*, Seiten 136–138. IEEE.
- Price, R. (1958). A useful theorem for nonlinear devices having Gaussian inputs. *IEEE Transactions on Information Theory*, 4(2), 69–72.
- Quiroga, R. Q., Reddy, L., Kreiman, G., Koch, C., und Fried, I. (2005). Invariant visual representation by single neurons in the human brain. *Nature*, 435(7045), 1102–1107.
- Radford, A., Metz, L., und Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv Vorabdruck arXiv:1511.06434*.
- Raiko, T., Yao, L., Cho, K., und Bengio, Y. (2014). Iterative neural autoregressive distribution estimator (NADE-k). Technischer Bericht, arXiv:1406.1485.
- Raina, R., Madhavan, A., und Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In L. Bottou und M. Littman, Hrsg., *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*, Seiten 873–880, New York, NY, USA. ACM.
- Ramsey, F. P. (1926). Truth and probability. In R. B. Braithwaite, Hrsg., *The Foundations of Mathematics and other Logical Essays*, Kapitel 7, Seiten 156–198. McMaster University Archive for the History of Economic Thought.
- Ranzato, M. und Hinton, G. H. (2010). Modeling pixel means and covariances using factorized third-order Boltzmann machines. In *CVPR'2010*, Seiten 2551–2558.
- Ranzato, M., Poultney, C., Chopra, S., und LeCun, Y. (2007a). Efficient learning of sparse representations with an energy-based model. In *NIPS'2006*.
- Ranzato, M., Huang, F., Boureau, Y., und LeCun, Y. (2007b). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'07)*. IEEE Press.

- Ranzato, M., Boureau, Y., und LeCun, Y. (2008). Sparse feature learning for deep belief networks. In *NIPS'2007*.
- Ranzato, M., Krizhevsky, A., und Hinton, G. E. (2010a). Factored 3-way restricted Boltzmann machines for modeling natural images. In *Proceedings of AISTATS 2010*.
- Ranzato, M., Mnih, V., und Hinton, G. (2010b). Generating more realistic images using gated MRFs. In *NIPS'2010*.
- Rao, C. (1945). Information and the accuracy attainable in the estimation of statistical parameters. *Bulletin of the Calcutta Mathematical Society*, **37**, 81–89.
- Rasmus, A., Valpola, H., Honkala, M., Berglund, M., und Raiko, T. (2015). Semi-supervised learning with ladder network. *arXiv Vorabdruck arXiv:1507.02672*.
- Recht, B., Re, C., Wright, S., und Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS'2011*.
- Reichert, D. P., Series, P., und Storkey, A. J. (2011). Neuronal adaptation for sampling-based probabilistic inference in perceptual bistability. In *Advances in Neural Information Processing Systems*, Seiten 2357–2365.
- Rezende, D. J., Mohamed, S., und Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *ICML'2014*. Vorabdruck: arXiv:1401.4082.
- Rifai, S., Vincent, P., Muller, X., Glorot, X., und Bengio, Y. (2011a). Contractive auto-encoders: Explicit invariance during feature extraction. In *ICML'2011*.
- Rifai, S., Mesnil, G., Vincent, P., Muller, X., Bengio, Y., Dauphin, Y., und Glorot, X. (2011b). Higher order contractive auto-encoder. In *ECML PKDD*.
- Rifai, S., Dauphin, Y., Vincent, P., Bengio, Y., und Muller, X. (2011c). The manifold tangent classifier. In *NIPS'2011*.
- Rifai, S., Bengio, Y., Dauphin, Y., und Vincent, P. (2012). A generative process for sampling contractive auto-encoders. In *ICML'2012*.
- Ringach, D. und Shapley, R. (2004). Reverse correlation in neurophysiology. *Cognitive Science*, **28**(2), 147–166.
- Roberts, S. und Everson, R. (2001). *Independent component analysis: principles and practice*. Cambridge University Press.
- Robinson, A. J. und Fallside, F. (1991). A recurrent error propagation network speech recognition system. *Computer Speech and Language*, **5**(3), 259–274.
- Rockafellar, R. T. (1997). Convex analysis. princeton landmarks in mathematics.

- Romero, A., Ballas, N., Ebrahimi Kahou, S., Chassang, A., Gatta, C., und Bengio, Y. (2015). Fitnets: Hints for thin deep nets. In *ICLR'2015, arXiv:1412.6550*.
- Rosen, J. B. (1960). The gradient projection method for nonlinear programming. part i. linear constraints. *Journal of the Society for Industrial and Applied Mathematics*, **8**(1), Seiten 181–217.
- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, **65**, 386–408.
- Rosenblatt, F. (1962). *Principles of Neurodynamics*. Spartan, New York.
- Roweis, S. und Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, **290**(5500).
- Roweis, S., Saul, L., und Hinton, G. (2002). Global coordination of local linear models. In T. Dietterich, S. Becker, und Z. Ghahramani, Hrsg., *Advances in Neural Information Processing Systems 14 (NIPS'01)*, Cambridge, MA. MIT Press.
- Rubin, D. B. et al. (1984). Bayesianly justifiable and relevant frequency calculations for the applied statistician. *The Annals of Statistics*, **12**(4), 1151–1172.
- Rumelhart, D., Hinton, G., und Williams, R. (1986a). Learning representations by back-propagating errors. *Nature*, **323**, 533–536.
- Rumelhart, D. E., Hinton, G. E., und Williams, R. J. (1986b). Learning internal representations by error propagation. In D. E. Rumelhart und J. L. McClelland, Hrsg., *Parallel Distributed Processing*, Band 1, Kapitel 8, Seiten 318–362. MIT Press, Cambridge.
- Rumelhart, D. E., McClelland, J. L., und die PDP Research Group (1986c). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, Cambridge.
- Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., und Fei-Fei, L. (2014a). ImageNet Large Scale Visual Recognition Challenge.
- Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2014b). Imagenet large scale visual recognition challenge. *arXiv Vorabdruck arXiv:1409.0575*.
- Russel, S. J. und Norvig, P. (2003). *Artificial Intelligence: a Modern Approach*. Prentice Hall.
- Rust, N., Schwartz, O., Movshon, J. A., und Simoncelli, E. (2005). Spatiotemporal elements of macaque V1 receptive fields. *Neuron*, **46**(6), 945–956.

- Sainath, T., Mohamed, A., Kingsbury, B., und Ramabhadran, B. (2013). Deep convolutional neural networks for LVCSR. In *ICASSP 2013*.
- Salakhutdinov, R. (2010). Learning in Markov random fields using tempered transitions. In Y. Bengio, D. Schuurmans, C. Williams, J. Lafferty, und A. Culotta, Hrsg., *Advances in Neural Information Processing Systems 22 (NIPS'09)*.
- Salakhutdinov, R. und Hinton, G. (2009a). Deep Boltzmann machines. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, Band 5, Seiten 448–455.
- Salakhutdinov, R. und Hinton, G. (2009b). Semantic hashing. In *International Journal of Approximate Reasoning*.
- Salakhutdinov, R. und Hinton, G. E. (2007a). Learning a nonlinear embedding by preserving class neighbourhood structure. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS'07)*, San Juan, Porto Rico. Omnipress.
- Salakhutdinov, R. und Hinton, G. E. (2007b). Semantic hashing. In *SIGIR'2007*.
- Salakhutdinov, R. und Hinton, G. E. (2008). Using deep belief nets to learn covariance kernels for Gaussian processes. In J. Platt, D. Koller, Y. Singer, und S. Roweis, Hrsg., *Advances in Neural Information Processing Systems 20 (NIPS'07)*, Seiten 1249–1256, Cambridge, MA. MIT Press.
- Salakhutdinov, R. und Larochelle, H. (2010). Efficient learning of deep Boltzmann machines. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, JMLR W&CP, Band 9, Seiten 693–700.
- Salakhutdinov, R. und Mnih, A. (2008). Probabilistic matrix factorization. In *NIPS'2008*.
- Salakhutdinov, R. und Murray, I. (2008). On the quantitative analysis of deep belief networks. In W. W. Cohen, A. McCallum, und S. T. Roweis, Hrsg., *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, Band 25, Seiten 872–879. ACM.
- Salakhutdinov, R., Mnih, A., und Hinton, G. (2007). Restricted Boltzmann machines for collaborative filtering. In *ICML*.
- Sanger, T. D. (1994). Neural network learning control of robot manipulators using gradually increasing task difficulty. *IEEE Transactions on Robotics and Automation*, **10**(3).

- Saul, L. K. und Jordan, M. I. (1996). Exploiting tractable substructures in intractable networks. In D. Touretzky, M. Mozer, und M. Hasselmo, Hrsg., *Advances in Neural Information Processing Systems 8 (NIPS'95)*. MIT Press, Cambridge, MA.
- Saul, L. K., Jaakkola, T., und Jordan, M. I. (1996). Mean field theory for sigmoid belief networks. *Journal of Artificial Intelligence Research*, **4**, 61–76.
- Savich, A. W., Moussa, M., und Areibi, S. (2007). The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. *Neural Networks, IEEE Transactions on*, **18**(1), 240–252.
- Saxe, A. M., Koh, P. W., Chen, Z., Bhand, M., Suresh, B., und Ng, A. (2011). On random weights and unsupervised feature learning. In *Proc. ICML'2011*. ACM.
- Saxe, A. M., McClelland, J. L., und Ganguli, S. (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *ICLR*.
- Schaul, T., Antonoglou, I., und Silver, D. (2014). Unit tests for stochastic optimization. In *International Conference on Learning Representations*.
- Schmidhuber, J. (1992). Learning complex, extended sequences using the principle of history compression. *Neural Computation*, **4**(2), 234–242.
- Schmidhuber, J. (1996). Sequential neural text compression. *IEEE Transactions on Neural Networks*, **7**(1), 142–146.
- Schmidhuber, J. (2012). Self-delimiting neural networks. *arXiv Vorabdruck arXiv:1210.0118*.
- Schölkopf, B. und Smola, A. J. (2002). *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT Press.
- Schölkopf, B., Smola, A., und Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, **10**, 1299–1319.
- Schölkopf, B., Burges, C. J. C., und Smola, A. J. (1999). *Advances in Kernel Methods — Support Vector Learning*. MIT Press, Cambridge, MA.
- Schölkopf, B., Janzing, D., Peters, J., Sgouritsa, E., Zhang, K., und Mooij, J. (2012). On causal and anticausal learning. In *ICML'2012*, Seiten 1255–1262.
- Schuster, M. (1999). On supervised learning from sequential data with applications for speech recognition.
- Schuster, M. und Paliwal, K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, **45**(11), 2673–2681.
- Schwenk, H. (2007). Continuous space language models. *Computer speech and language*, **21**, 492–518.

- Schwenk, H. (2010). Continuous space language models for statistical machine translation. *The Prague Bulletin of Mathematical Linguistics*, **93**, 137–146.
- Schwenk, H. (2014). Cleaned subset of WMT '14 dataset.
- Schwenk, H. und Bengio, Y. (1998). Training methods for adaptive boosting of neural networks. In M. Jordan, M. Kearns, und S. Solla, Hrsg., *Advances in Neural Information Processing Systems 10 (NIPS'97)*, Seiten 647–653. MIT Press.
- Schwenk, H. und Gauvain, J.-L. (2002). Connectionist language modeling for large vocabulary continuous speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Seiten 765–768, Orlando, Florida.
- Schwenk, H., Costa-jussà, M. R., und Fonollosa, J. A. R. (2006). Continuous space language models for the IWSLT 2006 task. In *International Workshop on Spoken Language Translation*, Seiten 166–173.
- Seide, F., Li, G., und Yu, D. (2011). Conversational speech transcription using context-dependent deep neural networks. In *Interspeech 2011*, Seiten 437–440.
- Sejnowski, T. (1987). Higher-order Boltzmann machines. In *AIP Conference Proceedings 151 on Neural Networks for Computing*, Seiten 398–403. American Institute of Physics Inc.
- Series, P., Reichert, D. P., und Storkey, A. J. (2010). Hallucinations in Charles Bonnet syndrome induced by homeostasis: a deep Boltzmann machine model. In *Advances in Neural Information Processing Systems*, Seiten 2020–2028.
- Sermanet, P., Chintala, S., und LeCun, Y. (2012). Convolutional neural networks applied to house numbers digit classification. *CoRR*, **abs/1204.3968**.
- Sermanet, P., Kavukcuoglu, K., Chintala, S., und LeCun, Y. (2013). Pedestrian detection with unsupervised multi-stage feature learning. In *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR'13)*. IEEE.
- Shilov, G. (1977). *Linear Algebra*. Dover Books on Mathematics Series. Dover Publications.
- Siegelmann, H. (1995). Computation beyond the Turing limit. *Science*, **268**(5210), 545–548.
- Siegelmann, H. und Sontag, E. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, **4**(6), 77–80.
- Siegelmann, H. T. und Sontag, E. D. (1995). On the computational power of neural nets. *Journal of Computer and Systems Sciences*, **50**(1), 132–150.

- Sietsma, J. und Dow, R. (1991). Creating artificial neural networks that generalize. *Neural Networks*, **4**(1), 67–79.
- Simard, D., Steinkraus, P. Y., und Platt, J. C. (2003). Best practices for convolutional neural networks. In *ICDAR'2003*.
- Simard, P. und Graf, H. P. (1994). Backpropagation without multiplication. In *Advances in Neural Information Processing Systems*, Seiten 232–239.
- Simard, P., Victorri, B., LeCun, Y., und Denker, J. (1992). Tangent prop - A formalism for specifying selected invariances in an adaptive network. In *NIPS'1991*.
- Simard, P. Y., LeCun, Y., und Denker, J. (1993). Efficient pattern recognition using a new transformation distance. In *NIPS'92*.
- Simard, P. Y., LeCun, Y. A., Denker, J. S., und Victorri, B. (1998). Transformation invariance in pattern recognition — tangent distance and tangent propagation. *Lecture Notes in Computer Science*, **1524**.
- Simons, D. J. und Levin, D. T. (1998). Failure to detect changes to people during a real-world interaction. *Psychonomic Bulletin & Review*, **5**(4), 644–649.
- Simonyan, K. und Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *ICLR*.
- Sjöberg, J. und Ljung, L. (1995). Overtraining, regularization and searching for a minimum, with application to neural networks. *International Journal of Control*, **62**(6), 1391–1407.
- Skinner, B. F. (1958). Reinforcement today. *American Psychologist*, **13**, 94–99.
- Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In D. E. Rumelhart und J. L. McClelland, Hrsg., *Parallel Distributed Processing*, Band 1, Kapitel 6, Seiten 194–281. MIT Press, Cambridge.
- Snoek, J., Larochelle, H., und Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *NIPS'2012*.
- Socher, R., Huang, E. H., Pennington, J., Ng, A. Y., und Manning, C. D. (2011a). Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *NIPS'2011*.
- Socher, R., Manning, C., und Ng, A. Y. (2011b). Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the Twenty-Eighth International Conference on Machine Learning (ICML'2011)*.
- Socher, R., Pennington, J., Huang, E. H., Ng, A. Y., und Manning, C. D. (2011c). Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP'2011*.

- Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., und Potts, C. (2013a). Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP'2013*.
- Socher, R., Ganjoo, M., Manning, C. D., und Ng, A. Y. (2013b). Zero-shot learning through cross-modal transfer. In *27th Annual Conference on Neural Information Processing Systems (NIPS 2013)*.
- Sohl-Dickstein, J., Weiss, E. A., Maheswaranathan, N., und Ganguli, S. (2015). Deep unsupervised learning using nonequilibrium thermodynamics.
- Sohn, K., Zhou, G., und Lee, H. (2013). Learning and selecting features jointly with point-wise gated Boltzmann machines. In *ICML'2013*.
- Solomonoff, R. J. (1989). A system for incremental learning based on algorithmic probability.
- Sontag, E. D. (1998). VC dimension of neural networks. *NATO ASI Series F Computer and Systems Sciences*, **168**, 69–96.
- Sontag, E. D. und Sussman, H. J. (1989). Backpropagation can give rise to spurious local minima even for networks without hidden layers. *Complex Systems*, **3**, 91–106.
- Sparkes, B. (1996). *The Red and the Black: Studies in Greek Pottery*. Routledge.
- Spitkovsky, V. I., Alshawi, H., und Jurafsky, D. (2010). From baby steps to leapfrog: how “less is more” in unsupervised dependency parsing. In *HLT'10*.
- Squire, W. und Trapp, G. (1998). Using complex variables to estimate derivatives of real functions. *SIAM Rev.*, **40**(1), 110—112.
- Srebro, N. und Shraibman, A. (2005). Rank, trace-norm and max-norm. In *Proceedings of the 18th Annual Conference on Learning Theory*, Seiten 545–560. Springer-Verlag.
- Srivastava, N. (2013). *Improving Neural Networks With Dropout*. Masterarbeit, U. Toronto.
- Srivastava, N. und Salakhutdinov, R. (2012). Multimodal learning with deep Boltzmann machines. In *NIPS'2012*.
- Srivastava, N., Salakhutdinov, R. R., und Hinton, G. E. (2013). Modeling documents with deep Boltzmann machines. *arXiv Vorabdruck arXiv:1309.6865*.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., und Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, **15**, 1929–1958.

- Srivastava, R. K., Greff, K., und Schmidhuber, J. (2015). Highway networks. *arXiv:1505.00387*.
- Steinkrau, D., Simard, P. Y., und Buck, I. (2005). Using GPUs for machine learning algorithms. *2013 12th International Conference on Document Analysis and Recognition*, **0**, 1115–1119.
- Stoyanov, V., Ropson, A., und Eisner, J. (2011). Empirical risk minimization of graphical model parameters given approximate inference, decoding, and model structure. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, Band 15 der *JMLR Workshop and Conference Proceedings*, Seiten 725–733, Fort Lauderdale. Zusätzlich ergänzende Unterlagen (4 Seiten) verfügbar.
- Sukhbaatar, S., Szlam, A., Weston, J., und Fergus, R. (2015). Weakly supervised memory networks. *arXiv Vorabdruck arXiv:1503.08895*.
- Supancic, J. und Ramanan, D. (2013). Self-paced learning for long-term tracking. In *CVPR'2013*.
- Sussillo, D. (2014). Random walks: Training very deep nonlinear feed-forward networks with smart initialization. *CoRR, abs/1412.6558*.
- Sutskever, I. (2012). *Training Recurrent Neural Networks*. Doktorarbeit, Department of computer science, University of Toronto.
- Sutskever, I. und Hinton, G. E. (2008). Deep narrow sigmoid belief networks are universal approximators. *Neural Computation*, **20**(11), 2629–2636.
- Sutskever, I. und Tieleman, T. (2010). On the Convergence Properties of Contrastive Divergence. In Y. W. Teh und M. Titterington, Hrsg., *Proc. of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, Band 9, Seiten 789–795.
- Sutskever, I., Hinton, G., und Taylor, G. (2009). The recurrent temporal restricted Boltzmann machine. In *NIPS'2008*.
- Sutskever, I., Martens, J., und Hinton, G. E. (2011). Generating text with recurrent neural networks. In *ICML'2011*, Seiten 1017–1024.
- Sutskever, I., Martens, J., Dahl, G., und Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *ICML*.
- Sutskever, I., Vinyals, O., und Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *NIPS'2014*, *arXiv:1409.3215*.
- Sutton, R. und Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

- Sutton, R. S., Mcallester, D., Singh, S., und Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *NIPS'1999*, Seiten 1057–1063. MIT Press.
- Swersky, K., Ranzato, M., Buchman, D., Marlin, B., und de Freitas, N. (2011). On autoencoders and score matching for energy based models. In *ICML'2011*. ACM.
- Swersky, K., Snoek, J., und Adams, R. P. (2014). Freeze-thaw Bayesian optimization. *arXiv Vorabdruck arXiv:1406.3896*.
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., und Rabinovich, A. (2014a). Going deeper with convolutions. Technischer Bericht, arXiv:1409.4842.
- Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., und Fergus, R. (2014b). Intriguing properties of neural networks. *ICLR, abs/1312.6199*.
- Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., und Wojna, Z. (2015). Rethinking the Inception Architecture for Computer Vision. *ArXiv e-prints*.
- Taigman, Y., Yang, M., Ranzato, M., und Wolf, L. (2014). DeepFace: Closing the gap to human-level performance in face verification. In *CVPR'2014*.
- Tandy, D. W. (1997). *Works and Days: A Translation and Commentary for the Social Sciences*. University of California Press.
- Tang, Y. und Eliasmith, C. (2010). Deep networks for robust visual recognition. In *Proceedings of the 27th International Conference on Machine Learning, 21-24. Juni 2010, Haifa, Israel*.
- Tang, Y., Salakhutdinov, R., und Hinton, G. (2012). Deep mixtures of factor analysers. *arXiv Vorabdruck arXiv:1206.4635*.
- Taylor, G. und Hinton, G. (2009). Factored conditional restricted Boltzmann machines for modeling motion style. In L. Bottou und M. Littman, Hrsg., *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*, Seiten 1025–1032, Montreal, Quebec, Kanada. ACM.
- Taylor, G., Hinton, G. E., und Roweis, S. (2007). Modeling human motion using binary latent variables. In B. Schölkopf, J. Platt, und T. Hoffman, Hrsg., *Advances in Neural Information Processing Systems 19 (NIPS'06)*, Seiten 1345–1352. MIT Press, Cambridge, MA.
- Teh, Y., Welling, M., Osindero, S., und Hinton, G. E. (2003). Energy-based models for sparse overcomplete representations. *Journal of Machine Learning Research*, 4, 1235–1260.

- Tenenbaum, J., de Silva, V., und Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, **290**(5500), 2319–2323.
- Theis, L., van den Oord, A., und Bethge, M. (2015). A note on the evaluation of generative models. arXiv:1511.01844.
- Thompson, J., Jain, A., LeCun, Y., und Bregler, C. (2014). Joint training of a convolutional network and a graphical model for human pose estimation. In *NIPS'2014*.
- Thrun, S. (1995). Learning to play the game of chess. In *NIPS'1994*.
- Tibshirani, R. J. (1995). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society B*, **58**, 267–288.
- Tieleman, T. (2008). Training restricted Boltzmann machines using approximations to the likelihood gradient. In W. W. Cohen, A. McCallum, und S. T. Roweis, Hrsg., *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, Seiten 1064–1071. ACM.
- Tieleman, T. und Hinton, G. (2009). Using fast weights to improve persistent contrastive divergence. In L. Bottou und M. Littman, Hrsg., *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*, Seiten 1033–1040. ACM.
- Tipping, M. E. und Bishop, C. M. (1999). Probabilistic principal components analysis. *Journal of the Royal Statistical Society B*, **61**(3), 611–622.
- Torralba, A., Fergus, R., und Weiss, Y. (2008). Small codes and large databases for recognition. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR'08)*, Seiten 1–8.
- Touretzky, D. S. und Minton, G. E. (1985). Symbols among the neurons: Details of a connectionist inference architecture. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'85, Seiten 238–243, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Töscher, A., Jahrer, M., und Bell, R. M. (2009). The BigChaos solution to the Netflix grand prize.
- Tu, K. und Honavar, V. (2011). On the utility of curricula in unsupervised learning of probabilistic grammars. In *IJCAI'2011*.
- Turaga, S. C., Murray, J. F., Jain, V., Roth, F., Helmstaedter, M., Briggman, K., Denk, W., und Seung, H. S. (2010). Convolutional networks can learn to generate affinity graphs for image segmentation. *Neural Computation*, **22**(2), 511–538.

- Turian, J., Ratinov, L., und Bengio, Y. (2010). Word representations: A simple and general method for semi-supervised learning. In *Proc. ACL'2010*, Seiten 384–394.
- Uria, B., Murray, I., und Larochelle, H. (2013). Rnade: The real-valued neural autoregressive density-estimator. In *NIPS'2013*.
- van den Oörd, A., Dieleman, S., und Schrauwen, B. (2013). Deep content-based music recommendation. In *NIPS'2013*.
- van der Maaten, L. und Hinton, G. E. (2008). Visualizing data using t-SNE. *J. Machine Learning Res.*, **9**.
- Vanhoucke, V., Senior, A., und Mao, M. Z. (2011). Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*.
- Vapnik, V. N. (1982). *Estimation of Dependences Based on Empirical Data*. Springer-Verlag, Berlin.
- Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer, New York.
- Vapnik, V. N. und Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, **16**, 264–280.
- Vincent, P. (2011). A connection between score matching and denoising autoencoders. *Neural Computation*, **23**(7).
- Vincent, P. und Bengio, Y. (2003). Manifold Parzen windows. In *NIPS'2002*. MIT Press.
- Vincent, P., Larochelle, H., Bengio, Y., und Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *ICML 2008*.
- Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., und Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *J. Machine Learning Res.*, **11**.
- Vincent, P., de Brébisson, A., und Bouthillier, X. (2015). Efficient exact gradient update for training deep networks with very large sparse targets. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, und R. Garnett, Hrsg., *Advances in Neural Information Processing Systems 28*, Seiten 1108–1116. Curran Associates, Inc.
- Vinyals, O., Kaiser, L., Koo, T., Petrov, S., Sutskever, I., und Hinton, G. (2014a). Grammar as a foreign language. Technischer Bericht, arXiv:1412.7449.

- Vinyals, O., Toshev, A., Bengio, S., und Erhan, D. (2014b). Show and tell: a neural image caption generator. arXiv 1411.4555.
- Vinyals, O., Fortunato, M., und Jaitly, N. (2015a). Pointer networks. *arXiv Vorabdruck arXiv:1506.03134*.
- Vinyals, O., Toshev, A., Bengio, S., und Erhan, D. (2015b). Show and tell: a neural image caption generator. In *CVPR'2015*. arXiv:1411.4555.
- Viola, P. und Jones, M. (2001). Robust real-time object detection. In *International Journal of Computer Vision*.
- Visin, F., Kastner, K., Cho, K., Matteucci, M., Courville, A., und Bengio, Y. (2015). ReNet: A recurrent neural network based alternative to convolutional networks. *arXiv Vorabdruck arXiv:1505.00393*.
- Von Melchner, L., Pallas, S. L., und Sur, M. (2000). Visual behaviour mediated by retinal projections directed to the auditory pathway. *Nature*, **404**(6780), 871–876.
- Wager, S., Wang, S., und Liang, P. (2013). Dropout training as adaptive regularization. In *Advances in Neural Information Processing Systems 26*, Seiten 351–359.
- Waibel, A., Hanazawa, T., Hinton, G. E., Shikano, K., und Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **37**, 328–339.
- Wan, L., Zeiler, M., Zhang, S., LeCun, Y., und Fergus, R. (2013). Regularization of neural networks using dropconnect. In *ICML'2013*.
- Wang, S. und Manning, C. (2013). Fast dropout training. In *ICML'2013*.
- Wang, Z., Zhang, J., Feng, J., und Chen, Z. (2014a). Knowledge graph and text jointly embedding. In *Proc. EMNLP'2014*.
- Wang, Z., Zhang, J., Feng, J., und Chen, Z. (2014b). Knowledge graph embedding by translating on hyperplanes. In *Proc. AAAI'2014*.
- Warde-Farley, D., Goodfellow, I. J., Courville, A., und Bengio, Y. (2014). An empirical analysis of dropout in piecewise linear networks. In *ICLR'2014*.
- Wawrynek, J., Asanovic, K., Kingsbury, B., Johnson, D., Beck, J., und Morgan, N. (1996). Spert-II: A vector microprocessor system. *Computer*, **29**(3), 79–86.
- Weaver, L. und Tao, N. (2001). The optimal reward baseline for gradient-based reinforcement learning. In *Proc. UAI'2001*, Seiten 538–545.
- Weinberger, K. Q. und Saul, L. K. (2004). Unsupervised learning of image manifolds by semidefinite programming. In *CVPR'2004*, Seiten 988–995.

- Weiss, Y., Torralba, A., und Fergus, R. (2008). Spectral hashing. In *NIPS*, Seiten 1753–1760.
- Welling, M., Zemel, R. S., und Hinton, G. E. (2002). Self supervised boosting. In *Advances in Neural Information Processing Systems*, Seiten 665–672.
- Welling, M., Hinton, G. E., und Osindero, S. (2003a). Learning sparse topographic representations with products of Student t-distributions. In *NIPS'2002*.
- Welling, M., Zemel, R., und Hinton, G. E. (2003b). Self-supervised boosting. In S. Becker, S. Thrun, und K. Obermayer, Hrsg., *Advances in Neural Information Processing Systems 15 (NIPS'02)*, Seiten 665–672. MIT Press.
- Welling, M., Rosen-Zvi, M., und Hinton, G. E. (2005). Exponential family harmoniums with an application to information retrieval. In L. Saul, Y. Weiss, und L. Bottou, Hrsg., *Advances in Neural Information Processing Systems 17 (NIPS'04)*, Band 17, Cambridge, MA. MIT Press.
- Werbos, P. J. (1981). Applications of advances in nonlinear sensitivity analysis. In *Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, Seiten 762–770.
- Weston, J., Bengio, S., und Usunier, N. (2010). Large scale image annotation: learning to rank with joint word-image embeddings. *Machine Learning*, **81**(1), 21–35.
- Weston, J., Chopra, S., und Bordes, A. (2014). Memory networks. *arXiv Vorabdruck arXiv:1410.3916*.
- Widrow, B. und Hoff, M. E. (1960). Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, Band 4, Seiten 96–104. IRE, New York.
- Wikipedia (2015). List of animals by number of neurons — Wikipedia, the free encyclopedia. [Online; Zugriff am 4. März 2015].
- Williams, C. K. I. und Agakov, F. V. (2002). Products of Gaussians and Probabilistic Minor Component Analysis. *Neural Computation*, **14**(5), 1169–1182.
- Williams, C. K. I. und Rasmussen, C. E. (1996). Gaussian processes for regression. In D. Touretzky, M. Mozer, und M. Hasselmo, Hrsg., *Advances in Neural Information Processing Systems 8 (NIPS'95)*, Seiten 514–520. MIT Press, Cambridge, MA.
- Williams, R. J. (1992). Simple statistical gradient-following algorithms connectionist reinforcement learning. *Machine Learning*, **8**, 229–256.
- Williams, R. J. und Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, **1**, 270–280.

- Wilson, D. R. und Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural Networks*, **16**(10), 1429–1451.
- Wilson, J. R. (1984). Variance reduction techniques for digital simulation. *American Journal of Mathematical and Management Sciences*, **4**(3), 277–312.
- Wiskott, L. und Sejnowski, T. J. (2002). Slow feature analysis: Unsupervised learning of invariances. *Neural Computation*, **14**(4), 715–770.
- Wolpert, D. und MacReady, W. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, **1**, 67–82.
- Wolpert, D. H. (1996). The lack of a priori distinction between learning algorithms. *Neural Computation*, **8**(7), 1341–1390.
- Wu, R., Yan, S., Shan, Y., Dang, Q., und Sun, G. (2015). Deep image: Scaling up image recognition. arXiv:1501.02876.
- Wu, Z. (1997). Global continuation for distance geometry problems. *SIAM Journal of Optimization*, **7**, 814–836.
- Xiong, H. Y., Barash, Y., und Frey, B. J. (2011). Bayesian prediction of tissue-regulated splicing using RNA sequence and cellular context. *Bioinformatics*, **27**(18), 2554–2562.
- Xu, K., Ba, J. L., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R. S., und Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In *ICML'2015*, arXiv:1502.03044.
- Yildiz, I. B., Jaeger, H., und Kiebel, S. J. (2012). Re-visiting the echo state property. *Neural networks*, **35**, 1–9.
- Yosinski, J., Clune, J., Bengio, Y., und Lipson, H. (2014). How transferable are features in deep neural networks? In *NIPS'2014*.
- Younes, L. (1998). On the convergence of Markovian stochastic algorithms with rapidly decreasing ergodicity rates. In *Stochastics and Stochastics Models*, Seiten 177–228.
- Yu, D., Wang, S., und Deng, L. (2010). Sequential labeling using deep-structured conditional random fields. *IEEE Journal of Selected Topics in Signal Processing*.
- Zaremba, W. und Sutskever, I. (2014). Learning to execute. arXiv 1410.4615.
- Zaremba, W. und Sutskever, I. (2015). Reinforcement learning neural Turing machines. arXiv:1505.00521.
- Zaslavsky, T. (1975). *Facing Up to Arrangements: Face-Count Formulas for Partitions of Space by Hyperplanes*. Number no. 154 in Memoirs of the American Mathematical Society. American Mathematical Society.

LITERATURVERZEICHNIS

- Zeiler, M. D. und Fergus, R. (2014). Visualizing and understanding convolutional networks. In *ECCV'14*.
- Zeiler, M. D., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q., Nguyen, P., Senior, A., Vanhoucke, V., Dean, J., und Hinton, G. E. (2013). On rectified linear units for speech processing. In *ICASSP 2013*.
- Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., und Torralba, A. (2015). Object detectors emerge in deep scene CNNs. ICLR'2015, arXiv:1412.6856.
- Zhou, J. und Troyanskaya, O. G. (2014). Deep supervised and convolutional generative stochastic network for protein secondary structure prediction. In *ICML'2014*.
- Zhou, Y. und Chellappa, R. (1988). Computation of optical flow using a neural network. In *Neural Networks, 1988., IEEE International Conference on*, Seiten 71–78. IEEE.
- Zöhrer, M. und Pernkopf, F. (2014). General stochastic networks for classification. In *NIPS'2014*.

Abkürzungsverzeichnis

Abk.	Bedeutung
ABC	approximate bayesian computation, dt. approximative bayessche Berechnung
ADALINE	adaptive linear element, dt. adaptives lineares Element
AIS	annealed importance sampling
ANN	artificial neural network, dt. künstliches neuronales Netz
ASIC	application-specific integrated circuits
ASR	automatic speech recognition, dt. automatische Spracherkennung
BFGS	Broyden-Fletcher-Goldfarb-Shanno algorithm, dt. Broyden-Fletcher-Goldfarb-Shanno-Algorithmus
BPTT	backpropagation through time
CAE	contractive autoencoder
CD	contrastive divergence
CNN	convolutional neural network
DAE	denoising autoencoder
DBM	deep Boltzmann machines
DBN	deep belief network, dt. Deep-Belief-Netz
DCGAN	deep convolutional generative adversarial network, dt. tiefes gefaltetes Generative-Adversarial-Netz
DRAW	deep recurrent attention writer (DRAW model)
EBM	energy-based model, dt. energiebasiertes Modell
ELBO	evidence lower bound

Abk.	Bedeutung
EM	expectation maximization (algorithm)
ESN	Echo State Network, dt. Echo-State-Netz
FPCD	fast persistent contrastive divergence
FPGA	field programmable gate array, dt. im Feld programmierbare (Logik-)Gatter-Anordnung
FVBN	fully-visible Bayes network, dt. vollständig sichtbares Bayes-Netz
GAN	generative adversarial network, dt. Generative-Adversarial-Netz
GCN	global contrast normalization, dt. globale Kontrastnormalisierung
GMM	Gaussian mixture model, dt. gaußsches Mischmodell
GRU	gated recurrent units
GSM	generalized score matching, dt. generalisiertes Score Matching
GSN	generative stochastic networks, dt. generative stochastische Netze
ICA	independent component analysis, dt. Unabhängigkeitsanalyse
KKT	Karush-Kuhn-Tucker
KL divergence	Kullback-Leibler divergence, dt. Kullback-Leibler-Divergenz
LIS	linked importance sampling
LSTM	long short-term memory
MAP	maximum a posteriori estimation, dt. Maximum-a-posteriori-Methode
MCMC	Markov chain Monte Carlo methods, dt. Markow-Ketten-Monte-Carlo-Verfahren
mcRBM	mean and covariance RBM
MLP	multilayer perceptron, dt. mehrschichtiges Perzeptron
MMD	maximum mean discrepancy, dt. maximale Mittelwertabweichung

ABKÜRZUNGSVERZEICHNIS

Abk.	Bedeutung
MNIST	NIST: National Institute of Standards and Technology (dt. Nationales Institut für Standards und Technologie der USA), M: modified (dt. geändert)
MP-DBM	multi-prediction deep Boltzmann machine
mPoT	mean product of Student t-distribution, dt. Mittelwertproduktmodell der studentschen t-Verteilung
MQF	mittlerer quadratischer Fehler, engl. mean squared error (MSE)
NADE	neural auto-regressive density estimator
NCE	noise-contrastive estimation
NICE	nonlinear independent components estimation, dt. nichtlineare Unabhängigkeitsschätzung
NLL	negative log-likelihood, dt. negative Log-Likelihood
NLM	neural language model, dt. neuronales Sprachmodell
NLP	natural language processing, dt. Verarbeitung natürlicher Sprache
NP	NP-complete, dt. NP-vollständig
PCA	principal component analysis, dt. Hauptkomponentenanalyse
PCD	persistent contrastive divergence
PDF	probability density function, dt. Wahrscheinlichkeitsdichtefunktion
PMF	probability mass function, dt. Wahrscheinlichkeitsfunktion
PRelu	parametric rectified linear units, dt. parametrische rektifizierte lineare Einheit
RBF	radial basis function, dt. radiale Basisfunktion
RBM	restricted Boltzmann machine
ReLU	rectified linear unit, dt. rektifizierte lineare Einheit
RNN	recurrent neural network, dt. rekurrentes neuronales Netz

ABKÜRZUNGSVERZEICHNIS

Abk.	Bedeutung
SGD	stochastic gradient descent, dt. stochastisches Gradientenabstiegsverfahren
SML	stochastic maximum likelihood, dt. stochastische Maximum Likelihood
SPN	sum-product network, dt. Sum-Product-Netz
ssRBM	spike and slab RBM
SVD	singular value decomposition, dt. Singulärwertzerlegung
SVM	support vector machine
TDNN	time-delay neural network, dt. zeitverzögerte neuronale Netze
VAE	variational autoencoder
VC dimension	Vapnik-Chervonenkis dimension, dt. Vapnik-Chervonenkis-Dimension
XOR	exclusive or, dt. exklusiv oder

Index

Symbols

0-1-Verlust **114**, 308

A

A-priori-Wahrscheinlichkeitsverteilung **150**

Ableitung 90

AdaGrad 343

Adam 346, 472

Adaptives lineares Element 16, 25, 26

Adversarial Example 299, 300

Adversarial Training 304, 596

Äquivarianz 377

Affin 121

AIS *siehe* Annealed Importance Sampling

Aktive Bedingung 103

Aktivierungsfunktion 188

Ancestral Sampling 651, 669

ANN *siehe* künstliches neuronales Netz

Annealed Importance Sampling 704, 753, 809

Approximative bayessche Berechnung 808

Approximative Inferenz 656

ASR *siehe* Spracherkennung, automatische

Asymptotisch erwartungstreu 138

Audio 112, 401, 511

Ausgabeschicht 185

Autoencoder 5, 396, **563**

Autoencoder mit Gewichten nach Wichtigkeit 787

B

Backprop *siehe* Backpropagation

Backpropagation 225

Backpropagation Through Time **427**

Bag-of-Words 525

Bagging 285

Batch-Normalisierung 298, 472

Bayes-Fehler **129**

Bayes-Netz *siehe* gerichtetes graphisches Modell

Bayessche Hyperparameter-Optimierung 484

Bayesscher Wahrscheinlichkeitsbegriff 60

Bayessche Statistik **149**

Bedingte Berechnung *siehe* dynamische Struktur

Bedingte RBM 772

Bedingte Unabhängigkeit 65

Bedingte Wahrscheinlichkeit 64

Beispiel 109

Belief-Netz *siehe* gerichtetes graphisches Modell

Berechnungsgraph 226

Bergsteigeralgorithmus 93

Bernoulli-Verteilung 68

Bestärkendes Lernen *siehe* Reinforcement Learning

Beziehungen 539

BFGS 353

Bias *siehe* Verzerrung

Biased Importance Sampling 667

Bigramm 515

Binäre Relation 539

Block-Gibbs-Sampling 674

Boltzmann-Verteilung 641
Boltzmann-Maschine 641, 737, 757
BPTT *siehe* Backpropagation Through Time
Broadcasting 36
Burn-In 672

C

CAE *siehe* Contractive Autoencoder
Calculus of variations *siehe* Variationsrechnung
CD *siehe* kontrastive Divergenz
Chord 650
Chordaler Graph 650
Cliques-Potenzial *siehe* Faktor (grafisches Modell)
CNN 18, 282, **369**, 472, 513
Collider *siehe* Explaining-Away-Effekt
Computer Vision 504
Concept drift *siehe* Konzept-Drift
Conditional computation *siehe* dynamische Struktur
Contractive Autoencoder 584
Contrastive divergence *siehe* kontrastive Divergenz
Convolutional Neural Network *siehe* CNN
Coordinate Ascent *siehe* Koordinatenanstieg
CTC-Framework 513
Curriculum Learning 367
Cyc 3

D

d-Separation 643
DAE *siehe* Denoising Autoencoder
Datengenerierender Prozess 122
Datengenerierende Verteilung **122**, 145
Datenparallelität 497
Datensatz 115
DBM *siehe* Deep Boltzmann Machine
DBM mit mehreren Vorhersagen *siehe* Multi-Prediction DBM
DCGAN 620, 791

Deckung 470
Decoder 5
Deep-Belief-Netz 26, 596, 711, 741, 743, 771, 780
Deep Blue 2
Deep Boltzmann Machine 25, 26, 596, 711, 735, 741, 747, 757, 771
Deep convolutional GAN 791
Deep Learning 2, 6
Denoising Autoencoder 573, 777
Denoising Score Matching 697
Diagonalmatrix 44
Dichteschätzung 113
Differentielle Entropie 80, 728
Diracsche Deltafunktion 71
Disambiguierung 542
Diskriminative Feinabstimmung 746
Diskriminative RBM 774
Domänenadaption 603
Domain adaptation *siehe* Domänenadaption
Doppelt zyklische Blockmatrix 373
Double Backpropagation 304
Doubly block circulant matrix *siehe* doppelt zyklische Blockmatrix
Dreiecksungleichung 42
DropConnect 297
Dropout **287**, 472, 478, 479, 757, 777
Dünnbesetzte Repräsentation 162, 166, 283
Dynamische Struktur 499, 500

E

E-Schritt 714
Early stopping *siehe* früher Abbruch
EBM *siehe* energiebasiertes Modell
Echo-State-Netz 25, 448
Eigenvektor 45
Eigenwert 46
Eigenwertzerlegung 45
Einfache Zelle 406, 407
Einheitsmatrix 38
Einheitsvektor 44
ELBO 713

- Elementweises Produkt *siehe* Hadamard-Produkt
- EM-Algorithmus *siehe* Expectation-Maximization-Algorithmus
- Embedding 580
- Empfehlungsdienste 533
- Empirische Risikominimierung 307
- Empirisches Risiko 307
- Empirische Verteilung 72
- Encoder 5
- Energiebasiertes Modell 640, 668, 737, 747
- Energiefunktion 640
- Ensemblemethoden 285
- Entscheidungsbaum **159**, 615
- Entwurfsmatrix **117**
- Epoche 275
- Erkennungsschicht 379
- Ersatzverlustfunktion 308
- Erwartungsmaximierungsalgorithmus *siehe* Expectation-Maximization-Algorithmus
- Erwartungsschritt 714
- Erwartungstreu 138
- Erwartungswert 66
- Erweitern des Datensatzes 303, 510
- ESN *siehe* Echo-State-Netz
- Euklidische Norm *siehe* Norm, euklidische
- Euler-Lagrange-Gleichung 727
- Evidence Lower Bound *siehe* ELBO, 745
- Expectation-Maximization-Algorithmus 714
- Explaining-Away 712, 726
- Explaining-Away-Effekt 645
- Exploitation 538
- Exploration 538
- Exponentialverteilung **71**
- F**
- F-Maß 470
- Faktor (graphisches Modell) 637
- Faktorenanalyse 548
- Faktoren der Variation 5
- Faktograph 651
- Faltung 369, 769
- Farbbilder 401
- Fast sichere Konvergenz 143
- Fehlende Eingangsdaten 109
- Fehlerfunktion *siehe* Zielfunktion
- Feinabstimmung 361
- Finite Differenzen 488
- Fluch der Dimensionalität 172
- Forget-Gate 342
- Fortsetzungsmethoden 366
- Forward-Propagation 225
- Fourier-Transformation 401, 403
- Fovea 408
- FPCD 691
- Freebase 540
- Freie Energie **642**, 766
- Freie Helmholtz-Energie *siehe* ELBO
- Frequentistischer Wahrscheinlichkeitsbegriff 60
- Frequentistische Statistik **149**
- Frobenius-Norm 50
- Früher Abbruch 273, 274, 277, 278, 472
- FVBN *siehe* vollständig sichtbares Bayes-Netz
- G**
- Gaborfunktion 410
- GAN *siehe* Generative-Adversarial-Netz
- GAN, tiefes gefaltetes 791
- Gated Recurrent Unit 472
- Gauß-Verteilung *siehe* Normalverteilung
- Gaussian mixture model 73
- Gaußsche Mischverteilung 209
- Gaußscher Kernel 157
- GCN 507
- Gehemmte Initialisierung 340
- Genauigkeit 470
- GeneOntology 540
- Generalisierung 121
- Generative-Adversarial-Netz 777, 788
- Generative-Moment-Matching-Netze 793
- Generator-Netze 782

Gerichtetes graphisches Modell 84,
569, 633, 780
Gewicht 15, 118
Gibbs-Verteilung 639
Gibbs-Sampling 653, 674
Gleichheitsbedingung 102
Gleichverteilung 62
Globale Kontrastnormalisierung *siehe*
GCN
GPU *siehe* Grafikprozessor
Gradient 92
Gradienten-Clipping 322, 459
Gradientenabstiegsverfahren 90, 93
Grafikprozessor 494
Graphisches Modell *siehe* strukturier-
tes probabilistisches Modell
Greedy-Algorithmus 361, 593
Greedy supervised pretraining *sie-
he* Überwachtes Pretraining
mit Greedy-Algorithmen
Grid search *siehe* Rastersuche

H

Hadamard-Produkt 37
Halb-überwachtes Lernen 270
Hard tanh 218
Harmonie-Theorie 642
Harmonium *siehe* Restricted Boltz-
mann Machine
Hauptdiagonale 35
Hauptkomponentenanalyse 51, 163,
164, 548, 711
Hesse-Matrix 94, 247
Heteroskedastisch 207
Hyperparameter 133, 478
Hyperparameter-Optimierung 480
Hypothesenraum 123, 131

I

Identifizierbarkeit eines Modells 317
ILSVRC *siehe* ImageNet Large Scale
Visual Recognition Chal-
lenge
ImageNet Large Scale Visual Recog-
nition Challenge 27
Immoralität 648

Importance Sampling 666, 703, 787
Inferenz 630, 656, 711, 713, 716, 718,
730, 733

Informationsgehalt 80
Informationsgewinnung 589
Inhaltsabhängige Empfehlungsdiens-
te 536

Inhaltsbasierte Adressierung 464
Initialisierung 335
Invarianz 379
Inverse einer Matrix 39
Isotrop 71

J

Jacobi-Matrix 78, 94

K

k-Means 404, 615
k-Nearest-Neighbor **158**, 615
Kachel-Faltung 391
Karush-Kuhn-Tucker 101
Karush-Kuhn-Tucker-Bedingungen
103, 263
Kategoriale Verteilung *siehe* Multi-
noulli-Verteilung
Kernel (Faltung) 371
Kernel-Maschine 615
Kernel-Trick 156
Kettenregel (Analysis) 228
KKT *siehe* Karush-Kuhn-Tucker
KKT-Bedingungen *siehe* Karush-
Kuhn-Tucker-Bedingungen
KL-Divergenz *siehe* Kullback-Leibler-
Divergenz
Klassifizierung 109
Klassisches dynamisches System 417
Kollaboratives Filtern 534
Komplexe Zelle 407
Konditionszahl 311
Konnektionismus 18, 493
Konsistenz 143, 574
Kontextabhängige Banditen 536
Kontextabhängige Unabhängigkeit
644
Kontrahierender Autoencoder *siehe*
Contractive Autoencoder

- Kontrast 506
Kontrastive Divergenz 324, 686, 757
Konvexe Optimierung 157
Konzept-Drift 604
Koordinatenabstieg 359
Koordinatenanstieg 756
Korrekttheit 469
Korrelation 67
Kostenfunktion *siehe* Zielfunktion
Kovarianz 66
Kovarianzmatrix 67
Kreuzentropie **83**, 146
Kreuzkorrelation 372
Kreuzvalidierung 135
Kritische Temperatur 679
Krylow-Verfahren 247
Künstliche Intelligenz 1
Künstliches neuronales Netz *siehe* Neuronales Netz
Kullback-Leibler-Divergenz **80**
- L**
Label-Glättung 270
Lagrange *siehe* Lagrange-Funktion, allgemeine
Lagrange-Funktion, allgemeine 101
Lagrange-Multiplikatoren 101, 728
LAPGAN 792
Laplace-Verteilung **71**, 556
Latente Variable 73
LCN 509
Leaky-Einheiten 452
Leaky ReLU 213
Lernrate 93
Lineare Abhängigkeit 41
Lineare Faktoremodellen 547
Lineare Regression **118**, 121, 155
Linearkombination 40
Liniensuche 93, 101
Link Prediction 541
Lipschitz-Konstante 100
Lipschitz-Stetigkeit 100
Liquid State Machine 448
Logistische Regression 3, **155**, 156
Logistische Sigmoidfunktion 8, 74
- Lokal bedingte Wahrscheinlichkeitsverteilung 634
Lokale Kontrastnormalisierung *siehe* LCN
Long short-term memory 19, 28, 342, **455**, 472
Loopy Belief Propagation 658
 L^p Norm 42
LSTM *siehe* Long short-term memory
- M**
M-Schritt 715
Machine Learning 3
Manifold Learning 178
Mannigfaltigkeit 177
Mannigfaltigkeit-Hypothese 179
Mannigfaltigkeit-Tangentenklassifikator 304
MAP-Approximation 153, 567
Markow-Kette 668
Markow-Kette-Monte-Carlo 668
Markow-Netz *siehe* ungerichtetes Modell
Markow-Zufallsfeld *siehe* ungerichtetes Modell
Maschinelle Übersetzung 111
Maß Null 77
Maßtheorie 77
Matrix 35
Matrixprodukt 36
Max-Pooling 379
Maximum Likelihood **145**
Maximumsnorm 43
Maxout 213, 472
MCMC *siehe* Markow-Kette-Monte-Carlo
Mehrschichtiges Perzepron 26
Memory-Netz 462, 463
Merkmalsauswahl 262
Mini-Batch 311
Mischen (Markow-Kette) 676
Mischmodell 209, 572
Mischverteilung 72
Mittlerer quadratischer Fehler 119
Mixture-Density-Netze 209

Mixture of Experts 501, 615
MLP 6
MNIST 22, 23, 757
Modellkomprimierung 498
Modellmittelung 285
Modellparallelität 497
Molekularfeld 719, 720, 757
Moment Matching 793
Moore-Penrose-Pseudoinverse 49, 266
Moralisierter Graph 648
MP-DBM *siehe* Multi-Prediction DBM
MQF *siehe* mittlerer quadratischer Fehler
MRF (Markow-Zufallsfeld) *siehe* ungerichtetes Modell
Multi-Prediction DBM 759
Multimodales Lernen 607
Multinomialverteilung 68
Multinoulli-Verteilung 68
Multitask Learning 271, 604
Multivariate Verteilung 62

N

N-Gramm **515**
NADE 799
Naive Bayes 3
Natürliches Bild 628
Nearest-Neighbor-Regression 127
Negativ definit 97
Negative Phase 524, 682, 685
Neocognitron 18, 25, 26, 409
Nesterow-Momentum 334
Netflix Grand Prize 286, 535
Neuronales Feedforward-Netz 185
Neuronales Netz 14
Neuronales Sprachmodell 517, 532
Neuronale Turing-Maschine 463
Neurowissenschaft 16
Newton-Verfahren 98, 348
Nicht Normalisierte Wahrscheinlichkeitsverteilung 637
Nichtparametrisches Modell **127**
Nit 80
NLM *siehe* neuronales Sprachmodell

NLP *siehe* Verarbeitung natürlicher Sprache
No-Free-Lunch-Theorem 130, 133, 220, 623
Noise-Contrastive Estimation 698
Norm 42
Norm, euklidische 42
Norm 1 44
Normalgleichungen **120**, 120, 124, 259
Normalisierte Initialisierung 338
Normalverteilung 69, 70, 139
Numerische Differentiation *siehe* finiten Differenzen

O

Objekterkennung 505
OMP-*k* *siehe* Orthogonal Matching Pursuit
One-Shot Learning 605
Operation 226
Optimierung 87, 90
Optimierung unter Nebenbedingungen 101, 263
Orthodoxe Statistik *siehe* frequentistische Statistik
Orthogonale Matrix 45
Orthogonalität 44
Orthogonal Matching Pursuit 26, **284**

P

Parallel Distributed Processing 18
Parameterinitialisierung 335, 450
Parameter Sharing 281, 374, 415, 417, 432
Parameter Tying *siehe* Parameter Sharing
Parametrisches Modell **127**
Partielle Ableitung 92
Partitionsfunktion 638, 681, 754
PCA *siehe* Hauptkomponentenanalyse
PCD *siehe* stochastische Maximum Likelihood
Persistente kontrastive Divergenz *siehe* stochastische Maximum Likelihood

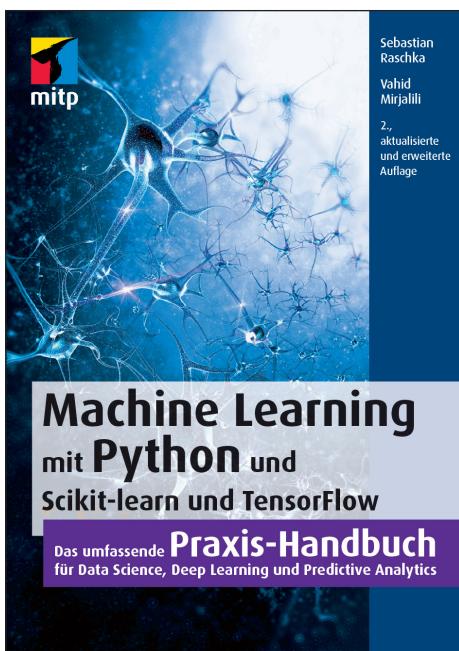
- Perzeptron 16, 26
Policy 536
Pooling 369, 770
Positiv definit 97
Positive Phase 524, 682, 685, 739, 753
Prädiktive dünnbesetzte Zerlegung 587
Präzision (einer Normalverteilung) 69, 70
Pretraining 361, 593
Primärer visueller Cortex 406
Probabilistische PCA 548, 549, 712
Probabilistisches Max-Pooling 770
Product of Experts 641
Produktregel für Wahrscheinlichkeiten 65
PSD *siehe* prädiktive dünnbesetzte Zerlegung
Pseudo-Likelihood 692
Punktschätzer 136
- Q**
Quadratmatrix 41
Quadraturpaar 412
Quasi-Newton-Verfahren 353
- R**
Radiale Basisfunktion 217
Random search *siehe* Zufallssuche
Randwahrscheinlichkeit 64
Rastersuche 481
Ratio Matching 696
RBF 217
RBM *siehe* Restricted Boltzmann Machine
RBM, bedingte 772
RBM, diskriminative 774
Regression 110
Regularisierer 132
Regularisierung 133, 133, 196, 253, 478
REINFORCE 777
Reinforcement Learning 29, 116, 536, 777
Rektifizierung des Absolutbetrags 213
Relationale Datenbank 540
- ReLU 191, 213, 472, 569
ReLU, parametrisch 213
Reparametrisierung 776
Repräsentative Kapazität 125
Representation Learning 3
Restricted Boltzmann Machine 396, 512, 536, 659, 711, 740, 762, 767, 769
Rezeptives Feld 376
Richtungsableitung 92
Ridge-Regression *siehe* Weight Decay
Risiko 307
RNN 26, 420
RNN-RBM 773
- S**
Sattelpunkte 318
Satz von Bayes 76
Schach 2
Schicht (neuronales Netz) 185
Schichtweises unüberwachtes Pretraining 593
Schleife 650
Score Matching 574, 695
Semantisches Hashing 589
Separation (probabilistische Modellierung) 643
Separierbare Faltung 403
SGD *siehe* stochastisches Gradientenabstiegsverfahren
Shannon-Entropie 80
Shortlist 520
Sichtbare Schicht 7
Sigmoid *siehe* logistische Sigmoidfunktion
Sigmoid-Belief-Netz 26
Singulärvektor *siehe* Singulärwertzerlegung
Singulärwert *siehe* Singulärwertzerlegung
Singulärwertzerlegung 48, 164, 535
Skalar 34
Skalarprodukt 37, 156
Slow Feature Analysis 553
SML *siehe* stochastische Maximum Likelihood

- softmax 203, 463, 501
Softplus 74, 217
Spamerkennung 3
Sparse Coding 359, 396, 556, 711, 780
Spearmint 484
Spektralradius 449
Sphering *siehe* Whitening
Spike and Slab Restricted Boltzmann Machine 767
SPN *siehe* Sum-Product-Netz
Spracherkennung, automatische 511
Sprachmodelle auf Klassenbasis 517
Spuroperator 50
ssRBM *siehe* Spike and Slab Restricted Boltzmann Machines
Standardabweichung 66
Standardfehler 140
Standardfehler des Mittelwerts 141, 309
Statistik 136
Statistische Lerntheorie 122
Steilster Abstieg *siehe* Gradientenabstiegsverfahren
Stichprobenmittelwert 139
Stochastische Backpropagation *siehe* Reparametrisierung
Stochastische Maximum Likelihood 689, 757
Stochastisches Gradientenabstiegsverfahren 16, 167, 311, **328**, 757
Stochastisches Pooling 297
Störungsanalyse *siehe* Reparametrisierung
Structure Learning 654
Strukturierte Ausgabe 111, 772
Strukturiertes probabilistisches Modell 83, 627
Sum-Product-Netz 622
Summenregel für Wahrscheinlichkeiten 64
Support Vector Machine 156
SVD *siehe* Singulärwertzerlegung
Symmetrische Matrix 44, 46
- T**
Tangenten-Propagation 301
Tangentendistanz 301
Tangentialebene 578
Tatsächliche Kapazität 125
TDNN *siehe* zeitverzögertes neuronales Netz
Teacher Forcing 425, 426
Tempering 678
Template Matching 157
Tensor 35
Testdatenmenge 122
Tiefes Feedforward-Netz 185, 472
Tikhonov-Regularisierung *siehe* Weight Decay
Tiled convolution *siehe* Kachel-Faltung
Toeplitz-Matrix 373
Topografische Unabhängigkeitsanalyse 552
Trainingsfehler 121
Transfer Learning 602
Transkription 110
Transponierte 35
Traumschlaf 686, 734
Trefferquote 470
triangulierter Graph *siehe* Chordaler Graph
Trigramm 515

U

- u.i.v.-Annahmen 122, 136, 299
Überprüfung der zweiten Ableitung 97
Überwachte Feinabstimmung 595, 746
Überwachtes Lernen **115**
Überwachtes Pretraining mit Greedy-Algorithmen 361
Unabhängige Unterraumanalyse 552
Unabhängigkeit 65
Unabhängigkeitsanalyse 549
Unabhängig und identisch verteilt *siehe* u.i.v.-Annahmen
Ungerichtetes graphisches Modell 84, 568
Ungerichtetes Modell 636
Ungleichheitsbedingung 102
Unigramm 515

- Universal-Approximation-Theorem 219
Universal Approximation 622
Unüberwachtes Lernen 115, 162
Unüberwachtes Pretraining 512, 593
- V**
- V-Struktur *siehe* Explaining-Away-Effekt V1 406
VAE *siehe* Variational Autoencoder
Vapnik-Chervonenkis-Dimension 126
Varianz 66, 254
Variational Autoencoder 777, 785
Variational Free Energy *siehe* ELBO
Variationsableitungen *siehe* Funktionalableitungen
Variationsrechnung 198
VC-Dimension *siehe* Vapnik-Chervonenkis-Dimension
Vektor 34
Verarbeitung natürlicher Sprache 514
Verdeckte Schicht 7, 185
Verfahren des steilsten Abstiegs *siehe* Gradientenabstiegsverfahren
Verlustfunktion *siehe* Zielfunktion
Verteilte Repräsentation 19, 167, 614
Verzerrung 138, 254
Verzerrungsparameter 121
Virtuelle Adversarial Examples 300
Vollständig sichtbares Bayes-Netz 796
- Volumetrische Daten 401
Vorverarbeitung 505
- W**
- Wahrscheinlichkeitsdichtefunktion 63
Wahrscheinlichkeitsfunktion 61
Wahrscheinlichkeitsfunktionsschätzung 113
Wahrscheinlichkeitsverteilung 61
Wake-Sleep 733, 745
Weight Decay 131, 196, 256, 479
Weight Space Symmetry 317
Wert, erwarteter *siehe* Erwartungswert
Whitening 509
Wikibase 540
Wissensbasis 2, 540
WordNet 540
Wort-Embedding 518
- Z**
- Zeitverzögertes neuronales Netz 409, 416
Zentraler Grenzwertsatz 70
Zentrierung (DBM) 758
Zero-Data Learning *siehe* Zero-Shot Learning
Zero-Shot Learning 605
Zielfunktion 90
Zufallssuche 482
Zufallsvariable 61
Zweite Ableitung 94



Sebastian Raschka • Vahid Mirjalili

Machine Learning mit Python und Scikit-learn und TensorFlow

Das umfassende Praxis-Handbuch
für Data Science, Deep Learning und
Predictive Analytics

Datenanalyse mit ausgereiften statistischen
Modellen des Machine Learnings

Anwendung der wichtigsten Algorithmen und
Python-Bibliotheken wie NumPy, SciPy, Scikit-
learn, TensorFlow, Matplotlib, Pandas und Keras

Best Practices zur Optimierung Ihrer Machine-
Learning-Algorithmen

ISBN 978-3-95845-733-1

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/733

Jake VanderPlas

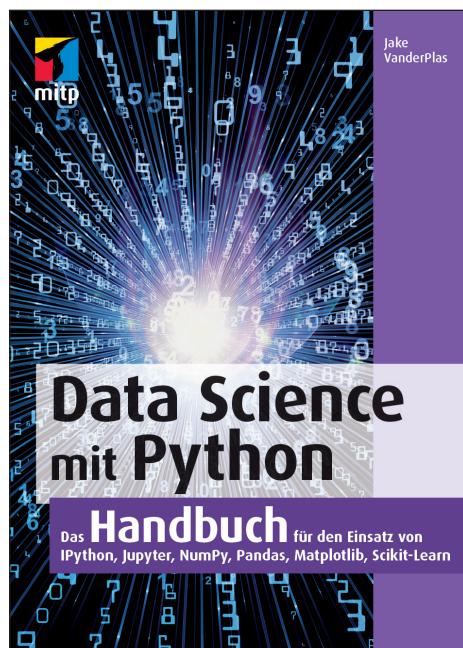
Data Science mit Python

Das Handbuch für den Einsatz von
IPython, Jupyter, NumPy, Pandas,
Matplotlib, Scikit-Learn

Die wichtigsten Tools für die Datenanalyse
und -bearbeitung im praktischen Einsatz
Python effizient für datenintensive Berech-
nungen einsetzen mit IPython und Jupyter
Laden, Speichern und Bearbeiten von Daten
und numerischen Arrays mit NumPy und Pandas
Visualisierung von Daten mit Matplotlib

ISBN 978-3-95845-695-2

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/695





Joachim Schlosser

Wissenschaftliche Arbeiten schreiben mit LATEX

Leitfaden für Einsteiger

Schnell zur fertig gesetzten Arbeit – ohne Vorkenntnisse

Lösungsorientierte und verständliche Beschreibungen

Von Tabellen und Formeln über Grafiken bis zum Literaturverzeichnis

ISBN 978-3-95845-289-3

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/289

Andrea Klein

Wissenschaftliche Arbeiten schreiben

Praktischer Leitfaden mit über
100 Software-Tipps

Alle Grundlagen zum Schreiben wissenschaftlicher Arbeiten

Methoden zur Selbstorganisation und Zeitplanung sowie Strategien für unterschiedliche Schreibtypen

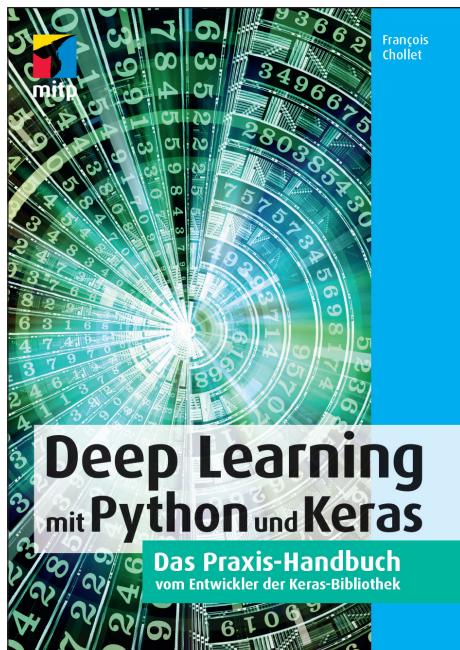
Literaturverwaltung, -recherche und -auswertung, Inhalte sammeln, strukturieren, schreiben und effektiv überarbeiten

Software als hilfreiche Unterstützung für alle Phasen der Arbeit

ISBN 978-3-95845-386-9

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/386





François Chollet

Deep Learning mit Python und Keras

Das Praxis-Handbuch
vom Entwickler der Keras-Bibliothek

Einführung in die grundlegenden Konzepte von
Machine Learning und Deep Learning

Zahlreiche praktische Anwendungsbeispiele
zum Lösen konkreter Aufgabenstellungen

Maschinelles Sehen, Sprachverarbeitung, Bild-
klassifizierung, Vorhersage von Zeitreihen,
Stimmungsanalyse, Erzeugung von Bildern und
Texten u.v.m.

ISBN 978-3-95845-838-3

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/838

François Chollet · J.J. Allaire

Deep Learning mit R und Keras

Das Praxis-Handbuch
von Entwicklern von Keras und RStudio

Einführung in die grundlegenden Konzepte
von Machine Learning und Deep Learning

Zahlreiche praktische Anwendungsbeispiele
zum Lösen konkreter Aufgabenstellungen:
Maschinelles Sehen, Sprachverarbeitung,
Bildklassifizierung, Vorhersage von Zeitreihen,
Stimmungsanalyse

CNNs, Rekurrente neuronale Netze, generative
Modelle wie Variational Autoencoder und Gene-
rative-Adversarial-Netze

ISBN 978-3-95845-893-2

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/893

