Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische
Friedrich-Wilhelms-Universität Bonn

Diplomarbeit

Empirische Evaluation für Gradientenabstiegsverfahren erster Ordnung in nicht-konvexen Modellen

Autor:
Lukas Koliogiannis

Erstprüfer:
Prof. Dr. Sven Behnke

Zweitprüfer:
Prof. Dr. Joachim K. Anlauf

Betreuer:
Hannes Schulz

Eingereicht: Montag, den 08.09.2014

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich die eingereichte Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich in jedem einzelnen Fall durch die Angabe der Quelle als Entlehnung kenntlich gemacht.
Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Diplomarbeit eingereicht wurde.

Ort, Datum

Unterschrift

Zusammenfassung

Die Verwendung von neuronalen Netzen hat in den letzten Jahren, vor allem im Bereich der Mustererkennung, an Bedeutung gewonnen. Dabei ist die Lernfähigkeit solcher Netze ein besonderer Vorteil. Je nach Netzgröße und Typ kann ein entsprechendes Training eine lange Zeit in Anspruch nehmen. Dadurch ist die Entwicklung und Verwendung von effizienten Lernalgorithmen zwingend erforderlich. Optimierungsverfahren erster Ordnung basieren oft auf dem klassischen Gradientenabstieg. Innerhalb eines solchen Verfahrens hängt die Trainingseffizienz häufig von der Wahl der Lernrate ab. Diese Arbeit gibt einen Überblick über aktuelle Algorithmen, welche individuelle, anpassungsfähige Lernraten verwenden.

Einer der bekanntesten Vertreter dieser Klasse ist der RPROP-Algorithmus. Dieser zeigt eine beeindruckende Effizienz beim Batchlearning, weist jedoch Schwächen bei der Verwendung mit Minibatches auf, welche in dieser Arbeit erläutert und experimentell belegt werden. Als Minibatchansatz von RPROP kann die RMSPRop Lernrate angesehen werden. Hierbei werden die Gradienten von zeitlich benachbarten Iterationen durch die Verwendung eines Mean-Squares normiert. Basierend auf dieser Lernrate werden die Algorithmen RMSProp, NA-RMSProp und RRMSProp formuliert. Eine zu RMSProp ähnliche Lernrate nutzt der AdaGrad-Algorithmus.

Sowohl RMSPRop als auch AdaGrad wurden erfolgreich für das Training von nicht-konvexen Modellen eingesetzt. Zum aktuellen Zeitpunkt liegen jedoch keine empirischen Informationen über die Optimierungsfähigkeit, die Hyperparameterempfindlichkeit und die Trainingsdauer dieser Algorithmen vor. Diese Arbeit führt eine empirische Evaluation dieser Algorithmen durch und liefert solche Informationen. Anhand der gewonnenen Daten werden Vergleiche zwischen den RMSProp-Versionen, AdaGrad, RPROP und dem klassischen Gradientenabstieg gezogen. Die Evaluation erfolgt durch Experimente auf MLPs und Konvolutionsnetzen, welche mit den Datensätzen MNIST und CIFAR-10 trainiert werden.

Inhaltsverzeichnis

1 Einleitung **1**

2 Theoretische Grundlagen **3**

2.1 Neuronale Netze . . . 3

2.1.1 Das künstliche Neuron . . . 4

2.1.2 Multi-Layer-Perzeptrons . . . 5

2.1.3 Konvolutionsnetze . . . 7

2.2 Training von Neuronalen Netzen . . . 10

2.2.1 Überwachtes Lernen . . . 10

2.2.2 Der Gradientenabstieg . . . 12

2.2.3 Präsentation und Verarbeitung der Muster . . . 13

2.2.4 Lernraten . . . 14

2.3 Regularisierung . . . 16

2.3.1 Early-Stopping . . . 17

2.3.2 Dropout . . . 18

3 Verwandte Arbeiten **21**

3.1 Gradientenabstieg mit Momentum . . . 21

3.2 Optimierungsverfahren zweiter Ordnung . . . 22

3.3 Verfahren nach Tom Schaul . . . 23

3.4 AdaGrad . . . 25

3.5 RPROP . . . 27

4 Das RMSProp-Lernverfahren **29**

4.1 Die Minibatch-Problematik von RPROP . . . 29

4.2 Der RMSProp-Ansatz . . . 30

4.3 RMSProp Versionen . . . 31

4.3.1 RMSProp . . . 31

4.3.2 Elastisches RMSProp . . . 32

4.3.3 Durch Nesterov-Momentum beschleunigtes RMSProp 32

5 Arbeitsauftrag und Arbeitshypothesen 35

6 Experimente 39
6.1 Datensätze 39
6.2 Netzarchitekturen 41
6.3 Kreuzvalidierung 42
6.4 Hyperopt 46
6.4.1 Der TPE-Algorithmus 47
6.5 Retraining 49
6.6 Implementierung und Ausführung 49
6.6.1 CUV und CUVNET 50
6.6.2 Das Learner-Konzept 51
6.6.3 Hyperoptclients 52

7 Ergebnisse 55
7.1 AdaGrad 55
7.2 GD 60
7.3 RMSProp 62
7.4 NA-RMSProp 66
7.5 RPROP 73
7.6 RRMSProp 77
7.7 Vergleich und Schlussfolgerungen 78
7.8 Empfehlungen 84

8 Ausblick 87

Anhang 89
1 Hyperparameter-Intervalle 89
2 Kreuzvalidierte Hyperparameter 92

Abbildungsverzeichnis

2.1 Die Aktivierungsfunktionen tanh und fermi 5
2.2 Beispiel einer MLP-Architektur 6
2.3 Konvergenzverhalten für verschiedene Lernraten 15

6.1 Beispiele für Trainingsmuster aus den Datensätzen MNIST und CIFAR-10 40
6.2 Konvolutionsnetz CNN-MNIST 43
6.3 Konvolutionsnetz CNN-CIFAR 44
6.4 Zur Kreuzvalidierung benötigte Aufteilung der Trainigsmenge . . . 45
6.5 Verdeutlichung der Kommunikation zwischen Hyperopt und der Kreuzvalidierung 48
6.6 Abhängigkeiten zwischen den zur Implementierung benötigten Komponenten 53

7.1 Kreuzvalidierte Werte für η und λ in AdaGrad 57
7.2 Verteilung der durch AdaGrad erzielten Klassifikationsfehler 58
7.3 Entwicklung der AdaGrad-Lernrate 59
7.4 Proportionalität zwischen Lernrate und Batchgröße in GD 60
7.5 Verteilung der durch GD erzielten Klassifikationsfehler 61
7.6 Kreuzvalidierte Werte für η und δ in RMSProp 63
7.7 Auftragung aller untersuchten Lernraten gegen resultierende Epochenanzahl für RMSProp 64
7.8 Verteilung der durch RMSProp erzielten Klassifikationsfehler 65
7.9 Entwicklung der RMSProp-Lernrate 66
7.10 Verteilung der durch NA-RMSProp erzielten Klassifikationsfehler . 68
7.11 Auftragung aller untersuchten Lernraten gegen resultierende Epochenanzahl für NA-RMSProp 69
7.12 Entwicklung der NA-RMSProp-Lernrate 70
7.13 Kreuzvalidierte Werte für γ und μ in NA-RMSProp 71

7.14 Kreuzvalidierte Werte für η, δ und ρ in NA-RMSProp 72
7.15 Verteilung der durch RPROP erzielten Klassifikationsfehler 74
7.16 Auftragung der zehn besten l_2-Regularisierungskonstante λ gegen die Batchgröße in RPROP 75
7.17 Entwicklung des Gradienten und der RPROP-Lernraten 76
7.18 Kreuzvalidierte Werte für λ in RPROP 76
7.19 Verteilung der durch RRMSProp erzielten Klassifikationsfehler . . . 79
7.20 Kreuzvalidierte Werte für Δ_0 und λ in RPROP 80

Tabellenverzeichnis

6.1 Spezifikationen der MLP-Architekturen 41
6.2 Aufteilung der Datensätze MNIST und CIFAR-10 46

7.1 Hyperparameter für AdaGrad mit geringstem Validierungsfehler . . 56
7.2 Hyperparameter für GD mit geringstem Validierungsfehler 59
7.3 Hyperparameter für RMSProp mit geringstem Validierungsfehler . . 62
7.4 Hyperparameter für NA-RMSProp mit geringstem Validierungsfehler 67
7.5 Hyperparameter für RPROP mit geringstem Validierungsfehler . . . 73
7.6 Hyperparameter für RRMSProp mit geringstem Validierungsfehler . 77
7.7 Übersicht über die geringsten erzielten Validierungsfehler 81
7.8 Vergleich CNN-MNIST mit aktuellen Arbeiten 84
7.9 Vergleich CNN-CIFAR mit aktuellen Arbeiten 84

1 Intervalle in denen die besten zehn Hyperparameter von AdaGrad liegen. 90
2 Intervalle in denen die besten zehn Hyperparameter von GD liegen. 90
3 Intervalle in denen die besten zehn Hyperparameter von RMSProp liegen. 90
4 Intervalle in denen die besten zehn Hyperparameter von NA-RMSProp liegen. 91
5 Intervalle in denen die besten zehn Hyperparameter von RPROP liegen. 91
6 Intervalle in denen die besten zehn Hyperparameter von RRMSProp liegenl.. 92

1 Einleitung

Tiefe neuronale Netze (z.B. Bengio (2009), Schulz und Behnke (2012)) haben in den vergangenen Jahren zunehmend an Bedeutung gewonnen. Das Training von solchen Netzen gestaltet sich jedoch schwierig. Da der Gradient in Richtung Eingabeschicht zunehmend an Informationsgehalt verliert, lassen sich die entsprechenden Schichten nur unzureichend optimieren. Dem kann durch ein geeignetes Vortraining vorgebeugt werden. Die bekanntesten Ansätze zur Realisierung eines Vortrainings sind Restricted Boltzmann Machines (RBM) (G. Hinton und Salakhutdinov, 2006), G. Hinton, Osindero u. a. (2006) oder Denaising Autoencoder (DAE) (Vincent u. a., 2010). Dabei wird jede Schicht separat unüberwacht vortrainiert und anschließend das gesamte Netz mit den vortrainierten Parametern durch ein überwachtes Training optimiert. Zu beachten ist, dass dabei das Training eine lange Zeit in Anspruch nehmen kann. So müssen in einem tiefen vollverknüpftem Netz mit bereits drei verdeckten Schichten, dass durch DAEs vortrainiert wird, vier verschiedene DAEs trainiert werden und anschließend das gesamte Netz.

Konvolutionsnetze sind innerhalb der Klasse von tiefen neuronalen Netzen von besonderer Bedeutung. Solche Netze besitzen durch ihre Weight-Sharing-Eigenschaft weniger freie Parameter, die durch ein Training einzustellen sind. In der Regel ist dieser Netztyp sehr rechenintensiv, so dass auch bei diesem Netztyp ein unüberwachtes Vortraining mit anschließender globaler Optimierung eine lange Zeit in Anspruch nehmen kann.

Auch ein Konvolutionsnetz, welches drei verdeckte Schichten besitzt und anschließend eine vollverknüpfte Klassifikationsschicht, kann durch Expansion der Konvolutions- und Poolingschichten aus acht Schichten bestehen (Inputschicht, 3 $\times$ Konvolutionsschicht, 3 $\times$ Poolingschicht und Outputschicht). In beiden Fällen kann das Training (z.B. durch den klassischen Gradientenabstieg) sehr lange dauern. Effiziente Lernalgorithmen sind somit zwingend erforderlich.

Die Lerngeschwindigkeit wird maßgeblich durch die Wahl der Lernrate beeinflusst. Es gibt viele Ansätze, die Lernrate eines Optimierungsverfahrens zu bestimmen. Dabei haben in den letzten Jahren solche Lernverfahren an Bedeutung

gewonnen, die für jedes Gewicht eine individuelle Lernrate verwenden, welche in jedem Iterationsschritt angepasst werden kann. Einer der bekanntesten Vertreter dieser Klasse von Lernalgorithmen ist RPROP (Riedmiller und Braun, 1993). Dieses Verfahren zeigt jedoch einige Schwächen in Bezug auf Minibatches. Diese Schwächen werden im Rahmen dieser Arbeit aufgezeigt und experimentell belegt.

Einen auf RPROP basierenden Ansatz zur Bestimmung von globalen, adaptiven Lernraten liefert RMSProp (G. Hinton, 2012). Dieser Ansatz bildet den Kern der vorliegenden Arbeit. Es werden drei verschiedene Algorithmen formuliert, welche diese RMSProp-Lernrate verwenden. Ein zu RMSProp sehr ähnliches Lernverfahren ist AdaGrad (Duchi, Hazan u. a., 2010). Zum aktuellen Zeitpunkt liegen keine empirischen Informationen über die Optimierungseigenschaften, die Hyperparameterempfindlichkeit und die Trainingsdauer dieser Algorithmen vor. Ziel dieser Arbeit ist es, solche Daten durch eine empirische Evaluation zu liefern.

Im Verlauf dieser Ausarbeitung werden zunächst einige theoretische Grundlagen zu neuronalen Netzen und ihrem Training gelegt, um anschließend die zu testenden Algorithmen näher zu erläutern. Nach diesem theoretischen Teil der Arbeit werden die zur Evaluation durchgeführten Experimente beschrieben und im Anschluss ein Überblick über die erzielten Ergebnisse geliefert.

2 Theoretische Grundlagen

Bevor näher auf die zu evaluierenden Algorithmen eingegangen werden kann, werden an dieser Stelle einige Grundlagen zum allgemeinen Verständnis von künstlichen neuronalen Netzen und deren Training geliefert.

2.1 Neuronale Netze

Die Entwicklung künstlicher neuronaler Netze ist inspiriert von der Informationsverarbeitung innerhalb eines biologischen Gehirns. In diesem werden Informationen von einer Vielzahl von Nervenzellen parallel verarbeitet. Aus dem biologischen Vorbild geht die folgende Arbeitsweise von Neuronen hervor:

1. Die Inputs werden im Zellkörper aufsummiert,
2. Überschreitet dieser Reiz eine gewisse Schwelle, so wird ein Signal an andere Nervenzellen weitergeleitet,
3. Dieses Signal kommt dann in anderen Zellen gehemmt oder verstärkt an, und wird dort als Input verarbeitet.

Ein großer Vorteil von neuronalen Netzen ist, ihre Fähigkeit zu lernen. Sie besitzen die Fähigkeit, selbstständig Lösungswege zu entwickeln und somit, komplexe Aufgaben (wie beispielsweise Klassifikationsaufgaben) mit Hilfe von Trainingsbeispielen zu lösen. Durch das Training kann das Netz eine Generalisierungsfähigkeit entwickeln, so dass eine Problemstellung unabhängig von den Trainingsbeispielen wird.

Das vielleicht größte Anwendungsgebiet für künstliche neuronale Netze ist die Mustererkennung. Ein entsprechen trainiertes Netz kann Muster innerhalb von Daten auffinden und Zusammenhänge zwischen den Mustern herstellen. Ein Teilgebiet der Mustererkennung stellt die Klassifikation von Daten dar. Ein entsprechend trainiertes neuronales Netz kann als Klassifikator genutzt werden und solche Klassifikationsaufgaben lösen. In dieser Arbeit werden neuronale Netze speziell für solche Aufgabenstellungen behandelt und trainiert.

2.1.1 Das künstliche Neuron

Das künstliche Neuron stellt die Grundeinheit von künstlichen neuronalen Netzen dar. Es erhält einen Inputvektor $i \in \mathbb{R}^D$ als Eingabe. Die Inputs werden anschließend gewichtet und aufsummiert. Auf diese gewichtete Summe wird anschließend eine nichtlineare Transferfunktion (auch Aktivierungsfunktion genannt) angewandt. Das daraus resultierende Ergebnis bildet den Output des Neurons. Die Ausgabe lässt sich somit wie folgt berechnen

$$o_{neuron} = \text{act}\left(\sum_{j=1}^{D} i_j w_j\right). \tag{2.1}$$

Dabei bezeichnet act die nichtlineare Transferfunktion und w_j das Gewicht zum Input i_j. Fasst man die Gewichte zu einem Vektor $w \in \mathbb{R}^D$ zusammen, lässt sich Gleichung (2.1) schreiben als Matrixmultiplikation

$$o_{neuron} = \text{act}\left(w^T \cdot i\right). \tag{2.2}$$

Die Transferfunktion bestimmt den Grad der Aktivierung des Neurons. Überschreitet die gewichtete Summe einen bestimmten Schwellwert, so wird das Neuron aktiviert. Aufgrund dieses Verhaltens, darf die Transferfunktion nicht linear sein und muss einen Schwellwert aufweisen. Typische Transferfunktionen sind:

- Die logistische Funktion

$$\text{fermi}(x) = \frac{1}{1 - e^{-x}} \tag{2.3}$$

- Der hyperbolische Tangens

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}} \tag{2.4}$$

In Abbildung 2.1 sind diese Funktionen abgebildet.

Der Schwellwert kann festgelegt werden, indem zur gewichteten Summe ein zusätzliches Gewicht w_b addiert wird. w_b bezeichnet man als Biasgewicht. Mit einem solchen Biasgewicht lässt sich Gleichung (2.2) umschreiben zu

$$o_{neuron} = \text{act}\left(w^T \cdot i + w_b\right). \tag{2.5}$$

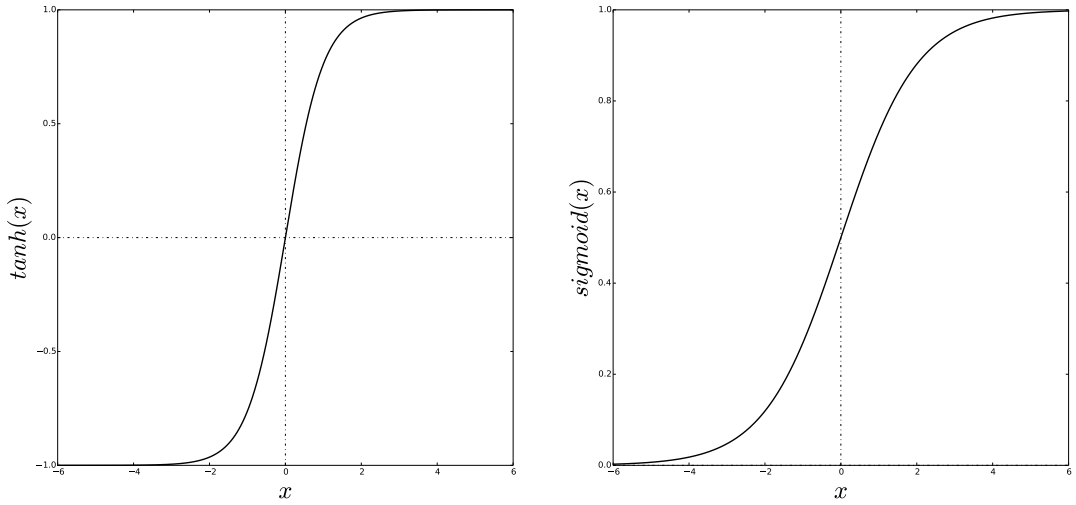


Abbildung 2.1: Die Aktivierungsfunktionen tanh und fermi

Ein künstliches neuronales Netz kann als gerichteter gewichteter Graph von einzelnen Neuronen aufgefasst werden. Dabei ist es zulässig, dass der Output o_j eines Neurons j als Input eines weiteren Neurons k verwendet wird. Das gesamte Netz erhält i als Inputvektor und generiert einen Outputvektor o.

2.1.2 Multi-Layer-Perzeptrons

Eine der bekanntesten Klassen von künstlichen neuronalen Netzen bildet das Multi-Layer-Perzeptron (MLP). Innerhalb eines solchen Netzes sind die einzelnen Neuronen schichtweise angeordnet. Innerhalb einer Schicht besteht keine Verbindung zwischen den einzelnen Neuronen. In jedem MLP existiert eine Input- und eine Outputschicht. Die Inputschicht stellt dem Netz den Eingabevektor i zur Verfügung und hat sonst keine Funktionsweise. Sie wird im Folgenden als erste Schicht des Netzes bezeichnet. Die Outputschicht berechnet den Outputvektor o des Netzes und stellt die letzte Schicht des Netzes dar.

Zwischen Input- und Outputschicht können sich mehrere verdeckte Schichten (Hidden Layer, HL) befinden. Bis auf die Inputschicht erhält jede weitere Schicht ihren Input aus der jeweils darüber liegenden Schicht. Für den Fall, dass ein MLP l Schichten besitzt, erhält die Schicht L_i als Input den Output der Schicht L_{i-1}. Die Inputschicht wird mit L_0 und die Outputschicht mit L_l bezeichnet. Dieses Verhalten bezeichnet man als vorwärtsgerichtet. Eine Schicht L_j kann somit keinen

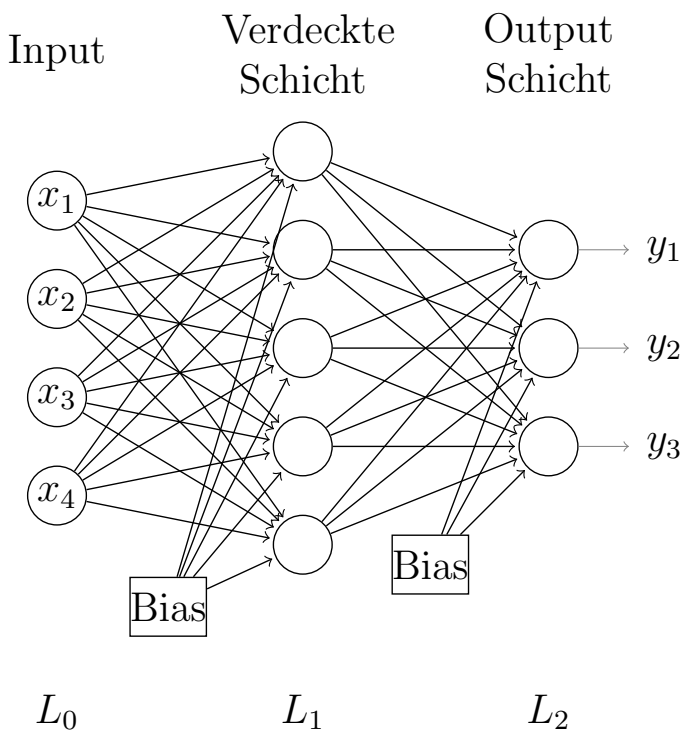


Abbildung 2.2: Beispiel einer MLP-Architektur. Das MLP besitzt vier Inputs und berechnet einen Outputvektor aus $\mathbb{R}^3$. Die verdeckte Schicht besitzt fünf Neuronen. Insgesamt beinhaltet das Netz 43 gerichtete Verbindungen mit ebenso vielen Gewichten.

Input aus tiefer liegenden Schichten L_k, mit $k > j$, erhalten. Eine Rückkopplung ist in einem solchen Netz nicht zulässig. Innerhalb einer Schicht verwendet jedes Neuron die selbe Transferfunktion.

Des Weiteren ist ein MLP vollverknüpft. Eine Vollverknüpfung bedeutet, dass der Input eines Neurons in L_j aus allen Outputs der Neuronen aus der Schicht L_{j-1} besteht. Dies gilt für alle Neuronen in L_j. Somit erhält jedes Neuron aus L_j einen identischen Inputvektor, dessen Komponenten aus den Outputs von L_{j-1} besteht. Es existiert also für jedes Neuron aus L_{i-1} eine gewichtete Verbindung zu allen Neuronen in L_i.

Ein MLP ist somit ein schichtweise aufgebautes, vollverknüpftes und vorwärtsgerichtetes neuronales Netz. In Abbildung 2.2 ist ein Beispiel dargestellt.

Innerhalb einer Schicht kann man jedem Neuron ein Biasgewicht zuordnen. Dazu erhält eine Schicht ein zusätzliches Input-Neuron, welches immer den Wert 1 liefert. Die Verbindungen, welche von diesem Biasneuron ausgehen, können dann mit den

Biasgewichten versehen werden, welche in einem Biasvektor b zusammengefasst werden. Zu beachten ist, dass ein Biasneuron keinen Input besitzt.

Angenommen in L_i befinden sich n und in L_{i+1} m Neuronen, dann gibt es zwischen diesen beiden Schichten $n \cdot m$ Verbindungen mit genauso vielen Gewichten. Bezeichne o_{L_j} den Outputvektor von L_j. Ordnet man die Gewichte zwischen diesen Schichten in einer Matrix $W \in \mathbb{R}^{m \times n}$ an, so berechnet sich der Outputvektor von L_{j+1} wie folgt

$$o_{L_{j+1}} = \text{act}\left(W \cdot o_{L_j} + b\right). \tag{2.6}$$

Dabei wird die Transferfunktion elementweise auf $W \cdot o_{L_j}$ angewendet.

2.1.3 Konvolutionsnetze

Der Aufbau und die Funktionsweise von Konvolutionsnetzen sind stark inspiriert von biologischen visuellen Systemen. Da in dieser Arbeit solche Netze auf Bilder angewendet werden, wird im Folgenden immer davon ausgegangen, dass der Input eines Netzes aus einem rechteckigen Bild mit $p \times p$ Pixeln besteht. Analog zu MLPs ist ein Konvolutionsnetz schichtartig aufgebaut und vorwärtsgerichtet. Die Netztypen unterscheiden sich jedoch in der Verarbeitung ihrer Eingaben und im Umgang mit ihren Gewichten. Eine Schicht innerhalb eines Konvolutionsnetzes verarbeitet Informationen in der Regel in zwei Arbeitsschritten:

1. Konvolutionsoperation: Bildeigenschaften werden anhand von Filtern extrahiert und innerhalb eines gefilterten Outputbildes gespeichert.

2. Poolingoperation: Wird eine Bildeigenschaft gefunden, so liefert ihre exakte Position innerhalb des Inputs keine wichtigen Informationen. Für Klassifikationsaufgaben ist nur die relative Position zwischen gefundenen Bildeigenschaften von Bedeutung. Eine Poolingoperation wird zur Auflösung lokaler Strukturen verwendet.

Eine Konvolutionsoperation wird genutzt, um Bildeigenschaften zu extrahieren. Bildeigenschaften werden im Folgenden Features genannt. Dazu werden Filter benötigt, die ein entsprechendes Feature erkennen können. Ein solcher Filter kann dann auf jeden Bereich des Inputbildes angewandt werden. Typische Filtergrößen sind 3×3, 5×5 oder 7×7.

Sei die Größe eines Filters gegeben durch $k \times k$ mit $k < p$, dann kann dieser Filter auf jede $k \times k$-Nachbarschaft des Inputbildes angewandt werden und erzeugt somit ein gefiltertes Outputbild, welches Featuremap genannt wird. Die entsprechende Nachbarschaft im Inputbild wird rezeptives Feld genannt. Neuronen einer Konvolutionsschicht sind somit innerhalb von Featuremaps angeordnet. Ein Neuron innerhalb einer Featuremap erhält nur Eingaben aus einem seiner Position entsprechenden rezeptiven Feld im Inputbild. Verbindungsgewichte eines Neurons zu einem solchen Feld bilden die Filter. Diese Beziehung zwischen Input und Featuremap führt direkt zum Prinzip des Weight-Sharing. Jede Featuremap besitzt k^2-Gewichte. Die informationsverarbeitenden Einheiten innerhalb einer Featuremap nutzen alle die gleichen Gewichte. Die Inputs aus einem rezeptiven Feld und die Gewichte werden dann durch eine gewichtete Summe und eine Transferfunktion zu einem Eintrag in der Featuremap verarbeitet.

Eine Konvolutionsschicht kann mehrere Featuremaps erzeugen, dazu besitzt jede Featuremap ihren eigenen Filter. Die Inputs eines Neurons müssen nicht nur auf ein Bild beschränkt sein. In der Regel bezieht ein Neuron seine Inputs aus mehreren Bildern (oder Featuremaps) der darüber liegenden Schicht. Auch hier muss für jedes Inputbild ein eigenständiger Filter definiert werden. Fasst man die Filter einer Featuremap zusammen, so ergibt sich eine Matrix mit $k \times k \times N$ Einträgen. Dabei bezeichnet N die Anzahl der Inputbilder eines jeden Neurons innerhalb einer Featuremap. Der Eintrag (x, y) der Featuremap j innerhalb der Schicht L_i wird somit berechnet durch

$$o_{x,y}^{(i,j)} = \text{act}\left(b^{(i,j)} + \sum_{n=0}^{N-1}\sum_{a=0}^{k-1}\sum_{b=0}^{k-1} w_{a,b}^{(i,j,n)} \cdot o_{x+a,x+b}^{(i-1,n)}\right). \tag{2.7}$$

Dabei bezeichnet $w_{a,b}^{(i,j,n)}$ das Gewicht an der Position a, b, n in der Gewichtsmatrix der j-ten Featuremap innerhalb der Schicht L_i und $b^{(i,j)}$ das Biasgewicht der Featuremap j. Für den Fall, dass ein Inputbild aus $p \times p$ Pixeln besteht, besitzt eine resultierende Featuremap nach der Konvolution eine Größe von $(p-k+1) \times (p-k+1)$ Pixel.

Für Klassifikationsaufgaben ist es oft sinnvoll, eine Poolingoperation nach der Konvolution auszuführen. Poolingoperationen werden genutzt, um lokale Strukturen aufzulösen und die Größe von Featuremaps zu reduzieren. Hierzu werden rezeptive Felder einer Featuremap verrechnet, indem man beispielsweise das Maximum oder das Mittel aus den entsprechenden Pixelwerten ermittelt. Bezeichne

$k_{pool} \times k_{pool}$ die Größe eines Poolingbereichs innerhalb einer Featuremap und bezeichne $r = (r_{i,j})_{i,j \leq k_{pool}}$ ein entsprechend großes rezeptives Feld. Die in dieser Arbeit benutzte Poolingoperation ist das max-Pooling. Ihr Output wird berechnet durch

$$\text{max-pool}(r) = \max_{i,j}(r_{i,j}). \tag{2.8}$$

Eine weitere weit verbreitete Poolingoperation ist das Average-Pooling

$$\text{avg-pool}(r) = \frac{1}{k_{pool}^2} \sum_{i=0}^{k_{pool}-1} \sum_{j=0}^{k_{pool}-1} (r_{i,j}). \tag{2.9}$$

Rezeptive Felder können sich überlappen. Dies gilt sowohl für Konvolutions- als auch für Poolingoperationen. Die Überlappungsgröße wird Stride genannt und gibt die Anzahl der sich nicht überlappenden Pixel in horizontaler und vertikaler Richtung an. In dieser Arbeit werden die gesamten Bildinformationen eines Inputs ausgenutzt, indem die rezeptiven Felder für Konvolutionen um höchstens ein Pixel in jeder Richtung verschoben werden.

Man fasst eine Konvolutions- und eine Poolingoperation zusammen in einer Konvolutionsschicht. Für das Lösen von Klassifikationsaufgaben ist es oft sinnvoll, vor der Outputschicht eine oder mehrere vollverknüpfte Schichten einzufügen. Sie verhalten sich analog zu verdeckten Schichten von MLPs.

Ein großer Vorteil von Konvolutionsnetzen gegenüber MLPs ist das Weight-Sharing-Prinzip. Es erlaubt die Verwendung einer Vielzahl von Neuronen bei einer verhältnismäßig kleinen Anzahl von Gewichten.

Goodfellow u. a. (2013) haben gezeigt, dass die maxout-Funktion sich besonders als Transferfunktion innerhalb von Konvolutionsschichten eignet. Produziere eine solche Schicht n verschiedene Featuremaps und bezeichne $h_{i,j}$ die gewichtete Summe eines Neurons i innerhalb der j-ten Featuremap. Die maxout-Funktion erhält als Eingabe $q \leq n$ Featuremaps $f_1, \ldots, f_q$ und fasst diese zusammen. Sie liefert eine neue Featuremap deren Einträge sich nach folgender Vorschrift berechnen lassen

$$\text{maxout}(h_{i,1}, \cdots, h_{i,q}) = \max_{j \in [1,q]}(h_{i,j}). \tag{2.10}$$

Die Arbeitsweise von maxout kann als Pooling entlang der produzierten Featuremaps angesehen werden.

2.2 Training von Neuronalen Netzen

Prinzipiell können neuronale Netze mit einer verdeckten Schicht und einer ausreichend hohen Anzahl an Neuronen jede Funktion approximieren. Dabei spielt nur die Belegung der Gewichte eine Rolle für eine erfolgreiche Approximation. In der Regel verfügen diese Netze über eine sehr hohe Anzahl an Gewichten (das kleinste in dieser Arbeit verwendete Netz nutzt 95410 Gewichte). Eine analytische Berechnung der geeigneten Gewichte ist somit nicht realisierbar. Es gibt jedoch Verfahren, die es einem solchen Netz erlauben, selbstständig seine Gewichte entsprechend seiner Aufgabenstellung einzustellen.

2.2.1 Überwachtes Lernen

Das Lernen von Klassifikationsaufgaben erfolgt typischerweise durch die Verwendung von Trainingsbeispielen. Eine Menge von Trainingsbeispielen wird Trainingsmenge T und die darin enthaltenen Elemente werden Trainingsmuster p genannt. Ein Trainingsmuster ist ein Paar $p = (x, \hat{y})$, bestehend aus einem Inputvektor x und einem Soll-Outputvektor $\hat{y}$, welcher im Folgenden Teacher genannt wird. Die Grundidee zum Trainieren eines neuronalen Netzes lässt sich in drei Schritte aufteilen:

1. Nutze x als Input des Netzes und bestimme den daraus resultierenden Output $y(x)$
2. Berechne die Abweichung zwischen $y(x)$ und $\hat{y}$
3. Passe die Gewichte des Netzes so an, dass diese Abweichung minimiert wird.

Ein solches Lernverfahren, dass externe Teacher verwendet, wird überwachtes Lernen genannt. Werden die Teacher nicht explizit angegeben (sondern beispielsweise während des Trainings berechnet) spricht man von unüberwachtem Lernen. In dieser Arbeit werden nur überwachte Lernvorgänge verwendet.

Im Folgenden wird immer angenommen, dass Teacher, Vektoren aus $\{0, 1\}^n$ sind mit $\sum_{i=1}^{n} y_i = 1$. Ein solcher Vektor wird auch als 1-aus-n Kodierung bezeichnet. Die durch diesen Vektor repräsentierte Klasse kann durch diejenige Position im Teacher identifiziert werden an der sich die 1 befindet.

Die Abweichung zwischen Teacher und tatsächlichem Netzoutput wird anhand einer Fehlerfunktion bestimmt. Als Lernen bezeichnet man nun die Modifikation

der Gewichte eines Netzes, so dass der Fehler minimiert wird. Zur Vereinfachung der Schreibweise werden alle Gewichte des Netzes in einem Gewichtsvektor w zusammengefasst. Bezeichne dazu E die Fehlerfunktion, welche alle Gewichte w und die Trainingsmenge $T = \{p_1, \ldots, p_d\}$ als Eingabe erhält, und sei $y, \hat{y} \in \mathbb{R}^n$. Die wohl bekannteste Fehlerfunktion ist die mittlere quadratische Abweichung (mean squared Error, MSE)

$$E_{MSE}(w, T) = \sum_{i=1}^{d} \sum_{j=1}^{n} (y_j(x_i) - \hat{y}_{i,j}). \tag{2.11}$$

Es hat sich gezeigt, dass für Klassifikationsaufgaben die Verwendung der Cross-Entropy Fehlerfunktion (CE) effizienter ist (Golik u. a., 2013). Diese Fehlerfunktion ist definiert als

$$E_{CE}(w, T) = -\sum_{i=1}^{d} \sum_{j=1}^{n} \left(\hat{y}_{i,j} \cdot \log \left(\frac{\exp(y_j(x_i))}{\sum_{k=1}^{n} \exp(y_k(x_i))} \right) \right) \tag{2.12}$$

Dabei kann der logarithmische Ausdruck als Transferfunktion der Outputschicht verwendet werden. Diese Funktion wird Softmax genannt und ist definiert als

$$\underset{j}{\text{softmax}}(x_i) = \frac{\exp(y_j(x_i))}{\sum_{k=1}^{n} \exp(y_k(x_i))}. \tag{2.13}$$

Innerhalb dieser Arbeit wird immer der zur Cross-Entropy ähnliche multinomiale logistische Fehler verwendet. Dieser berechnet sich durch

$$E_{ML}(w, T) = -\sum_{i=1}^{d} \left(\sum_{j=1}^{n} \hat{y}_{i,j} y_j(x_i) - \log \left(\sum_{j=1}^{n} \exp(y_j(x_i)) \right) . \right) . \tag{2.14}$$

Ist die Trainingsmenge bekannt, so wird im Folgenden auch $E(w)$ statt $E(w, T)$ geschrieben. Zu beachten ist, dass alle hier vorgestellten Fehlerfunktionen differenzierbar sind, und sich somit für einen Gradientenabstieg eignen.

Wird eine der hier vorgestellten Fehlerfunktionen für das Training eines neuronalen Netzes mit mindestens einer verdeckten Schicht verwendet, so ist die Funktion nicht-konvex. Das entsprechende Netz wird dann auch nicht-konvexes Modell genannt.

2.2.2 Der Gradientenabstieg

Für das Training eines Modells muss das Ziel sein, eine Gewichtskonfiguration w^* zu finden, an der sich ein Minimum der Fehlerfunktion befindet

$$w^* = \arg\min_w(E(w)). \tag{2.15}$$

Aufgrund der Vielzahl der Parameter in w ist eine analytische Berechnung von w^* in der Regel nicht realisierbar. Das Minimum der Fehlerfunktion muss approximativ berechnet werden. Das klassische Verfahren zur approximativen Suche eines solchen Minimums ist der Gradientenabstieg (Gradient Descent, GD) auf der Oberfläche von E. Dazu startet man an einer zufälligen Position auf der Oberfläche und verändert den Gewichtsvektor in negativer Richtung des Gradienten von E. Der Gradient $\nabla E(w)$ wird im Folgenden, wenn nicht anders angegeben, mit g und seine i-te Komponente mit g_i bezeichnet. Die Richtung des Gradientenabstiegs ist durch $-g$ gegeben. Die grundlegende iterative Vorschrift zur Gewichtsänderung ist gegeben durch

$$w_t = w_{t-1} - \eta g_{t-1}. \tag{2.16}$$

Dabei bezeichnet w_t den Gewichtsvektor zum Iterationszeitpunkt t und g_{t-1} den Gradienten der Fehlerfunktion ausgewertet an w_{t-t}. Weiterhin bezeichnet η die Lernrate. Sie skaliert die Gewichtsänderung in jedem Iterationsschritt.

Für den Zeitpunkt $t = 0$ wird die Startposition auf der Fehleroberfläche durch eine zufällige Initialisierung w_0 festgelegt. Glorot und Bengio (2010) schlagen vor, die Gewichte, welche Schicht L_i mit der Schicht L_{i+1} verbinden, zufällig normalverteilt aus dem Intervall

$$\left[-\frac{\sqrt{6}}{\sqrt{n_i + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{j+1}}}\right] \tag{2.17}$$

zu ziehen. Dabei bezeichnet n_i bzw. n_{i+1} die Anzahl an Neuronen in L_i und L_{i+1}.

Eine große Herausforderung im Umgang mit solchen Lernverfahren ist die Bestimmung der Lernrate η. Sie muss vor Trainingsbeginn manuell festgelegt werden. Solche Parameter, die nicht während des Trainings durch den Lernalgorithmus selbstständig eingestellt werden, sondern manuell vorgegeben werden müssen, werden als Hyperparameter bezeichnet. In der Regel reagieren solche Gradienten-

abstiegsverfahren sehr sensibel auf die Wahl der Lernrate. Der einfachste Ansatz für deren Belegung ist, η mit einem konstanten skalaren Wert zu belegen. Belegt man η mit einem geeigneten Wert, so konvergiert w für $t \to \infty$ gegen eine optimale Gewichtskonfiguration w^*.

Im Allgemeinen benötigen solche Gradientenabstiegsverfahren eine Vielzahl an Iterationen bis sich eine ausreichende Optimierung der Gewichtsparameter einstellt. Um eine schnelle Konvergenz zu einem Optimum zu erhalten, ist es dringend notwendig, gute Lernraten zu verwenden und somit jeden Schritt auf der Fehleroberfläche optimal zu skalieren.

2.2.3 Präsentation und Verarbeitung der Muster

Trainingsmuster können einem Netz während des Trainings auf verschiedene Arten präsentiert werden. Man unterscheidet drei Präsentationsmöglichkeiten:

- Onlinelearning:
  Jedes Muster wird einzeln präsentiert und verarbeitet. Eine Gewichtsanpassung erfolgt, nachdem genau ein Muster dem Netz präsentiert wird. Ein Nachteil dieser Musterverarbeitung ist, dass eine parallele Implementation durch die Verwendung von geeigneten Grafikkarten nicht ausgereizt werden kann.

- Batchlearning:
  Eine Gewichtsänderung wird nicht nach jeder Musterpräsentation durchgeführt, sondern erst, sobald alle Muster dem Netz präsentiert wurden. Dieses Vorgehen kann vor allem bei einem begrenzten Speicher problematisch werden. Zu jedem Muster muss der ihm entsprechende Gradient im Speicher verbleiben, bis alle Muster abgearbeitet wurden und die Gewichtsänderung vollzogen werden konnte.

- Minibatchlearning:
  Die Minibatchvariante bietet einen Kompromiss zwischen Online- und Batchlearning. Dabei wird die Trainingsmenge in gleichgroße Teile aufgeteilt, den sogenannten Minibatches. Eine Gewichtsänderung wird ausgeführt, sobald ein Minibatch dem Netz präsentiert wurde.

Wie bereits ausgeführt, reizt eine Onlinepräsentation der Trainingsmuster die Rechenleistung von GPUs nicht aus. Dagegen benötigt die Batchvariante zu viel

Speicherplatz bei der Verwendung von großen Trainingsmengen, die zusätzlich noch aus großen Mustern bestehen kann. Der Speicher von modernen Grafikkarten ist in der Regel stark begrenzt, so dass ein Batchverfahren sehr schnell an seine physischen Grenzen stößt. Die Verwendung von Minibatches eignet sich speziell bei parallel arbeitenden GPU-Implementierungen. Jedoch bildet die Größe der einzelnen Minibatches einen Hyperparameter, der für jeden Lernalgorithmus einzustellen ist.

2.2.4 Lernraten

Gradientenbasierte Lernverfahren, welche den Gradienten mit einer Lernrate skalieren, sind sehr sensibel gegenüber dieser. Sie trägt maßgeblich zum Lernerfolg bei. Viele Verfahren zur Bestimmung der Lernrate basieren auf der quadratischen Approximation der Fehlerfunktion (z.B. Bishop u. a. (1995) und LeCun, Bottou, Orr u. a. (1998)). Diese ergibt sich durch die Expansion von E mit Hilfe der Taylor-Reihe um den Entwicklungspunkt $\overline{w}$

$$E(w) \approx E(\overline{w}) + (w - \overline{w})g(\overline{w}) + \frac{1}{2}(w - \overline{w})^T H_{\overline{w}}(w - \overline{w}) \tag{2.18}$$

wobei $H_{\overline{w}}$ die Hessematrix ausgewertet am Punkt $\overline{w}$ bezeichnet. Daraus folgt für den Gradienten

$$g = g(\overline{w}) + H_{\overline{w}}(w - \overline{w}). \tag{2.19}$$

Bezeichne w^* den Punkt, an dem E ihr Minimum annimmt. Für einen solchen Entwicklungspunkt gilt

$$E(w) = E(w^*) + \frac{1}{2}(w - w^*)^T H_{w^*}(w - w^*). \tag{2.20}$$

Zu beachten ist, dass in diesem Fall $g(w^*) = 0$ gilt.

An eine Lernrate werden die folgenden Anforderungen gestellt:

- η muss groß genug sein, um eine schnelle Konvergenz zu erreichen. Für den Fall, dass w in nur einem Updateschritt nach w^* überführt wird, spricht man von einer optimalen Lernrate η^*. Wird die Lernrate mit einem zu kleinen Wert belegt, kann das Training eine lange Zeit in Anspruch nehmen.

- η darf nicht zu groß sein, um eine Divergenz zu vermeiden. Zu große Lernra-

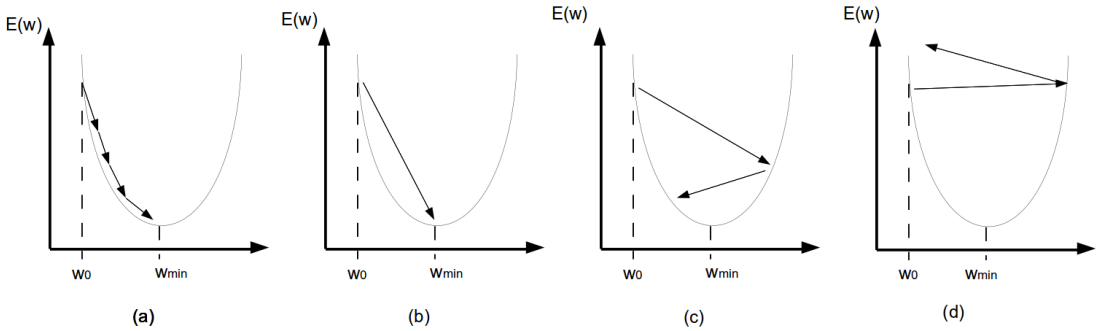


Abbildung 2.3: Konvergenzverhalten bei verschiedenen Lernraten auf einer quadratischen Fehlerfunktion im eindimensionalen Parameterraum. (a) $\eta < \eta^*$ Konvergenz stellt sich erst nach vielen Iterationsschritten ein. (b) Optimale Lernrate. (c) und (d) $\eta^* < \eta$ Minimum wird übersprungen oder eine Divergenz setzt ein.

ten führen dazu, dass durch einen Updateschritt ein akzeptables Minimum auf der Fehleroberfläche übersprungen wird, oder dass sich w immer weiter von w^* entfernt.

Abbildung 2.3 macht diese Anforderungen deutlich. Eine ausführliche Analyse ist in LeCun, Bottou, Orr u. a. (1998) zu finden. Konstante skalare Lernraten können diese Anforderungen nicht garantieren. Effektiver ist es, für jedes Gewicht eine individuelle Lernrate zu bestimmen.

Ausgehend von Gleichung (2.18) kann festgestellt werden, dass sich optimale, individuelle Lernraten durch die Eigenwerte λ der Hessematrix bestimmen lassen

$$\eta_i^* = \frac{1}{\lambda_i}. \tag{2.21}$$

Ein ausführlicher Beweis ist in (LeCun, Bottou, Orr u. a., 1998) zu finden.

Die Verwendung der Hessematrix ist jedoch äußerst kostenintensiv und die Berechnung solcher Lernraten nicht effizient realisierbar. Wünschenswert ist es, ein Verfahren zu entwickeln, dass die Eigenwerte von H ohne die explizite Berechnung von H liefert. Dies ist jedoch nur eingeschränkt möglich. LeCun, Simard u. a. (1993) haben ein Verfahren entwickelt, welches zumindest den größten Eigenwert von H approximiert.

2.3 Regularisierung

Durch das Training eines neuronalen Netzes sollen die Gewichte so eingestellt werden, so dass ein Netz auch auf nicht präsentierte Daten einen minimalen Fehler liefert. Eine solche Eigenschaft wird Generalisierungsfähigkeit genannt. Um die Generalisierungseigenschaften während des Trainings zu kontrollieren, wird ein Teil der Trainingsmenge zur Validierung genutzt. Die Muster innerhalb der Validierungsmenge werden dem Netz nicht für das Training zur Verfügung gestellt. Sie dienen nur zur Überprüfung des Lernforschritts auf einem unbekannten Datensatz. Eine solche Überprüfung kann beispielsweise immer nach fünf Trainingsepochen erfolgen. Man unterscheidet somit während des Trainings zwischen zwei Arten von Fehlern:

- Der Trainingsfehler: Dieser berechnet sich durch eine Auswertung der Fehlerfunktion auf den Trainingsmustern.
- Der Validierungsfehler: Dieser wird zur Überprüfung der Generalisierungseigenschaften des Netzes genutzt. Dazu werden dem Netz unbekannte Daten präsentiert anhand derer die Fehlerfunktion ausgewertet wird.

Es ist nicht zwingend erforderlich, dass beide Fehler die selbe Fehlerfunktion nutzen. Vor allem für Klassifikationsaufgaben ist es hilfreich, eine Fehlerfunktion zu nutzen, welche den prozentualen Anteil der Muster berechnet, welche nicht korrekt klassifziert werden können. Eine solche Funktion ist in der Regel nicht differenzierbar und kann nicht für das Training mit Gradientenabstiegsverfahren genutzt werden. Jedoch kann sie genutzt werden, um den Klassifikationsfehler CERR auf der Validierungsmenge zu bestimmen. Ziel des Lernens muss somit sein, das Netz auf der Trainingsmenge mit Hilfe des Trainingsfehlers zu trainieren und das Lernen zu beenden, sobald sich ein minimaler Validierungsfehler eingestellt hat.

Zusätzlich kann eine weitere Teilmenge der Trainingsmuster zur Trainingsbewertung nach abgeschlossenem Training genutzt werden. Sie wird als Testmenge bezeichnet.

Während des Trainings kann es zu Diskrepanzen zwischen Trainingsfehler und Validierungsfehler kommen. Dabei kann der Fall auftreten, dass der Trainingsfehler zwar minimiert wird, jedoch der Validierungsfehler, ab einem bestimmten Zeitpunkt, zu steigen beginnt. Dieses Phänomen wird als Overfiting bezeichnet. Zur Vermeidung und zur Kompensation des Overfitings gibt es zahlreiche Möglichkeiten.

Overfiting ist ein Indiz, dass ein Netz die präsentierten Muster auswendig gelernt hat und auf neue Daten nicht ausreichend reagieren kann. Um dem entgegenzuwirken, sollte man ein Netz nicht zu groß wählen. Kleinere Netze besitzen weniger Gewichte und können deshalb auch weniger Muster auswendig lernen.

Eine weitere Möglichkeit das Netz zu beschränken, bietet die Regularisierung (z.B. Moody u. a. (1995)). Unter Verwendung eines Regularisierungsterms in der Fehlerfunktion kann man verhindern, dass Gewichte zu groß angepasst werden. Die Fehlerfunktion muss in diesem Fall wie folgt angepasst werden

$$E_{reg}(w) = E(w) + \lambda\phi(w). \tag{2.22}$$

Dabei bezeichnet ϕ den Regularisierungsterm. Mit Regularisierung ändert sich Gleichung (2.16) zu

$$w_t = w_{t-1} - \eta g_{t-1} - \lambda\phi(w). \tag{2.23}$$

Populäre Regularisierungsterme sind die l_1-Norm und die l_2-Norm. Unter Verwendung dieser Normen spricht man auch von

- l_1-Regularisierung mit

$$\phi_{l_1}(w) = \|w\|_1 = \sum_i |w_i|. \tag{2.24}$$

- l_2-Regularisierung mit

$$\phi_{l_2}(w) = \frac{1}{2}\|w\|_2 = \sum_i w_i^2. \tag{2.25}$$

2.3.1 Early-Stopping

Um ein Overfiting zu umgehen, kann man während des Lernens das Early-Stopping als Abbruchbedingung verwenden (z.B. Prechelt (1998)). Dieses bricht das Training ab, sobald sich während des Trainings keine signifikante Verbesserung des Validierungsfehlers mehr einstellt. Im Fall eines Overfitings kann das Training abgebrochen werden, sobald sich der Validierungsfehler verschlechtert. Problematisch bei der Verwendung des Early-Stoppings ist, dass nach einer Verschlechterung des Validierungsfehlers dieser auch wieder fallen kann. Bei der Implementierung eines Early-Stoppers müssen daher einige Parameter beachtet werden.

Zunächst muss definiert werden wie viele Epochen zwischen zwei Auswertungen auf der Validierungsmenge liegen dürfen. Erfahrungen aus dieser Arbeit haben gezeigt, dass fünf Epochen ein akzeptabler Wert für diese Epochenanzahl ist. Dem Training muss zu Beginn eine anfängliche Early-Stopping Toleranz eingeräumt werden, das heißt das Netz wird zu Lernbeginn eine gewisse Anzahl an Epochen trainiert, bis ein Abbruch des Trainings erfolgen darf. Wird während dieser Zeit eine Verbesserung des Validierungsfehlers festgestellt, wird diese Zeit mit einem Faktor multipliziert. Nun kann dem Netz diese neue Zeit gewährt werden, bis ein Abbruch erfolgen kann. Dieses Verhalten kann dann sooft wiederholt werden, bis sich der Validierungsfehler nicht mehr verbessert. Der Faktor mit dem die Epochenanzahl multipliziert wird, muss vor Trainingsbeginn vordefiniert werden. Zusätzlich muss festgelegt werden, wann von einer signifikanten Verbesserung des Validierungsfehlers gesprochen werden darf. Hierzu wird ein fester Prozentsatz festgelegt, um den sich der Validierungsfehler seit der letzten Auswertung ändern muss. Wird dieses Kriterium innerhalb der aktuell gewährten Zeit nicht erfüllt, so wird das Training abgebrochen.

2.3.2 Dropout

Eine weitere Methode zur Regularisierung bietet Dropout (G. E. Hinton u. a., 2012; Srivastava u. a., 2014). Dieser Ansatz liefert eine Möglichkeit, ein Overfiting zu verhindern und gleichzeitig exponentiell viele verschiedene Netze miteinander zu kombinieren. Dabei werden innerhalb einer Schicht im Netz Neuronen unabhängig voneinander mit einer Wahrscheinlichkeit p ausgeschaltet. p wird dabei als Dropoutrate bezeichnet. Genauer werden während der Bearbeitung einer Gewichtsänderung sowohl die entsprechenden Neuronen, als auch ihre gewichteten Verbindungen aus dem Netz entfernt. Dadurch wird die Kapazität des Netzes verringert. Für Neuronen innerhalb von verdeckten Schichten hat sich eine Dropoutrate von 0,5 und innerhalb einer Inputschicht von 0,2 bewährt.

Da Neuronen zufällig ausgeschaltet werden, ist dies gleichzusetzen mit einem Training von exponentiell vielen verschiedenen Netzen. Alle Netze teilen sich dabei die eingeschalteten Gewichte. Da es sich um exponentiell viele verschiedene Netze handelt, wird jedes Netz nur „ein wenig“trainiert.

Problematisch bei der Verwendung von Dropout ist die Bewertung des Trainings auf der Validierungsmenge. Hierbei müssen alle trainierten Netze wieder zusammengeführt werden. Dazu werden im Ausgangsnetz alle Neuronen eingeschaltet

und die Gewichte werden während der Validierung mit der ihnen entsprechenden Dropoutrate multipliziert. Es hat sich gezeigt, dass durch die Verwendung von Dropout der Validierungsfehler eines Netzes verringert werden kann. Auf diese Weise trainierte Netze verfügen über gesteigerte Generalisierungseigenschaften.

3 Verwandte Arbeiten

Im folgenden Kapitel wird ein Überblick über verwandte Arbeiten gegeben. Im Fokus stehen dabei Algorithmen zum Training von neuronalen Netzen, welche anpassungsfähige, individuelle Lernraten verwenden.

3.1 Gradientenabstieg mit Momentum

Der bereits in Kapitel 2 angesprochene Gradientenabstieg berechnet in jedem Iterationsschritt Gewichtsänderungen

$$\Delta w_{t+1} = -\eta g_t \tag{3.1}$$

und wendet diesen Wert auf das aktuelle Gewicht an (Gleichung (2.16)). In der Regel benötigt dieses Verfahren eine große Anzahl an Iterationen, bis ein akzeptables Minimum erreicht wird. Der Algorithmus kann beschleunigt werden, indem man für die Berechnung von Δw_{t+1} Informationen aus früheren Gewichtsänderungen verwendet (Polyak, 1964). Die Gewichtsänderung berechnet sich dann aus

$$\Delta w_{t+1} = -\eta g_t + \mu \Delta w_t. \tag{3.2}$$

Dabei wird $\mu \Delta w_t$ als Momentum-Term bezeichnet. Die Momentum-Konstante μ skaliert dabei die Gewichtsänderung aus dem vorherigen Iterationsschritt. Die Idee dahinter ist, dass falls $-g_t$ und Δw_t in die gleiche Richtung zeigen, der Gradientenabstieg beschleunigt wird.

Eine Momentum-Methode mit besseren Konvergenzeigenschaften bietet das Nesterov-Momentum (Nesterov, 1983). Hierbei wird ein Gewicht mit Hilfe eines Zwischenschritts nach folgender Vorschrift geändert

$$v_t = w_t - \eta g_t \tag{3.3}$$

$$w_{t+1} = v_t + \mu(v_t - v_{t-1}). \tag{3.4}$$

Der hierdurch erzeugte Gradientenabstieg wird als Nesterov-beschleunigt bezeichnet (Nesterov accelerated gradient descent, NAG).

Durch die Beschleunigung des Gradientenabstiegs mit Hilfe eines Momentums, muss der zusätzliche Hyperparameter μ berücksichtigt werden.

3.2 Optimierungsverfahren zweiter Ordnung

Approximiert man die Fehlerfunktion E durch die Taylor-Reihe entsprechend Gleichung (2.18) erhält man eine lokale quadratische Approximation an einem Entwicklungspunkt $\overline{w}$. An einem Minimum dieser Funktion gilt für den Gradienten

$$g(\overline{w}) + H_{\overline{w}}(w - \overline{w}) = 0. \tag{3.5}$$

Dies gilt für

$$w = \overline{w} - H_{\overline{w}}^{-1} g(\overline{w}). \tag{3.6}$$

Bildet man nun wieder an w die lokale quadratische Approximation und berechnet das entsprechende Minimum erhält man die folgende iterative Vorschrift

$$w_{t+1} = w_t - H_{w_t}^{-1} g(w_t) \tag{3.7}$$

welche als Newton-Methode bekannt ist. Dabei wird $p_{t+1} = -H_{w_t}^{-1} g(w_t)$ auch Newton-Richtung genannt. Im Gegensatz zum bereits vorgestellten Gradientenabstieg nutzt diese Methode nicht nur Informationen aus dem Gradienten, sondern auch die Informationen aus den partiellen Ableitungen zweiter Ordnung in Form der Hessematrix H. Lernverfahren, welche die Einträge der Hessematrix zur Berechnung einer Gewichtsänderung benötigen, nennt man Lernverfahren zweiter Ordnung.

Die Verwendung der Hessematrix kann eine schnellere Konvergenz zu einem Minimum bewirken. Jedoch gibt es im Umgang mit der Hessematrix einige Probleme. Zunächst benötigt sie einen Speicherplatz der Größenordnung $O(n^2)$ (wobei n die Anzahl der Einträge im Gewichtsvektor w bezeichnet). Sie muss zusätzlich invertiert werden, wozu eine Laufzeit von etwa $O(n^3)$ benötigt wird. Des Weiteren muss sichergestellt werden, dass sie positiv definit ist, was nicht garantiert werden kann. Da in der Regel die Anzahl an Gewichten innerhalb eines neuronalen Netzes hoch

ist, ist die Verwendung der Newton-Methode sehr kostenintensiv.

Eine Möglichkeit, um die direkte Verwendung von H zu umgehen, bietet die Approximation der Hessematrix. Solche Verfahren, die statt H eine Approximation verwenden, werden Quasi-Newton-Methoden genannt. Der bekannteste Vertreter dieser Methoden ist das BFGS- bzw. L-BFGS-Verfahren (benannt nach Broyden - Fletcher - Goldfarb - Shanno). Hierbei wird H in jedem Iterationsschritt nur durch Verwendung der ersten partiellen Ableitungen approximiert. Bezeichne dazu B die Approximation von H. Nachdem eine Gewichtsänderung entlang der Richtung $p_t = -B_{w_t}^{-1} g(w_t)$ vollzogen wurde, wird B iterativ nach folgender Vorschrift verändert

$$B_{w_{t+1}} = (I - \rho_t s_t y_t^T) B_{w_t} (I - \rho_t y_t s_t^T) + \rho_t s_t s_t^T, \tag{3.8}$$

dabei bezeichnet I die Einheitsmatrix, mit

$$s_t = w_{t+1} - w_t \tag{3.9}$$

$$y_t = g(w_{t+1}) - g(w_t) \tag{3.10}$$

$$\rho_t = (y_t^T s_t)^{-1}. \tag{3.11}$$

Für eine hohe Anzahl an Gewichten ist jedoch auch die Verwendung von B sehr speicherintensiv. Abhilfe schafft da das L-BFGS-Lernverfahren (Limited Memory BFGS). Hierbei wird nicht die gesamte Matrix B gespeichert, sondern nur die Paare (s_t, y_t) innerhalb eines bestimmten Zeitfensters.

Eine weitere Möglichkeit, um die Verwendung der Hessematrix zu umgehen, bietet die Approximation von H durch die Jakobimatrix, welche bei der Gauss-Newton-Methode zum Einsatz kommt, oder der Einsatz des Conjugate-Gradient-Algorithmus, um die Newton-Richtung zu approximieren (beispielsweise in Martens (2010) und Wiesler u. a. (2013)).

Obwohl sich diese Arbeit nur auf Lernverfahren erster Ordnung beschränkt, muss erwähnt werden, dass sich auch Optimierungsverfahren zweiter Ordnung zum Trainieren von neuronalen Netzen eignen. Besonders das L-BFGS-Verfahren hat gezeigt, dass es eine sehr gute Trainingsleistung liefert (Ngiam u. a., 2011).

3.3 Verfahren nach Tom Schaul

Schaul, Zhang u. a. (2012) berechnen Lernraten durch den Erwartungswert und die Varianz des Gradienten. Jedes Trainingsmuster unterliegt einer Wahrschein-

lichkeitsverteilung und alle Muster beeinflussen durch ihren Einzelfehler $E^p(w)$ ($p = 1, \dots P$ bezeichnet das p-te Muster aus der Mustermenge) den erwarteten Wert des globalen Fehlers $E(w) = \mathbb{E}rw[E^p(w)]$. Der optimale Parametervektor wird durch diesen globalen Fehler analog zu Gleichung (2.15) definiert durch

$$w^* = \underset{w}{\operatorname{argmin}}(E(w)). \tag{3.12}$$

Aus Gleichung (2.20) erhält man für die Komponenten des Gradienten

$$g_i^p = h_{i,i}(w_i - w_i^*). \tag{3.13}$$

Besitze die Verteilung der zu jedem Muster passenden w_i^* den Mittelwert $w_{i,\mu}$ und Varianzmatrix Σ. Im Folgenden wird nun angenommen, dass die Hessematrix und Σ in einer diagonalen Form vorliegen, mit Diagonalelementen $h_{1,1}, \dots, h_{D,D}$ und $\sigma_{1,1}, \dots, \sigma_{D,D}$. Für den Erwartungswert des Gradienten gilt nun

$$\mathbb{E}rw[g^p(w_i)] = h_{i,i}(w_i - w_{i,\mu}) \tag{3.14}$$

und für die Varianz

$$\mathbb{V}ar[g^p(w_i)] = h_{i,i}^2\sigma_{i,i}^2. \tag{3.15}$$

Setzt man dies nun in Gleichung (2.16) ein und formt diese Gleichung um, erhält man

$$\begin{aligned}
\eta_{i,t} &= \frac{(w_{i,t} - w_{i,\mu})^2}{h_{(i,i),t+1}((w_{i,t} - w_{i,\mu})^2 + \sigma_{i,i}^2)} && (3.16)\\
&= \frac{1}{h_{(i,i),t+1}} \frac{\mathbb{E}rw[g_{i,t}^p]^2}{\mathbb{E}rw[g_{i,t}^p]^2 + \mathbb{V}ar[g_{i,t}^p]} && (3.17)\\
&= \frac{1}{h_{(i,i),t+1}} \frac{\mathbb{E}rw[g_{i,t}^p]^2}{\mathbb{E}rw[(g_{i,t}^p)^2]} \ . && (3.18)
\end{aligned}$$

Im Allgemeinen stehen die Werte für $\mathbb{E}rw[g_{i,t}^p]$ und $\mathbb{E}rw[(g_{i,t}^p)^2]$ nicht zur Verfügung. Diese Werte können jedoch approximiert werden durch

$$\begin{aligned}
\mathbb{E}rw[g_{i,t+1}^p] &\approx \alpha_{i,t+1} = (1 - \rho_{i,t+1})\alpha_{i,t}) + \rho_{i,t+1}g_{i,t}^p && (3.19)\\
\mathbb{E}rw[(g_{i,t+1}^p)^2] &\approx \beta_{i,t+1} = (1 - \rho_{i,t+1})\beta_{i,t} + \rho_{i,t+1}(g_{i,t}^p)^2 \ . && (3.20)
\end{aligned}$$

Eine Möglichkeit zur Bestimmung von ρ ist gegeben durch

$$\rho_{i,t+1} = \left(\left(1 - \frac{\alpha_{i,t}^2}{\beta_{i,t}^2} \right) \rho_{i,t} + 1 \right)^{-1} . \tag{3.21}$$

Auch bei diesem Verfahren ist die Bestimmung der Diagonalelemente von H problematisch. Verfahren zur Approximation dieser Elemente finden sich in den Arbeiten von LeCun, Bottou, Orr u. a. (1998) und Becker und Le Cun (1988). Durch die Verwendung eines solchen Verfahrens ergibt sich die folgende Approximation für $h_{i,i}$

$$\overline{h_{(i,i),t+1}} = (1 - \rho_i)\overline{h_{(i,i),t+1}} + \rho_i h_{(i,i),t+1} . \tag{3.22}$$

Analoge Überlegungen gelten auch für globale Lernraten und Minibatches (Schaul und LeCun, 2013).

3.4 AdaGrad

Duchi, Hazan u. a. (2010) geben mit ihrem AdaGrad Algorithmus eine weitere Methode an, um individuelle Lernraten zu bestimmen. Als Basis dient hier die Composite Mirror Descent Methode (Duchi, Shalev-Shwartz u. a., 2010), in der sich die Gewichtsänderung berechnet durch

$$w_{t+1} = \underset{w}{\operatorname{argmin}} \left(\eta \langle g_t, w \rangle + \eta \phi(w) + B_\psi(w, w_t) \right) . \tag{3.23}$$

Dabei bezeichnet $\phi(.)$ den Regulierungsterm. In dieser Arbeit wird immer zur Regularisierung von AdaGrad immer die die l_1-Regulierung verwendet. B_ψ bezeichnet die Bergmann-Divergenz mit Proximalfunktion

$$\psi_t(w) = \langle w, M_t w \rangle \tag{3.24}$$

und

$$M_t = diag \left(\sum_{\tau=1}^{t} g_\tau g_\tau^T \right)^{1/2} . \tag{3.25}$$

M_t bezeichnet eine Diagonalmatrix, deren Hauptdiagonale aus den jeweiligen Summen der quadrierten Gradientenkomponenten besteht. Setzt man dies in Glei-

chung (3.23) ein erhält man

$$w_{t+1} = \underset{w}{\operatorname{argmin}} \left(\eta_0 \langle g_t, w \rangle + \eta_0 \lambda \|w\|_1 + \psi(w) - \psi(w_t) - \langle \nabla \psi(w_t), w - w_t \rangle \right) \quad (3.26)$$

Daraus folgt für die i-te Komponente des optimalen Gewichtsvektors w^*

$$\begin{aligned} \eta_0 g_{i,t} + \eta_0 \lambda + M_{t,ii} w_i^* - M_{t,ii} w_{i,t} &= 0 && (3.27) \\ w_i^* &= w_{i,t} - \frac{\eta_0}{M_{t,ii}} g_{i,t} - \frac{\lambda}{M_{t,ii}}. && (3.28) \end{aligned}$$

Vernachlässigt man den Regulierungsterm in Abschnitt 3.4 ergibt sich die Lernrate für Gewicht i zum Zeitpunkt t durch

$$\eta_{i,t} = \frac{\eta_0}{\sqrt{\sum_{\tau=1}^{t} g_{i,\tau}^2}}. \quad (3.29)$$

Um zu verhindern, dass der Bruch in der obigen Gleichung nummerisch instabil wird, ist die Verwendung einer Stabilisierungskonstante δ hilfreich. Aus Gleichung (3.29) folgt dann

$$\eta_{i,t} = \frac{\eta_0}{\sqrt{\sum_{\tau=1}^{t} g_{i,\tau}^2 + \delta}}. \quad (3.30)$$

Die Summe innerhalb der AdaGrad-Lernrate beinhaltet Informationen aus den Gradienten bis zum aktuellen Zeitpunkt t. Sie kann als Historie des Lernverlaufs angesehen werden. Für die Gewichtsänderungen werden somit historische Informationen genutzt. Bemerkenswert an diesem Verfahren ist, dass die Summe unter der Wurzel in Gleichung (3.29) bei jedem Iterationsschritt anwächst und somit die Lernrate kleiner wird. Im Gegensatz zu anderen Lernverfahren, bei denen ein großer Gradient zu großen Schritten auf der Fehleroberfläche führt, und somit den Lernfortschritt beschleunigt, verhält sich AdaGrad genau umgekehrt. Große Gradienten führen hier zu kleiner werdenden Schritten auf der Fehleroberfläche.

Die Berechnung der adaptiven Lernraten η_i benötigt drei Hyperparameter:

- Die globale Lernrate η
- Die Stabilisierungskonstante δ
- Die l_1-Regulierungskonstante λ

AdaGrad wurde speziell für konvexe Optimierungsprobleme konzipiert. Der Algorithmus wurde jedoch auch bei nicht-konvexen Problemstellungen angewandt

(Dean u. a., 2012). Es liegen zum aktuellen Zeitpunkt keine detaillierten Informationen vor, wie sich AdaGrad in nicht-konvexen Modellen verhält.

Eine Modifikation von AdaGrad liefert das AdaDelta Lernverfahren (Zeiler, 2012). Hierbei wird die anwachsende Summe in Gleichung (3.29) auf ein festgelegtes Zeitfenster beschränkt und die globale Lernrate η durch die Newton-Gleichung approximiert.

3.5 RPROP

Der von Riedmiller und Braun (1993) beschriebene RPROP-Algorithmus ist ein elastisches Verfahren für den Gradientenabstieg. Die Gewichtsänderung wird dabei nicht von der Größe des Gradienten, sondern von seinem Vorzeichen bestimmt. Dazu wird jedem Gewicht w_i ein individueller Updatewert Δ_i zugeordnet. Δ_i wird zu einem Zeitpunkt t wie folgt berechnet

$$\Delta_{i,t} = \begin{cases} \eta^+ \Delta_{i,t-1} & , \; falls \quad g_{i,t-1} \cdot g_{i,t} > 0 \\ \eta^- \Delta_{i,t-1} & , \; falls \quad g_{i,t-1} \cdot g_{i,t} < 0 \\ g_{i,t-1} & , \; sonst \end{cases} \tag{3.31}$$

mit $0 < \eta^- < 1 < \eta^+$. Jedes Mal wenn g_i sein Vorzeichen ändert, wird Δ_i multiplikativ um η^- verringert. Der Begriff der Elastizität beschreibt die Anpassung der Δ_i entsprechend der Gradientenentwicklung. Die Vorzeichenänderung impliziert, dass die letzte Gewichtsänderung zu stark war und das Minimum verfehlt wurde. Falls die Vorzeichen von $g_{i,t-1}$ und $g_{i,t}$ identisch sind, wird Δ_i multiplikativ um η^+ vergrößert, um eine beschleunigte Konvergenz zu erreichen. Die Berechnung der Gewichtsänderung Δw_i erfolgt dann nach folgender Vorschrift

$$\Delta w_{i,t} = \begin{cases} -\Delta_{i,t} & , \; falls \quad g_{i,t} > 0 \\ +\Delta_{i,t} & , \; falls \quad g_{i,t} < 0 \\ -\Delta w_{i,t-1} & , \; falls \quad g_{i,t-1} \cdot g_{i,t} < 0 \\ 0 & , \; sonst. \end{cases} \tag{3.32}$$

Das Gewicht w_i wird dann nach der bereits bekannten Form geändert

$$w_{i,t} = w_{i,t-1} + \Delta w_{i,t}. \tag{3.33}$$

Eine Ausnahme bei dieser Gewichtsanpassung tritt bei einem Vorzeichenwechsel in kraft. Findet ein Vorzeichenwechsel statt, so wird die Gewichtsanpassung rückgängig gemacht durch

$$\Delta w_{i,t} = -\Delta w_{i,t-1} \tag{3.34}$$

und der Wert für den entsprechenden Gradienten wird auf null gesetzt ($g_{i,t-1} = 0$). Diese Ausnahmeregel wird Weight-Backtracking genannt.

Im Vergleich zum klassischen Gradientenabstieg besitzt RPROP mehr Hyperparameter (η^+, η^-, Δ_i). Zur Belegung dieser Werte werden die folgenden Werte vorgeschlagen:

- $\eta^+ = 1{,}2$. Hier sollte eine Wert > 1 benutzt werden, um die Konvergenz zum Minimum zu beschleunigen,
- $\eta^- = 0{,}5$. Hier wird ein Wert < 1 benutzt, um ein Überspringen eines Minimums zu verhindern,
- $\Delta_{i,0} = \Delta_0 = 0{,}1$. Zur Initialisierung der Updatewerte werden alle auf einen Wert Δ_0 gesetzt.

Bei der Verwendung von Batchlearning haben sich diese Hyperparameter als sehr robust erwiesen. Der RPROP-Algorithmus reagierte unempfindlich auf deren Belegung.

4 Das RMSProp-Lernverfahren

Alle vorgestellten Lernverfahren besitzen ihre Vor- und Nachteile. So benötigt der klassische Gradientenabstieg eine hohe Anzahl von Iterationen bis sich ein akzeptables Minimum einstellen kann. Die Verwendung von Lernverfahren zweiter Ordnung ist durch ihre Abhängigkeit von der Hessematrix problematisch und das Verfahren von Schaul, Zhang u. a. (2012) benötigt Diagonalelemente der Hessematrix, welche approximiert werden müssen.

4.1 Die Minibatch-Problematik von RPROP

RPROP hat sich als sehr robustes Verfahren erwiesen. Seine Hyperparameter sind unempfindlich gegenüber ihrer Initialisierung. Dieses Verhalten ist jedoch nur im Batchlearning garantiert. Verwendet man RPROP mit Minibatches zeigt der Algorithmus Schwächen.

Im Minibatchlearning werden viele kleine Gewichtsänderungen (im Vergleich zum Batchlearning) durchgeführt. Für den Fall, dass die Gradientenkomponente g_i passend zum Gewicht w_i in vielen hintereinander durchgeführten Iterationsschritten ihr Vorzeichen ständig wechselt, würde die RPROP-Lernrate Δ_i in jeder Iteration verkleinert werden. Das ständige Springen des Vorzeichens impliziert, dass Δ_i durch η^- ständig multiplikativ verringert wird. Dadurch fallen auch die Gewichtsänderungen immer kleiner aus und das Lernen kommt zum Erliegen.

Eine weitere Schwäche von RPROP in Bezug auf Miniatches soll das folgende Beispiel verdeutlichen.

Beispiel: Der Gradient eines Gewichts w_i beträgt in neun aufeinander folgenden Minibatch-Iterationen 0,1 und im zehnten Durchlauf 0,9. Wünschenswert ist in diesem Fall, dass sich der Wert des Gewichts vor dem ersten und nach dem zehnten Durchlauf nicht ändert. Tatsächlich ist der Wert von w_i jedoch nach dem zehnten Durchlauf höher, als der Wert vor dem ersten. Dies wird durch die neunmalige multiplikative Erhöhung von Δ_i um η^+ bewirkt. Die Mittelungseigenschaft von

Minibatches geht durch die Verwendung von RPROP verloren.

Beide hier aufgeführten Schwächen von RPROP bei der Verwendung von Minibatches sind durch den Umgang mit Vorzeichenwechsel im Gradienten begründet (Gleichung (3.31)).

Des Weiteren hat sich gezeigt, dass $\eta^- = 0{,}5$ und $\eta^+ = 1{,}2$ gute Belegungen für die multiplikativen Konstanten sind. Dies gilt jedoch zunächst nur bei der Verwendung von RPROP im Batchlearning. Ob diese Werte mit Minibatches ähnliche Ergebnisse erzielen können, und ob der Algorithmus auch hier eine ähnliche Unempfindlichkeit gegenüber seinen Hyperparametern zeigt, ist nicht bekannt.

4.2 Der RMSProp-Ansatz

Aus den Schwächen von RPROP stellt sich direkt die Frage, ob man die RPROP-Idee mit der Effizienz von Minibatches so verbinden kann, dass die natürliche Mittelung des Gradienten durch Minibatches erhalten bleibt. G. Hinton (2012) bietet einen Ansatz, die Minibatch-Problematik von RPROP zu umgehen.

Die Kernidee von RPROP ist, eine Gewichtsänderung nur mit Informationen aus dem Vorzeichen des Gradienten durchzuführen und nicht aus seinem exakten Wert. Lässt man die Spezialfälle für den Umgang des Gradienten bei einem Vorzeichenwechsel außer Acht, kann man das Gleichung (3.32) schreiben als

$$w_{i,t+1} = w_{i,t} - \operatorname{sign}(g_{i,t})\Delta_{i,t}. \tag{4.1}$$

Dabei bezeichnet $\operatorname{sign}(g_i)$ das Vorzeichen der Gradientenkomponente g_i. Dieses Vorzeichen lässt sich ausdrücken durch

$$\operatorname{sign}(g_i) = \frac{g_i}{|g_i|}. \tag{4.2}$$

Setzt man dies in Gleichung (4.1), ein erhält man

$$w_{i,t+1} = w_{i,t} - \frac{g_{i,t}}{|g_{i,t}|}\Delta_{i,t}. \tag{4.3}$$

Die Mittelungseigenschaft von Minibatches geht verloren, da der Gradient in (4.3) bei jedem Iterationsschritt durch verschiedene Werte $|g_{i,t}|$ dividiert wird. G. Hinton (2012) schlägt nun vor, dass die Gradienten von zeitlich nahen Iterationen durch ähnliche Werte dividiert werden, die jedoch von den Werten der vorange-

gangenen Gradienten abhängen. Eine solche Möglichkeit bietet der Austausch des Absolutbetrags durch ein quadratisches Mittel (Mean-Square, MS)

$$\mathrm{MS}(g_i)_t = \rho\,\mathrm{MS}(g_i)_{t-1} + (1-\rho)g_{i,t}^2 \tag{4.4}$$

und anschließend daraus die Wurzel ziehen, wodurch sich das Lernen vereinfachen soll. Daraus ergibt sich die RMSProp-Lernrate (root mean square)

$$\eta_{i,t} = \frac{1}{\sqrt{\mathrm{MS}(g_i)_t}}. \tag{4.5}$$

Zu beachten ist, dass diese Lernrate den Hyperparameter ρ beinhaltet. Die RMSProp-Lernrate bietet eine Möglichkeit jedem Gewicht w_i eine individuelle Lernrate η_i zuzuordnen. Diese entwickelt sich bei jedem Iterationsschritt weiter.

Für die Implementation dieser Lernrate muss erwähnt werden, dass der Bruch eine Stabilisierungskonstante δ benötigt. Dadurch ändert sich Gleichung (4.6) zu

$$\eta_{i,t} = \frac{1}{\sqrt{\mathrm{MS}(g_i)_t + \delta}}. \tag{4.6}$$

4.3 RMSProp Versionen

Aus der RMSProp-Lernrate kann man verschiedene Verfahren zur Gewichtsänderung ableiten. In dieser Arbeit werden die drei im Folgenden beschriebenen RMSProp-Versionen verwendet.

4.3.1 RMSProp

Die RMSProp-Lernrate kann innerhalb des klassischen Gradientenabstiegs (Gleichung (2.16)) verwendet werden. Dadurch wird die globale konstante Lernrate ersetzt durch adaptive, für jedes Gewicht individuelle, Lernraten. Die Änderung der Gewichte wird dann berechnet durch

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{\mathrm{MS}(g_i)_t + \delta}} g_{t-1}. \tag{4.7}$$

Das auf dieser Änderung der Gewichtsänderung basierende Lernverfahren wird im Folgenden RMSProp genannt.

Es sei an dieser Stelle darauf hingewiesen, dass eine große Ähnlichkeit zwischen

RMSProp und AdaGrad existiert. In beiden Verfahren unterscheidet sich die Lernrate nur durch den Term unter der Wurzel. Die potenziell nach oben hin unbeschränkte Summe $\sum_t g_t^2$ in AdaGrad wird in RMSProp ersetzt durch den Mean-Square des Gradienten. Anschaulich kann man dies verstehen als Beschränkung der AdaGrad-Summe auf ein Zeitfenster. Falls das Zeitfenster k Iterationen umfasst, müsste man für die exakte Berechnung der entsprechenden Summe alle k Werte des Fensters im Speicher aufheben. Dieses Vorgehen impliziert hohe Speicherkosten. Der Mean-Square bietet eine Möglichkeit die Summe innerhalb des Fensters zu approximieren.

4.3.2 Elastisches RMSProp

Eine elastische Version von RMSProp erhält man, indem die RMSProp-Lernrate in RPROP eingebaut wird. Angelehnt an RPROP wird dieses Lernverfahren RRMSProp genannt (Resilient RMSProp). Setzt man die Berechnung der Werte Δ_i und das analoge Verhalten bei einem Vorzeichenwechsel entsprechend RPROP (Gleichung (3.31) und Gleichung (3.32)) voraus, so erhält man die RRMSProp Anpassung der Gewichte aus

$$w_{i,t} = w_{i,t-1} + \frac{1}{\sqrt{\mathrm{MS}(g_i)_t + \delta}} \Delta w_{i,t}. \tag{4.8}$$

4.3.3 Durch Nesterov-Momentum beschleunigtes RMSProp

Angelehnt an (Tieleman und G. Hinton, 2012) kann RMSProp mit einem Nesterov-Momentum kombiniert werden. Die resultierende Berechnungsvorschrift lautet

$$v_{i,t+1} = w_{i,t} - \frac{\eta}{\sqrt{\mathrm{MS}(g_i)_t + \delta}} g_{i,t} \tag{4.9}$$

$$w_{i,t+1} = v_{i,t+1} + \mu(v_{i,t+1} - v_{i,t}). \tag{4.10}$$

Das entsprechende Lernverfahren wird NA-RMSProp (Nesterov accelerated RMSProp) genannt.

Auch diese Version kann mit einer elastischen Komponente versehen werden. Dazu wird die globale Lernrate η ersetzt durch eine für jedes Gewicht individuelle Lernrate η_i. Diese kann in jedem Iterationsschritt angepasst werden. Dazu definiert

man

$$\alpha_{i,t} = \mu(v_{i,t} - v_{i,t-1}) \tag{4.11}$$

und

$$\beta_{i,t} = \frac{\eta_{i,t}}{\sqrt{\mathrm{MS}(g_i)_t + \delta}} g_{i,t}. \tag{4.12}$$

Die Anpassung der individuellen Lernrate η_i erfolgt durch den Hyperparameter γ nach folgender Vorschrift

$$\eta_{i,t+1} = \eta_{i,t} \cdot \begin{cases} (1+\gamma) & , \; falls \quad \mathrm{sign}(\alpha_{i,t}) = \mathrm{sign}(\beta_{i,t} + alpha_{i,t}) \\ (1-\gamma) & , \; sonst \quad . \end{cases} \tag{4.13}$$

Die Idee hinter dieser Formulierung ist, die Lernrate zu erhöhen, falls die Gewichtsänderung $\alpha_i + \beta_i$ und das Momentum β_i die gleiche Richtung besitzen. Falls die Richtungen nicht übereinstimmen, soll die Lernrate verringert werden. Die elastische Konstante γ ist ein Hyperparameter des Lernverfahrens.

5 Arbeitsauftrag und Arbeitshypothesen

RMSProp wurde bereits erfolgreich im Bereich des Deep Learning eingesetzt (Höft u. a., 2014). Obwohl AdaGrad für eine konvexe Optimierung konzipiert wurde, ist auch dieses Verfahren erfolgreich für nicht-konvexe Optimierungen eingesetzt worden (Dean u. a., 2012). Jedoch existieren für beide Verfahren zur Zeit keine empirischen Informationen über ihre Optimierungsfähigkeiten in Bezug auf nicht-konvexe Modelle. Vor allem die Lernverfahren NA-RMSProp und RRMSProp bedürfen einer näheren Untersuchung. Aufgrund dessen wird in dieser Arbeit eine Evaluation dieser Algorithmen anhand von Experimenten durchgeführt. Zusätzlich soll die Minibatch Problematik von RPROP aufgezeigt werden. Da all diese Algorithmen über individuelle, adaptive Lernraten verfügen, wird auch der klassische Gradientenabstieg evaluiert, um einen Vergleich zwischen solchen Lernraten und konstanten globalen Lernraten ziehen zu können.

Die Evaluation soll Informationen über die Optimierungsfähigkeit der Algorithmen in Bezug auf nicht-konvexe Modelle in Kombination mit Minibatches liefern. Diese Modelle werden in Experimenten durch MLPs und Konvolutionsnetze vertreten. Von besonderer Bedeutung ist eine Untersuchung der Hyperparameter. Hier soll festgestellt werden, wie gut sich geeignete Werte für die einzelnen Hyperparameter finden lassen und wie die Algorithmen auf ihre Hyperparameter reagieren. Des Weiteren muss festgestellt werden, ob vielleicht einige Hyperparameter nicht mit festen Werten belegt werden können (ähnlich zu η^+, η^-). Bei der Verwendung des entsprechenden Algorithmus können dann solche Parameter als Konstanten betrachtet werden. Eine weitere interessante Fragestellung in Bezug auf die Hyperparameter ist, ob Korrelationen zwischen diesen bestehen. Für berechnungsintensive Modelle ist eine schnelle Konvergenz eines Trainingsverfahrens sehr wichtig. Deswegen muss die Trainingszeit der Lernverfahren ein weiteres Evaluationskriterium sein. Nachdem experimentelle Daten vorliegen, wird ein Vergleich der Lernverfahren durchgeführt. Im Fokus dieses Vergleichs liegen die folgenden

Kriterien:

- Wie gut sind die Optimierungseigenschaften der Algorithmen?
- Wie empfindlich reagieren die Algorithmen auf eine Belegung ihrer Hyperparameter?
- Wie einfach ist es für die empfindlichen Hyperparameter, akzeptable Werte zu finden?
- Können einige Hyperparameter auch mit konstanten Werten belegt werden?
- Gibt es Korrelationen zwischen den Hyperparametern?
- Wieviel Zeit benötigen die Algorithmen für das Training?

Vor der Ausführung der Experimente werden an dieser Stelle die folgenden Arbeitshypothesen und Erwartungen formuliert. Die durchgeführten Experimente werden anschließend anhand dieser Hypothesen ausgewertet.

- **Hypothese 1**: *RPROP und RRMSProp liefern mit großen Minibatches bessere Ergebnisse.* Da RPROP aufgrund der Minibatch-Problematik als Batchverfahren angesehen werden kann und da viele kleine Gewichtsanpassungen mit ständig wechselnden Vorzeichen des Gradienten diese Problematik noch verstärken können, wird angenommen, dass RPROP und RRMSProp mit hohen Batchgrößen bessere Ergebnisse liefern.

- **Hypothese 2**: η^- *und* η^+ *müssen näher an 1 initialisiert werden.* Um immer kleiner werdenden Lernraten Δ_i entgegenzuwirken, wird angenommen, dass die multiplikativen Konstanten η^+ und η^- im Minibatchlearning angepasst werden müssen. Die Erwartung ist, dass diese Hyperparameter mit Werten näher an 1 belegt werden müssen, als im Batchlearning.

- **Hypothese 3**: Δ_i *konvergiert sehr schnell gegen* Δ_{min}. Es wird erwartet, dass, aufgrund der möglichen Vorzeichenwechsel im Gradienten, die RPROP-Lernraten Δ_i sehr schnell gegen die vordefinierte untere Grenze konvergieren. Dadurch werden die Lernraten während des Lernens mit sehr kleinen Werten belegt und ein Lernfortschritt bleibt ab einem gewissen Zeitpunkt aus. Dadurch wird zusätzlich angenommen, dass RPROP im Vergleich zu den restlichen Algorithmen einen deutlich schlechteren finalen Klassifikationsfehler erreichen wird.

- **Hypothese 4**: *In AdaGrad muss* η *mit großen Werten belegt werden.* Die bei steigender Epochenanzahl immer größer werdende Summe der quadrier-

ten Gradientenkomponenten innerhalb der AdaGrad-Lernraten kann dazu führen, dass der Lernverlauf schon nach wenigen Epochen zu stagnieren beginnt. Eine Möglichkeit dem entgegenzuwirken ist die Verwendung von großen globalen Konstanten η. Aufgrund dessen wird angenommen, dass AdaGrad eine große globale Lernrate benötigt.

- **Hypothese 5**: *AdaGrad und RMSProp liefern niedrigere Fehler als GD.* Aufgrund der aktuell hohen Popularität von individuellen, anpassungsfähigen Lernraten wird erwartet, dass sowohl AdaGrad als auch die RMSProp-Varianten niedrigere Klassifikationsfehler liefern als GD.

- **Hypothese 6**: *RMSProp und AdaGrad liefern ähnliche Ergebnisse.* Die Ähnlichkeit zwischen RMSProp und AdaGrad führt zu der Annahme, dass eine geeignete Approximation der AdaGrad-Summe durch einen entsprechenden Mean-Square dazu führen kann, dass beide Verfahren eine ähnliche Trainingsleistung liefern können.

- **Hypothese 7**: *RMSProp reagiert empfindlich auf ρ.* Eine daraus resultierende Erwartung ist, dass alle RMSProp-Varianten empfindlich auf den Hyperparameter ρ reagieren.

- **Hypothese 8**: *NA-RMSProp konvergiert schneller als RMSProp.* Erweiterung von RMSProp um ein Momentum führt zu der Formulierung von NA-RMSProp. Da durch die Verwendung eines Momentums eine Konvergenz beschleunigt werden soll, wird angenommen, dass NA-RMSProp schneller konvergiert als RMSProp.

- **Hypothese 9**: *NA-RMSProp benötigt kleine Belegungen für γ.* Ähnlich zu RPROP und RRMSProp kann ein hoher Wert für die elastische Konstante γ zu Problemen führen. Eine Belegung von γ mit kleinen Werten führt dazu, dass die individuellen Parameter η_i in NA-RMSProp mit Werten nahe an 1 multipliziert wird und somit die Elastizität abgeschwächt wird. Es wird angenommen, dass solche Werte dem Lernen dienlicher sind als hohe Belegungen von γ.

6 Experimente

Die hier durchgeführten Experimente lassen sich in zwei Kategorien aufteilen: Experimente mit MLPs und Experimente mit Konvolutionsnetzen. In beiden Kategorien werden die Netze auf den Datensätzen MNIST und CIFAR-10 trainiert. Das Training erfolgt durch die bereits besprochenen Algorithmen AdaGrad, RMSProp, NA-RMSProp, RRMSPROP, RPROP und GD. Um für diese Algorithmen die besten Hyperparameter zu finden, wird zunächst eine Kreuzvalidierung durchgeführt und anschließend das Netz mit den besten gefundenen Hyperparametern neu trainiert.

6.1 Datensätze

Für die Evaluation werden die standardisierten Datensätze MNIST (LeCun, Cortes u. a., 1998; LeCun, Bottou, Bengio u. a., 1998) und CIFAR-10 (Krizhevsky und G. Hinton, 2009) genutzt (Abbildung 6.1).

Der MNIST-Datensatz besteht aus insgesamt 70000 Graustufenbildern mit 28×28 Pixeln von handschriftlich verfassten Ziffern. Jedes Bild besitzt ein Label, welches als Teacherwert verwendet werden kann. Der Datensatz wird in 60000 Bilder in der Trainingsmenge und 10000 Bilder in der Testmenge aufgeteilt.

Der CIFAR-10 Datensatz besteht aus 60000 Farbbildern mit 32×32 Pixeln und besitzt somit drei Farbkanäle. Jedes Bild lässt sich eindeutig zu einer von zehn unterschiedlichen Klassen zuordnen. Analog zu MNIST besitzt auch hier jedes Bild ein passendes Label, welches als Teacherwert genutzt werden kann. Der Datensatz wird in 50000 Bilder in der Trainingsmenge und 10000 Bilder in der Testmenge aufgeteilt.

(a) MNIST **(b)** CIFAR-10

Abbildung 6.1: Beispiele für Trainingsmuster aus den Datensätzen MNIST und CIFAR-10

Name	Input	Hidden-Layer	Output	# Gewichte
MNIST-1	784	120	10	95410
MNIST-2	784	500-300	10	545810
CIFAR-1	3072	360	10	1109890

Tabelle 6.1: Spezifikationen der MLP-Architekturen. Als Aktivierungsfunktion kommt der hyperbolische Tangens tanh zum Einsatz. Den Output bildet eine Softmax-Schicht.

6.2 Netzarchitekturen

Alle verwendeten Netzarchitekturen sind auf Klassifikationsaufgaben ausgelegt. Es wurden drei verschiedene MLP-Architekturen und zwei Konvolutionsnetze verwendet.

Zu Vergleichszwecken mit anderen Arbeiten wurden die MLP-Architekturen von Schaul, Zhang u. a. (2012) übernommen, welche bereits von Zeiler (2012) zur Evaluation von AdaDelta verwendet wurden. Für Experimente auf dem MNIST-Datensatz wurden zwei MLP-Architekturen verwendet mit jeweils einer bzw. zwei verdeckten Schichten. Das MLP, welches mit CIFAR-10 trainiert wird, besitzt nur eine verdeckte Schicht. Alle MLPs verwenden den hyperbolischen Tangens tanh als Aktivierungsfunktion. Den Output bildet eine Klassifikationsschicht zur Klassifikation. Die genauen Netzspezifikationen findet man in Tabelle 6.1.

Es sei an dieser Stelle erwähnt, dass auch MLP-Experimente ohne verdeckte Schicht durchgeführt wurden. Diese MLPs sind konvexe Modelle. In diesen Experimenten zeigte sich bei keinem Algorithmus eine bemerkenswerte Empfindlichkeit gegenüber den Hyperparametern. Alle Algorithmen zeigten ähnliche Werte für den finalen Klassifikationsfehler auf der Testmenge. Diese Beobachtungen sind auf die sehr einfache Netzarchitektur zurückzuführen. Da man anhand dieser Experimente keine Aussage über die Qualität der Hyperparameter treffen kann, werden sie in dieser Arbeit nicht weiter beachtet.

Die Architekturen der verwendeten Konvolutionsnetze orientieren sich stark an den Maxout-Netzen von Goodfellow u. a. (2013), die zum aktuellen Zeitpunkt den State-of-the-Art für Konvolutionsnetze, die auf den Datensätzen MNIST und CIFAR-10 trainiert wurden, halten.

Das Konvolutionsnetz CNN-MNIST, welches mit dem MNIST Datensatz trainiert wird, besteht aus drei verdeckten Schichten. Jede dieser Schichten besteht

aus einer Konvolutions- und einer Poolingschicht. Nach jeder Konvolution kommt die Aktivierungsfunktion maxout zum Einsatz. Diese fasst je zwei Featuremaps zusammen. Für das anschließende Pooling wurde Max-Pooling mit einem Stride von 2 in jeder Richtung verwendet. In der ersten verdeckten Schicht wird der Input mit einem Dropout versehen. Die Wahrscheinlichkeit mit der ein Inputpixel ausgeschaltet wird beträgt 0,2. Eine Klassifikationsschicht bildet den Output des Netzes. Abbildung 6.2 verdeutlicht diese Netzarchitektur.

Analog zu CNN-MNIST ist das Konvolutionsnetz CNN-CIFAR aufgebaut. CNN-CIFAR wird mit dem CIFAR-10 Datensatz trainiert und erhält dementsprechend seinen Input aus drei Farbkanälen. Auch hier wird der Input mit einer Dropoutrate von 0,2 versehen. Die drei verdeckten Konvolutions- und Poolingschichten unterscheiden sich nur in der Anzahl und Größe der Filter und der resultierenden Anzahl und Größe der Featuremaps. Die genauen Werte können Abbildung 6.3 entnommen werden. Bevor der Output durch eine Klassifikationsschicht gebildet wird, kommt in diesem Netz zusätzlich eine vollverknüpfte Schicht zum Einsatz. Auch hier wird maxout zur Aktivierung genutzt, wobei je fünf Featuremaps zusammengefasst werden.

6.3 Kreuzvalidierung

Die Suche und Bereitstellung von Hyperparametern erfolgt durch das Python-Package Hyperopt, welches in Abschnitt 6.4 näher erläutert wird.

Für die Kreuzvalidierung wird die Trainingsmenge aufgeteilt in k gleichgroße Teile $K_1, \ldots, K_k$, die sogenannten Folds. Jeder Kreuzvalidierungsdurchlauf besteht aus k verschiedenen Testdurchläufen. In jedem Testdurchlauf werden $k-1$ Teile der Trainingsmenge für das Training des Netzes, und das verbleibende Stück zur Trainingsbewertung genutzt. Die gesamte Trainingsmenge T wird dadurch aufgeteilt in eine Trainingsmenge T_i für den entsprechenden Testdurchlauf i und eine Validierungsmenge V_i. Für den Testdurchlauf $i \in \{1, \ldots, k\}$ ergibt sich für die Validierungsmenge $V_i = K_i$ und für die Trainingsmenge $T_i = T \backslash V_i = T \backslash K_i$. Abbildung 6.4 verdeutlicht diese Aufteilung. Die Größe von V_i und T_i wird bestimmt durch die Anzahl an Folds. Es ergibt sich für die Trainingsmenge $|T_i| = |T|(k-1)/k$ und für die Validierungsmenge $|V_i| = |T_i|/k$. In dieser Arbeit wurde für alle Experimente mit MLPs $k = 10$ und für alle Experimente mit Konvolutionsnetzen $k = 5$ gewählt. Eine entsprechende Aufteilung von MNIST und CIFAR-10 findet man in

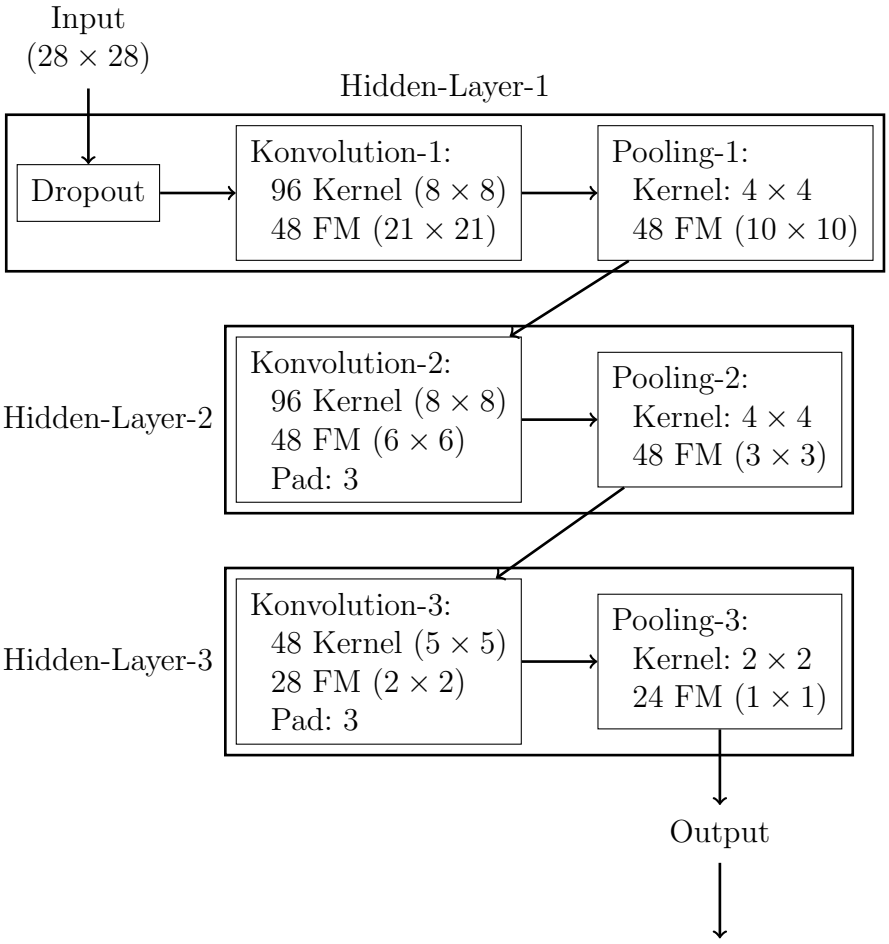


Abbildung 6.2: Das Konvolutionsnetz CNN-MNIST wird mit dem MNIST-Datensatz trainiert. Es besitzt drei verdeckte Schichten in denen abwechselnd Konvolutionen und Max-Pooling ausgeführt werden. Als Aktivierungsfunktion dient maxout für je zwei Featuremaps. Der Input wird mit einer Dropoutrate von 0,2 ausgeschaltet. Den Output bildet eine Klassifikationsschicht. Für jede Konvolutionsoperation wird die Anzahl und die Größe der Filter sowie der resultierenden Featuremaps angegeben. Für die Poolingoperationen wird die Ausdehnung der Operation und die Größe der resultierenden Featuremaps angegeben.

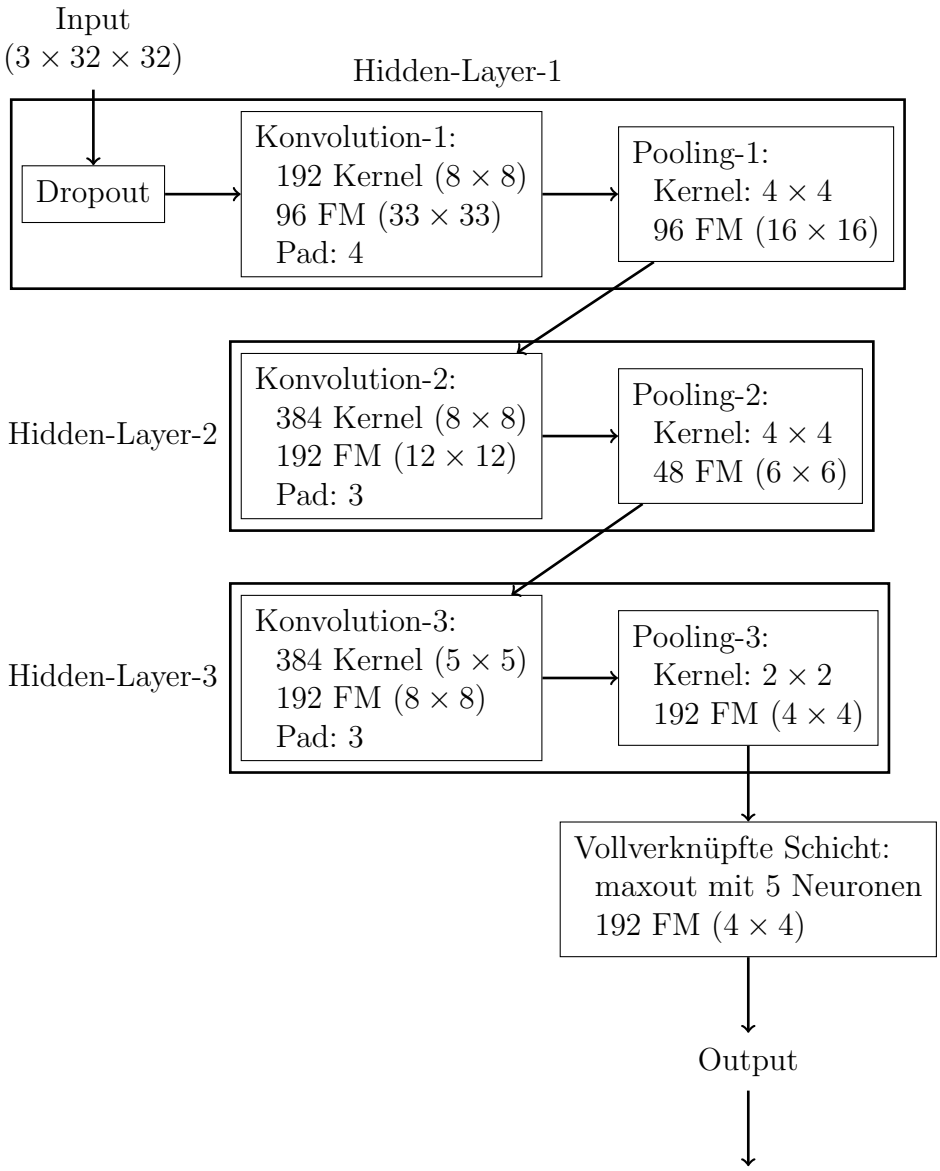


Abbildung 6.3: Das Konvolutionsnetz CNN-CIFAR wird mit dem CIFAR-10-Datensatz trainiert. Es besitzt drei verdeckte Schichten in denen abwechselnd Konvolutionen und Max-Pooling ausgeführt werden und eine verdeckte vollverknüpfte maxout-Schicht. Als Aktivierungsfunktion dient maxout für je zwei Featuremaps nach den Konvolutionen, bzw. fünf Featuremaps in der vollverknüpften Schicht. Der Input wird mit einer Dropoutrate von 0,2 ausgeschaltet. Den Output bildet eine Klassifikationsschicht. Für jede Konvolutionsoperation wird die Anzahl und die Größe der Filter sowie der resultierenden Featuremaps angegeben. Für die Poolingoperationen wird die Ausdehnung der Operation und die Größe der resultierenden Featuremaps angegeben.

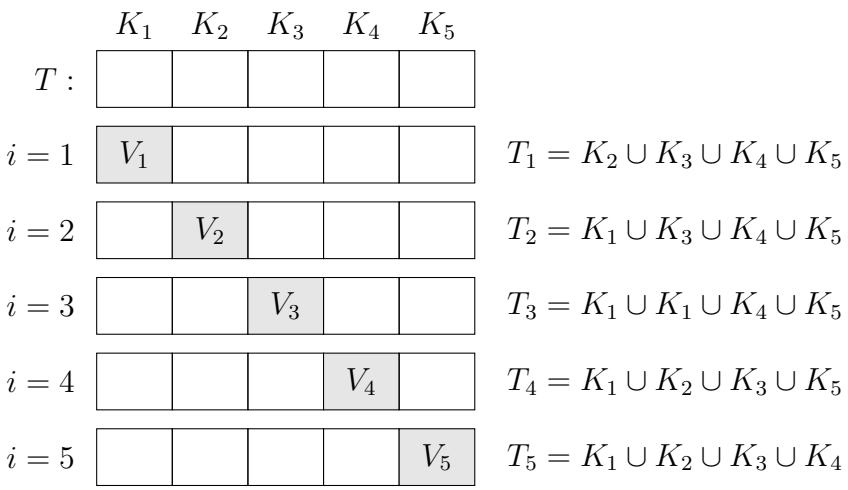


Abbildung 6.4: Aufteilung der Trainingsmenge T in Trainingsmenge (weiß) und Validierungsmenge (grau) für eine 5-Fold Kreuzvalidierung.

Tabelle 6.2.

Für jeden Fold wird das zu trainierende Netz zufällig initialisiert. Um sicherzustellen, dass jeder Kreuzvalidierungsdurchlauf die gleichen Gewichte benutzt, wurden vor Beginn der Kreuzvalidierung k verschiedene Randomseeds fest gewählt. Jedem der k Folds kann somit ein fester Seed zugeordnet werden. Nach jedem Testdurchlauf werden die finalen Ergebnisse für Trainingsfehler, Klassifikationsfehler und benötigte Epochen gespeichert. Nach einem Kreuzvalidierungsdurchlauf werden aus diesen Werten die jeweiligen Mittel gebildet und als Ergebnis des gesamten Durchlaufs ausgegeben. Durch die Bildung des Mittelwertes soll sichergestellt werden, dass die aktuell behandelte Hyperparameterkombination objektiv bewertet wird. Würde man nur den besten Wert aus den k Folds als Ergebnis des gesamten Durchlaufs betrachten, kann nicht sichergestellt werden, dass dieses durch die Wahl der Hyperparameter erreicht wurde. In diesem Fall kann das Ergebnis auch durch eine „glückliche“Initialisierung der Gewichte beeinflusst werden. Ein entsprechendes Listing ist in Algorithmus 1 gegeben.

Während eines Testdurchlaufs wird das Netz trainiert, bis sich keine signifikante Verbesserung des Klassifikationsfehlers auf der Validierungsmenge einstellt. Der Trainingsabbruch wird dabei durch die Verwendung von Early-Stopping gesteuert. Falls bereits Ergebnisse aus vorherigen Kreuzvalidierungsdurchläufen zur Verfügung stehen, kann man diese verwenden, um den aktuellen Durchlauf vorzeitig abzubrechen. Dies kann durch einen Vergleich zwischen dem bisher erzielten besten Ergebnis und dem aktuellen Ergebnis erzielt werden. Wenn schon nach der

	Testmenge	Trainingsmenge				
			$k = 5$		$k = 10$	
		$\|T\|$	$\|T_i\|$	$\|V_i\|$	$\|T_i\|$	$\|V_i\|$
MNIST	10 000	60 000	48 000	12 000	54 000	6 000
CIFAR-10	10 000	50 000	40 000	10 000	45 000	5 000

Tabelle 6.2: Aufteilung der Anzahl der Bilder der Datensätze MNIST und CIFAR-10 in Trainingsmenge und Testmenge. Die Trainingsmenge wird zur Kreuzvalidierung ein weiteres Mal aufgeteilt in Trainingsmenge und Validierungsmenge. Angegeben sind die Anzahl an Bildern in der Testmenge, der gesamten Trainingsmenge und für die Trainingsmenge und Validierungsmenge einer 5-Fold bzw. 10-Fold Kreuzvalidierung.

Abarbeitung von wenigen Folds (z.B. nach drei Folds) feststeht, dass das Ergebnis des Durchlaufs signifikant schlechter ist, als das bisher beste erzielte Ergebnis, kann der aktuelle Durchlauf abgebrochen werden. Ein solches vorzeitiges Abbrechen wurde durch einen T-Test realisiert werden.

Nachdem die gesamte Kreuzvalidierung abgeschlossen wurde, kann man Aussagen über die getesteten Hyperparameter treffen. Die beste Hyperparameterkombination ist diejenige, die im Durchschnitt den kleinsten Klassifikationsfehler auf der Validierungsmenge geliefert hat.

6.4 Hyperopt

Die Suche und Bereitstellung von Hyperparametern erfolgt durch das Python-Package Hyperopt (J. Bergstra u. a., 2013). Hyperopt benötigt zur Suche die Angabe der Suchintervalle für die einzelnen Hyperparameter und die entsprechenden Ergebnisse der Auswertung durch die Kreuzvalidierung. Bis auf wenige Ausnahmen wurden alle Hyperparameter aus dem Intervall $[10^{-7},\ 1]$ gezogen. Die Ausnahmen bilden die Hyperparameter $\lambda \in [10^{-7},\ 1]$ für die l_2-Regularisierung von RPROP und RRMSProp, $\eta^+ \in [1,\ 1{,}6]$ und $\eta^- \in [0{,}4,\ 1]$. Für AdaGrad wurde das Suchintervall für die globale Lernrate η auf $[10^{-7},\ 10]$ festgelegt (vergleiche Hypothese 4). Die Batchgrößen wurden so gewählt, dass während des Trainings die gesamte Trainingsmenge ausgenutzt wird.

Jede generierte Hyperparameterkombination wird in eine Datenbank geschrie-

Algorithm 1: Kreuzvalidierung

Input: Das zu trainierende Netz net_h mit Hyperparameterkombination h
Trainigsmenge T
Anzahl der Folds k
Randomseeds $s_1, \dots s_k$
Output: Gemittelte Werte für
Klassifikationsfehler auf Validierungsmenge CERR
Trainingsfehler auf der Trainingsmenge LOSS
benötigte Epochen e
Teile T auf in $K_1, \dots, K_k$;
Definiere CERR $= 0$, LOSS $= 0$, $e = 0$;
for $j = 1, \dots, k$ **do**
  Reinitialisierte net_h mit dem Randomseed s_j ;
  Trainiere net_h auf T_j und bewerte Trainingsfortschritt auf V_j; Summiere Ergebnisse mit CERR, LOSS, e auf;
end
Bilde Mittel über alle Folds:; CERR $/ = k$, LOSS $/ = k$, $e/ = k$;
return CERR, LOSS, e

ben. Von dort kann sie durch einen Client gelesen werden und eine Kreuzvalidierung gestartet werden. Die Ergebnisse der Kreuzvalidierung werden anschließend für Hyperopt in der gleichen Datenbank bereitgestellt. Durch diese Ergebnisse können nun wieder neue Hyperparameter generiert werden. Abbildung 6.5 verdeutlicht dieses Vorgehen.

6.4.1 Der TPE-Algorithmus

Der TPE-Algorithmus (Tree-Structured-Parse-Estimator, J. S. Bergstra u. a. (2011)) liefert iterativ Hyperparameterkombinationen. Ziel ist es, nach einigen Iterationen solche Hyperparameter zu finden, für die ein Modell immer bessere Ergebnisse liefert. Im folgenden Abschnitt wird die prinzipielle Arbeitsweise des TPE-Algorithmus erläutert.

Sei M ein Modell, welches mit einer Hyperparameterkombination x arbeitet und bezeichne y_x den Output des Modells mit Konfiguration x. Ziel ist es, das Expected Improvement (EI) zu maximieren. Dieser ist definiert durch

$$\mathrm{EI}(x)_b = \int_{-\infty}^{\infty} \max(b - y, 0)\, \mathrm{p}(y|x) dy \tag{6.1}$$

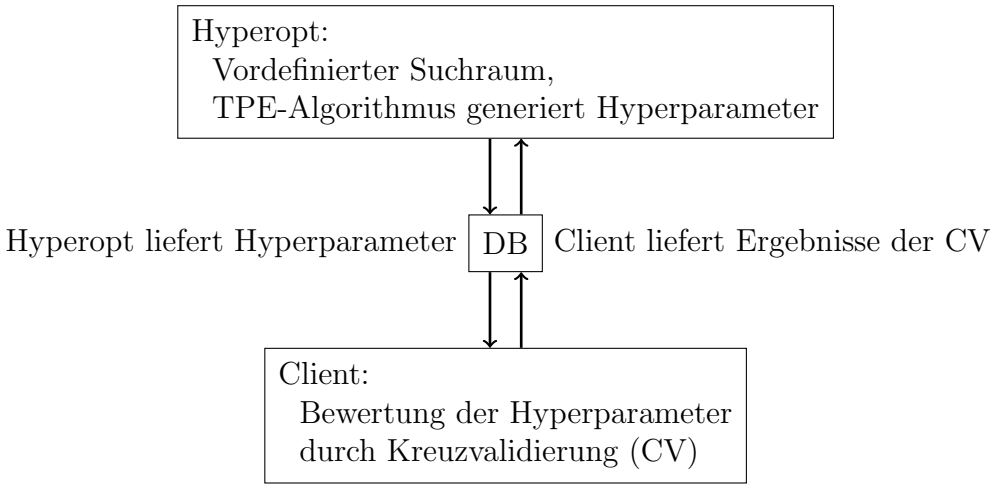


Abbildung 6.5: Verdeutlichung der Kommunikation zwischen Hyperopt und der Kreuzvalidierung. Die Kommunikation erfolgt durch eine Datenbank auf der Hyperopt und ein Client, der die Kreuzvalidierung startet, Zugriff haben.

und beschreibt die Erwartung mit der M unter der Konfiguration x einen Output y_x liefert, der unter einer Grenze b liegt. Um EI zu maximieren, muss eine Konfiguration gefunden werden mit $y_x < b$. Aufgrund diese Überlegungen und der Verwendung des Satzes von Bayes lässt sich Gleichung (6.1) umschreiben zu

$$\mathrm{EI}(x)_b = \int_{-\infty}^{b} (b - y) \frac{\mathrm{p}(x|y)\,\mathrm{p}(y)}{\mathrm{p}(x)} dy. \tag{6.2}$$

Seien nun zum Zeitpunkt t die Konfigurationen $\{x_i, \ldots, x_t\}$ bekannt. Aus diesen lassen sich nun die Wahrscheinlichkeitsdichten l und g konstruieren. Für diese gilt

$$\mathrm{p}(x|y) = \begin{cases} l(x) & \text{falls } y_x < b \\ g(x) & \text{falls } y_x \geq b \end{cases} \tag{6.3}$$

l und g werden konstruiert, indem für jede Konfiguration x_i eine Gauskurve an x_i zentriert wird und die Standardabweichung, entsprechend dem größten Abstand zu einem direkten Nachbarn, gesetzt wird. Aus den einzelnen Gauskurven werden anschließend Mischverteilungen gebildet. Dabei werden für die Konstruktion von l nur Konfigurationen x_i genutzt mit $y_{x_i} < b$ und für g Konfigurationen mit $y_{x_i} \geq b$. Um genügend Daten für die Modellierung von l und g zur Verfügung zu haben, sollte man b nicht mit dem besten erzielten y gleichsetzen, sondern b auf ein bestimmtes Quantil γ der Daten setzen (z.B. 15% der Daten sind besser als b). Aus

diesen Überlegungen und aus Gleichung (6.3) folgt

$$\mathrm{p}(x) = \int_{\mathbb{R}} \gamma l(x) + (1-\gamma) g(x) dy \tag{6.4}$$

und

$$\int_{-\infty}^{b} (b-y)\,\mathrm{p}(x|y)\,\mathrm{p}(y) dy = l(x) \int_{-\infty}^{b} (b-y)\,\mathrm{p}(y) dy. \tag{6.5}$$

Daraus ergibt sich die folgende Proportionalität

$$\mathrm{EI}(x)_b \propto \frac{l(x)}{g(x)} dy. \tag{6.6}$$

Somit muss zur Maximierung von Gleichung (6.1) eine Konfiguration x gefunden werden, welche einen großen Wert $l(x)$ und einen kleinen Wert $g(x)$ liefert.

6.5 Retraining

Wurde durch die Kreuzvalidierung die beste Hyperparameterkombination gefunden, kann man nun das Netz mit diesen Parametern neu trainieren. Für dieses Training wird die gesamte Trainingsmenge ausgenutzt. Eine Aufteilung in Trainingsmenge und Validierungsmenge ist hierbei nicht nötig. Dazu wird das Netz neu initialisiert und so lange auf der gesamten Trainingsmenge trainiert, bis der finale Trainingsfehler aus dem entsprechenden Kreuzvalidierungsdurchlauf erreicht wurde. Anschließend wird die Trainingsleistung bewertet, indem der Klassifikationsfehler auf der Testmenge berechnet wird.

6.6 Implementierung und Ausführung

Für die Ausführung der Experimente standen insgesamt 18 CUDA-fähige Grafikkarten zur Verfügung. Diese haben annähernd fünf Monate am Stück durchgearbeitet, um die im anschließenden Kapitel beschriebenen Ergebnisse zu erzielen. In den durchgeführten Experimenten wurden 2739 Kreuzvalidierungsdurchläufe ausgeführt. Diese verteilen sich wie folgt auf die getesteten Algorithmen:

- AdaGrad: 434
- GD: 479

- RMSProp: 535
- NA-RMSProp: 424
- RPROP: 479
- RRMSProp: 388

Im Folgenden sollen einige benötigte Implementationen erläutert werden. Neben diesen Implementationen wurden zahlreiche Werkzeuge entwickelt, welche der Auswertung der Ergebnisse und zur visuellen Verdeutlichung der Hyperparameter, des Lernfortschritts und der Entwicklung der Lernparameter dienten. Nahezu alle in dieser Arbeit enthaltenen Bilder wurden mit Hilfe dieser Werkzeuge erstellt.

6.6.1 CUV und CUVNET

Die für diese Arbeit benötigten Implementationen basieren größtenteils auf den C++ Bibliotheken CUV und CUVNET (Schulz, 2014a).

Die CUV-Bibliothek stellt Datenstrukturen für Matritzen, in Form von n-dimensionalen Arrays, und Basisoperationen auf diesen Arrays zur Verfügung. Die bereitgestellte Funktionalität kann sowohl auf der CPU als auch auf GPUs ausgeführt werden. Solche Operationen sind beispielsweise Matritzenmultiplikationen oder Konvolutionsoperationen. Von besonderem Interesse für diese Arbeit ist die Möglichkeit, innerhalb der CUV Funktionen für Gewichtsänderungen zu integrieren. So waren vor Beginn dieser Ausarbeitung die Gewichtsänderungen für die Algorithmen AdaGrad, GD, RMSProp und RPROP bereits in der CUV enthalten. Während dieser Arbeit mussten entsprechende Erweiterungen für NA-RMSProp und RRMSProp in der CUV ergänzt werden. Diese wurden sowohl in Form einer CPU Implementierung, als auch durch GPU-Kernels realisiert.

Basierend auf der CUV stellt die Bibliothek CUVNET diverse Möglichkeiten für die Implementierung von neuronalen Netzen und deren Optimierung zur Verfügung. Dem Benutzer werden zahlreiche Operationen (die wiederum in vielen Fällen Funktionen aus der CUV verwenden) bereitgestellt, die zum Aufbau eines neuronalen Netzes genutzt werden können. Das entsprechende Netz wird als Modell bezeichnet. Die Optimierung der Gewichte kann durch diverse Gradientenabstiegsverfahren erfolgen. Die hierfür benötigten Differenzierungen werden ebenfalls von der CUVNET bereitgestellt. Im Rahmen dieser Arbeit wurden die Gradientenabstiegsverfahren NA-RMSProp und RRMSProp der Bibliothek hinzugefügt. Alle Gradientenabstiegsverfahren innerhalb der CUVNET benutzen die bereitgestellten Möglichkeiten zur Differenzierung.

Um genauere Angaben über den Lernfortschritt und über die Entwicklung der Gewichte, des Gradienten und der Lernraten zu treffen, war es sinnvoll genau diese Werte während des Lernens abzuspeichern. Zu diesem Zweck wurden die genutzten Gradientenabstiegsverfahren mit Tracern versehen, welche die gewünschten Entwicklungen für zehn zufällig gewählte Gewichte nach jeder Gewichtsanpassung herrausschreiben.

Zusätzlich wurde die CUVNET um einen Crossvalidator erweitert. Dieser führt die Kreuzvalidierung durch und sammelt Ergebnisse aus jedem Fold. Die Ergebnisse können durch eine entsprechend implementierte Funktionalität an Hyperopt über eine Datenbank übermittelt werden

6.6.2 Das Learner-Konzept

Für das Training eines Modells werden innerhalb der CUVNET die folgenden drei Komponenten benötigt:

- Das zu trainierende Modell, das Daten durch die n-dimensionalen Datenstruckturen der CUV repräsentiert werden können. Die einzelnen Arbeitsschritte innerhalb des Netzes können mit Operationen der CUVNET realisiert werden.
- Der Datensatz, aus dem die Trainingsmuster stammen.
- Das Optimierungsverfahren, mit dessen Hilfe die Gewichte des Netzes eingestellt werden sollen.

Diese einzelnen Komponenten werden in einem Learner zusammengeführt. Dieser stellt einen vom Benutzer vorgegebenen Datensatz und das Lernverfahren zur Verfügung. Die Hyperparameter eines Verfahrens können innerhalb des Learners konfiguriert werden. Auch die benötigte Bedingung für einen Lernabbruch kann innerhalb des Learners definiert werden.

Der Benutzer muss bei der Verwendung des Learners innerhalb der CUVNET das Modell definieren, angeben mit welchen Daten dieses trainiert werden soll, den Lernalgorithmus bestimmen und diesem passende Hyperparameter zuweisen. Die Abbruchbedingung für das Lernen wird in allen Experimenten so festgelegt, dass ein Abbruch erfolgt, sobald sich keine signifikante Verbesserung des Klassifikationsfehlers auf einer definierten Validierungsmenge mehr einstellt (Early-Stopping).

Im entwickelten Crossvalidator wird der Learner zur Abarbeitung der einzelnen Folds genutzt.

6.6.3 Hyperoptclients

Für die Verwendung von Hyperopt werden Clients benötigt, welche die Kommunikation mit der Hyperopt-Datenbank sicherstellen und mit den ermittelten Hyperparametern eine Kreuzvalidierung starten können. Die Rahmenbedingung für diese Clients stellt MDBQ (Schulz, 2014b) zur Verfügung. Bei der Implementierung eines solchen Clients muss die zu verwendende Datenbank angegeben werden und welche Arbeitsschritte ausgeführt werden sollen, sobald eine neue Hyperparameterkombination aus der Datenbank gelesen wurde. In dieser Arbeit wird, sobald der Client neue Hyperparameter erhalten hat, eine Kreuzvalidierung durchgeführt und anschließend die erzielten Ergebnisse zurück in die Datenbank geschrieben.

Für die Hyperparameteroptimierung mussten somit folgende Komponenten implementiert werden:

- Ein Python-Skript, in dem die Suchräume für die einzelnen Hyperparameter definiert werden
- Die Kreuzvalidierung, welche die Hyperparameter bewerten soll
- Ein Hyperoptclient, welcher die Hyperparametersuche und die Auswertung der Hyperparameter miteinander verbindet.

Nachdem die gesamte Kreuzvalidierung abgeschlossen wurde, können die besten Hyperparameter aus der Datenbank gelesen werden und innerhalb eines Retrainings verwendet werden. Auch hierfür liefert MDBQ die erforderliche Funktionalität. Der benötigte Retrainer definiert einen entsprechenden Learner mit den bereitgestellten Hyperparametern und trainiert ein Modell von Beginn an mit der gesamten Trainingsmenge. Auch ein solcher Retrainer musste im Rahmen dieser Arbeit implementiert werden.

Die Abhängigkeiten zwischen CUV, CUVNET, Hyperoptclient und den vom Benutzer zu definierenden Modellen und Datensätzen werden in Abbildung 6.6 verdeutlicht.

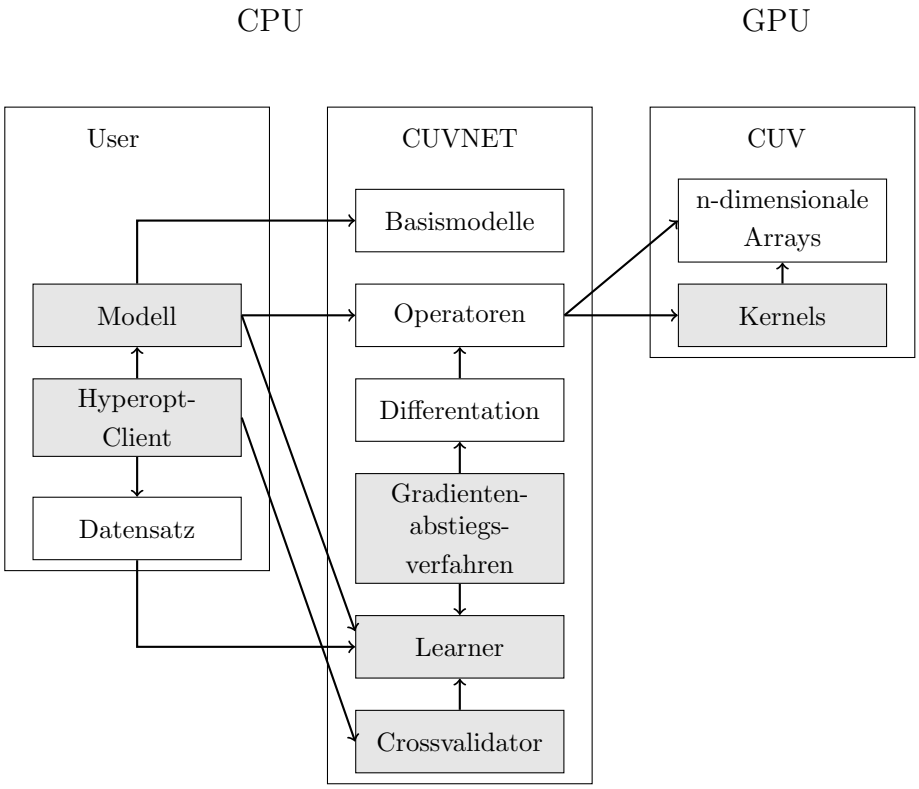


Abbildung 6.6: Abhängigkeiten zwischen den zur Implementierung benötigten Komponenten. Die Abhängigkeit erstreckt sich auf drei Ebenen zwischen CUV, CUVNET und der userspezifischen Implementierung. Die farblich gekennzeichneten Knoten wurden im Rahmen dieser Arbeit weiterentwickelt oder implementiert. Eine Verbindung von einem Knoten A zu einem Knoten B ist so zu verstehen, dass A abhängig von B ist.

7 Ergebnisse

In diesem Kapitel werden die erzielten Ergebnisse präsentiert. Zunächst werden für jeden Algorithmus die Ergebnisse separat in Bezug auf ihre Hyperparameterempfindlichkeit, Trainingsdauer und Entwicklung der Lernraten und Gradienten diskutiert. Anschießend werden die Algorithmen miteinander verglichen und Schlussfolgerungen gezogen. Diese nehmen direkten Bezug auf die im Kapitel 5 aufgestellten Hypothesen. Ausführliche Tabellen und Bilder zu den kreuzvalidierten Belegungen der Hyperparameter aus allen durchgeführten Experimenten sind im Anhang zu finden.

7.1 AdaGrad

AdaGrad wurde mit allen vorgestellten Netzarchitekturen getestet. Die erzielten Ergebnisse werden in Tabelle 7.1 aufgezeigt.

In allen Experimenten reagierte der Algorithmus empfindlich auf die Wahl der Lernrate (Abbildung 7.1). Die Vermutung, dass AdaGrad bessere Ergebnisse mit eher großen Lernraten erzielt, wurde nicht bestätigt (Hypothese 4). Betrachtet man die besten Lernraten aus Tabelle 7.1, stellt man fest, dass alle Werte kleiner 0,2 gewählt wurden. Die Verwendung von Lernraten mit Werten > 1 zeigte keine nennenswerte Verbesserung der Trainingsleistung. In beinahe allen Experimenten zeigte sich, dass die besten Ergebnisse mit eher kleinen Lernraten erzielt wurden. Es wurde auch festgestellt, dass λ ein weiterer empfindlicher Parameter ist. Die Intervalle, aus denen die besten Werte für λ gewählt wurden, sind nahezu identisch. Dieser Parameter besitzt jedoch in jedem Experiment eine unterschiedlich stark ausgeprägte Wichtigkeit. So ist aufgefallen, dass in den Experimenten MNIST-2 und CNN-MNIST der Parameter unwichtig ist, wohingegen in den restlichen Experimenten der Parameter für die Minimierung des Klassifikationsfehlers eine Rolle spielte. Der Stabilisierungsparameter δ und die Batchgröße zeigten keine relevante Empfindlichkeit. Eine deutliche Korrelation der einzelnen Hyperparameter

	MNIST-1	MNIST-2	CIFAR-1
η	$2{,}46 \cdot 10^{-2}$	$1{,}58 \cdot 10^{-1}$	$1{,}38 \cdot 10^{-3}$
λ	$3{,}37 \cdot 10^{-5}$	$4{,}22 \cdot 10^{-6}$	$9{,}06 \cdot 10^{-5}$
δ	$4{,}68 \cdot 10^{-3}$	$3{,}20 \cdot 10^{-2}$	$1{,}10 \cdot 10^{-2}$
bs	750	30	500
	CNN-MNIST	CNN-CIFAR	
η	$1{,}45 \cdot 10^{-2}$	$5{,}98 \cdot 10^{-3}$	
λ	$2{,}90 \cdot 10^{-6}$	$1{,}00 \cdot 10^{-5}$	
δ	$5{,}06 \cdot 10^{-5}$	$1{,}06 \cdot 10^{-2}$	
bs	480	5	

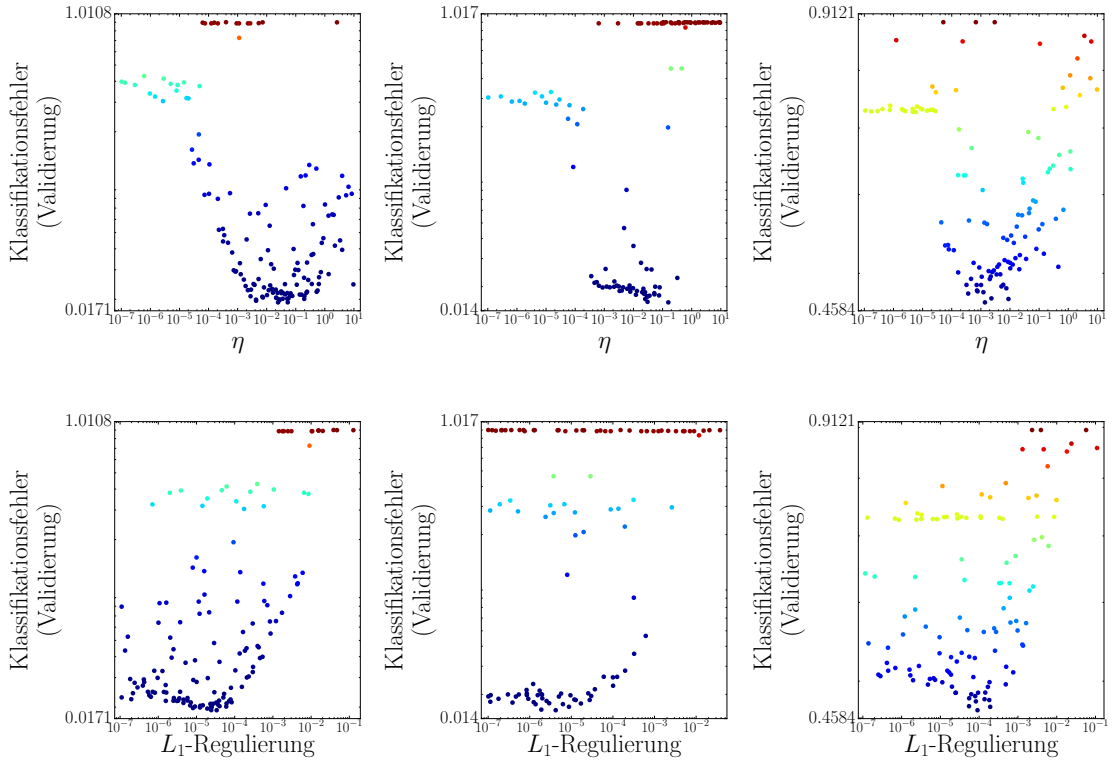
Tabelle 7.1: Aufgelistet sind die Hyperparameterkombinationen für AdaGrad, welche im jeweiligen Experiment den geringsten Klassifikationsfehler auf der Validierungsmenge lieferten.

untereinander konnte nicht festgestellt werden.

Die Suche nach geeigneten Hyperparametern gestaltete sich bei Netzarchitekturen mit mehr als einer verdeckten Schicht schwierig (Abbildung 7.2). Ein möglicher Grund hierfür ist, dass die Bereiche, aus denen die besten Werte für die (empfindlichen) Hyperparameter η und λ gezogen wurden, eng ausfallen. Vor allem die Bereiche, aus denen die besten Werte für die Lernrate gezogen wurden, erstreckten sich in nahezu allen Experimenten nur über eine Zehnerpotenz. Im Experiment MNIST-1, war AdaGrad am einfachsten zu optimieren. Gerade in diesem Experiment ist der Bereich, aus dem die besten Werte für η gezogen wurden, auch am größten.

Eine Aussage über die benötigte Trainingszeit lässt sich nur schwer tätigen. Zwar kann sich mit der richtigen Hyperparameterkombination recht schnell ein akzeptabler Lernerfolg einstellen, da jedoch eine solche Kombination nur schwer zu finden ist, benötigt das Training in der Regel eine relativ große Anzahl an Epochen. Die besten Klassifikationsfehler während der Kreuzvalidierung wurden für MNIST-1 nach 460 Epochen, in MNIST-2 bereits nach 59 Epochen und CIFAR-1 nach 2644 Epochen erzielt. Auch hier spiegelt sich die Empfindlichkeit gegenüber der Lernrate wider.

Betrachtet man den Verlauf des AdaGrad-Trainings (Abbildung 7.3), so fällt in jedem durchgeführten Experiment auf, dass die Werte des Gradienten für jedes Gewicht nur in der letzten Gewichtsschicht abnehmen. In allen darüber liegenden Schichten, ist keine bemerkbare Abnahme dieser Werte zu verzeichnen. Die Gra-



(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.1: Auftragung der kreuzvalidierten AdaGrad-Hyperparameter η und der l_1-Regulierungskonstante λ gegen den resultierenden Klassifikationsfehler auf der Validierungsmenge in den Experimenten MNIST-1, MNIST-2 und CIFAR-1. Auf beide Hyperparameter reagiert AdaGrad empfindlich.

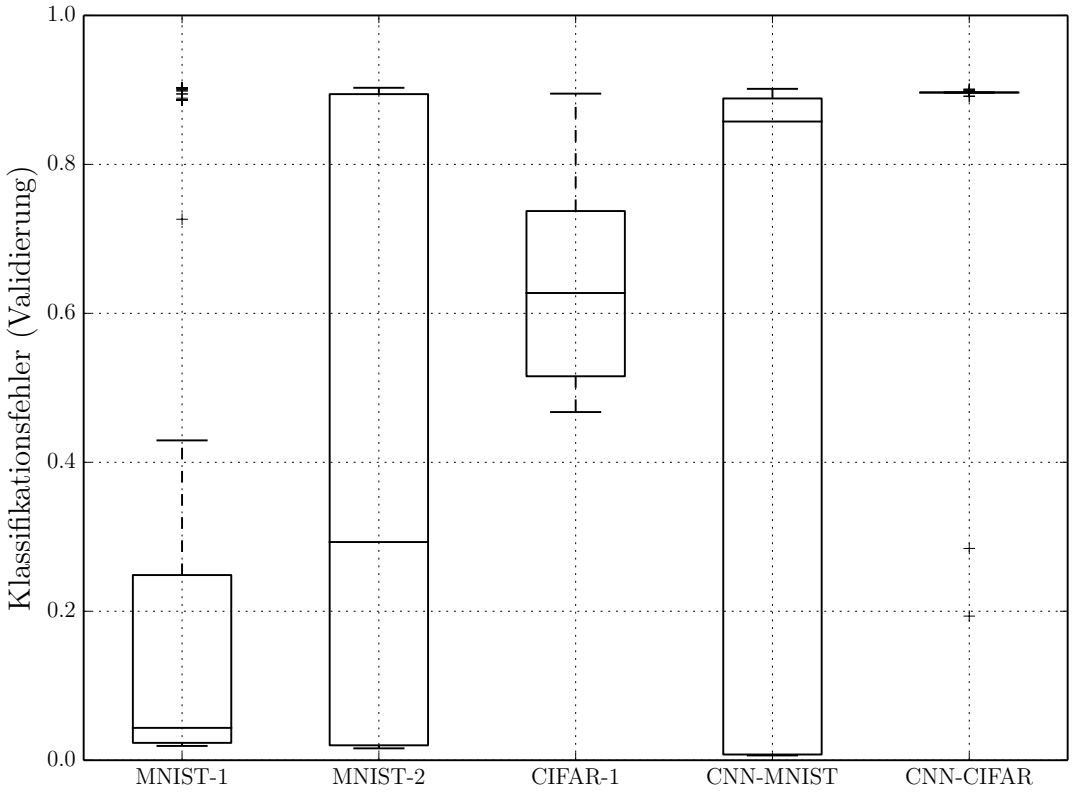
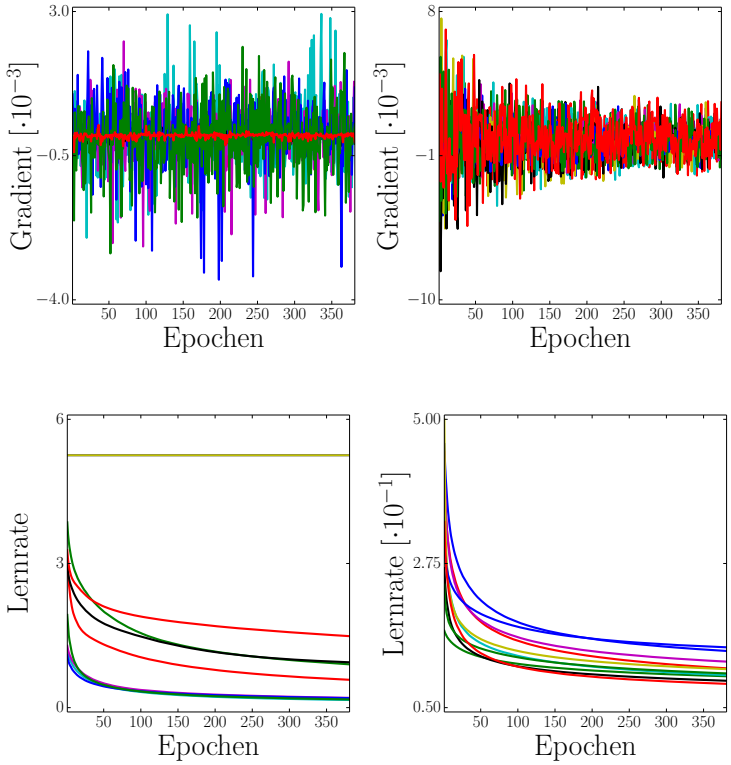


Abbildung 7.2: Verteilung der durch AdaGrad erzielten Klassifikationsfehler auf der Validierungsmenge nach abgeschlossener Kreuzvalidierung. Für Netzarchitekturen mit mehr als einer verdeckten Schicht zeigt sich, dass AdaGrad immer schwieriger zu optimieren ist.

dientenkomponenten schwanken in diesen Schichten sehr stark. Geht man davon aus, dass im Laufe des Trainings ein Minimum auf der Fehleroberfläche angesteuert wird, so ist zu erwarten, dass der Gradient mit zunehmender Epochenanzahl abnimmt. Da ein solches Verhalten nur in der Outputschicht bemerkbar ist, kann dies ein Hinweis darauf sein, dass AdaGrad die Schichten über der Outputschicht nicht ausreichend optimiert. Auf Schwankungen im Gradienten kann die Lernrate nicht reagieren. Betrachtet man die Entwicklung der AdaGrad-Lernrate, so stellt man fest, dass sie mit zunehmender Epochenanzahl exponentiell fällt.

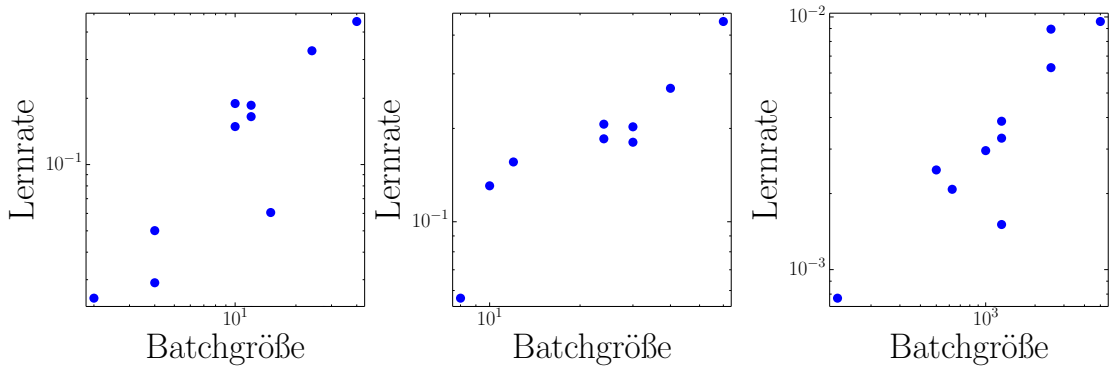


(a) Layer 1 **(b)** Layer 2

Abbildung 7.3: Entwicklung der AdaGrad-Lernraten und Gradientenkomponenten im Experiment MNIST-1 für zehn zufällig gewählte Gewichte in jeder Schicht. Zu sehen ist, dass die Werte der Gradientenkomponenten nur in der letzten Gewichtsschicht gegen 0 tendieren. In der ersten Schicht zeigt sich eine solche Tendenz nicht. Die entsprechenden Lernraten fallen exponentiell bei steigender Epochenanzahl. In allen anderen Experimenten wurde ein ähnliches Verhalten beobachtet.

	MNIST-1	MNIST-2	CIFAR-1
η	$1{,}89 \cdot 10^{-1}$	$1{,}85 \cdot 10^{-1}$	$9{,}59 \cdot 10^{-3}$
bs	10	24	5000
	CNN-MNIST	CNN-CIFAR	
η	$3{,}34 \cdot 10^{-2}$	$4{,}27 \cdot 10^{-3}$	
bs	400	5	

Tabelle 7.2: Aufgelistet sind die Hyperparamterkombinationen für GD, welche im jeweiligen Experiment den geringsten Klassifikationsfehler auf der Validierungsmenge lieferten.



(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.4: Auftragung der besten zehn Werte aus den MLP-Experimenten für die Batchgröße gegen η in GD, mit logarithmischer Achsenskalierung. In allen Bildern zeigt sich eine klare Proportionalität zwischen der Batchgröße und der Lernrate.

7.2 GD

Der klassische Gradientenabstieg wurde auf alle Netzarchitekturen und Datensätze angewendet. Der Algorithmus ist der einzige, der keine adaptive Lernrate besitzt. In diesem Fall ist die Lernrate eine skalare Konstante. Dementsprechend arbeitet dieses Lernverfahren nur mit zwei Hyperparametern, der Lernrate und der Größe der Minibatches. Diejenigen Hyperparameterkombinationen, welche den geringsten Validierungsfehler lieferten, sind in Tabelle 7.2 aufgeführt.

Trotz der konstant gewählten Lernrate zeigte sich der Algorithmus bei allen MLP-Experimenten sehr robust. Die Suche nach geeigneten Werten für η gestaltete sich in diesen Experimenten als sehr einfach. Für die Wahl der Batchgröße ließen sich keine allgemeingültigen Aussagen treffen. Auffällig war, dass in den Experimenten MNIST-1 und MNIST-2 sehr kleine Batchgrößen zu guten Ergebnissen führten. Bei den restlichen Experimenten erstreckte sich das Intervall, in dem sich die zehn besten Batchgrößen befanden, fast über den gesamten Suchraum. Bemerkenswert ist auch, dass die Hyperparametersuche für die Experimente MNIST-1 und MNIST-2 annähernd die gleichen Hyperparameter geliefert hat. Es ist weiterhin aufgefallen, dass scheinbar in allen MLP-Experimenten eine proportionale Korrelation zwischen der Lernrate und der Größe der Minibatches existierte (Abbildung 7.4).

Bei den Experimenten mit konvolutionalen Netzarchitekturen zeigte der Algorithmus große Schwächen. So ist das Intervall, in dem diejenigen Werte für η enthal-

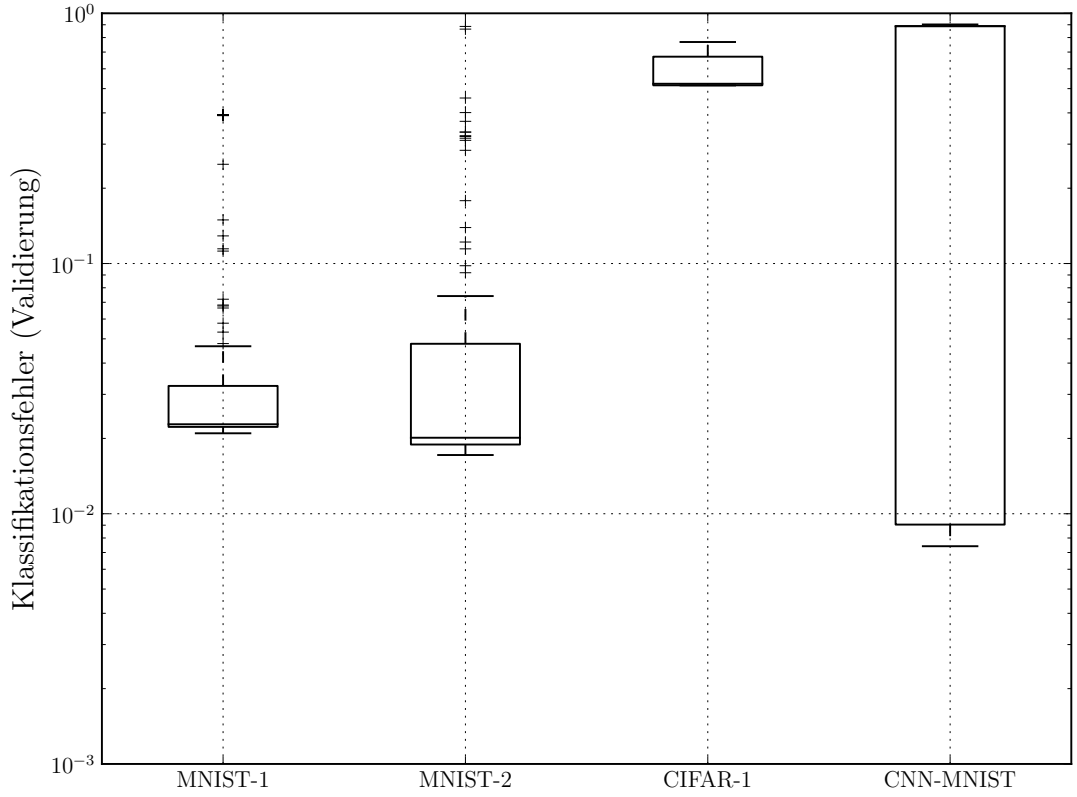


Abbildung 7.5: Verteilung der durch GD erzielten Klassifikationsfehler auf der Validierungsmenge nach abgeschlossener Kreuzvalidierung. Der Median im Experiment CNN-MNIST liegt bei 0,89. Bemerkenswert ist, dass die Verteilung in den MLP-Experimenten sehr eng ausfällt und, dass der Median in diesen Experimenten sehr nah am minimalen Klassifikationsfehler auf der Validierungsmenge liegt. In den konvolutionalen Experimenten zeigt sich ein inverses Bild. Der Median liegt sehr nahe am schlechtesten erzielten Klassifikationsfehler.

ten sind, die akzeptable Klassifikationsfehler auf der Validierungsmenge lieferten, sehr eng im Vergleich zu denen aus den MLP-Experimenten. Diese Ambivalenz wird auch in Abbildung 7.5 deutlich. In den Experimenten mit Konvolutionsnetzen zeigte sich keine entscheidende Empfindlichkeit gegenüber der Wahl für die Batchgröße. Auch konnte keine Korrelation zwischen der Lernrate und der Batchgröße festgestellt werden.

Fasst man die Diskussion für alle Experimente zusammen, so kann man die Lernrate als einzigen empfindlichen Hyperparameter identifizieren. Es könnte eine mögliche Korrelation zwischen der Lernrate und der Batchgröße existieren, die jedoch in den konvolutionalen Experimenten nicht bestätigt werden konnte. Die

	MNIST-1	MNIST-2	CIFAR-1
ρ	$4{,}63 \cdot 10^{-5}$	$1{,}23 \cdot 10^{-3}$	$9{,}95 \cdot 10^{-1}$
η	$7{,}38 \cdot 10^{-3}$	$1{,}18 \cdot 10^{-3}$	$1{,}74 \cdot 10^{-5}$
δ	$6{,}72 \cdot 10^{-2}$	$2{,}21 \cdot 10^{-2}$	$1{,}06 \cdot 10^{-4}$
bs	8	2	200
	CNN-MNIST	CNN-CIFAR	
ρ	$5{,}31 \cdot 10^{-2}$	$2{,}12 \cdot 10^{-1}$	
η	$7{,}52 \cdot 10^{-4}$	$1{,}24 \cdot 10^{-3}$	
δ	$1{,}07 \cdot 10^{-2}$	$1{,}43 \cdot 10^{-1}$	
bs	120	16	

Tabelle 7.3: Aufgelistet sind die Hyperparamterkombinationen für RMSProp, welche im jeweiligen Experiment den geringsten Klassifikationsfehler auf der Validierungsmenge lieferten.

Robustheit des Algorithmus in den MLP-Experimenten kann vielleicht mit der geringen Anzahl an Hyperparametern erklärt werden. Dadurch ergeben sich nur kleine Hyperparameterkombinationen, in denen nur die Lernrate als empfindlich identifiziert werden kann.

Ähnlich zu AdaGrad lässt sich auch bei diesem Algorithmus keine klare Aussage über die benötigte Trainingszeit treffen. Das Training kann hierbei auch mit einer guten Hyperparameterkombination eine relativ lange Zeit benötigen, bis sich eine Konvergenz des Klassifikationsfehlers einstellt.

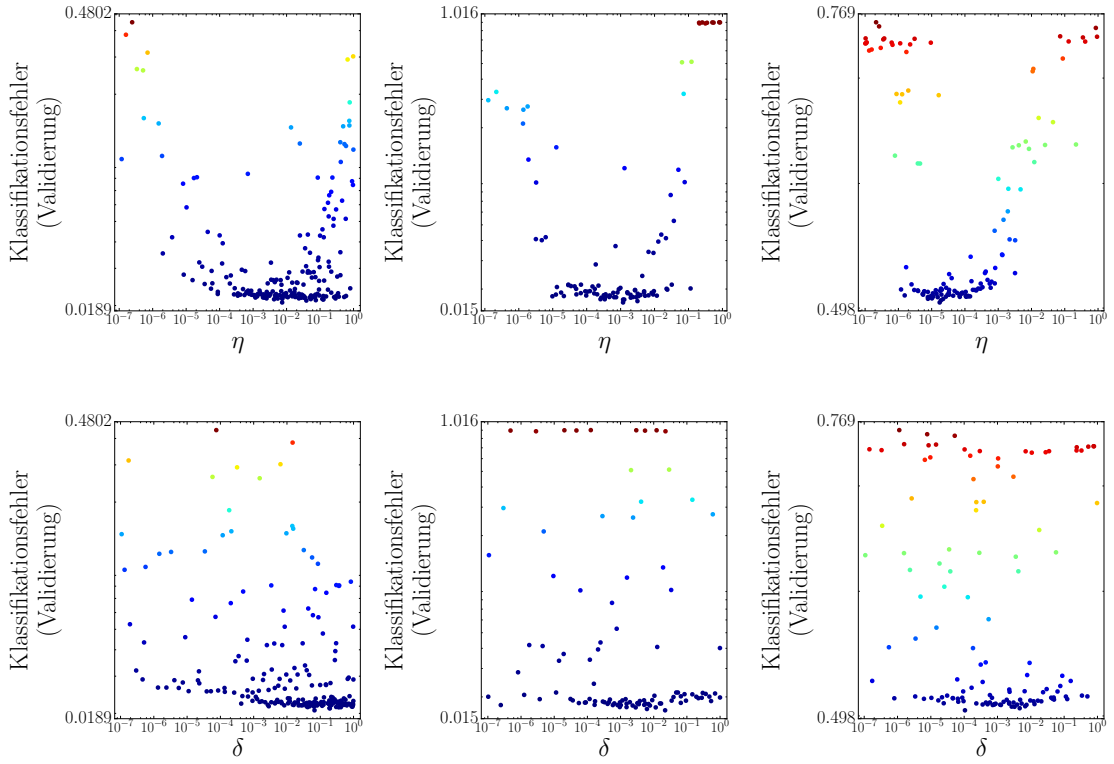
Bei der Betrachtung des Lernverlaufs zeigte sich kein charakteristisches Verhalten für die Entwicklung des Gradienten.

7.3 RMSProp

Das RMSProp-Lernverfahren wurde mit allen Netzarchitekturen und Datensätzen getestet. Die besten gefundenen Hyperparameter sind in Tabelle 7.3 aufgelistet.

In allen Experimenten zeigte RMSProp ein sehr robustes Verhalten. Als einziger nicht empfindlicher Hyperparameter konnte die Mean-Square-Konstante ρ identifiziert werden. Bei der Hyperparametersuche nahm ρ Werte aus dem gesamten Suchintervall an. Selbst bei der Betrachtung der jeweils besten Werte für ρ (also diejenigen, die den besten Klassifikationsfehler auf der Validierungsmenge) in jedem Experiment, zeigte sich keine Tendenz.

Für alle weiteren Hyperparameter zeigte der Algorithmus ein empfindliches Verhalten (Abbildung 7.6). Bei der Belegung für die Lernrate η zeigte sich, dass hier sehr kleine Werte ($< 10^{-2}$) die besten Ergebnisse liefern. Die Intervalle, aus denen die besten Werte für die Belegung von η stammen, sind sehr breit und erstrecken sich in einigen Experimenten sogar über den Bereich von mehr als zwei Zehnerpotenzen. Der Stabilisierungsparameter δ zeigte sich in den MLP-Experimenten empfindlich, jedoch ist auch hier zu bemerken, dass die entsprechenden Intervalle sehr breit ausfallen. Nur in den konvolutionalen Experimenten nahm δ Werte aus dem gesamten Suchbereich an. Die besten Klassifikationsfehler auf der Validierungsmenge lieferte RMSProp mit einer eher kleinen Batchgröße (< 200). Betrachtet man die Intervalle, in denen die zehn besten Werte zur Belegung der Batchgröße beinhaltet sind, lässt sich diese Aussage bestätigen.

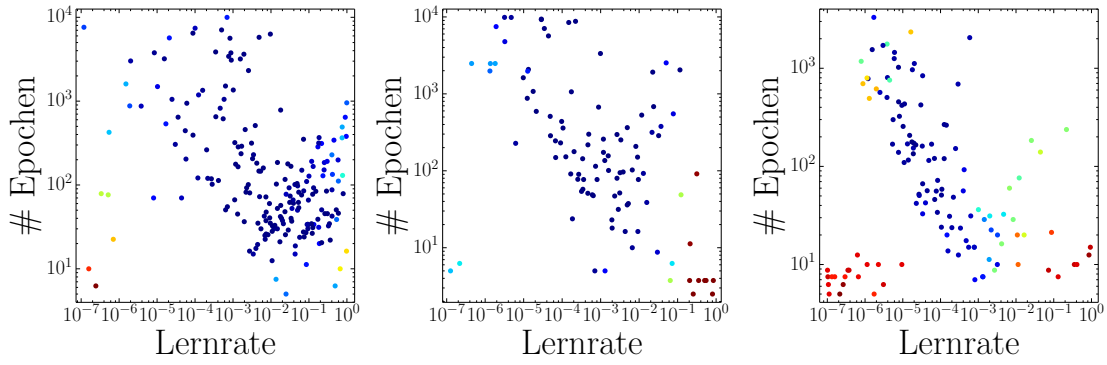


(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.6: Auftragung der kreuzvalidierten RMSProp-Hyperparameter η und δ gegen den resultierenden Klassifikationsfehler auf der Validierungsmenge in den Experimenten MNIST-1, MNIST-2 und CIFAR-1. Auf beide Hyperparameter reagiert RMSProp empfindlich, jedoch können geeignete Belgungen auf breiten Intervallen gewählt werden.

Es wurde zwar festgestellt, dass der Algorithmus auf drei seiner vier Hyperparameter empfindlich reagiert, jedoch fallen für diese Parameter die entscheidenden Intervalle breit aus. Dadurch wird die Suche nach geeigneten Hyperparametern deutlich erleichtert. Diese Tatsache bringt RMSProp einen entscheidenden Vorteil ein. Dies zeigt sich vor allem bei der Verwendung mit komplexen und rechenaufwendigen Netzarchitekturen, wie sie die hier verwendeten Konvolutionsnetze bilden. Trainiert man solche Netze mit RMSProp, kann sehr schnell eine Hyperparameteroptimierung durchgeführt werden. Die besonders einfache Hyperparametersuche zeigt sich in Abbildung 7.8.

Betrachtet man die Trainingsdauer, so kann man eine Beziehung zwischen der Wahl der Lernrate und der Anzahl an Trainingsepochen feststellen. Bei einer gut eingestellten Lernrate zeigt sich, dass der Algorithmus schnell konvergiert. Somit ist die Wahl der Lernrate wichtig für die Trainingsdauer. Mit einer entsprechenden Belegung von η, kann eine Minimierung der Epochenanzahl erreicht werden. Abbildung 7.7 verdeutlicht diesen Aspekt. Durch die Verwendung der besten Hyperparameter konnte in den hier durchgeführten Experimenten eine Konvergenz bereits nach relativ wenigen Epochen erreicht werden.



(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.7: Auftragung aller untersuchten Lernraten gegen die resultierende benötigte Epochenanzahl für alle MLP-Experimente mit RMSProp bei logarithmischer Achsenskalierung. In allen Bildern zeigt sich, dass die Anzahl der benötigten Epochen durch eine geeignete Belegung von η minimiert werden kann. Die Farbskalierung entspricht dem Klassifikationsfehler auf der Validierungsmenge.

Durch die Betrachtung des Lernverlaufs wird deutlich, dass die RMSProp Lernrate auf Schwankungen im Gradienten reagieren kann. Der Grund hierfür ist, dass die Mean-Square Komponente von RMSProp nicht nur einen historischen Aspekt besitzt, sondern auch auf aktuelle Werte des Gradienten reagieren kann. Dieses

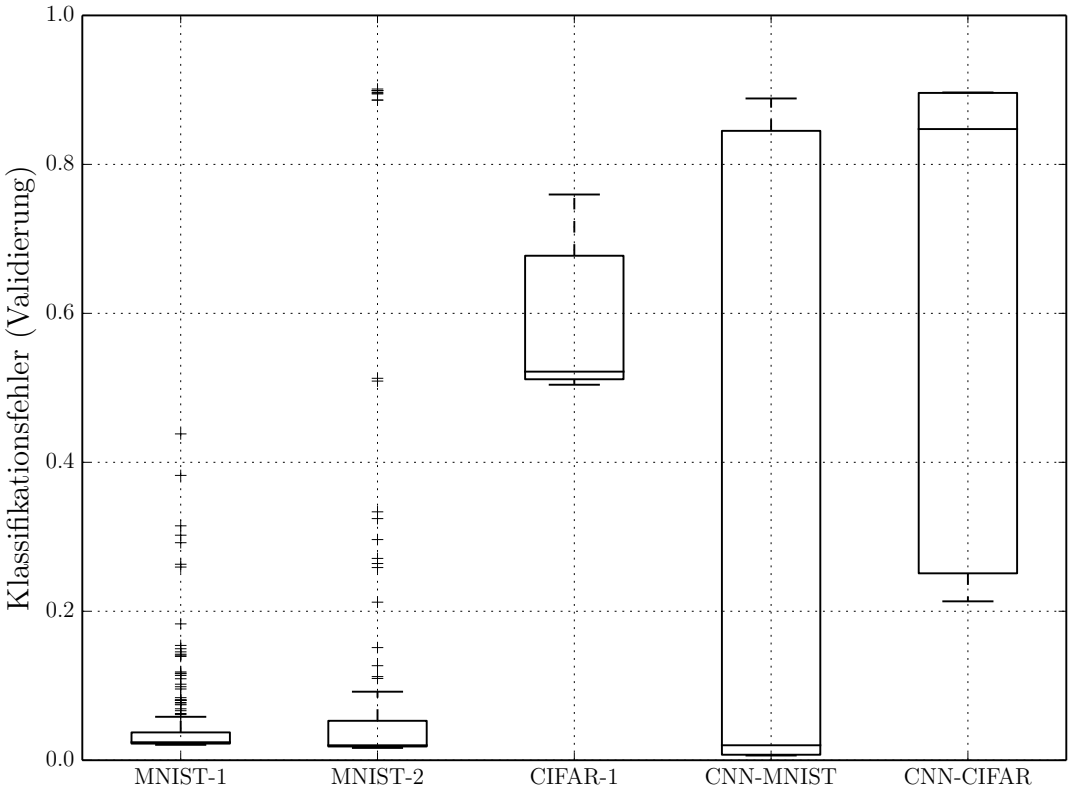
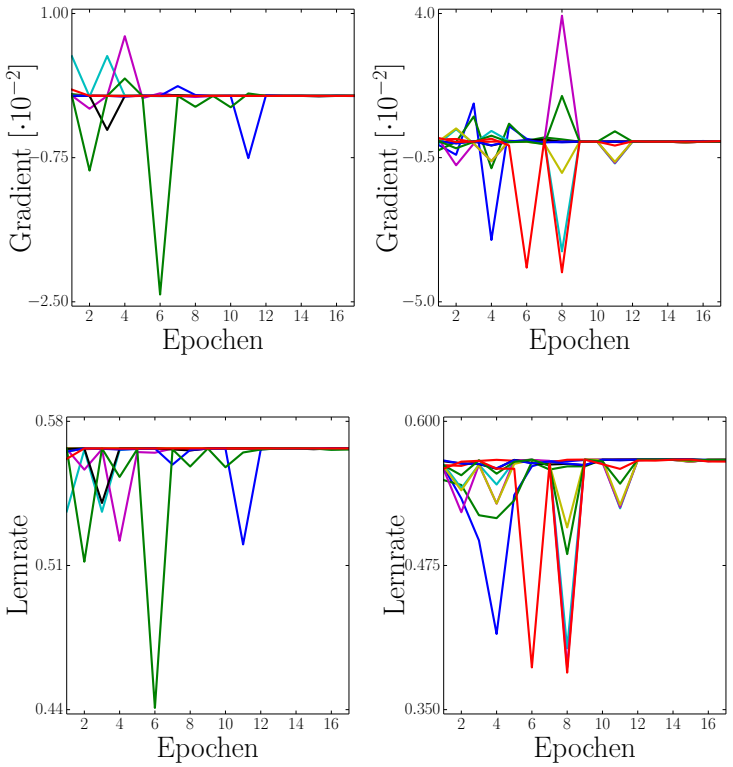


Abbildung 7.8: Verteilung der durch RMSProp erzielten Klassifikationsfehler auf der Validierungsmenge nach abgeschlossener Kreuzvalidierung. Es zeigt sich, dass sich die Verteilungen (besonders in den MLP-Experimenten) in einem sehr engen Bereich befinden. Bemerkenswert ist, dass sich der Median in jedem Experiment sehr nahe am minimalen erreichten Klassifikationsfehler auf der Validierungsmenge befindet.

Verhalten ist vor allem bei großen Schwankungen im Gradienten von Vorteil. Eine solche Schwankung kann durch den Mean-Square kompensiert werden, sodass der resultierende Updateschritt nicht zu groß ausfällt. Abbildung 7.9 verdeutlicht dieses Verhalten. Sind über einem längeren Zeitraum keine oder nur geringe Schwankungen zu verzeichnen, so nimmt die Lernrate einen annähernd konstanten Wert an. Gerade in solchen Fällen ist es wünschenswert, dass die Lernrate sensibler auf kleine Schwankungen reagieren soll, um ein verfeinertes Lernen zu ermöglichen. Ein solches Verhalten weist RMSProp nicht auf. Eine mögliche Modifikation ist, das Zeitfenster im Mean-Square zu verkleinern, so dass aktuelle Werte des Gradienten stärker berücksichtigt werden können.



(a) Layer 1 **(b)** Layer 2

Abbildung 7.9: Entwicklung der RMSPROP-Lernraten und Gradientenkomponenten im Experiment MNIST-1 für zehn zufällig gewählte Gewichte in jedem Layer. Gut zu sehen ist, dass die Lernrate auf die aktuelle Größe der Gradientenkomponente reagiert. Bemerkenswert ist, dass sobald die Gradientenkomponenten annähernd konstant bleiben auch die Lernrate sich dementsprechend verhält.

7.4 NA-RMSProp

NA-RMSProp wurde auf alle Netzarchitekturen und Datensätze angewandt. Die resultierenden Hyperparameterkombinationen sind in Tabelle 7.4 zu finden.

Der Algorithmus zeigt sich empfindlich gegenüber allen Hyperparametern. In nahezu allen Experimenten zeigte sich, dass η mit zum Teil sehr kleinen Werten belegt werden muss. Jedoch erstreckt sich, ähnlich zu RMSProp, das Intervall in dem die besten Lernraten liegen über mehrere Zehnerpotenzen. Die Mean-Square-Konstante ρ zeigt nur eine leichte Empfindlichkeit in den Experimenten MNIST-2 und CIFAR-1. Bei der Durchführung der restlichen Experimente zeigte sich der Algorithmus unempfindlich gegenüber diesem Hyperparameter. Die Stabilisierungskonstante δ verhält sich ähnlich zur Lernrate. Zwar ist δ ein empfindlicher Hyper-

	MNIST-1	MNIST-2	CIFAR-1
ρ	$2{,}75 \cdot 10^{-4}$	$2{,}79 \cdot 10^{-1}$	$6{,}84 \cdot 10^{-7}$
η	$1{,}79 \cdot 10^{-1}$	$1{,}72 \cdot 10^{-3}$	$1{,}82 \cdot 10^{-6}$
γ	$1{,}34 \cdot 10^{-7}$	$3{,}34 \cdot 10^{-7}$	$3{,}47 \cdot 10^{-7}$
μ	$1{,}17 \cdot 10^{-5}$	$7{,}75 \cdot 10^{-7}$	$6{,}94 \cdot 10^{-6}$
δ	$4{,}20 \cdot 10^{-1}$	$3{,}11 \cdot 10^{-4}$	$2{,}17 \cdot 10^{-3}$
bs	30	12	125
	CNN-MNIST	CNN-CIFAR	
ρ	$3{,}19 \cdot 10^{-1}$	$1{,}25 \cdot 10^{-2}$	
η	$3{,}99 \cdot 10^{-3}$	$2{,}09 \cdot 10^{-3}$	
γ	$3{,}72 \cdot 10^{-4}$	$1{,}33 \cdot 10^{-4}$	
μ	$2{,}35 \cdot 10^{-1}$	$2{,}87 \cdot 10^{-3}$	
δ	$2{,}28 \cdot 10^{-7}$	$3{,}22 \cdot 10^{-3}$	
bs	200	25	

Tabelle 7.4: Aufgelistet sind die Hyperparamterkombinationen für NA-RMSProp, welche im jeweiligen Experiment den geringsten Klassifikationsfehler auf der Validierungsmenge lieferten.

parameter, jedoch ist auch hier das relevante Intervall sehr breit. Abbildung 7.14 verdeutlicht die Empfindlichkeit gegenüber η, ρ und δ

Die Hypothese, dass die adaptive Konstante γ einen kleinen Wert annehmen muss, wurde bei allen Experimenten bestätigt (Hypothese 9). So wurden die besten Ergebnisse bei allen MLP-Experimenten mit einem Wert der Größenordnung 10^{-7} und bei konvolutionalen Experimenten mit $\gamma \approx 10^{-4}$ erzielt. Die entsprechenden relevanten Intervalle fallen bei nahezu allen Experimenten identisch aus. Auch hier sind die Intervalle breit.

Die Momentum-Konstante μ ist ein weiterer empfindlicher Hyperparameter. Überraschend wurden die besten Ergebnisse bei allen MLP-Experimenten mit sehr kleinen Werten für μ erzielt. In den konvolotionalen Experimenten zeigte sich der Algorithmus unempfindlich gegenüber μ. Eine grafische Verdeutlichung der Empfindlichkeit von NA-RMSProp gegenüber γ und μ ist in Abbildung 7.13 zu finden.

Betrachtet man die besten Hyperparameterkombinationen, so stellt man fest, dass der Algorithmus die besten Ergebnisse mit relativ kleinen Batchgrößen erzielt hat (< 200). Diese Tendenz wird von den relevanten Intervallen in nahezu allen Experimenten bestätigt.

Die Suche nach geeigneten Hyperparameterkombinationen gestaltete sich für

dieses Lernverfahren einfach (vergleiche Abbildung 7.10). Zwar reagiert der Algorithmus auf alle Hyperparameter empfindlich, jedoch sind akzeptable Hyperparameter in sehr breiten Intervallen innerhalb des Suchbereichs zu finden. Darüber hinaus gibt es noch klare Tendenzen hin zu kleinen Werten für die Batchgröße, die Momentumkonstante μ und die adaptive Konstante γ.

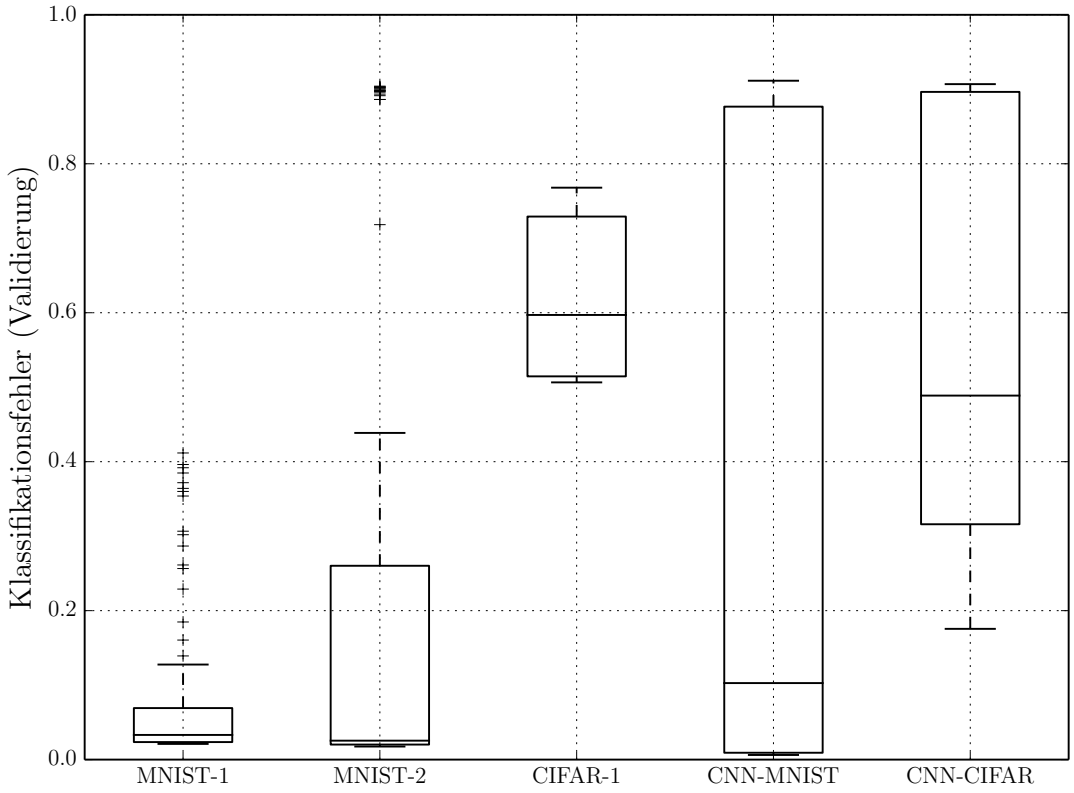
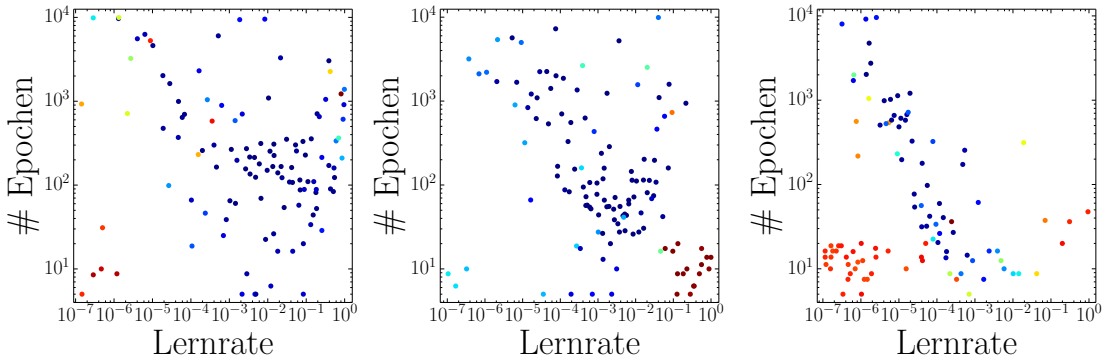


Abbildung 7.10: Verteilung der durch NA-RMSProp erzielten Klassifikationsfehler auf der Validierungsmenge nach abgeschlossener Kreuzvalidierung. Bemerkenswert ist, dass sich der Median in nahezu allen Experimenten sehr nahe am minimalen erreichten Klassifikationsfehler auf der Validierungsmenge befindet, wodurch die Suche nach geeigneten Hyperparametern vereinfacht wird.

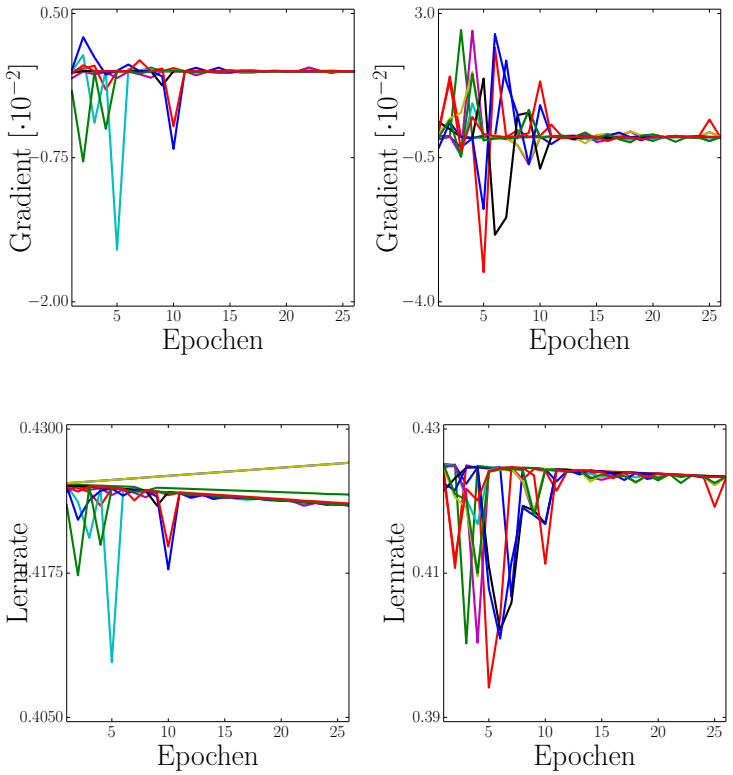
Ähnlich zu RMSProp verhält sich die Trainingsdauer. Hier besteht eine Beziehung zwischen der Lernrate und der benötigten Anzahl an Epochen. Bei einer entsprechenden Wahl der Lernrate kann die Epochenanzahl minimiert werden (Abbildung 7.11).



(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

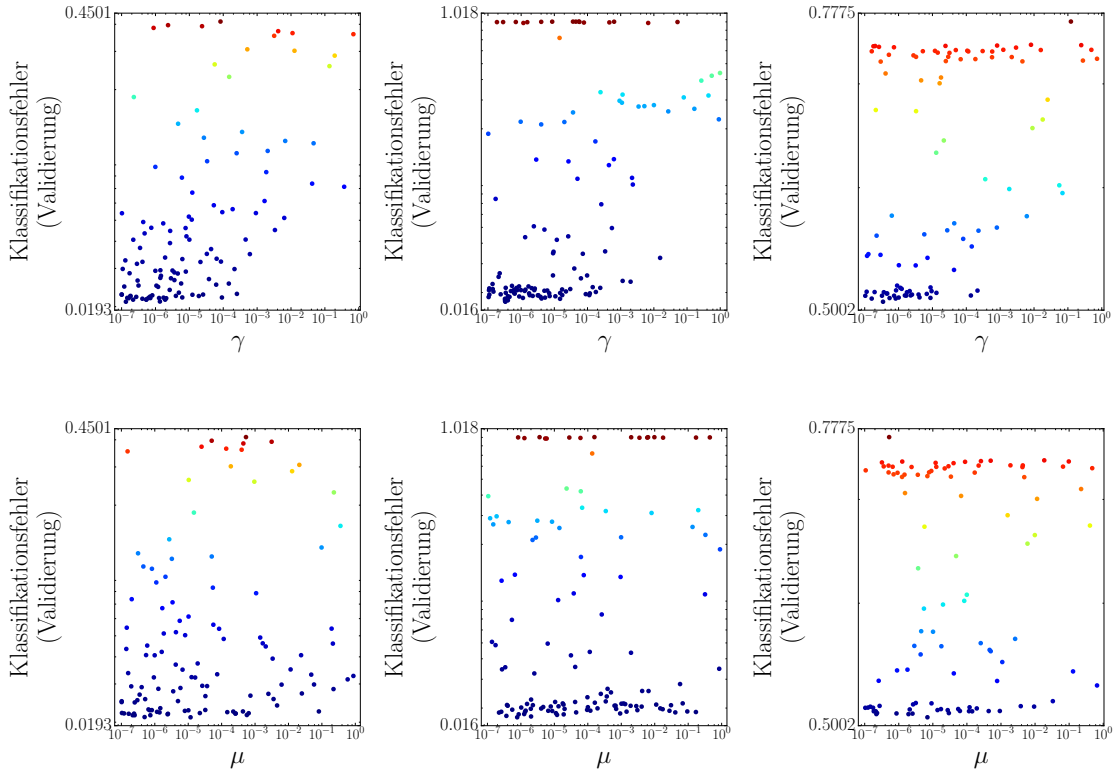
Abbildung 7.11: Auftragung aller in NA-RMSProp untersuchten Lernraten gegen die resultierende benötigte Epochenanzahl für alle MLP-Experimente bei logarithmischer Achsenskalierung. In allen Bildern zeigt sich, dass die Anzahl der benötigten Epochen durch eine geeignete Belegung von η minimiert werden kann. Die Farbskalierung entspricht dem Klassifikationsfehler auf der Validierungsmenge.

Bei der Betrachtung des Lernverlaufs zeigte sich, dass sich die Lernrate nahezu identisch zu der von RMSProp verhält. Jedoch besitzt NA-RMSProp die elastische Komponente γ (Abbildung 7.12). Diese kann dazu führen, dass die globale Lernrate η mit fortschreitender Epochenanzahl verringert wird und dadurch das Verfahren gegenüber kleinen Werten im Gradienten sensibler wird.



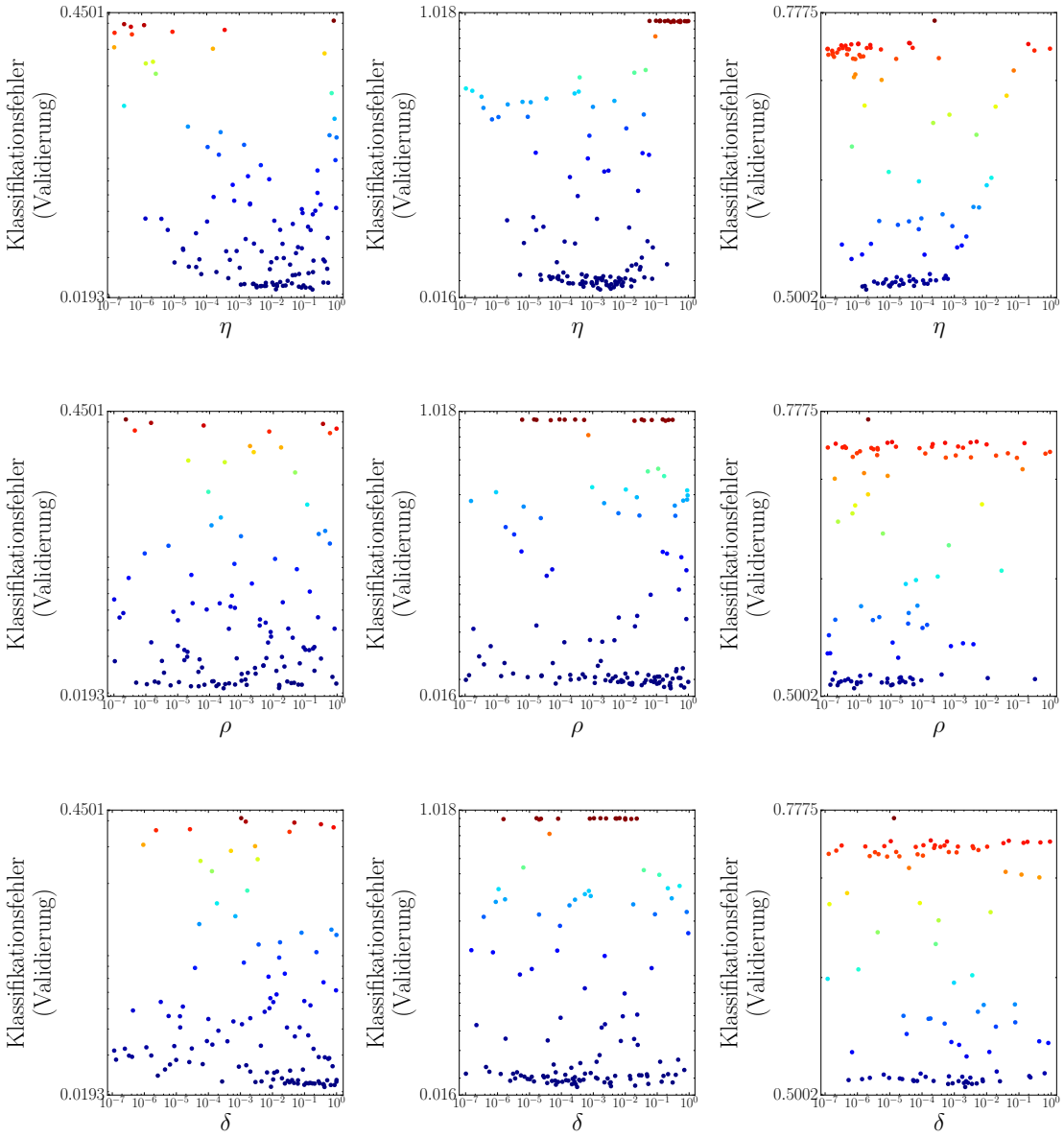
(a) Layer 1 **(b)** Layer 2

Abbildung 7.12: Entwicklung der NA-RMSProp-Lernraten und Gradientenkomponenten im Experiment MNIST-1 für zehn zufällig gewählte Gewichte in jeder Schicht. Gut zu sehen ist, dass die Lernrate auf die aktuelle Größe der Gradientenkomponente reagiert. Bemerkenswert ist, dass sobald die Gradientenkomponenten annähernd konstant bleiben, sich auch die Lernrate dementsprechend verhält.



(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.13: Auftragung der kreuzvalidierten NA-RMSProp-Hyperparameter γ und μ gegen den resultierenden Klassifikationsfehler auf der Validierungsmenge in den Experimenten MNIST-1, MNIST-2 und CIFAR-1. Auf beide Hyperparameter reagiert NA-RMSProp empfindlich, jedoch können geeignete Belegungen aus breiten Intervallen gewählt werden.



(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.14: Auftragung der kreuzvalidierten NA-RMSProp-Hyperparameter η, δ und ρ gegen den resultierenden Klassifikationsfehler auf der Validierungsmenge in den Experimenten MNIST-1, MNIST-2 und CIFAR-1. Auf alle Hyperparameter reagiert NA-RMSProp empfindlich, jedoch können geeignete Belegungen aus breiten Intervallen gewählt werden.

	MNIST-1	MNIST-2
λ	$2{,}81 \cdot 10^{-1}$	$2{,}54 \cdot 10^{-3}$
η^+	1,04	1,00
η^-	$4{,}54 \cdot 10^{-1}$	$4{,}43 \cdot 10^{-1}$
bs^*	30	3000
	CIFAR-1	CNN-MNIST
λ	1,04	$3{,}13 \cdot 10^{-1}$
η^+	1,37	1,45
η^-	$4{,}70 \cdot 10^{-1}$	$4{,}00 \cdot 10^{-1}$
bs	1000	250

Tabelle 7.5: Aufgelistet sind die Hyperparameterkombinationen für RPROP, welche im jeweiligen Experiment den geringsten Klassifikationsfehler auf der Validierungsmenge lieferten.

7.5 RPROP

Das RPROP-Lernverfahren wurde auf allen MLP-Architekturen und auf dem Konvolutionsnetz CNN-MNIST angewendet.

Während der ersten Experimente zeigte sich, dass eine l_2-Regulierung für den Lernverlauf hilfreich war. Daraufhin wurde diese in allen Experimenten übernommen und der Hyperparameter λ evaluiert.

Der Algorithmus zeigte sich insbesondere in den MLP-Experimenten extrem unempfindlich gegenüber den Belegungen seiner Hyperparameter (vergleiche hierzu Abbildung 7.15). Die initiale Belegung der Updatewerte Δ_i wurde, in Anlehnung an die Originalarbeit (Riedmiller und Braun, 1993), auf den Wert 0,1 gesetzt. Akzeptable Werte für die Belegung der multiplikativen Konstanten η^- und η^+ sind nahezu im gesamten Suchraum zu finden, jedoch zeigte sich in allen Experimenten, dass eine Belegung von η^- mit einem Wert um 0,45 die besten Ergebnisse erzielte. In beiden MLP-Experimenten mit dem MNIST-Datensatz zeigte eine Belegung von η^+ mit einem Wert nahe bei 1 die beste Leistung, wohingegen in den restlichen Experimenten ein Wert über 1,3 die besten Ergebnisse erzielte. Zwar wurden die besten Ergebnisse in allen Experimenten nicht mit den Belegungen $\eta^- = 0{,}5$ und $\eta^+ = 1{,}2$ erzielt, jedoch muss bemerkt werden, dass aufgrund der extremen Unempfindlichkeit des Algorithmus gegenüber seinen Hyperparametern auch diese Werte eine gute Trainingsleistung liefern können. Die Hypothese, dass beide Parameter im Minibatchlearning näher an 1 rücken müssen, konnte durch

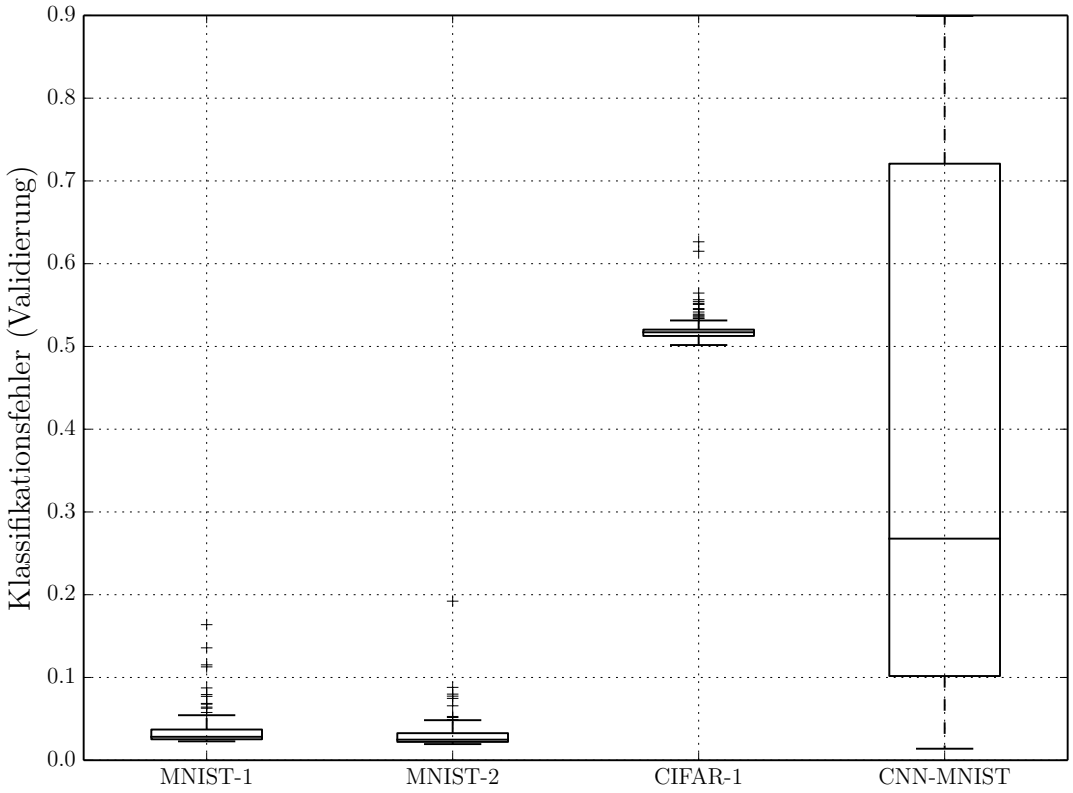


Abbildung 7.15: Verteilung der durch RPROP erzielten Klassifikationsfehler auf der Validierungsmenge nach abgeschlossener Kreuzvalidierung. Vor allem in den MLP-Experimenten zeigen sich extrem dichte Verteilungen des Klassifikationsfehlers.

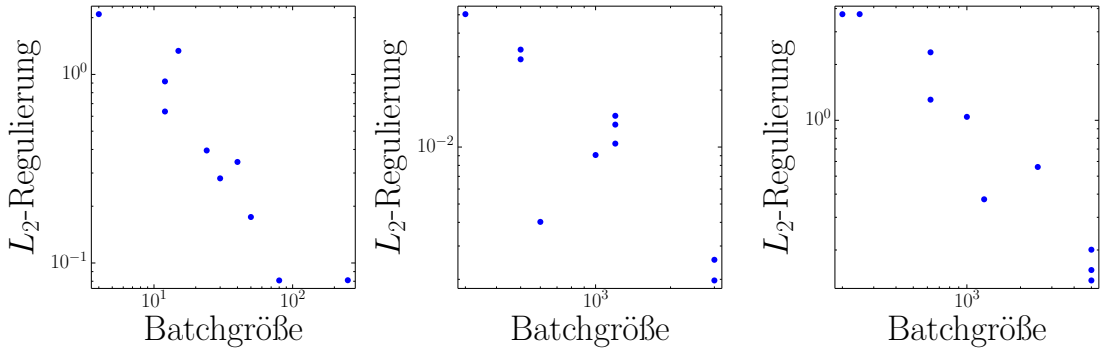
die hier durchgeführten Experimente nicht bestätigt werden (Hypothese 2). Die besten Ergebnisse wurden mit nicht zu kleinen Batchgrößen erzielt. Bemerkenswert ist, dass in MNIST-2 und CIFAR-1 die geringsten Klassifikationsfehler auf der Validierungsmenge mit sehr hohen Werten für die Belegung der Batchgröße erreicht wurden. Aus Tabelle 5 (Anhang) kann man schließen, dass der Algorithmus eher große Batches bevorzugt. Infolgedessen muss Hypothese 1 als bestätigt angesehen werden. λ ist der einzige Hyperparameter, den es einzustellen gilt (Abbildung 7.16).

Aus den zehn besten Hyperparameterkombinationen geht hervor, dass eine Korrelation zwischen der Batchgröße und der Wahl der l_2-Regularisierungskonstante λ existiert. Abbildung 7.16 zeigt, dass es sich um eine antiproportionale Beziehung der beiden Hyperparameter handelt. Weitere Korrelationen zwischen den Hyper-

parametern wurden nicht festgestellt.

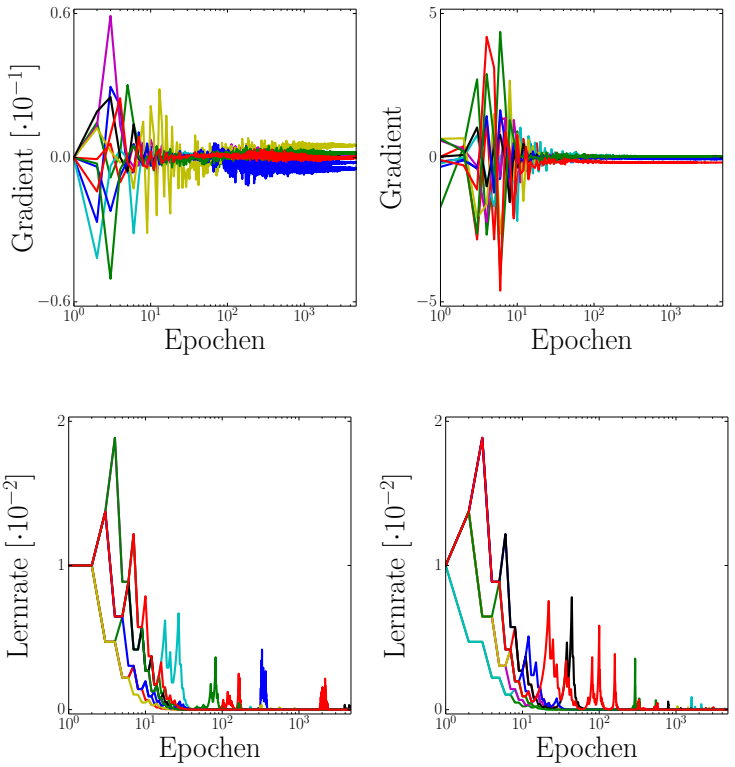
Bei der Betrachtung der Trainingsdauer zeigte der Algorithmus eine klare Tendenz hin zu einer hohen Epochenanzahl. Eine mögliche Erklärung kann in der Minibatch-Problematik von RPROP liegen. Aufgrund von ständigen Vorzeichenwechsel im Gradienten können nur immer kleiner werdende Gewichtsanpassungen vorgenommen werden, wodurch sich der Lernverlauf auf eine lange Trainingsperiode erstreckt. Ein zusätzlicher Grund für die Trainingsdauer kann auch im Weight-Backtracking gefunden werden. Dadurch werden Gewichtsanpassungen nach einem Vorzeichenwechsel rückgängig gemacht.

Betrachtet man die Entwicklung des Gradienten und die der daraus resultierenden Updatewerte Δ (Abbildung 7.17), zeigte sich in allen Experimenten ein charakteristisches Verhalten. Der Gradient wird durch RPROP schon nach wenigen Epochen auf einen engen Bereich um 0 beschränkt. Dieses Verhalten kann man für jede Schicht im Netz erkennen. Die Werte Δ_i konvergieren dabei gegen die vordefinierte untere Grenze von 10^{-8}. Ausschläge nach oben sind nur dann bemerkbar, wenn der Gradient über einem längeren Zeitraum sein Vorzeichen nicht ändert. Dieses Verhalten der Lernraten ist auch in der Entwicklung der Gewichte bemerkbar. Während man in den ersten Epochen eine starke Aktivität der Gewichtsänderungen bemerken kann, bleibt diese jedoch aus, sobald Δ_i zu stark gegen seine untere Grenze konvergiert. In diesem Fall werden nur sehr kleine Gewichtsänderungen durchgeführt und das Gewicht ändert kaum seinen Wert.



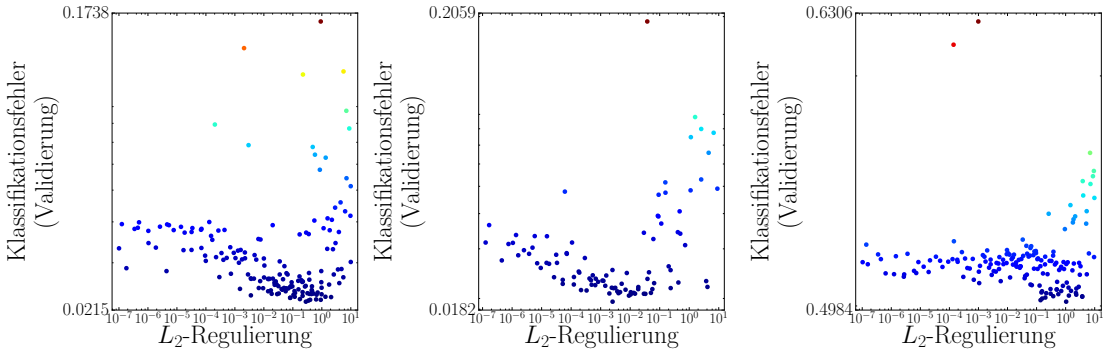
(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.16: Aufgetragen sind die zehn besten Batchgrößen gegen die entsprechenden l_2-Regularisierungskonstante in RPROP. Aus allen drei Bildern geht eine klare antiprobortionale Beziehung der beiden Parameter hervor.



(a) Layer 1 **(b)** Layer 2

Abbildung 7.17: Entwicklung der RPROP-Lernraten und Gradientenkomponenten im Experiment CIFAR-1 für zehn zufällig gewählte Gewichte in jeder Schicht. Gut zu sehen ist, dass die Gewichte sehr schnell auf einen engen Bereich um 0 beschränkt werden. Die Lernraten konvergieren im gesamten Lernverlauf gegen die vordefinierte untere Schranke. Ein Wachstum von Δ ist nur zu sehen, falls sich das Vorzeichen des Gradienten über einem längeren Zeitraum nicht ändert.



(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.18: Auftragung des kreuzvalidierten l_2-Regulierungsparameter λ für RPROP gegen den resultierenden Klassifikationsfehler auf der Validierungsmenge in den Experimenten MNIST-1, MNIST-2 und CIFAR-1.

	MNIST-1	MNIST-2
ρ	$1{,}06 \cdot 10^{-2}$	$3{,}37 \cdot 10^{-7}$
λ	$3{,}63 \cdot 10^{-2}$	$1{,}17 \cdot 10^{-2}$
Δ_0	$3{,}99 \cdot 10^{-4}$	$8{,}00 \cdot 10^{-4}$
η^+	1,42	1,32
η^-	$5{,}68 \cdot 10^{-1}$	$5{,}83 \cdot 10^{-1}$
δ	$5{,}75 \cdot 10^{-2}$	$8{,}65 \cdot 10^{-2}$
bs	750	2000
	CIFAR-1	CNN-MNIST
ρ	$2{,}54 \cdot 10^{-5}$	$4{,}61 \cdot 10^{-3}$
λ	6,13	$3{,}62 \cdot 10^{-2}$
Δ_0	$4{,}53 \cdot 10^{-5}$	$2{,}99 \cdot 10^{-3}$
η^+	1,17	1,10
η^-	$6{,}68 \cdot 10^{-1}$	$6{,}03 \cdot 10^{-1}$
δ	$2{,}10 \cdot 10^{-1}$	$2{,}09 \cdot 10^{-1}$
bs	125	800

Tabelle 7.6: Aufgelistet sind die Hyperparameterkombinationen für RRMSProp, welche im jeweiligen Experiment den geringsten Klassifikationsfehler auf der Validierungsmenge lieferten.

7.6 RRMSProp

Das RRMSProp-Lernverfahren wurde mit allen Netzarchitekturen, bis auf CNN-CIFAR getestet. Tabelle 7.6 beinhaltet die erzielten Ergebnisse.

Der Algorithmus zeigte sich unempfindlich gegenüber der Mean-Square-Konstanten ρ und gegenüber der Wahl der Batchgröße. Für beide Hyperparameter sind die zehn besten Werte über den gesamten Suchraum verteilt. Somit konnte die Hypothese, dass der RPROP ähnliche Algorithmus eher große Batchgrößen benötigt, nicht bestätigt werden.

Unerwarteter Weise reagierte RRMSProp empfindlich auf die initiale Belegung der Werte Δ_i. In nahezu allen Experimenten wurde das beste Ergebnis mit annähernd gleichen Werten erzielt (mit Werten um $4 \cdot 10^{-4}$). Auch die zur l_2-Regularisierung benötigte Konstante λ zeigte sich empfindlich. Vor allem im Experiment CIFAR-1 zeigte sich, dass für die Belegung von λ sehr große Werte benötigt werden (der größte Wert aus den zehn besten λ-Werten liegt bei 9,88). Die Stabilisierungskonstante δ zeigte zwar auch eine Empfindlichkeit, jedoch wurden in allen Experimenten die besten Ergebnisse mit Werten zwischen 10^{-2} und 1 erzielt. Der

Wert könnte somit bei der Verwendung von RRMSProp fest aus diesem Bereich gewählt werden. Die aus RPROP stammenden multiplikativen Konstanten η^+ und η^- zeigten ein zu δ ähnliches Verhalten. In nahezu allen Experimenten zeigte sich eine Belegung von η^- mit 0,6 und η^+ mit 1,3 als die beste Wahl. Abbildung 7.20 liefert einen Überblick über die empfindlichen Hyperparameter.

Somit reagiert RRMSProp auf fünf seiner sieben Hyperparameter empfindlich. Jedoch muss nur die Regularisierungskonstante λ aufwändig eingestellt werden. In nahezu allen Experimenten konnten die restlichen empfindlichen Hyperparameter auf einen festen Bereich innerhalb der Suchintervalle beschränkt werden. Die Suche nach geeigneten Hyperparametern konnte dadurch erheblich vereinfacht werden. Dies wird auch in Abbildung 7.19 deutlich. Eine bemerkenswerte Korrelation zwischen den einzelnen Hyperparameter konnte nicht festgestellt werden.

Eine Betrachtung der Entwicklung der Lernraten und des Gradienten zeigte kein charakteristisches Verhalten.

Die von RRMSProp benötigte Trainingsdauer zeigt eine klare Tendenz hin zu einer sehr hohen Epochenanzahl.

7.7 Vergleich und Schlussfolgerungen

Bei einem Vergleich der getesteten Algorithmen fällt zunächst auf, dass alle erzielten finalen Klassifikationsfehler in einem ähnlichen Bereich lagen. Lediglich RPROP konnte keine mit den restlichen Verfahren vergleichbare Leistung im Experiment CNN-MNIST erreichen. Betrachtet man die erreichten finalen Klassifikationsfehler so erkennt man, dass AdaGrad in nahezu jedem MLP-Experiment den kleinsten Wert geliefert hat. Im konvolutionalen Fall liefert zwar NA-RMSPROP den geringsten Fehler auf der Validierungsmenge, jedoch steigt dieser Wert bei der Auswertung auf der Testmenge. Auf der Trainingsmenge liefert auch hier AdaGrad den geringsten Klassifikationsfehler.

Der Vergleich der Trainingszeiten fällt zu Gunsten von RMSProp und NA-RMSProp aus. Beide Algorithmen benötigten in jedem Experiment eine relativ niedrige Anzahl an Epochen. In keinem getesteten Fall überstieg die Epochenanzahl von RMSProp den Wert 260. Ein vergleichbares Verhalten zeigte NA-RMSProp. Bemerkenswert ist, dass dieser Algorithmus in nahezu allen Experimenten das Training mit einer zweistelligen Epochenanzahl abgeschlossen hat. Obwohl AdaGrad fast in jedem Experiment den geringsten Testfehler zeigte, benötigte der

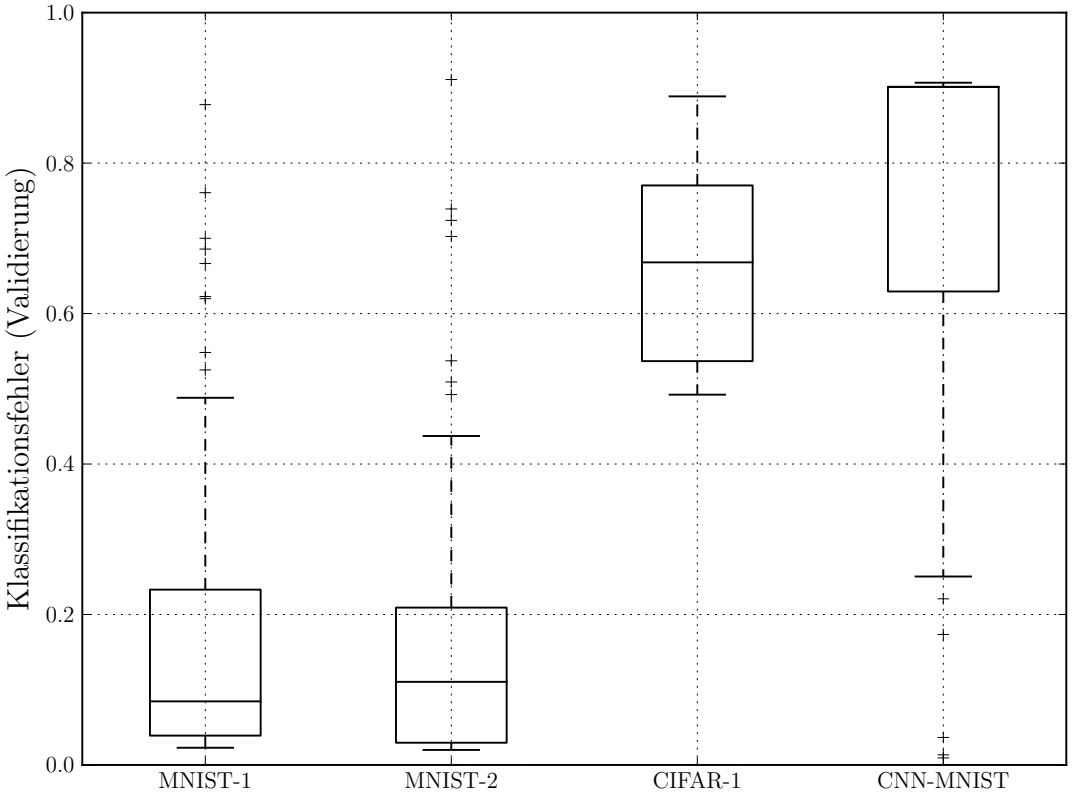
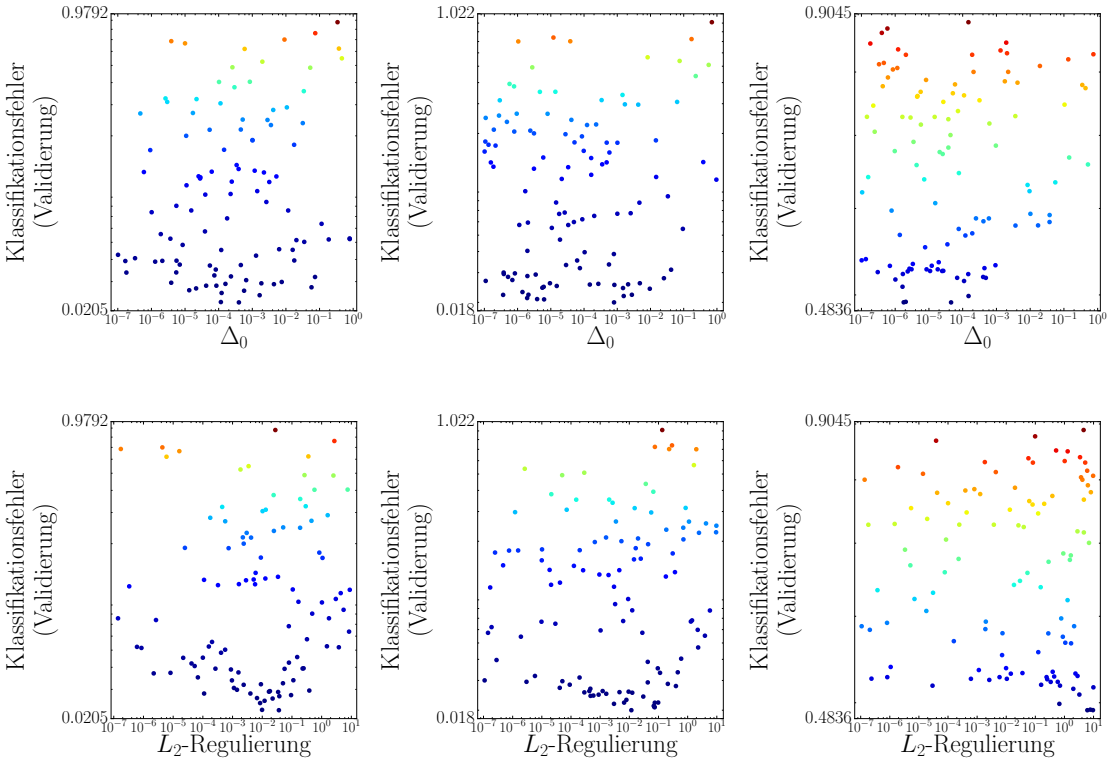


Abbildung 7.19: Verteilung der durch RRMSProp erzielten Klassifikationsfehler auf der Validierungsmenge nach abgeschlossener Kreuzvalidierung. Vor allem in den MLP-Experimenten zeigen sich nahezu gleichgroße Boxen. Für CNN-MNIST zeigte RRMSProp deutliche Schwächen. In diesem Fall kann das beste erzielte Ergebnis als Ausreißer betrachtet werden.



(a) MNIST-1 **(b)** MNIST-2 **(c)** CIFAR-1

Abbildung 7.20: Auftragung der kreuzvalidierten Werte für die initiale Belegung von Δ und des l_2-Regulierungsparameters λ für RRMSProp gegen den resultierenden Klassifikationsfehler auf der Validierungsmenge in den Experimenten MNIST-1, MNIST-2 und CIFAR-1.

	CERR_{cv}	CERR_{test}	Epochen
		MNIST-1	
AdaGrad	**1,92**	**1,82**	460
GD	2,10	2,04	**25**
NA-RMSProp	2,11	2,05	85
RMSProp	2,07	2,04	40
RPROP	2,28	2,28	1520
RRMSProp	2,29	2,21	7170
		MNIST-2	
AdaGrad	**1,6**	**1,64**	60
GD	1,72	1,71	115
NA-RMSProp	1,76	1,70	**55**
RMSProp	1,65	1,61	260
RPROP	1,94	1,94	7800
RRMSProp	2,0	1,93	7985
		CIFAR-1	
AdaGrad	**46,75**	**47,02**	2645
GD	51,51	51,59	1375
NA-RMSProp	50,65	51,55	2745
RMSProp	50,41	50,79	**180**
RPROP	50,17	51,20	1590
RRMSProp	49,22	49,69	7280
		CNN-MNIST	
AdaGrad	0,64	**0,69**	545
GD	0,74	0,74	320
NA-RMSProp	**0,62**	0,72	**60**
RMSProp	0,63	0,79	150
RPROP	1,39	1,44	1745
RRMSProp	0,94	0,80	1940
		CNN-CIFAR	
AdaGrad	19,35	20,14	**25**
GD	21,28	21,22	**25**
NA-RMSProp	**17,56**	**19,17**	35
RMSProp	21,33	21,82	110

Tabelle 7.7: Übersicht über die erzielten geringsten Klassifikationsfehler auf der Validierungsmenge und die zugehörigen Epochen. Die Fehlerwerte bezeichnen den prozentualen Anteil der nicht korrekt klassifizierten Validierungsmuster (CERR_{cv}) und Testmuster (CERR_{test}).

Algorithmus zum Teil sehr viele Epochen bis sich dieser einstellte. RPROP und RRMSProp benötigten in nahezu allen Experimenten eine große Anzahl an Epochen bis sich der geringste Klassifikationsfehler einstellte. Im Vergleich zu allen Experimenten arbeitete RRMSProp immer mit der höchsten Epochenanzahl. Diese erreichte in nahezu allen durchgeführten Experimenten einen Wert von bis zu 7985 Epochen.

Die Suche nach geeigneten Hyperparametern gestaltete sich für die Algorithmen RMSProp, NA-RMSProp, RRMSProp und RPROP einfach. Dabei zeigte sich, dass akzeptable Hyperparameter für RMSProp und NA-RMSProp aus breiten Intervallen gewählt werden können. RRMSProp und RPROP besitzen zwar sehr viele Hyperparameter, jedoch konnten die meisten Hyperparameter mit festen Werten belegt werden. Für diese Algorithmen kann man eine Hyperparameteroptimierung sehr schnell durchführen. Die multiplikativen Konstanten η^+ und η^- in RPROP konnten bei der Verwendung mit Minibatches mit nahezu identischen Werten versehen werden wie im Batchlearning. Somit kann Hypothese 2 nicht bestätigt werden. Ein ähnliches Bild zeigt sich auch für RRMSProp. Auch hier können diese Hyperparameter als konstant angesehen werden.

Ein anderes Verhalten zeigte sich bei AdaGrad. Dieses Lernverfahren reagierte empfindlich auf Hyperparameter. Die Suche für eine geeignete Belegung gestaltete sich aufwändig, da die entsprechenden Intervalle sehr eng ausfallen.

Betrachtet man die Entwicklung der Lernraten Δ in RPROP, so stellt man fest, dass diese Werte sehr schnell gegen die vordefinierte untere Grenze konvergieren. Ein Lernfortschritt wurde in diesen Fällen nur sehr langsam erreicht. Die entsprechende Hypothese 3 muss daher als bestätigt gelten.

Lässt man RPROP und RRMSProp außer acht und vergleicht die Testfehler der restlichen Algorithmen, so fällt auf, dass in nahezu jedem Fall ein Algorithmus mit einer adaptiven individuellen Lernrate einen geringeren Klassifikationsfehler liefert, als GD. Obwohl man hier sicherlich keinen allgemeingültigen Vergleich zwischen konstanten Lernraten und individuellen Lernraten ziehen kann, so muss man zumindest anmerken, dass die Popularität von adaptiven individuellen Lernraten nachvollziehbar ist. Trotz der Vielzahl an Hyperparametern scheinen Lernverfahren mit solchen Lernraten gegenüber konstanten Lernraten einen Vorteil in Bezug auf Optimierungsfähigkeit und Trainingsdauer zu besitzen. Hypothese 5 muss somit zum Teil bestätigt werden. Zwar lieferten die Agorithmen AdaGrad, RMSProp und NA-RMSProp bessere oder gleichwertige Ergebnisse im Vergleich zu GD, jedoch muss festgestellt werden, dass RRMSProp in fast allen Experimenten einen

schlechteren Testfehler lieferte.

Trotz der Ähnlichkeit zwischen AdaGrad und RMSProp zeigte letzteres Lernverfahren keine Empfindlichkeit gegenüber dem Mean-Square-Parameter ρ. Ähnlich unempfindlich zeigte sich auch NA-RMSProp gegenüber ρ. Beide Verfahren können somit als unempfindlich gegenüber der Approximation des Zeitfensters bezeichnet werden. Lediglich RRMSProp zeigte eine schwache Abhängigkeit gegenüber ρ. Entsprechend kann Hypothese 7 nur für RRMSProp bestätigt werden.

AdaGrad, RMSProp und NA-RMSProp haben in den Experimenten vergleichbare Klassifikationsfehler geliefert. Dabei zeigten sich die beiden RMSProp-Varianten als unempfindlich gegenüber ρ. Vergleichbare Ergebnisse können somit mit verschiedenen Belegungen von ρ erzielt werden. Die Approximation der AdaGrad-Summe spielt daher nur eine untergeordnete Rolle. Hypothese 6 kann somit als bedingt erfüllt angesehen werden. Die Algorithmen liefern vergleichbare Ergebnisse, jedoch werden diese nur bedingt durch die Approximation der AdaGrad-Summe hervorgerufen.

Bei einem Vergleich zwischen RMSProp und NA-RMSProp zeigt sich zwar ein gewisser Vorteil von NA-RMSProp in Bezug auf die Trainingszeit, jedoch lieferten beide Verfahren nahezu gleiche Testfehler. Bei der Verwendung von NA-RMSProp muss man zusätzlich beachten, dass dieses Verfahren mehr Parameter für die Berechnung der Lernrate zwischenspeichern muss (η_i, den Mean-Square und das Nesterov-Momentum). Für eine Anwendung in großen Netzen mit vielen Gewichten kann es daher eher zu Speicherproblemen kommen, als bei der Verwendung von RMSProp. Die Annahme, dass die Verwendung eines Nesterov-Momentums zu einer schnelleren Konvergenz führt (Hypothese 8) kann somit bestätigt werden.

Betrachtet man MLP- und Konvolutionsexperimente voneinander getrennt, so stellt man fest, dass alle Algorithmen einen vergleichbaren Testfehler in den MLP-Experimenten lieferten. Der durch RPROP und RRMSProp erzielte Testfehler in den Experimenten mit Konvolutionsnetzen war höher als die entsprechenden Fehler der restlichen Algorithmen. Ferner muss bemerkt werden, dass diese Netze sehr rechenintensiv arbeiten und somit die Abarbeitung einer Epoche eine lange Zeit benötigte. Da RPROP und RRMSProp eine große Anzahl an Epochen benötigen, ist die Verwendung dieser Algorithmen für das Training von Konvolutionsnetzen und MLPs mit vielen Gewichten nicht zu empfehlen.

Ein Vergleich der Ergebnisse aus den MLP-Experimenten mit anderen Arbeiten ist nur schwer möglich. Zwar wurden die selben Architekturen in den Arbeiten von Schaul, Zhang u. a. (2012) und Zeiler (2012) verwendet, jedoch wurden die Netze

Goodfellow u. a. (2013)	0,45
G. E. Hinton u. a. (2012)	0,79
CNN-MNIST	0,69

Tabelle 7.8: Vergleich des besten erzielten Ergebnisses auf der Testmenge im Experiment CNN-MNIST mit aktuellen Arbeiten. Der entsprechende Wert wurde mit AdaGrad erzielt.

Goodfellow u. a. (2013)	11,68
G. E. Hinton u. a. (2012)	15,6
CNN-CIFAR	19,17

Tabelle 7.9: Vergleich des besten erzielten Ergebnisses auf der Testmenge im Experiment CNN-CIFAR mit aktuellen Arbeiten. Der entsprechende Wert wurde mit NA-RMSProp erzielt.

nur sechs Epochen lang online trainiert. Dennoch kann an dieser Stelle festgestellt werden, dass die Ergebnisse der hier durchgeführten Evaluierung besser ausfallen als alle Ergebnisse dieser beiden Arbeiten.

Die Ergebnisse der durchgeführten Konvolutionsexperimente kann man vor allem mit der Arbeit von Goodfellow u. a. (2013) vergleichen. Auch G. E. Hinton u. a. (2012) verwendet ähnliche Netzarchitekturen. Eine tabellarische Gegenüberstellung dieser Ergebnisse mit den im Rahmen dieser Arbeit erzielten findet man in Tabelle 7.8 und in Tabelle 7.9.

7.8 Empfehlungen

Aufgrund der erzielten Ergebnisse und der gezogenen Schlussfolgerungen werden nun einige Empfehlungen zur Verwendung der getesteten Algorithmen gegeben.

Betrachtet man nur die finalen Testfehler stellt man fest, dass sich alle Algorithmen für das Training von MLPs eignen. Dabei erzielten RPROP und RRMSProp die schlechtesten Fehlerwerte. Diese Verfahren benötigten auch die meisten Epochen, bis das Lernen abgeschlossen werden konnte. Aufgrund dessen, sind RPROP und RRMSProp für das Training von MLPs nicht zu empfehlen. Auch bei der Verwendung mit Konvolutionsnetzen konnten die durch RPROP und RRMSProp erzielten Fehler auf der Testmenge und die benötigte Epochenanzahl nicht überzeugen. Somit ist auch ein Training von Konvolutionsnetzen durch diese Al-

gorithmen nicht empfehlenswert.

Der klassische Gradientenabstieg konnte in allen Experimenten ähnliche Testfehler liefern wie AdaGrad, RMSProp und NA-RMSProp. Jedoch hat sich gezeigt, dass die konstante, globale Lernrate im konvolutionalen Fall nur schwer mit einem geeigneten Wert zu belegen ist. Daher kann dieses Verfahren nur für das Trainieren von MLPs empfohlen werden.

AdaGrad, RMSProp und NA-RMSProp sind, sowohl für MLPs, als auch für Konvolutionsnetze zu empfehlen. Alle drei Verfahren liefern vergleichbare Testfehler. Jedoch bieten letztere Algoritmen einen Vorteil bei der Suche nach Hyperparametern und bei der Trainingsdauer gegenüber AdaGrad.

Bei den getesteten Lernverfahren eines zu favorisieren, fällt schwer. Aufgrund der schlechten Optimierungsfähigkeit und der langen Trainingsdauer verbietet sich die Verwendung von RPROP und RRMSProp. NA-RMSProp bietet eine schnelle Konvergenz und eine einfache Hyperparameteroptimierung, jedoch auch einen höheren Speicheraufwand als RMSProp und AdaGrad. Der GD-Algorithmus ist, von den getesteten am einfachsten zu implementieren. Der gelieferte Testfehler wurde jedoch von AdaGrad und RMSProp unterboten.

Ein Vorteil von RMSProp und NA-RMSProp gegenüber AdaGrad ist die entsprechende Reaktionsfähigkeit auf Schwankungen im Gradienten. Temporale Ausschläge im Gradienten führen dazu, dass AdaGrad seine Lernrate stark verkleinert und diese nicht wieder erhöht. Die RMSProp-Varianten können solche Schwankungen kompensieren. Tritt ein großer Ausschlag in der Entwicklung des Gradienten auf, so kann die RMSProp-Lernrate zunächst verkleinert werden und darauf reagieren. Anschließend kann die Lernrate wieder anwachsen. Dies hat zur Folge, dass zeitlich benachbarte Gewichtsanpassungen eine ähnliche Größe besitzen. (Vergleiche hierzu Abbildung 7.3 Abbildung 7.9, Abbildung 7.12.)

Als Favoriten unter den getesteten Algorithmen haben sich RMSProp und AdaGrad herauskristallisiert. Beide Verfahren besitzen Vorteile die für den jeweiligen Algorithmus sprechen. So hat AdaGrad in nahezu jedem Experiment den geringsten Testfehler geliefert, seine Hyperparameter waren jedoch nur schwer zu optimieren. Eine Hyperparametersuche kann somit vor allem bei rechenintensiven Netzen eine sehr lange Zeit in Anspruch nehmen. RMSProp lieferte ähnliche Testfehler wie AdaGrad. Die Suche nach geeigneten Hyperparametern gestaltete sich für dieses Verfahren sehr einfach. Dadurch ist RMSProp vor allem für das Training von rechenintensiven Netzen, wie beispielsweise großen Konvolutionsnetzen oder auch tiefen neuronalen Netzen, sehr empfehlenswert.

Für die Suche nach Hyperparametern empfiehlt sich das folgende Vorgehen:

- **RPROP**: Die multiplikativen Werte η^- und η^+ lassen sich aus dem Batchlearning übernehmen und fest mit den Werten 0,5 und 1,2 belegen. Lediglich die Suche nach geeigneten Werte für die l_2-Regularisierungskonstante λ muss aufwändiger durchgeführt werden. Dabei hat sich gezeigt, dass Werte aus einem breiten Suchraum verwendet werden können. Der Bereich $10^{-3} < \lambda < 10$ scheint gute Werte zu beinhalten.

- **RRMSProp**: Die initiale Belegung Δ_0 lässt sich fest wählen und mit einem Wert um 10^{-4} belegen. Auch hier können die Parameter η^- und η^+ fest gewählt werden. Es empfehlen sich dabei die Werte 0,6 bzw. 1,3. Eine Hyperparametersuche muss nur für ρ, λ und δ durchgeführt werden. Dabei kann nur der Suchbereich für δ eingeschränkt werden durch $10^{-2} < \delta < 1$.

- **GD**: Die Hyperparametersuche für die Lernrate η kann beschränkt werden auf den Bereich $10^{-3} < \eta < 1$.

- **AdaGrad**: Die Suche nach geeigneten Belegungen für η kann beschränkt werden auf $10^{-4} < \eta < 1$. Für λ ist eine Suche innerhalb des Bereichs $10^{-6} < \lambda < 5 \cdot 10^{-5}$ empfehlenswert. Die restlichen Hyperparameter sind als unempfindlich identifiziert worden.

- **RMSProp**: In diesem Algorithmus wurde festgestellt, dass η und δ empfindlich auf ihre Belegung reagieren. Eine Beschränkung des Suchbereichs ist zwar nicht möglich, jedoch gestaltet sich die Hyperparametersuche für diese Parameter einfach. Sie können Werte aus einem breiten Bereich innerhalb des Suchintervalls annehmen. Die Algorithmus reagiert unempfindlich auf ρ.

- **NA-RMSProp**: Hier kann die Suche für η beschränkt werden auf $10^{-4} < \eta < 1$, für μ auf $10^{-7} < \mu < 10^{-4}$ und für γ auf $10^{-7} < \gamma < 10^{-4}$. Nur für den empfindlichen Parameter δ kann der Suchbereich nicht beschränkt werden. Der Mean-Square-Parameter ρ reagiert unempfindlich.

8 Ausblick

In dieser Arbeit wurden experimentelle Ergebnisse durch die Verwendung der einzelnen Algorithmen in Kombination mit MLPs und Konvolutionsnetzen erzielt . Diese wurden auf den Datensätzen MNIST und CIFAR-10 trainiert. Die Experimente wurden dabei nur auf Klassifikationsaufgaben beschränkt.

Die durch die erzielte Ergebnisse getroffenen Schlussfolgerungen bedürfen einer Bestätigung durch Experimente mit weiteren Datensätzen. Von besonderem Interesse ist dabei eine Evaluation auf komplexeren Datensätzen, welche sowohl größere als auch mehr Muster beinhalten. Hierbei kann eine Klassifikation mit mehr als zehn Klassen erfolgen. Ein solcher Datensatz ist beispielsweise ImageNet (Russakovsky u. a., 2014).

Eine weitere mögliche Reihe von Experimenten könnte darin bestehen die vorgestellten Algorithmen auf andere Aufgabenstellungen als Klassifikation anzuwenden. Eine solche Aufgabenstellung könnte das Entrauschen von Bildern darstellen. In Bezug zur Deep Learning Problematik ist dabei die Verwendung Denoising Autoencoder als Netzarchitektur von besonderem Interesse. Hier können die Algorithmen hinsichtlich ihrer Eignung im Bezug auf das Vortraining untersucht werden.

Zu Vergleichszwecken mit konstanten globalen Lernraten wurde GD verwendet. Dieser lässt sich auf verschiedene Arten modifizieren. Er kann durch die Verwendung eines Momentums beschleunigt werden. Die globale Lernrate kann durch ein entsprechendes Schedule anpassungsfähig gemacht werden. Für einen Vergleich zwischen einem modifizierten GD und den hier getesteten Algorithmen kann eine Evaluation der verschiedenen Lernraten-Schedules von Vorteil sein.

Ein solches Schedule kann auch in Kombination mit dem globalen Hyperparameter η in RMSProp interessant sein. Wie weiter oben erwähnt kann RMSProp kein verfeinertes Lernen mit kleinen Gradientenwerten durchführen. Eine zeitliche Anpassung des Mean-Square-Parameters ρ könnte dieser Tatsache dienlich sein. Auch eine solche Anpassung kann durch die entsprechende Implementierung eines Schedules erreicht werden.

Aufgrund der Tatsache, dass L-BFGS sehr gute Trainingsergebnisse auf neuronalen Netzen liefert (Ngiam u. a., 2011), ist auch ein Vergleich zwischen den evaluierten Algorithmen und Optimierungsverfahren zweiter Ordnung vorstellbar.

Anhang

1 Hyperparameter-Intervalle

In den folgenden Tabellen werden diejenigen Intervalle aufgelistet, aus denen durch die Kreuzvalidierung die zehn besten Belegungen für den jeweiligen Algorithmus im jeweiligen Experiment gezogen wurden.

	MNIST-1	MNIST-2
η	$[4{,}30 \cdot 10^{-3}, 1{,}95 \cdot 10^{-1}]$	$[1{,}22 \cdot 10^{-2}, 1{,}58 \cdot 10^{-1}]$
λ	$[8{,}09 \cdot 10^{-6}, 6{,}00 \cdot 10^{-5}]$	$[1{,}13 \cdot 10^{-7}, 1{,}70 \cdot 10^{-5}]$
δ	$[1{,}05 \cdot 10^{-7}, 1{,}50 \cdot 10^{-1}]$	$[5{,}15 \cdot 10^{-7}, 3{,}20 \cdot 10^{-2}]$
bs	[120, 1500]	[4, 80]
	CIFAR-1	CNN-MNIST
η	$[3{,}68 \cdot 10^{-4}, 8{,}87 \cdot 10^{-3}]$	$[3{,}71 \cdot 10^{-3}, 5{,}90 \cdot 10^{-2}]$
λ	$[2{,}45 \cdot 10^{-5}, 1{,}92 \cdot 10^{-4}]$	$[1{,}02 \cdot 10^{-7}, 2{,}07 \cdot 10^{-5}]$
δ	$[1{,}10 \cdot 10^{-7}, 5{,}46 \cdot 10^{-1}]$	$[4{,}55 \cdot 10^{-5}, 2{,}06 \cdot 10^{-1}]$
bs	[50, 500]	[12, 500]

Tabelle 1: Intervalle in denen die besten zehn Hyperparameter von AdaGrad liegen.

	MNIST-1	MNIST-2
η	$[2{,}47 \cdot 10^{-2}, 4{,}47 \cdot 10^{-1}]$	$[3{,}54 \cdot 10^{-2}, 4{,}43 \cdot 10^{-1}]$
bs	[2, 40]	[4, 60]
	CIFAR-1	CNN-MNIST
η	$[7{,}71 \cdot 10^{-4}, 9{,}59 \cdot 10^{-3}]$	$[5{,}52 \cdot 10^{-3}, 5{,}34 \cdot 10^{-2}]$
bs	[125, 5000]	[50, 500]

Tabelle 2: Intervalle in denen die besten zehn Hyperparameter von GD liegen.

	MNIST-1	MNIST-2
ρ	$[3{,}88 \cdot 10^{-7}, 6{,}97 \cdot 10^{-1}]$	$[1{,}16 \cdot 10^{-6}, 1{,}78 \cdot 10^{-1}]$
η	$[1{,}17 \cdot 10^{-3}, 2{,}90 \cdot 10^{-1}]$	$[2{,}55 \cdot 10^{-4}, 3{,}13 \cdot 10^{-3}]$
δ	$[3{,}93 \cdot 10^{-2}, 1{,}26 \cdot 10^{-1}]$	$[5{,}86 \cdot 10^{-4}, 2{,}21 \cdot 10^{-2}]$
bs	[8, 50]	[2, 40]
	CIFAR-1	CNN-MNIST
ρ	$[6{,}70 \cdot 10^{-2}, 9{,}95 \cdot 10^{-1}]$	$[3{,}69 \cdot 10^{-6}, 9{,}18 \cdot 10^{-1}]$
η	$[7{,}69 \cdot 10^{-6}, 1{,}08 \cdot 10^{-4}]$	$[2{,}44 \cdot 10^{-4}, 1{,}49 \cdot 10^{-2}]$
δ	$[1{,}06 \cdot 10^{-4}, 7{,}43 \cdot 10^{-3}]$	$[1{,}64 \cdot 10^{-7}, 6{,}73 \cdot 10^{-1}]$
bs	[40, 500]	[30, 1000]

Tabelle 3: Intervalle in denen die besten zehn Hyperparameter von RMSProp liegen.

	MNIST-1	MNIST-2
ρ	$[5{,}99 \cdot 10^{-6}, 1{,}55 \cdot 10^{-2}]$	$[1{,}57 \cdot 10^{-2}, 7{,}41 \cdot 10^{-1}]$
η	$[3{,}46 \cdot 10^{-3}, 1{,}79 \cdot 10^{-1}]$	$[5{,}17 \cdot 10^{-4}, 1{,}63 \cdot 10^{-2}]$
γ	$[1{,}34 \cdot 10^{-7}, 2{,}08 \cdot 10^{-6}]$	$[1{,}05 \cdot 10^{-7}, 1{,}49 \cdot 10^{-5}]$
μ	$[1{,}66 \cdot 10^{-7}, 5{,}68 \cdot 10^{-4}]$	$[2{,}22 \cdot 10^{-7}, 9{,}27 \cdot 10^{-4}]$
δ	$[8{,}99 \cdot 10^{-3}, 8{,}67 \cdot 10^{-1}]$	$[3{,}60 \cdot 10^{-5}, 3{,}04 \cdot 10^{-1}]$
bs	[4, 40]	[2, 400]
	CIFAR-1	CNN-MNIST
ρ	$[1{,}23 \cdot 10^{-7}, 5{,}02 \cdot 10^{-5}]$	$[3{,}27 \cdot 10^{-7}, 8{,}61 \cdot 10^{-1}]$
η	$[1{,}38 \cdot 10^{-6}, 1{,}47 \cdot 10^{-4}]$	$[3{,}62 \cdot 10^{-3}, 7{,}39 \cdot 10^{-2}]$
γ	$[1{,}04 \cdot 10^{-7}, 8{,}74 \cdot 10^{-6}]$	$[4{,}94 \cdot 10^{-5}, 9{,}48 \cdot 10^{-4}]$
μ	$[1{,}84 \cdot 10^{-7}, 1{,}43 \cdot 10^{-4}]$	$[3{,}55 \cdot 10^{-7}, 2{,}45 \cdot 10^{-1}]$
δ	$[6{,}43 \cdot 10^{-4}, 2{,}58 \cdot 10^{-2}]$	$[2{,}28 \cdot 10^{-7}, 5{,}14 \cdot 10^{-3}]$
bs	[8, 1250]	[20, 200]

Tabelle 4: Intervalle in denen die besten zehn Hyperparameter von NA-RMSProp liegen.

	MNIST-1	MNIST-2
λ	$[8{,}08 \cdot 10^{-2}, 2{,}09]$	$[1{,}97 \cdot 10^{-3}, 5{,}04 \cdot 10^{-2}]$
η^+	[1,02, 1,11]	[1,00, 1,32]
η^-	$[4{,}19 \cdot 10^{-1}, 9{,}40 \cdot 10^{-1}]$	$[4{,}01 \cdot 10^{-1}, 5{,}98 \cdot 10^{-1}]$
bs	[4, 250]	[300, 3000]
	CIFAR-1	CNN-MNIST
λ	$[1{,}37 \cdot 10^{-1}, 3{,}73]$	$[1{,}84 \cdot 10^{-7}, 3{,}13 \cdot 10^{-1}]$
η^+	[1,18, 1,45]	[1,12, 1,50]
η^-	$[4{,}01 \cdot 10^{-1}, 5{,}30 \cdot 10^{-1}]$	$[4{,}00 \cdot 10^{-1}, 6{,}39 \cdot 10^{-1}]$
bs	[200, 5000]	[30, 800]

Tabelle 5: Intervalle in denen die besten zehn Hyperparameter von RPROP liegen.

	MNIST-1	MNIST-2
ρ	$[1{,}27 \cdot 10^{-3}, 5{,}06 \cdot 10^{-1}]$	$[3{,}37 \cdot 10^{-7}, 5{,}54 \cdot 10^{-1}]$
λ	$[1{,}10 \cdot 10^{-4}, 9{,}34 \cdot 10^{-2}]$	$[4{,}69 \cdot 10^{-4}, 9{,}77 \cdot 10^{-2}]$
Δ_0	$[1{,}33 \cdot 10^{-5}, 5{,}77 \cdot 10^{-2}]$	$[3{,}68 \cdot 10^{-7}, 9{,}54 \cdot 10^{-3}]$
η^+	[1,16, 1,59]	[1,04, 1,44]
η^-	$[4{,}04 \cdot 10^{-1}, 7{,}74 \cdot 10^{-1}]$	$[4{,}14 \cdot 10^{-1}, 8{,}34 \cdot 10^{-1}]$
δ	$[1{,}70 \cdot 10^{-2}, 8{,}42 \cdot 10^{-1}]$	$[2{,}15 \cdot 10^{-2}, 7{,}86 \cdot 10^{-1}]$
bs	[300, 3000]	[150, 2000]
	CIFAR-1	CNN-MNIST
ρ	$[5{,}01 \cdot 10^{-7}, 3{,}31 \cdot 10^{-1}]$	$[1{,}29 \cdot 10^{-7}, 9{,}45 \cdot 10^{-1}]$
λ	$[2{,}94 \cdot 10^{-5}, 9{,}88]$	$[1{,}05 \cdot 10^{-7}, 4{,}31]$
Δ_0	$[1{,}01 \cdot 10^{-6}, 4{,}86 \cdot 10^{-4}]$	$[3{,}56 \cdot 10^{-5}, 2{,}99 \cdot 10^{-3}]$
η^+	[1,00, 1,24]	[1,00, 1,34]
η^-	$[4{,}27 \cdot 10^{-1}, 8{,}76 \cdot 10^{-1}]$	$[4{,}03 \cdot 10^{-1}, 8{,}40 \cdot 10^{-1}]$
δ	$[5{,}31 \cdot 10^{-3}, 5{,}61 \cdot 10^{-1}]$	$[2{,}09 \cdot 10^{-2}, 9{,}62 \cdot 10^{-1}]$
bs	[40, 5000]	[50.0, 800.0]

Tabelle 6: Intervalle in denen die besten zehn Hyperparameter von RRMSProp liegenl..

2 Kreuzvalidierte Hyperparameter

Die folgenden Bilder zeigen alle kreuzvalidierten Hyperparameter aufgetragen gegen den resultierenden Klassifikationsfehler auf der Validierungsmenge. Die Farbkodierung entspricht diesem Validierungsfehler. In jedem Bild sind beide Achsen logarithmisch skaliert.

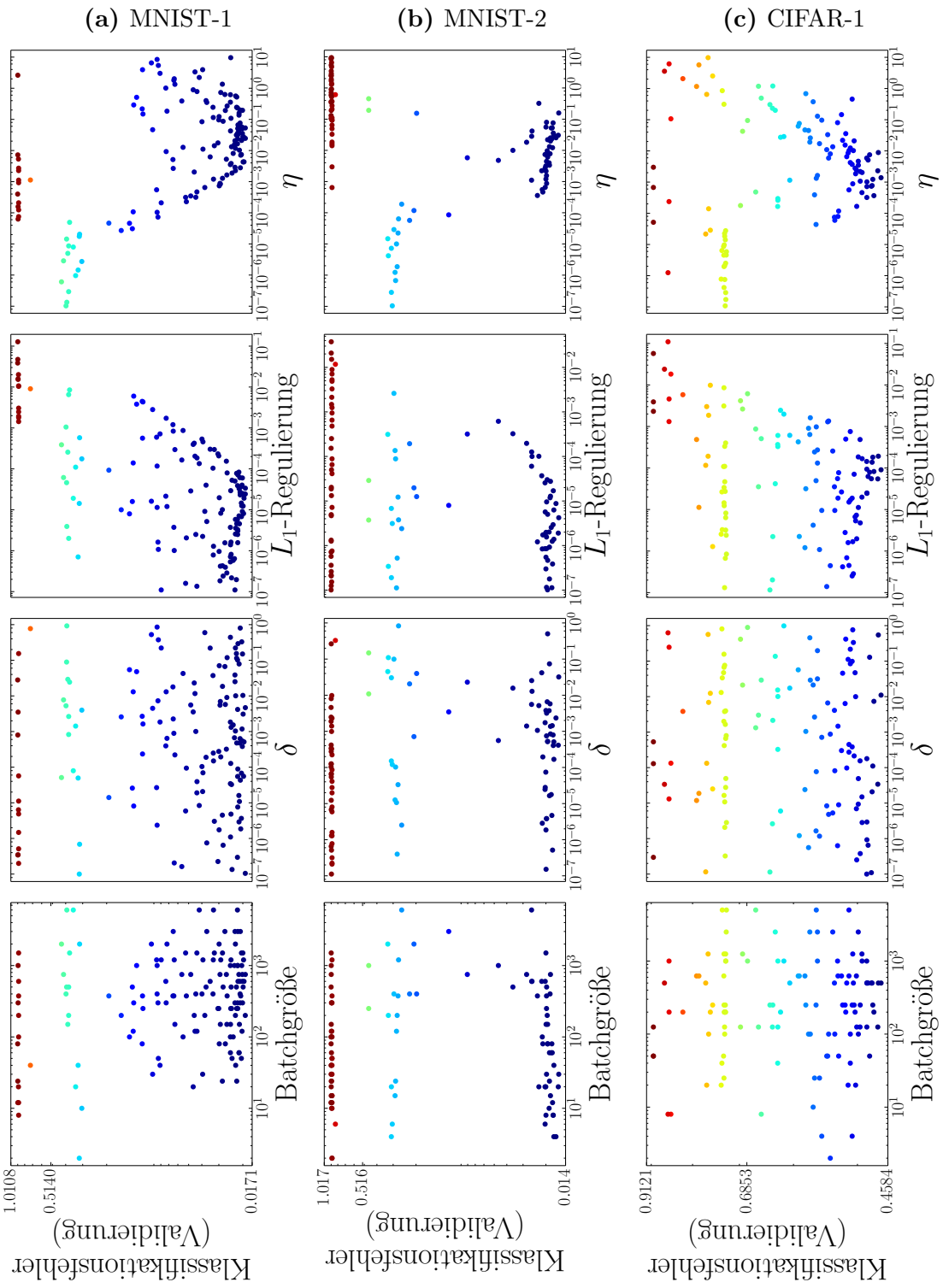


Abbildung 1: Kreuzvalidierte Hyperparameterkombinationen von AdaGrad.

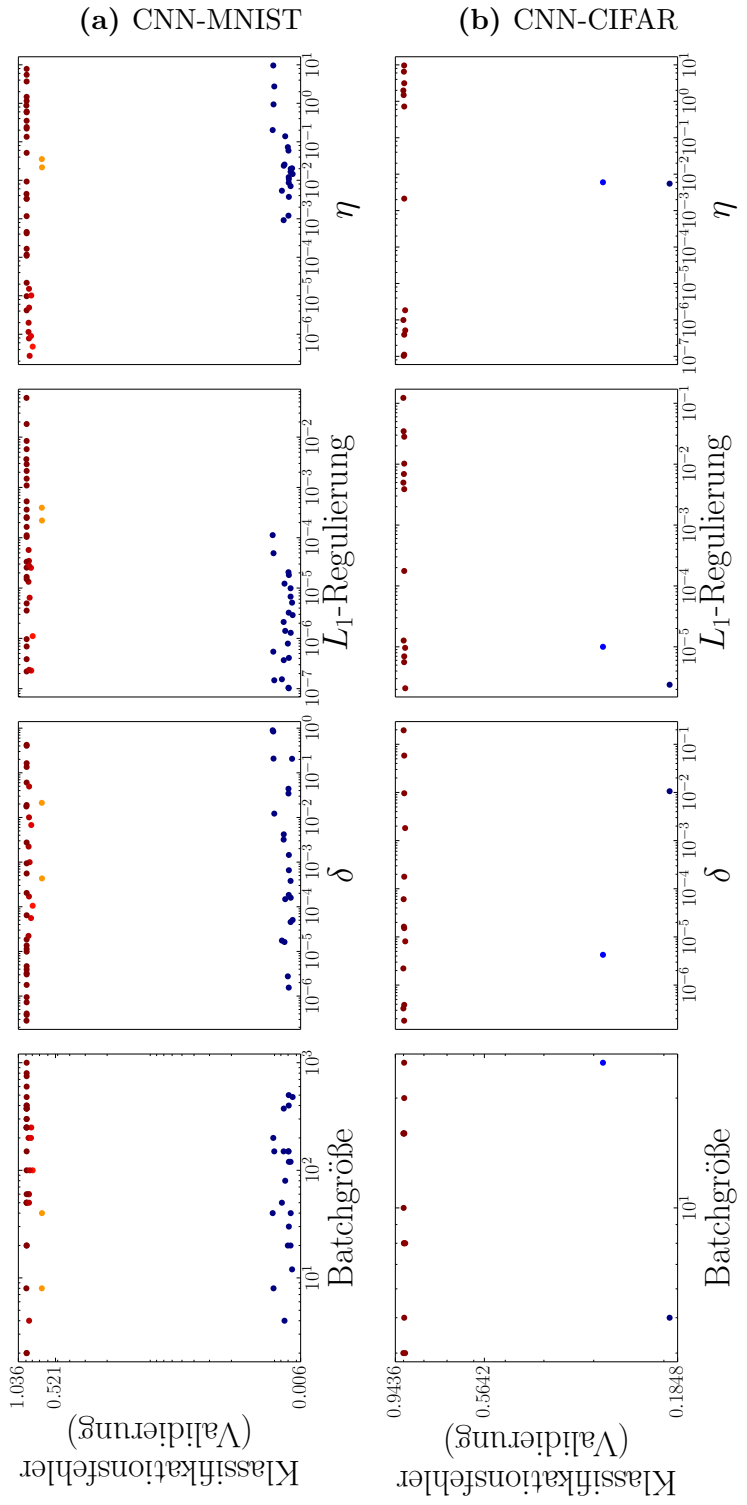


Abbildung 2: Kreuzvalidierte Hyperparameterkombinationen von AdaGrad.

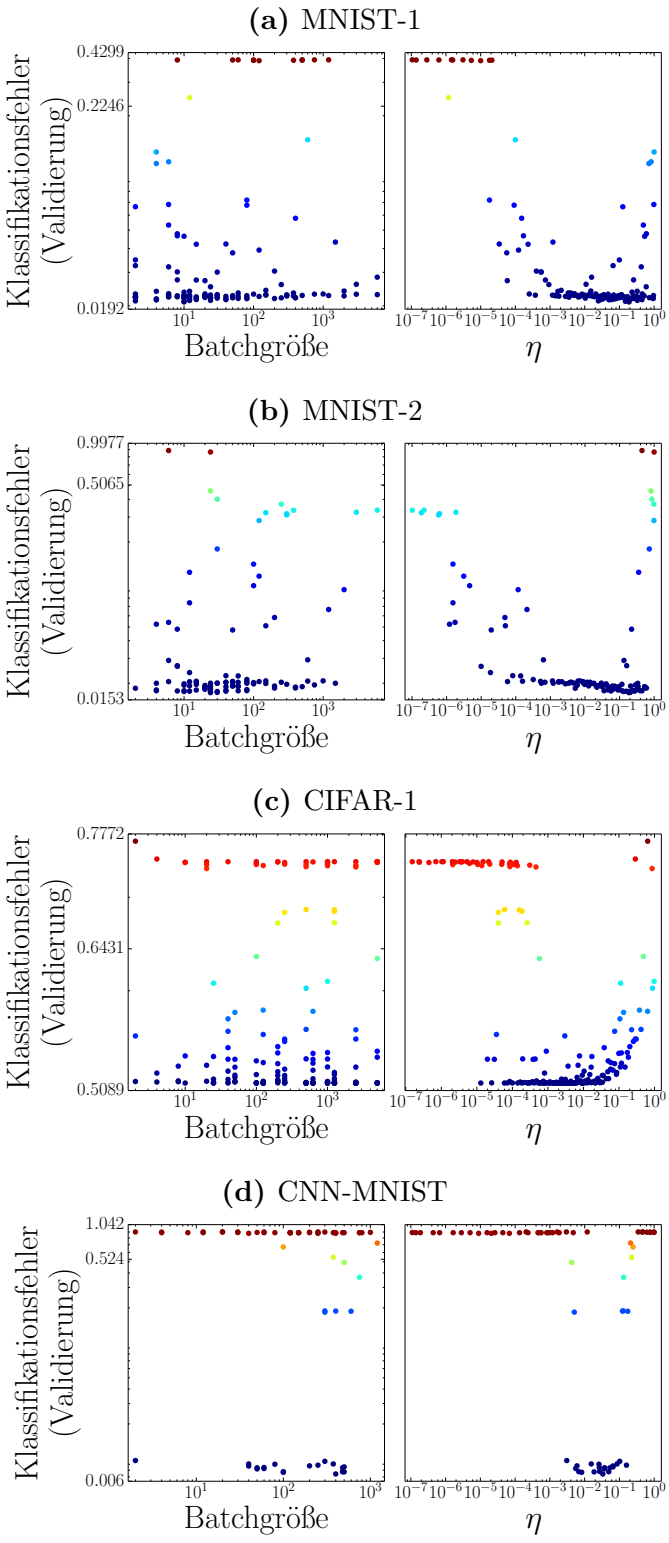


Abbildung 3: Kreuzvalidierte Hyperparameterkombinationen von GD.

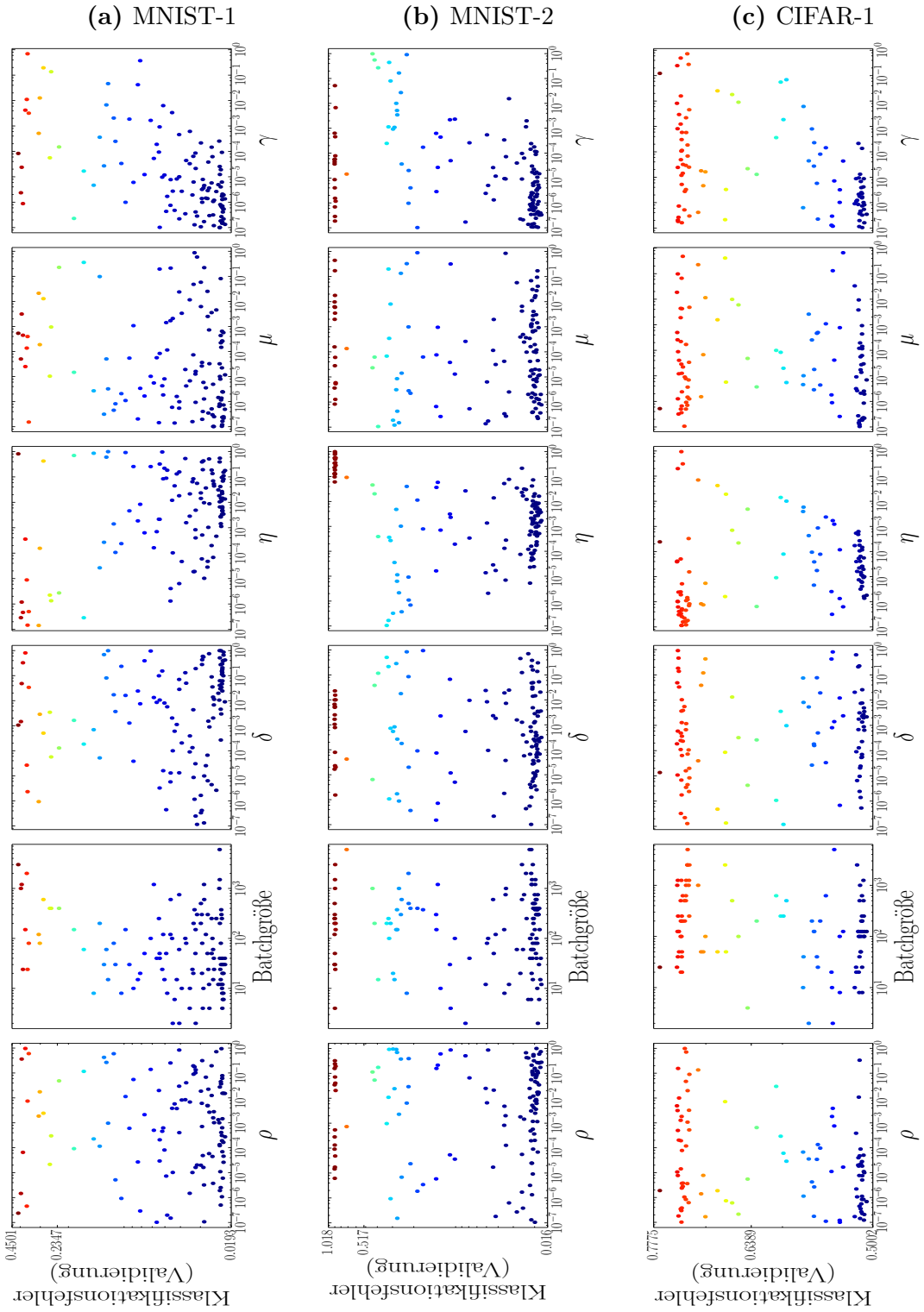


Abbildung 4: Kreuzvalidierte Hyperparameterkombinationen von NA-RMSProp.

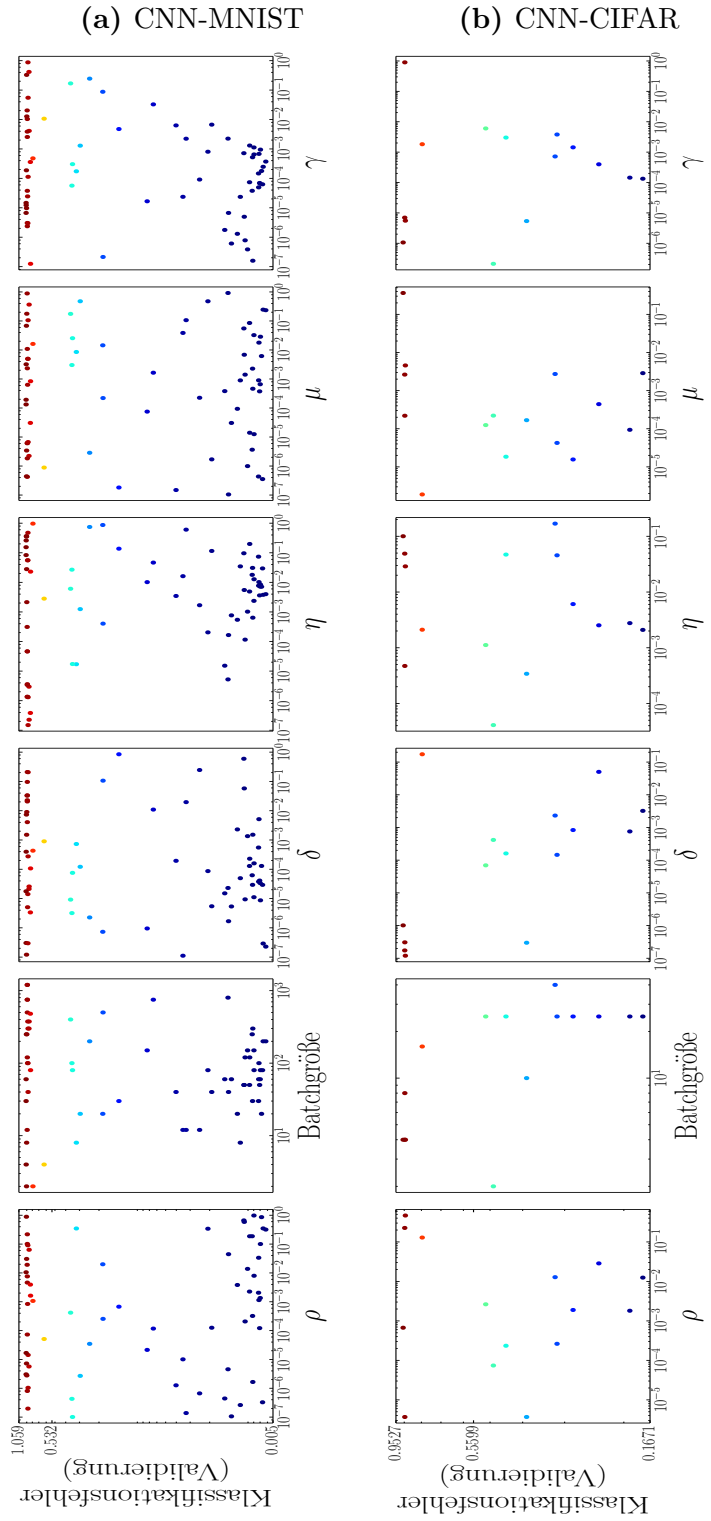


Abbildung 5: Kreuzvalidierte Hyperparameterkombinationen von NA-RMSProp.

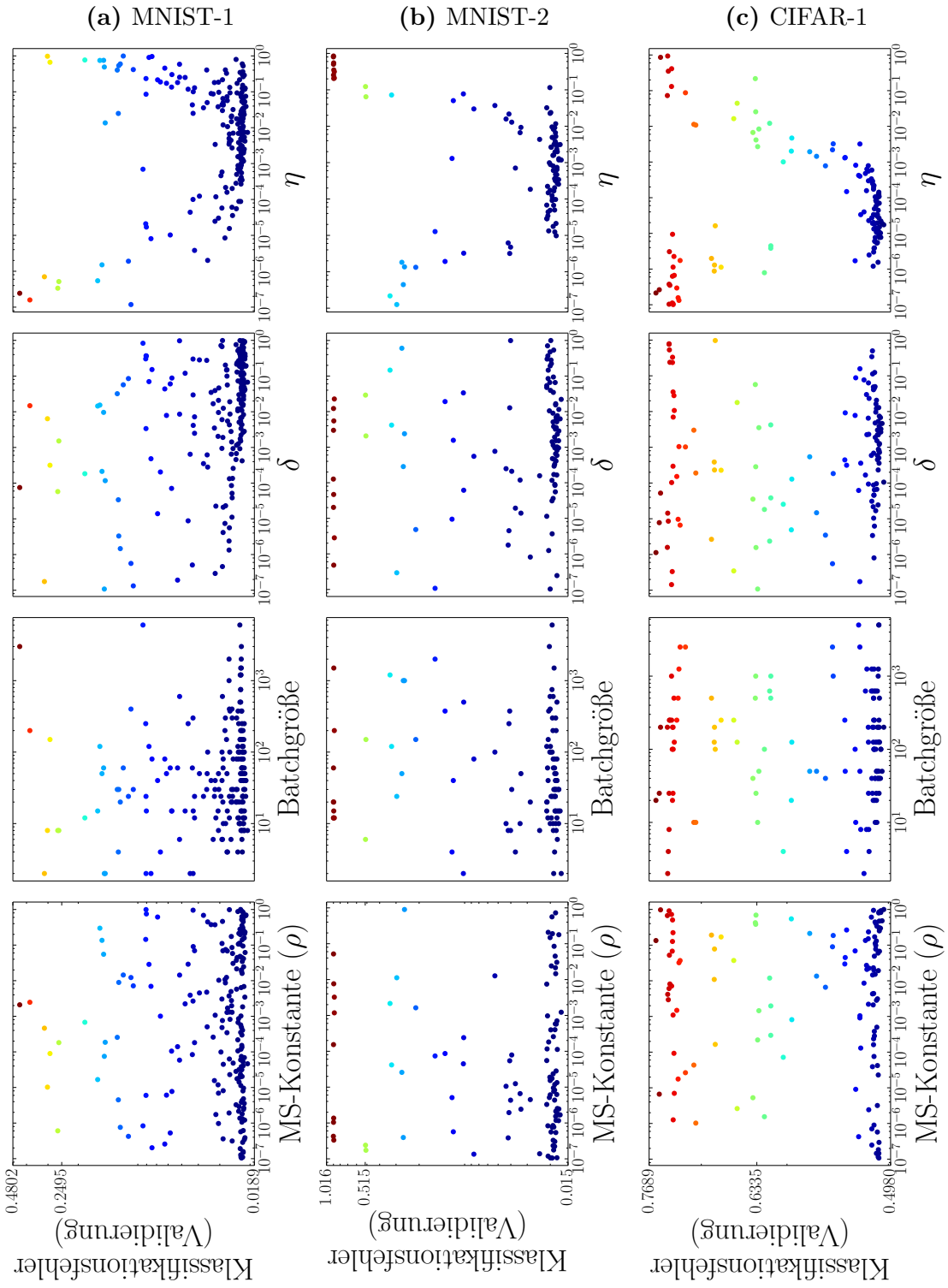


Abbildung 6: Kreuzvalidierte Hyperparameterkombinationen von RMSProp.

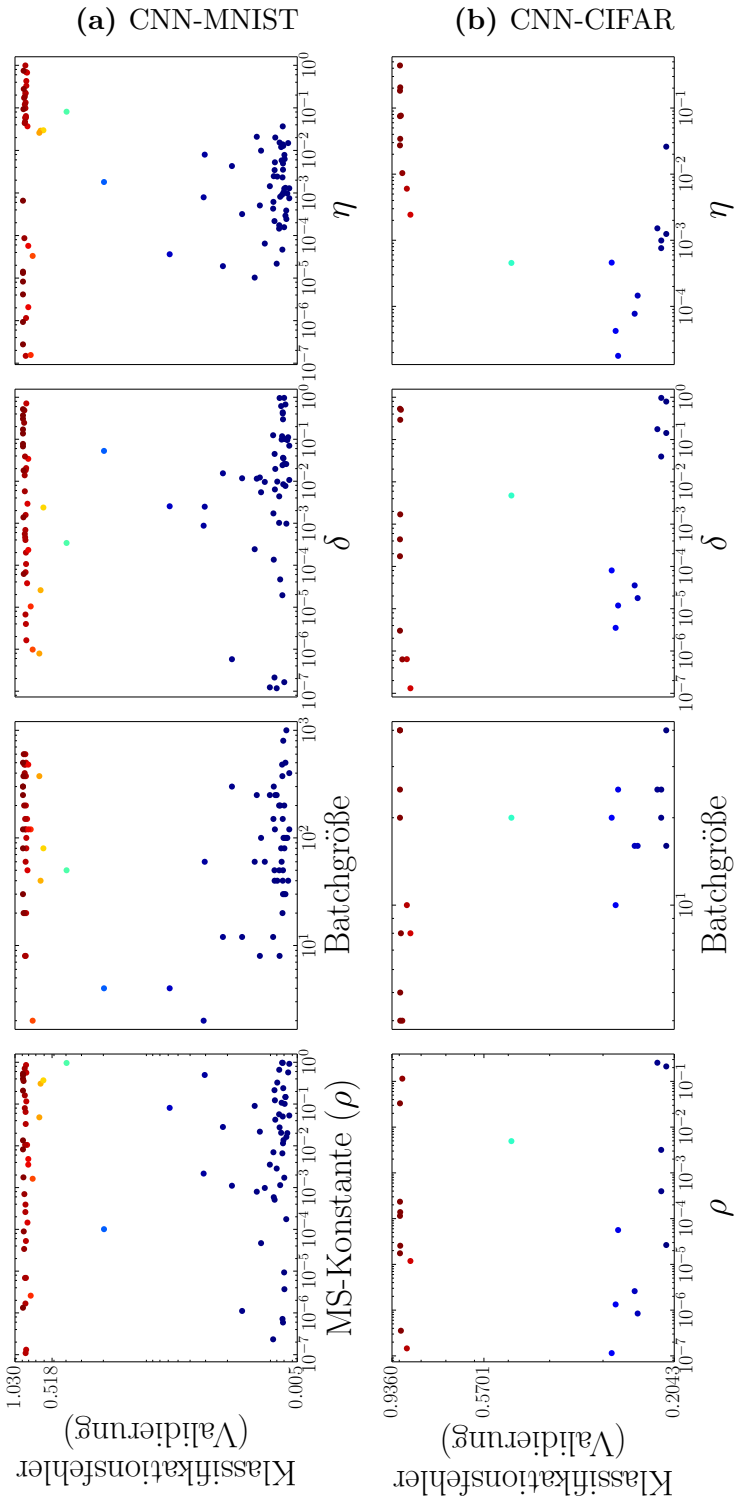


Abbildung 7: Kreuzvalidierte Hyperparameterkombinationen von RMSProp.

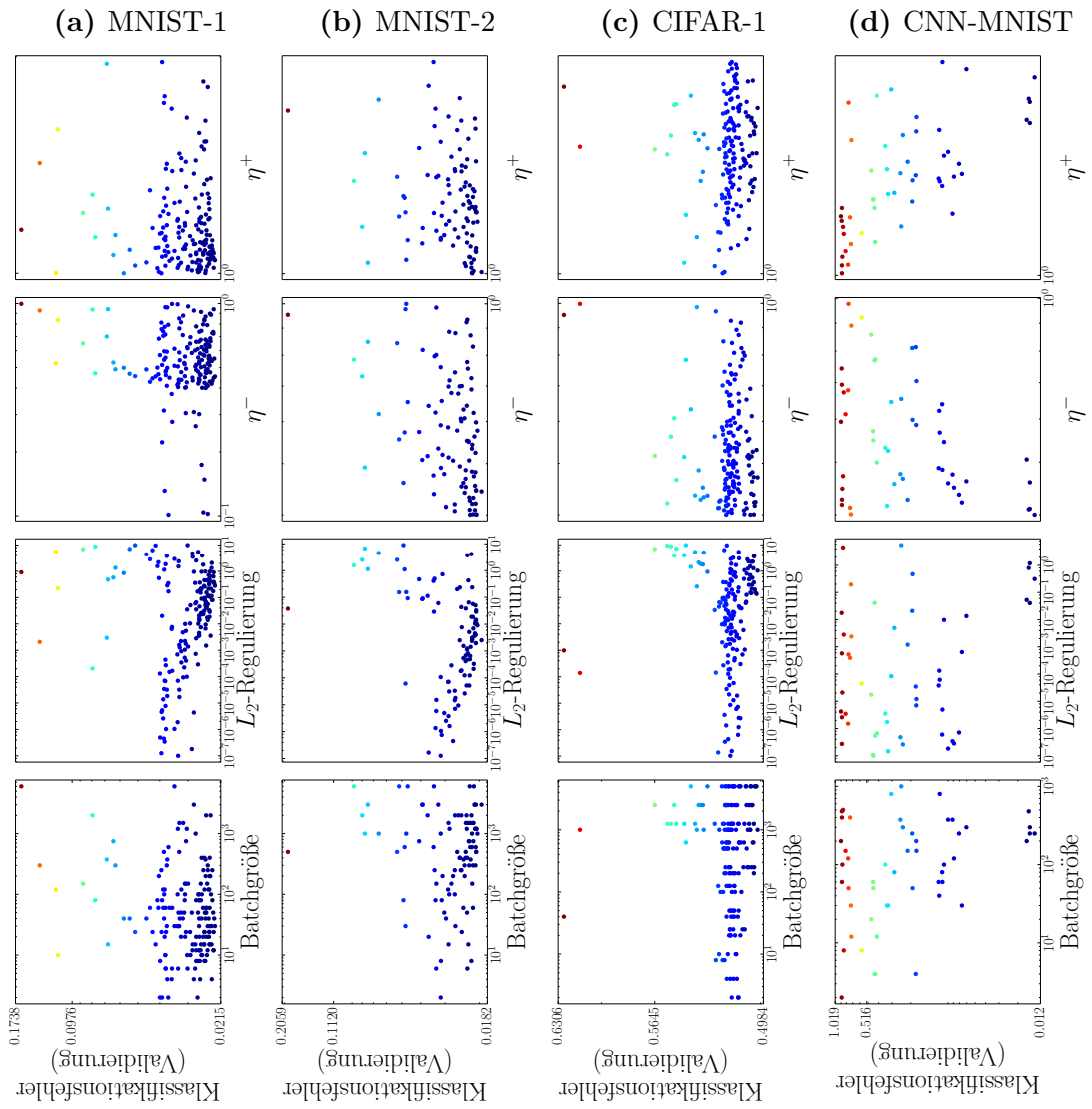


Abbildung 8: Kreuzvalidierte Hyperparameterkombinationen von RPROP.

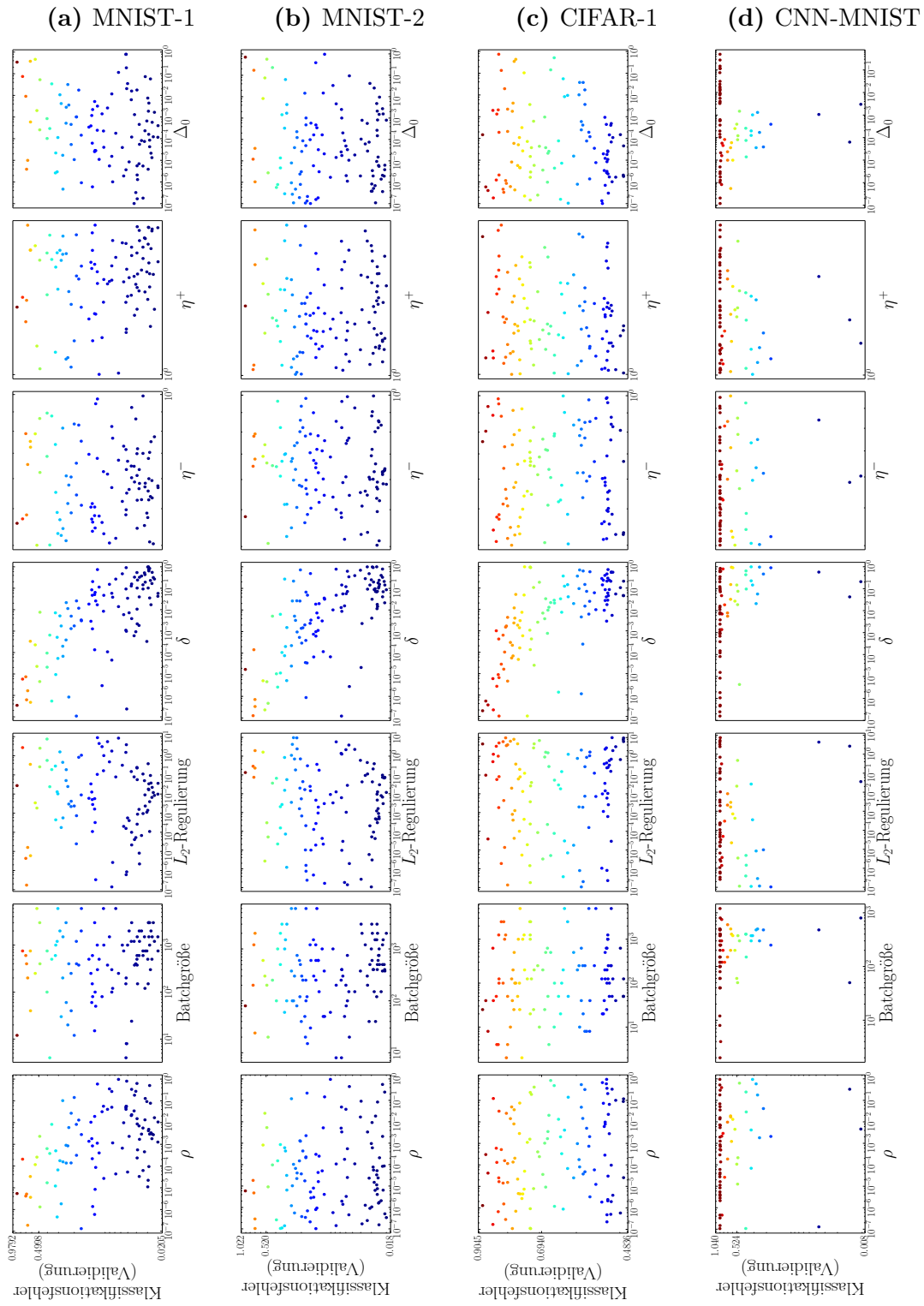


Abbildung 9: Kreuzvalidierte Hyperparameterkombinationen von RRMSProp.

Literatur

Becker, S. und Y. Le Cun (1988). „Improving the convergence of back-propagation learning with second order methods“. In: *Proceedings of the 1988 connectionist models summer school* (siehe S. 25).

Bengio, Y. (2009). „Learning deep architectures for AI“. In: *Foundations and Trends in Machine Learning* (siehe S. 1).

Bergstra, J. S., R. Bardenet, Y. Bengio und B. Kégl (2011). „Algorithms for hyper-parameter optimization“. In: *Advances in Neural Information Processing Systems* (siehe S. 47).

Bergstra, J., D. Yamins und D. D. Cox (2013). „Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms“. In: *Proc. Python for Scientific Computing Conference (SciPy)* (siehe S. 46).

Bishop, C. M. u. a. (1995). *Neural networks for pattern recognition*. Clarendon press Oxford (siehe S. 14).

Dean, J., G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang und A. Y. Ng (2012). „Large Scale Distributed Deep Networks“. In: *Advances in Neural Information Processing Systems (NIPS)* (siehe S. 27, 35).

Duchi, J., E. Hazan und Y. Singer (2010). „Adaptive subgradient methods for online learning and stochastic optimization“. In: *Journal of Machine Learning Research* (siehe S. 2, 25).

Duchi, J., S. Shalev-Shwartz, Y. Singer und A. Tewari (2010). „Composite objective mirror descent“. In: *Conference on Learning Theory (COLT)* (siehe S. 25).

Glorot, X. und Y. Bengio (2010). „Understanding the difficulty of training deep feedforward neural networks“. In: *Proceedings of the International Conference on Artificial Intelligence and Statistics. Society for Artificial Intelligence and Statistics* (siehe S. 12).

Golik, P., P. Doetsch und H. Ney (2013). „Cross-entropy vs. squared error training: a theoretical and experimental comparison.“ In: *Annual Conference of International Speech Communication Association (INTERSPEECH)* (siehe S. 11).

Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. Courville und Y. Bengio (2013). „Maxout networks“. In: *International Conference on Machine Learning (ICML)* (siehe S. 9, 41, 84).

Höft, N., H. Schulz und S. Behnke (2014). „Fast Semantic Segmentation of RGB-D Scenes with GPU-Accelerated Deep Neural Networks“. In: *German Conference on Artificial Intelligence (KI)* (siehe S. 35).

Hinton, G. (2012). *Rmsprop: Divide the gradient by a running average of its recent magnitude*. URL: http://www.amara.org/de/videos/vrXNiLBHyW92/info/rmsprop-divide-the-gradient-by-a-running-average-of-its-recent-magnitude/ (siehe S. 2, 30).

Hinton, G., S. Osindero und Y.-W. Teh (2006). „A fast learning algorithm for deep belief nets". In: *Neural computation* (siehe S. 1).

Hinton, G. und R. Salakhutdinov (2006). „Reducing the dimensionality of data with neural networks". In: *Science* (siehe S. 1).

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever und R. R. Salakhutdinov (2012). „Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (siehe S. 18, 84).

Krizhevsky, A. und G. Hinton (2009). „Learning multiple layers of features from tiny images". In: *Master's thesis, Department of Computer Science, University of Toronto* (siehe S. 39).

LeCun, Y., L. Bottou, G. Orr und K. Müller (1998). „Efficient backprop". In: *Neural networks: Tricks of the trade* (siehe S. 14, 15, 25).

LeCun, Y., C. Cortes und C. Burges (1998). *The mnist dataset of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/ (siehe S. 39).

LeCun, Y., P. Y. Simard und B. Pearlmutter (1993). „Automatic learning rate maximization by on-line estimation of the Hessians eigenvectors". In: *Advances in neural information processing systems* (siehe S. 15).

LeCun, Y., L. Bottou, Y. Bengio und P. Haffner (1998). „Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* (siehe S. 39).

Martens, J. (2010). „Deep learning via Hessian-free optimization". In: *International Conference on Machine Learning (ICML)* (siehe S. 23).

Moody, J., S. Hanson, A. Krogh und J. A. Hertz (1995). „A simple weight decay can improve generalization". In: *Advances in neural information processing systems* (siehe S. 17).

Nesterov, Y. (1983). „A method of solving a convex programming problem with convergence rate O (1/k2)". In: *Soviet Mathematics Doklady* (siehe S. 21).

Ngiam, J., A. Coates, A. Lahiri, B. Prochnow, A. Ng und Q. V. Le (2011). „On optimization methods for deep learning". In: *International Conference on Machine Learning (ICML)* (siehe S. 23, 88).

Polyak, B. T. (1964). „Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* (siehe S. 21).

Prechelt, L. (1998). „Early stopping-but when?" In: *Neural Networks: Tricks of the trade*. Springer (siehe S. 17).

Riedmiller, M. und H. Braun (1993). „A direct adaptive method for faster backpropagation learning: The RPROP algorithm". In: *IEEE International Conference on Neural Networks* (siehe S. 2, 27, 73).

Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg und L. Fei-Fei (2014). *ImageNet*

Large Scale Visual Recognition Challenge. eprint: `arXiv:1409.0575` (siehe S. 87).

Schaul, T. und Y. LeCun (2013). „Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients“. In: *International Conference on Learning Representations (ICLR)* (siehe S. 25).

Schaul, T., S. Zhang und Y. LeCun (2012). „No More Pesky Learning Rates“. In: *International Conference on Machine Learning (ICML)* (siehe S. 23, 29, 41, 83).

Schulz, H. (2014a). *CUV Library*. URL: `http://www.ais.uni-bonn.de/deep_learning/downloads.html` (siehe S. 50).

— (2014b). *MDBQ*. URL: `https://github.com/temporaer/MDBQ` (siehe S. 52).

Schulz, H. und S. Behnke (2012). „Deep Learning - Layer-Wise Learning of Feature Hierarchies“. In: *Künstliche Inteligenz* (siehe S. 1).

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever und R. Salakhutdinov (2014). „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* (siehe S. 18).

Tieleman, T. und G. Hinton (2012). *Lecture 6.5 - rmsprop, COURSERA: Neural Networks for Machine Learning*. URL: `http://climin.readthedocs.org/en/latest/rmsprop.html#id1` (siehe S. 32).

Vincent, P., H. Larochelle, I. Lajoie, Y. Bengio und P.-A. Manzagol (2010). „Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion“. In: *The Journal of Machine Learning Research* (siehe S. 1).

Wiesler, S., J. Li und J. Xue (2013). „Investigations on hessian-free optimization for cross-entropy training of deep neural networks.“ In: *Annual Conference of International Speech Communication Association (INTERSPEECH)* (siehe S. 23).

Zeiler, M. D. (2012). „ADADELTA: An Adaptive Learning Rate Method“. In: *CoRR* (siehe S. 27, 41, 83).