Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Victoria Bibaeva

Evolutionäre Optimierung von
Deep Neural Networks

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Victoria Bibaeva

Evolutionäre Optimierung von Deep Neural Networks

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter : Prof. Dr. Bettina Buth

Abgegeben am 26. Juli 2017

Victoria Bibaeva

Thema der Arbeit

Evolutionäre Optimierung von Deep Neural Networks

Stichworte

Deep Learning, Faltungsnetze, Optimierung der Hyperparameter, Evolutionäre Algorithmen, Simulierte Abkühlung, Metaheuristiken

Kurzzusammenfassung

Faltungsnetze sind eine beliebte Klasse der neuronalen Netzwerke im Bereich Deep Learning mit einer speziellen Architektur, die ihre hervorragende Leistung in vielen Einsatzgebieten wie Bilderkennung, Spracherkennung usw. begründet. Die Architektur der Faltungsnetze verfügt über viele (Hyper-)Parameter, die auf ihre Erkennungsgenauigkeit Einfluss nehmen. Trotz des enormen wissenschaftlichen Interesses an Faltungsnetzen erfolgt die Suche nach guten Hyperparameterwerten meist manuell, was extrem viel Zeit beansprucht und mit dem Risiko verbunden ist, einige erfolgsversprechende Werte zu übersehen. Gegenstand dieser Arbeit ist das Entwerfen von Algorithmen zur automatisierten Hyperparametersuche für Faltungsnetz-architekturen. Anhand des bestehenden Wissens über die Faltungsnetze sollen sie durch eine geschickte Suchstrategie in relativ kurzer Zeit einige sehr gute Parameterwerte liefern. Es werden drei solcher Algorithmen basierend auf bekannten Metaheuristiken wie Evolutionäre Optimierung und Lokale Suche präsentiert, entsprechend dem Anwendungsfall implementiert und miteinander verglichen. Anhand verschiedener Datasets wird ermittelt, welcher Algorithmus unter welchen Bedingungen zu den besten Faltungsnetzarchitekturen führt.

Victoria Bibaeva

Title of the paper

Evolutionary optimization techniques for Deep Neural Networks

Keywords

Deep Learning, convolutional neural networks, hyper-parameter search, evolutionary algorithms, simulated annealing, metaheuristics

Abstract

Convolutional neural networks is a widely spread class of powerful models from the deep learning domain. They have a specific architecture which allows them to tackle successfully many tasks such as image and speech recognition, video analysis etc. Convolutional architectures have a number of (hyper-)parameters which influence the final recognition error rate. Despite the fact that convolutional networks attract ever increasing interest within the research community, a search for good values for their hyper-parameters is carried out for the most part manually, which takes an extremely long time and is prone to overlook some promising values. The subject of this study is designing the algorithms to automatically search for hyper-parameters for convolutional architectures. These algorithms should encompass the existing knowledge about convolutional networks and yield very good hyper-parameter values in a relatively short time due to an appropriate search strategy. In this paper, three of such algorithms based on well-known metaheuristics named evolutionary algorithms and local search will be presented, adjusted to the use case of convolutional architectures and compared. Furthermore, it will be shown which algorithm produces the most successful architectures under which circumstances, using different datasets for image recognition.

Inhaltsverzeichnis

1 Einführung 6

1.1 Motivation 8

1.2 Zielsetzung 8

1.3 Aufbau der Arbeit 8

2 Grundlagen 10

2.1 Layertypen und ihre Hyperparameter 11

2.1.1 Filter Bank Layer 12

2.1.2 Non-Linearity Layer 14

2.1.3 Feature Pooling Layer 17

2.1.4 Full Connection Layer 18

2.2 Auswahl der Filter 21

2.3 Trainingsparameter 24

3 Lösungsansätze zur Architektursuche 27

3.1 Genetischer Algorithmus 33

3.2 Simulated Annealing 36

3.3 Memetischer Algorithmus 39

4 Implementierung 41

4.1 Kodierung der Architektur 44

4.2 Genetischer Algorithmus 48

4.3 Simulated Annealing 51

4.4 Memetischer Algorithmus 53

4.5 Datasets 54

5 Qualitätssicherung 57

6 Ergebnisse 61

6.1 Versuchsaufbau 61

6.2 Auswahl der Metaparameter 63

6.3 Experimente mit MNIST und Verfahrensvergleich 71

6.4 Experimente mit CIFAR-10 78

7 Zusammenfassung 80

7.1 Fazit 80

7.2 Ausblick 81

8 Glossar 83

9 Abbildungsverzeichnis 86

10 Tabellenverzeichnis 88

11 Literaturverzeichnis 89

12 Anhänge 96

1 Einführung

In den letzten Jahren haben enorme wissenschaftliche Fortschritte im Bereich Machine Learning stattgefunden. Unter anderem hat sich das Teilgebiet Deep Learning abgezweigt, das sich mit vielschichtigen neuronalen Netzwerken und entsprechenden Lerntechniken beschäftigt [1]. Durch eine komplexe Architektur solcher Netzwerke wird das Ziel verfolgt, eine Hierarchie von Merkmalen aus den Daten zu erlernen, die die nachfolgende Objektklassifizierung bzw. Mustererkennung ermöglicht [2].

Zu den meist verbreiteten Netzwerkklassen in Deep Learning zählen die sogenannten Convolutional Neural Networks (dt. „**Faltungsnetze**"). Dies ist eine Variante des Multilayer Perzeptrons (**MLP**) mit einer speziellen Architektur, präsentiert zum ersten Mal 1989 von Yann LeCun et al. [3], deren Inspiration die Sehrinde der Säugetiere war. Der weltweite Erfolg der Faltungsnetze begann 2012, als sie bahnbrechende Ergebnisse beim größten Wettbewerb für Objekterkennung namens ILSVRC zeigten [4]. Dabei bestand das Dataset aus 1,4 Millionen Bilder und 1000 Objektkategorien, und die erzielte Fehlerrate lag bei 16,4 % – fast halb so hoch wie im Vorjahr. Seitdem gehören sie zu den „State-of-the-Art"-Methoden in Anwendungsbereichen wie Bild- und Spracherkennung, Objektklassifizierung und Videoanalyse, und ihre Leistung nähert sich immer mehr der des Menschen an [5].

Die Allgegenwärtigkeit der Faltungsnetze erklärt sich neben ihrer überragenden Leistung auch durch stets wachsende Rechenkapazität, Verwendung programmierbarer GPUs, Verfügbarkeit diverser Datasets für viele Problemstellungen sowie einfallsreiche Techniken zur Steigerung der Erkennungsrate. Sie unterscheiden sich gegenüber den anderen Objektklassifizierungsverfahren durch ihre inhärente Robustheit bezüglich Objekttransformationen wie Rotation, Verschiebung und Skalierung [6], was letztendlich zu einer deutlichen Reduktion der Trainingsmenge und -zeit führt. Der Grund für diese Robustheit ist das Anlernen mehrerer Faltungskerne während des Trainings. Jeder Faltungskern dient dazu, ein bestimmtes Merkmal von zu klassifizierenden Objekten innerhalb eines Bildes zu erkennen, z.B. Linien, Rechtecke oder Kreise. Die Neuronen, die zu

demselben Faltungskern gehören, teilen miteinander die Netzgewichte, wodurch die Anzahl der trainierbaren Parameter im Vergleich zu einem allgemeinen MLP stark reduziert wird.

Im Jahr 2014 lag beim ILSVRC die niedrigste Fehlerrate bereits unter 6,7 %. Dabei nutzte das Sieger-Team „GoogLeNet" sieben 22-lagige Faltungsnetze mit jeweils 5 Millionen trainierbaren Parametern [7]. Zum Vergleich: ein solches Netzwerk von den sieben lieferte eine Fehlerrate von 7,89 %. Der Gewinner von ILSVRC-2015 [8] verwendete 6 Netze, zwei davon mit 152 Schichten, und erzielte 3,57 % als Fehlerrate. Seitdem geht der Trend eher zur Nutzung von mehreren Netzen, deren Ergebnisse aggregiert werden (Stichwort „Ensemble Learning" [9]), sowie zur Vertiefung der Netzarchitektur bei gleichzeitiger Senkung der Rechenkomplexität [8]. Dabei liegt das Verbesserungspotenzial schon im Nachkommastellenbereich.

Die steigende Komplexität der Architektur bringt allerdings viele Probleme mit sich, unter anderem die Notwendigkeit des effektiveren Trainings, das Ausweichen von lokalen Optima durch geschickte Initialisierung und die Vermeidung von *Overfitting* (Überanpassung an das Trainingsset). Deswegen existieren neuerdings auch Versuche, ein konkretes Netz so zu komprimieren, dass es weniger Rechenkapazität beansprucht, dennoch die Fehlerrate auf dem gleichen Niveau bleibt (SqueezeNet, [10]).

Die Einflussfaktoren auf die Performanz eines Faltungsnetzes, die als Erkennungs- bzw. Fehlerrate bei der Objektklassifizierung definiert ist, sind folgende (siehe Glossar und Abbildung 29, Kapitel 8):

- Netzarchitektur (inkl. dazugehörige Hyperparameter)
- Trainingsparameter
- Dataset

Zu den beiden letztgenannten Faktoren liegen viele wissenschaftliche Untersuchungen vor (z.B. siehe [11] und [12]). Die Architektur wird dagegen üblicherweise durch das unsystematische Variieren von bewährten Parametern und Techniken ermittelt. Der Leistungsvorteil von einem Netz gegenüber einem anderen wird meist durch Experimente begründet, und es mangelt immer noch an der theoretischen Grundlage für die gewonnenen Erkenntnisse. Nur wenige Forschungsbeiträge beschäftigen sich mit der empirischen Auswirkung einzelner Hyperparameter (vgl. [13], [14], [15]). Der gegenseitige Einfluss dieser Hyperparameter bleibt dagegen weitestgehend unbekannt, allein aus dem Grund, dass er unter Umständen von den beiden anderen Faktoren abhängig sein kann. Es fehlt also immer noch das genauere Verständnis, warum Faltungsnetze so viel leisten und wie diese Leistung verbessert werden kann.

1.1 Motivation

Ein guter Ansatz bei der Entwicklung eines (neuen) Deep-Learning-Modells besteht darin, nicht nur eine, sondern mehrere Einstellungen der Hyperparameter gegeneinander zu testen. Damit bleibt kein Zweifel, dass der erste eingetretene Erfolg nicht zufällig war, und es wird unter Umständen ein noch effizienteres Modell gefunden. Die manuelle Suche nimmt allerdings extrem viel Zeit in Anspruch, und es entsteht die Gefahr, dass der Forscher einige gute Hyperparameter übersieht. Eine automatisierte Hyperparametersuche würde diese mühsame Arbeit abnehmen und mithilfe einer geschickten Suchstrategie die ursprüngliche Performanz des Modells steigern. Diese Arbeit entstand aus der Idee heraus, ein solches Verfahren zu entwerfen und zu implementieren, welches auf den vorhandenen wissenschaftlichen Erkenntnissen im Bereich Faltungsnetze basiert. Dabei sollen in dieser Arbeit die erfolgsversprechenden Techniken für die Hyperparametersuche bei MLP zum ersten Mal auf Faltungsnetze übertragen werden.

1.2 Zielsetzung

In dieser Arbeit werden drei verschiedene Lösungsverfahren zur Hyperparametersuche präsentiert, die bereits bei Architekturen von MLP erfolgreich angewendet worden sind. Es wird anschließend aufgezeigt, wie sie auf den Anwendungsfall von Faltungsnetzen angepasst werden können. Diese Lösungsansätze müssen implementiert und miteinander verglichen werden, um festzustellen, welcher die beste Faltungsnetzarchitektur liefert. Um den Vergleich zu generalisieren, werden repräsentative Datasets zum Training der Faltungsnetze ausgewählt. Unterschiedliche Vergleichskriterien (Lösungsqualität, Komplexität, Laufzeit) werden bei allen Verfahren analysiert und ausgewertet, ferner wird der Einfluss des Datasets und der Trainingsparameter auf die Performanz des Endergebnisses ermittelt.

1.3 Aufbau der Arbeit

In Kapitel 2 wird der Stand der Wissenschaft zum Thema Hyperparameter der Faltungsnetze vorgestellt. Diese Erkenntnisse sollen dabei helfen, die Hyperparametersuche deutlich einzuschränken und eventuelle Fallstricke zu vermeiden. Anschließend beschreibt Kapitel 3 das Verhältnis von Architektursuche zu anderen Optimierungsproblemen sowie die möglichen Lösungsrichtungen – sowohl im Fall von MLP als auch für Faltungsnetze. Es wird außerdem detailliert auf drei Lösungsansätze eingegangen, die für die Umsetzung bei der Architektursuche von Faltungsnetzen ausgewählt worden sind. Das erste Verfahren basiert auf Evolutionärer Optimierung, das zweite – auf Lokaler Suche, und das dritte Verfahren ist ein Hybrid aus den beiden erstgenannten.

Ferner wird in Kapitel 4 berichtet, wie die Implementierung dieser drei Algorithmen erfolgte. Zunächst geht es dort um eine binäre Repräsentation der Faltungsnetzarchitektur, die für die Anwendung der Algorithmen notwendig ist. Danach wird analysiert, wie die jeweiligen verfahrensspezifischen Parameter gesetzt werden müssen, um das beste Ergebnis mit Faltungsnetzen zu erzielen. Im darauffolgenden Kapitel 5 werden die umgesetzten qualitätssichernden Maßnahmen erläutert, die zur Gewährleistung der reibungslosen Ausführung der Verfahren beitragen sollen.

Kapitel 6 widmet sich dem Vergleich der ausgewählten Lösungsverfahren anhand bestimmter Kriterien, um das beste Verfahren zur automatisierten Hyperparametersuche zu bestimmen. Zusätzlich werden die Faktoren ausgewertet, die auf das Ergebnis Einfluss haben können. Im letzten Kapitel 7 werden die wichtigsten Erkenntnisse zusammengefasst und die verbliebenen Verbesserungspotentiale kurz angerissen.

2 Grundlagen

Um eine Objektklassifizierung zu gewährleisten, bekommt ein Faltungsnetz als Input ein zweidimensionales Bild. Aus diesem werden zunächst bestimmte Objektmerkmale schrittweise extrahiert, sodass die Merkmale der späteren Schichten (z.B. Rechtecke, Kreise) aus Merkmalen der früheren Schichten (z.B. Linien, Ecken, Bögen) zusammengesetzt werden. Dadurch entsteht im Laufe der Verarbeitung die Merkmalshierarchie.

Im ersten Schritt werden aus einem Input-Bild mehrere sogenannte **Feature-Maps** (dt. Abbildung von Merkmalen) abgeleitet, wobei jede Feature-Map für ein Merkmal zuständig ist [6]. In weiteren Schritten können die Merkmale kombiniert oder verfeinert werden, wodurch aus dieser Menge von Feature-Maps weitere Feature-Maps erzeugt werden.

Ein Faltungsnetz besteht typischerweise aus mehreren **Feature-Extraction-Phasen** (dt. Markmalextraktion). Eine solche Phase kann wiederum aus verschiedenen Neuronenschichten bzw. **Layertypen** zusammengesetzt werden [13], und zwar aus *Filter Bank Layer*, *Non-Linearity Layer* und *Pooling Layer*. Jeder Layertyp bildet eine Menge von Feature-Maps auf die nächste Menge ab. Nach den Feature-Extraction-Phasen folgt ein **Klassifikator**, im Grunde ein klassisches MLP, der die Zugehörigkeit zu einer der bekannten Objektklassen berechnet.

Ein bekanntes Beispiel einer Faltungsnetzarchitektur ist in Abbildung 1 demonstriert. Dieses Netz, genannt „LeNet-5“, wurde in [6] präsentiert und für die Klassifizierung von handschriftlichen Ziffern erfolgreich angewendet. Es besteht aus zwei Feature-Extraction-Phasen (C1 mit S2, C3 mit S4), gefolgt von einem dreilagigen Klassifikator (C5, F6, OUTPUT). Hierbei ist „Convolution“ ein Synonym für Faltung im Filter Bank Layer (siehe Kapitel 2.1.1), bei „Subsampling“ handelt es sich um einen Feature Pooling Layer, und der dritte Layertyp, Non-Linearity Layer, kann implizit diesen beiden folgen. Die Anzahl und die Dimensionen der erzeugten Feature-Maps ist mit einer speziellen Notation anzugeben, siehe Abbildung 1 oben.

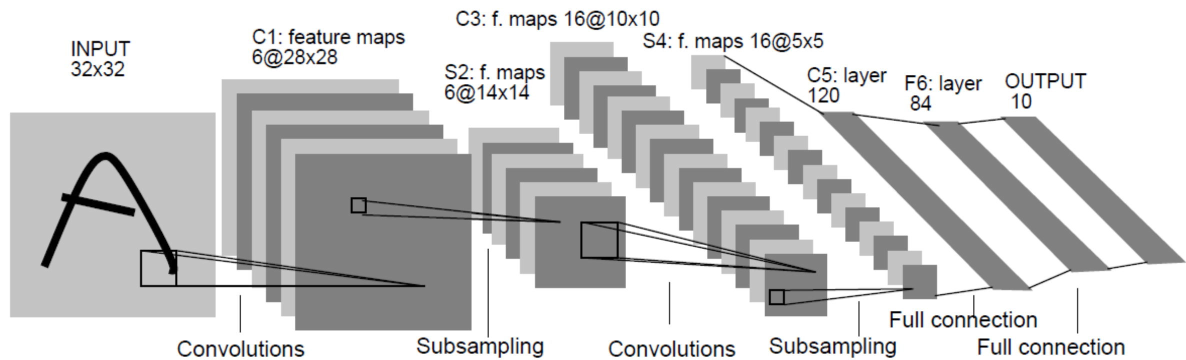

Abbildung 1. Architektur von „LeNet-5“ [6].

Wie kann eine Faltungsnetzarchitektur variiert werden, um die bestehende Erkennungsrate positiv zu beeinflussen? Mit anderen Worten, aus welchen **Hyperparametern** besteht die Architektur?

- Die Anzahl der Feature-Extraction-Phasen könnte modifiziert werden, um die „Tiefe“ des Netzes zu verändern.
- Jede Feature-Extraction-Phase kann wiederum diverse Arten des Non-Linearity Layers oder sogar weitere Kombinationen der drei genannten Layertypen nutzen.
- Auch der Klassifikator kann im Prinzip mehrlagig sein und einige Neuronenverbindungen ausfallen lassen.

Im Folgenden wird zunächst auf die soeben aufgelisteten Punkte eingegangen. Anschließend wird die Auswahl der Filter erläutert, mit welchen der Filter Bank Layer initialisiert werden soll, sowie die Trainingsparameter, die ebenfalls entscheidend für die Performanz einer Architektur sind.

2.1 Layertypen und ihre Hyperparameter

Jeder Layertyp in einem Faltungsnetz erfüllt seinen eigenen Zweck. So ist der Filter Bank Layer derjenige, der für die Extraktion der Merkmale zuständig ist. Dessen wichtige Eigenschaft ist die Unempfindlichkeit gegenüber Verschiebung und Verformung des Objektes im Bild. Im Non-Linearity Layer wird eine nichtlineare Aktivierungsfunktion auf das Output der vorherigen Schicht pixelweise angewendet, um die eingehenden Signale (das Vorhandensein der Merkmale) weiterzuleiten bzw. zu unterdrücken. Der Pooling Layer dient dazu, das Netzwerk unabhängig von der geringfügigen Merkmalstranslation zu halten und die Bildauflösung zu reduzieren. Schließlich ermöglicht ein Full Connection Layer die

Objektkategorisierung, indem er die Zusammenhänge zwischen den Merkmalen in seinem Input lernt.

Um besser einschätzen zu können, wie viel Variation in einer Architektur möglich ist, werden anschließend die wichtigsten Layertypen betrachtet:

2.1.1 Filter Bank Layer

Diesem Layertyp haben Faltungsnetze ihren Namen zu verdanken, denn hier wird **Faltung** ausgeführt. Dies ist eine Technik aus dem Gebiet der digitalen Bildverarbeitung, die als Kantendetektor sowie zur Bildglättung und -schärfung verwendet wird [13]. Faltung ist eine Abbildung von einem gegebenen Quellbild auf ein Zielbild und erfolgt mithilfe eines Faltungsoperators, der einen Faltungskern (auch **Filter** genannt) auf die überlappenden Bildabschnitte anwendet und dadurch ein bestimmtes Merkmal überall im Quellbild hervorheben kann. Ein Filter ist im Prinzip eine Zahlenmatrix, die elementenweise mit den Pixelwerten jedes Quellbildabschnittes multipliziert wird, und die Summe der resultierenden Elemente wird als Pixelwert an der entsprechenden Stelle des Zielbildes gespeichert. Somit wird jeder Quellbildabschnitt zu einem Zielbildpixel „gefaltet", was einen atomaren Teil des Faltungsoperators darstellt.

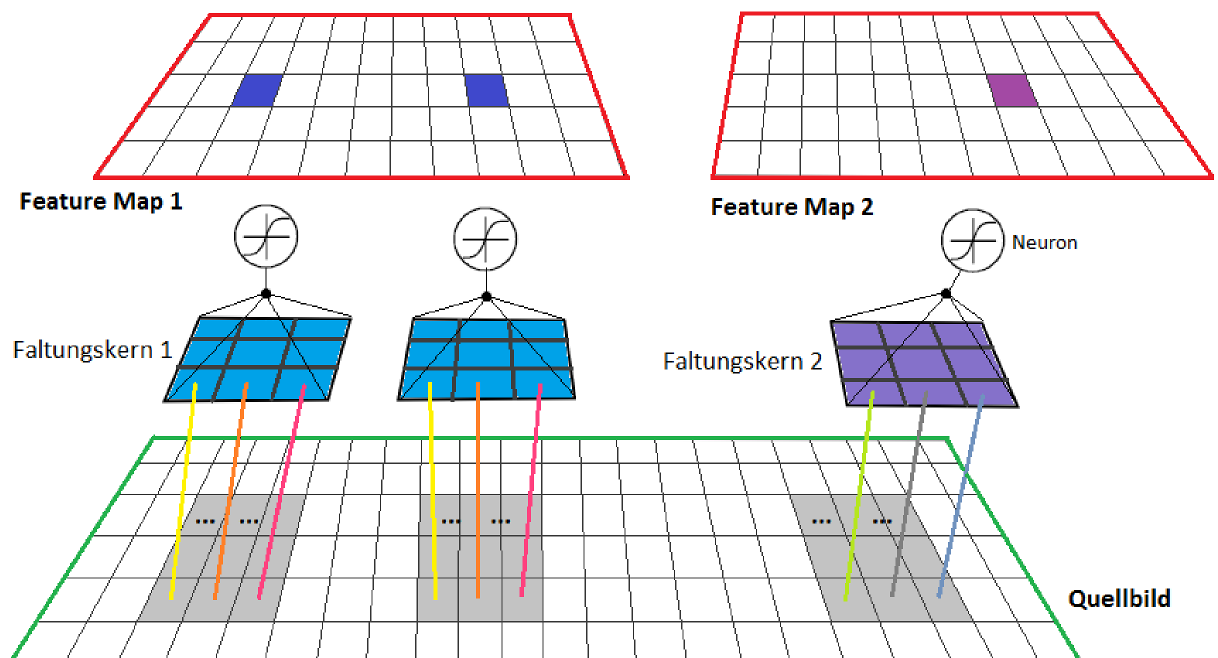

Abbildung 2. Funktionsweise des Filter Bank Layers (nach [16]).

Dementsprechend bekommt der Filter Bank Layer aus der ersten Feature-Extraction-Phase ein zweidimensionales Bild als Input. Die Grauwerte aus den benachbarten Pixeln dieses Quellbildes sind die Eingangswerte für jedes Neuron des Layers, siehe Abbildung 2. Da ein klassisches Neuron an sich nichts anderes macht als die Input-Werte mit einer Matrix der Gewichte zu multiplizieren und zusammenzuaddieren (ggf. mit Bias), ist seine Funktionsweise einem atomaren Teil des Faltungsoperators äquivalent.

Die Neuronen des Filter Bank Layers sind auf mehrere Ebenen aufgeteilt [6], sodass die Neuronen aus einer Ebene die gleichen Gewichte haben (in Abbildung 2 durch die gleiche Farbe dargestellt). Jede Neuronenebene agiert wie ein vollständiger Faltungsoperator mit einem und demselben Filter, der das Quellbild auf eine Feature-Map abbildet. Diese erfüllt den Zweck, ein konkretes Merkmal zu erkennen. Um mehrere Merkmale zu extrahieren, werden allerdings mehrere Faltungsoperatoren bzw. Filter benötigt, daher kommt der Begriff „Filter Bank". Der Layer-Output ist demnach eine Menge von Feature-Maps, die von den nachfolgenden Schichten verarbeitet wird.

Im allgemeinen Fall besteht der Input eines Filter Bank Layers aus n_1 Feature-Maps x_i (für $i \in 1..n_1$), deren Größe $n_2 \times n_3$ beträgt. Für die erste Feature-Extraction-Phase gilt $n_1 = 1$ oder $n_1 = 3$, wenn das Input-Bild farbig ist. Der Output des Layers seien m_1 Feature-Maps der Größe $m_2 \times m_3$, bezeichnet als y_j mit $j \in 1..m_1$. Die Filter Bank besteht also aus m_1 Faltungskernen k_j der Größe $l_2 \times l_3$. Dann wird jede Output-Feature-Map folgendermaßen berechnet [13]:

$$y_j = b_j + \sum_i k_j * x_i$$

Das Symbol $*$ steht hier für den Faltungsoperator; die Summanden b_j (Bias) sowie alle Faltungskerne k_j (Neuronengewichte) können trainiert werden und gehören daher zu den **trainierbaren Parametern** eines Faltungsnetzes. Die maximale Größe der Output-Feature-Map wird erreicht, wenn bei der Faltung die Schrittgröße 1 Pixel verwendet wird. In diesem Fall gelten folgende Formeln:

$$m_2 = n_2 - l_2 + 1 \; ; \; m_3 = n_3 - l_3 + 1$$

D.h. der Filter wird auf jeden Bereich einer Input-Feature-Map von links nach rechts und von oben nach unten angewendet, sodass er vollständig innerhalb des Bildes bleibt, – so entsteht die sogenannte *„valide" Faltung* [17]. Generell, wenn die Schrittgröße als $s_2 \times s_3$ angegeben ist (s_2 für horizontale Pixelreihen, s_3 für vertikale), wird die Output-Größe durch folgende Formeln berechnet (die eckigen Klammern stehen dabei für die Abrundungsfunktion):

$$m_2 = \left\lfloor \frac{n_2 - l_2}{s_2} \right\rfloor + 1 \; ; \; m_3 = \left\lfloor \frac{n_3 - l_3}{s_3} \right\rfloor + 1$$

Um den Informationsverlust zu vermeiden, können die Werte so gewählt werden, dass bei obiger Division kein Rest entsteht.

Nach diesen Berechnungen beträgt die Gesamtanzahl der trainierbaren Parameter $m_1 \cdot (l_2 \cdot l_3 + 1)$. Es gibt hingegen 5 Hyperparameter der Architektur (m_1, l_2, l_3, s_2, s_3), mithilfe dessen die anderen Variablen berechnet werden können. In der Realität werden allerdings stets quadratische Faltungskerne mit ungerader Länge verwendet [11], was aus 5 Hyperparameter lediglich 3 macht (m_1, l, s). Trotzdem gibt es unzählige Kombinationsmöglichkeiten ihrer Werte, wodurch das Durchtesten aller Varianten sehr viel Zeit beansprucht.

Glücklicherweise haben sich nur einige Filtergrößen etabliert, und zwar von 3×3 bis 11×11 – mit der Schrittgröße meistens von 1×1 bis 5×5, aber nie die Filtergröße übersteigend (siehe [7], [18]). Die Filter müssen am besten so dimensioniert sein, dass sie noch klein sind (im Verhältnis zu den Input-Bildern) und die Entstehung einer Merkmalshierarchie zulassen. Was die Werte von m_1 betrifft, existieren nur heuristische Einschätzungen, je nachdem, wie viele Filter bzw. zu erkennende Features erwartet werden. Für große Datasets mit komplexen Bildern und vielen feinen Details können beispielsweise bis zu 100 Faltungskerne für die erste Feature-Extraction-Phase verwendet werden [15]. Folglich gibt es $23 \cdot 100 = 2300$ Möglichkeiten, um den ersten Filter Bank Layer zu entwerfen. In weiteren Phasen wird m_1 deutlich größer als 100, um alle Feature-Kombinationen zu erfassen, aber nicht zu groß, um alle resultierende Netzgewichte noch speichern zu können.

Zudem muss berücksichtigt werden, dass eine feststehende Netzwerkarchitektur für Bilder anderer Dimensionen nicht ohne weiteres verwendbar ist, weil alle Feature-Map-Dimensionen neu berechnet werden müssen, um am Ende auf die fest vorgegebene Klassenanzahl zu kommen [6].

2.1.2 Non-Linearity Layer

Dieser Layertyp wendet eine nichtlineare Aktivierungsfunktion auf den Output des Filter Bank Layers punktweise an und wird nicht selten als dessen Bestandteil angesehen [13]. Traditionell kamen hier z.B. eine Sigmoidfunktion $g_1(x) = \frac{1}{1+e^{-x}}$ oder Tangens hyperbolicus $g_2(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ zum Einsatz. Diese Funktionen sind sättigend, d.h. ihr Wertebereich ist beschränkt, und beliebige Ausgangswerte der vorherigen Neuronenschicht werden demzufolge auf ein Intervall „gequetscht“ [6]. Allerdings wurde festgestellt, dass solche Funktionen das *Problem des verschwindenden Gradients* verursachen (siehe Kapitel 2.3) und dadurch das Training verlangsamen [19].

Nachfolgend wurde eine neue Funktion $g_3(x) = \max(0, x)$ vorgeschlagen, die das Training der Deep-Learning-Modelle um ein Vielfaches beschleunigte (siehe [5]). Ein Neuron mit dieser Aktivierungsfunktion nennt sich **ReLU** (engl. „rectified linear unit“) und hat den Vorteil, dass es nicht gesättigt werden kann und deswegen keine Normalisierung der Eingangswerte braucht. Ein starkes Eingangssignal wird einfach weitergeleitet, anstatt übermäßig gedämpft

zu werden, wie es bei sättigenden Aktivierungsfunktionen der Fall wäre [11]. Somit wird ein ReLU nicht dem Problem des verschwindenden Gradients ausgesetzt. Außerdem sichert es einen Deaktivierungszustand, der robust gegenüber Rauschen im Eingangssignal ist [19].

Heutzutage gehört ReLU zu den beliebtesten Aktivierungsfunktionen in Faltungsnetzen, trotz der Tatsache, dass der Mittelwert seiner Aktivierungen nicht bei 0 liegt. Dadurch agieren mehrere ReLUs jedoch wie ein Bias für die nächste Schicht und lösen das sogenannte „bias shift" aus, welches vom Netz ständig durch Gewichteveränderungen korrigiert werden muss [19]. Ein weiterer Mangel ist das *Problem der toten ReLUs* [11]: Wenn x immer negative Werte einnimmt, dann liefert $g_3(x)$ stets 0 und es findet kein Lernen mehr statt. Dieses Problem könnte eventuell durch eine geschicktere Initialisierung der Gewichte vermieden werden.

Aus der Notwendigkeit, die Nachteile von ReLUs zu eliminieren, wurde daraufhin eine Reihe von Aktivierungsfunktionen präsentiert, die auf den besten Eigenschaften von ReLU basieren, aber dennoch negative Ausgabewerte zulassen. In einer Studie [15] wurden diese miteinander verglichen, und zwar anhand einer konkreten Architektur auf dem ILSVRC Dataset. Demnach wurden die höchsten Erkennungsraten bei ELU, Maxout und ihrer Kombination festgestellt.

ELU (engl. „exponential linear unit") ist ein Neuron mit folgender Aktivierungsfunktion [19]:

$$g_4(x) = \begin{cases} x, & wenn\ x > 0 \\ \varepsilon \cdot (e^x - 1), & sonst \end{cases}$$

Dabei kontrolliert $\varepsilon > 0$ die untere Grenze des Wertebereiches, siehe Abbildung 3 für den von Autoren empfohlenen Wert $\varepsilon = 1$. Dank seiner Linearität im positiven Definitionsbereich verhindert ELU das Problem des verschwindenden Gradients genauso gut wie ReLU, nähert aber den Mittelwert von allen Aktivierungen zu 0 und beschleunigt daher das Training. Die Sättigung zu $-\varepsilon$ im negativen Definitionsbereich bedeutet wiederum weniger Varianz in den (negativen) Ausgabewerten und einen nicht informativen Deaktivierungszustand. Mit anderen Worten, die spezifische Eigenschaft von ELU besteht darin, dass es den Grad der Merkmalsanwesenheit kodiert, aber weder die Merkmalsabwesenheit quantifiziert noch die Ursachen der Abwesenheit unterscheidet [19].

Maxout wurde von [20] entworfen als eine Technik zur Approximation jeder beliebigen Aktivierungsfunktion und ist sowohl in Neuronen von MLP als auch von Faltungsnetzen verwendbar. Hierbei werden auf n Inputs einer verdeckten Schicht aus m Neuronen zunächst k verschiedene lineare Funktionen angewendet, um anschließend ihren maximalen Wert zu ermitteln. Das i-te Neuron der Maxout-Schicht ($i \in 1..m$) berechnet also folgende Aktivierungsfunktion:

$$g_5^i(x) = \max_j z_{ij} = \max_j \left(\sum_{l=1}^{n} x_l \cdot W_{lij} + b_{ij} \right)$$

Bei W_{lij} und b_{ij} handelt es sich um trainierbare Parameter – eine Matrix der Gewichte und Bias; $j \in 1..k$ und $x = (x_1, x_2, \dots x_n)$ ist ein Vektor der Input-Werte. Deswegen kann ein Maxout-Neuron als abschnittsweise definierte Approximationsfunktion für eine beliebige konvexe Aktivierungsfunktion interpretiert werden, und ein Maxout-Netz mit nur zwei solchen verdeckten Neuronen kann dadurch jede kontinuierliche Zielfunktion approximieren [21]. So versucht z.B. die Maxout-Kurve in Abbildung 3, mit $k = 3$ sich einer quadratischen Aktivierungsfunktion (Parabel) anzunähern.

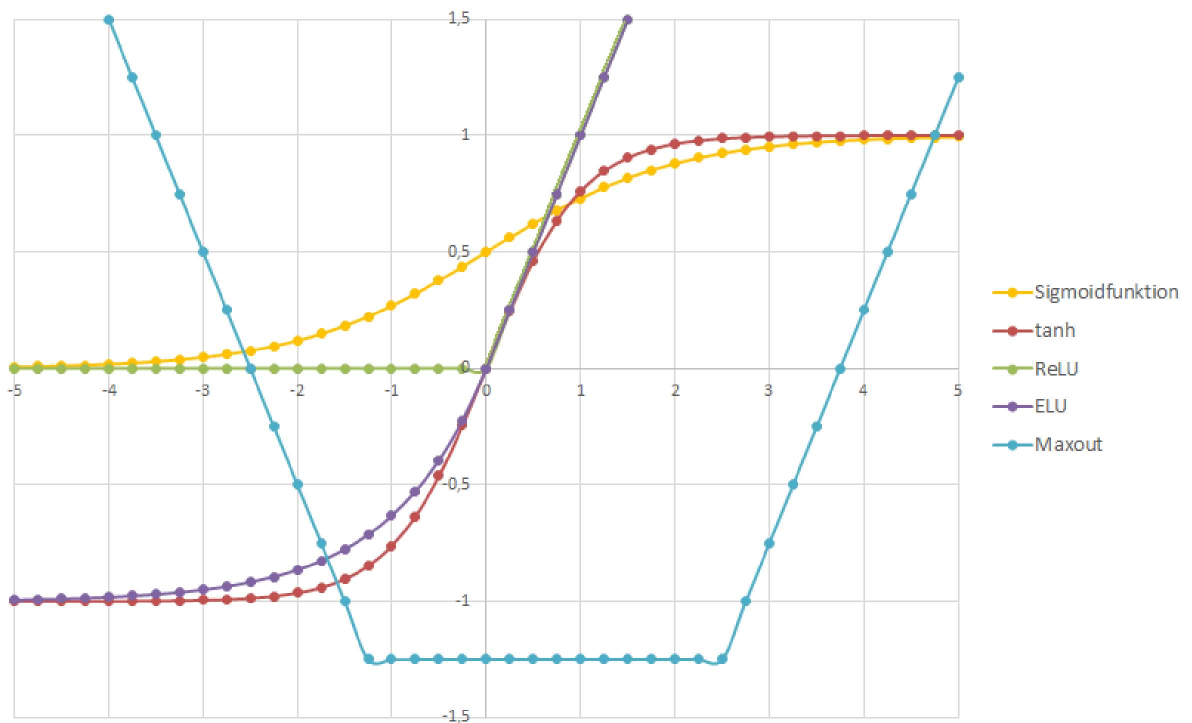

Abbildung 3. Aktivierungsfunktionen.

Ein Maxout-Netzwerk lernt demzufolge nicht nur das Verhältnis zwischen den Neuronen durch die Netzgewichte, sondern auch die Aktivierungsfunktion pro Neuron. Dies erhöht jedoch die Anzahl der trainierbaren Parameter pro Schicht um Faktor k und verlangsamt das Training [15]. Während ein Standard-Neuron eine Funktion $z = \sum_{l=1}^{n} x_l \cdot W_l + b$ berechnet, würden m solche Neuronen insgesamt $m \cdot (n+1)$ Gewichte inkl. Bias lernen. Ein Maxout-Neuron benötigt k Standard-Neuronen (mit $k \cdot (n+1)$ Parametern), um aus den

resultierenden k Ausgangswerten den maximalen zu wählen und als seine Aktivierung zu nutzen, sodass sich pro Schicht mit m Maxout-Neuronen $m \cdot k \cdot (n+1)$ trainierbare Parameter ergeben – k Mal so viel wie bei m Standard-Neuronen.

Beim Einsatz von Maxout in Faltungsnetzen sollte aber bedacht werden, dass die Neuronen einer Schicht in Feature-Maps gruppiert sind. Wenn eine Erhöhung der Parameterzahl dort nicht erwünscht ist, kann die Menge von m Feature-Maps in k Gruppen aufgeteilt werden, sodass pro Gruppe aus m/k Feature-Maps eine Aktivierungsfunktion entsteht (anstatt eine Funktion pro Feature-Map) [21]. Diese Variante von Maxout soll aus Performanz-Gründen bevorzugt werden.

Noch besser als ELU und Maxout hat sich bei Experimenten in [15] ihre Kombination erwiesen, wobei nach einem Filter Bank Layer immer ein ELU folgte, und im Klassifikator nur Maxout zum Einsatz kam. Die Motivation hinter dieser Konstellation ist es, dass Maxout in mittleren Schichten viele redundante Parameter enthält und es dort durch weniger komplexe Aktivierungsfunktionen ersetzt werden kann [21].

Insgesamt ergeben sich daher 4 Varianten des Non-Linearity Layers: ReLU, ELU, Maxout sowie eine Kombination aus ELU und Maxout. Dies entspricht einem zusätzlichen Hyperparameter.

2.1.3 Feature Pooling Layer

Der letzte Layertyp der Feature-Extraction-Phase reduziert die Auflösung der Feature-Maps aus der vorherigen Schicht. Dafür wird in jeder Feature-Map der gleiche Filter auf die überlappenden Regionen angewendet, der nicht mehr die gewichtete Summe, sondern den maximalen oder mittleren Wert je Region berechnet [11]. Der Layer wird dementsprechend als „Max-“ oder „Average Pooling“ bezeichnet. Daraus resultieren genauso viele neue, kleinere Feature-Maps, währenddessen ein Teil der eingehenden Information verloren geht.

Die Hyperparameter sind hier also die Filtergröße $l'_2 \times l'_3$ und Schrittgröße $s'_2 \times s'_3$ (mit $s'_i \leq l'_i$ für $i = 2{,}3$), und die Dimensionen der Output-Feature-Maps können wieder mit den oben genannten Formeln für m_2, m_3 berechnet werden, wobei die Anzahl von Input-Feature-Maps gleich der Anzahl von Output-Feature-Maps ist. Traditionell (siehe z.B. [5], [14], [18]) werden hierfür Filter der Größe 2×2 oder 3×3 mit der Schrittgröße 1 oder 2 benutzt. Diese wenigen Alternativen sind dadurch bedingt, dass das Verwenden eines 2×2 Filters ohne Überlappung (Schrittgröße = 2) ohnehin 75% Informationsverlust bedeutet, da aus 4 Pixel der Region nur ein Pixel entsteht. Es wird außerdem empfohlen [15], bei einem Filter von 3×3 und Schrittgröße 2 die Input-Feature-Maps um 1 Pixel in jede Richtung zu vergrößern (engl. „*padding*“), weil der Output dadurch größer wird und mehr Informationen enthält.

Die Entscheidung für eine konkrete Variante des Pooling Layers, die erwartungsgemäß bessere Erkennungsraten liefern würde, kann nicht generell für alle Netzarchitekturen getroffen werden. Einige Forscher [13] stellten fest, dass Average Pooling für ihre

Problemstellung effizienter ist. Andere dagegen diskutieren [22], dass diese Variante nachteilig ist, da sie alle Aktivierungen der vorherigen Schicht berücksichtigt, auch wenn viele davon sehr klein sind (z.B. 0 bei ReLU). Somit werden die wenigen starken Signale abgeschwächt. Solches Verhalten kommt bei der Max-Pooling-Variante nicht vor, allerdings neigt diese lt. den Forschern eher zum Overfitting [22]. Gleichzeitig gibt es einige erfolgreiche Architekturen, die beide Varianten abwechselnd beinhalten [11].

Neuere Erkenntnisse [23] bestätigen, dass Max und Average Pooling sich für unterschiedliche Arten von Datentransformationen eignen, d.h. anders auf Rotation, Verschiebung und Skalierung des Merkmals reagieren. Aufgrund dessen wurde vorgeschlagen, die Summe von Max und Average Pooling zu verwenden. Die Wirksamkeit dieser Kombination wurde in [15] experimentell belegt. Die intuitive Erklärung davon ist, dass der Max-Pooling-Summand die Selektivität und Invarianz der Merkmale erhält, und der Average-Pooling-Summand alle verfügbaren Informationen nutzt, statt sie wegzuwerfen.

Letztendlich hat ein Pooling Layer 3 Hyperparameter – Pooling-Typ (Max, Average oder ihre Summe), Filtergröße l' und Schrittgröße s'. Nach den oben genannten Richtlinien für l' und s' ergibt sich eine Auswahl aus 12 verschiedenen Ausprägungen von Pooling Layer.

2.1.4 Full Connection Layer

Dieser Layertyp gehört zum Klassifikator und agiert auf die gleiche Weise wie eine verdeckte Schicht in einem MLP. Wie der Name des Layertyps sagt, ist er mit der vorherigen Neuronenschicht vollständig verbunden. Mit anderen Worten, als Eingangswerte für seine Neuronen dienen alle einzelnen Pixelwerte des Vorgängerlayers. Aus diesem Grund enthält er die meisten trainierbaren Parameter im ganzen Faltungsnetz.

Angenommen, die letzte Feature-Extraction-Phase vom Faltungsnetz liefert n_1 Feature-Maps der Größe $n_2 \times n_3$, und die Zugehörigkeit des Input-Bildes zu einer der C Klassen muss festgestellt werden. Dann hat ein Full Connection Layer $n_1 \cdot n_2 \cdot n_3$ Eingänge. Falls er aus m_1 Neuronen besteht, so ergeben sich $m_1 \cdot (n_1 \cdot n_2 \cdot n_3 + 1)$ Gewichte inkl. Bias, die noch gelernt werden müssen. Der Output des Layers kann als m_1 Feature-Maps der Größe 1×1 betrachtet werden.

Optional kann zu einem Klassifikator noch eine oder zwei verdeckte Schichten mit entsprechend m_1' und C Neuronen geschaltet werden. Diese sind ebenfalls vollständig miteinander verbunden, damit die letzte Schicht C Ausgänge umfasst. Allerdings müssen insgesamt $m_1 \cdot (n_1 \cdot n_2 \cdot n_3 + 1) + m_1' \cdot (m_1 + 1) + C \cdot (m_1' + 1)$ Parameter gelernt werden, und es gilt: $m_1 \geq m_1' \geq C$. Der Klassifikator enthält demzufolge weniger Neuronen als z.B. ein Filter Bank Layer mit der gleichen Anzahl an Output-Feature-Maps, aber dafür viel mehr Verbindungen bzw. Netzgewichte, weil diese von den Neuronen nicht miteinander geteilt werden.

Trotz der Tatsache, dass der Klassifikator aus wenigen Full Connection Layers besteht, und zwar typischerweise aus zwei (siehe [13], [15]) oder drei (wie bei [24] oder [5], dem Gewinner von ILSVRC 2012), kann er je nach Layer-Größe viele irrelevante Zusammenhänge zwischen den Merkmalen lernen, was unter Umständen zu starkem Overfitting führt [25].

Um das Overfitting zu vermeiden, wurde für Klassifikatoren eine vielversprechende Technik namens **Dropout** in [11] präsentiert. Sie setzt zur Trainingszeit den Output jedes Neurons aus dem gegebenen Full Connection Layer mit der Wahrscheinlichkeit p auf 0 und lässt es somit aus dem Netz ausfallen, siehe Abbildung 4. Dadurch können sich die Neuronen nicht mehr auf alle Signale der Vorgängerschicht verlassen, sondern sind gezwungen, robustere Features zu lernen [5]. Dropout kann gleichzeitig aus einem anderen Blickwinkel betrachtet werden [11]: Jede Trainingsiteration resultiert in einer anderen Netzwerkarchitektur, weil stets unterschiedliche Neuronen aus dem Netz ausfallen. Folglich besteht die Wirkung des Dropouts darin, dass die mittlere Fehlerrate über viele Architekturkandidaten berechnet wird, ohne diese explizit trainieren zu müssen.

Dropout gehört mittlerweile zu den beliebten Methoden bei diversen Deep-Learning-Modellen und bringt eine deutliche Verbesserung der Erkennungsrate auf allen gängigen Datasets, wobei die optimale Dropout-Wahrscheinlichkeit auf $p \approx 0{,}5$ geschätzt wurde [25].

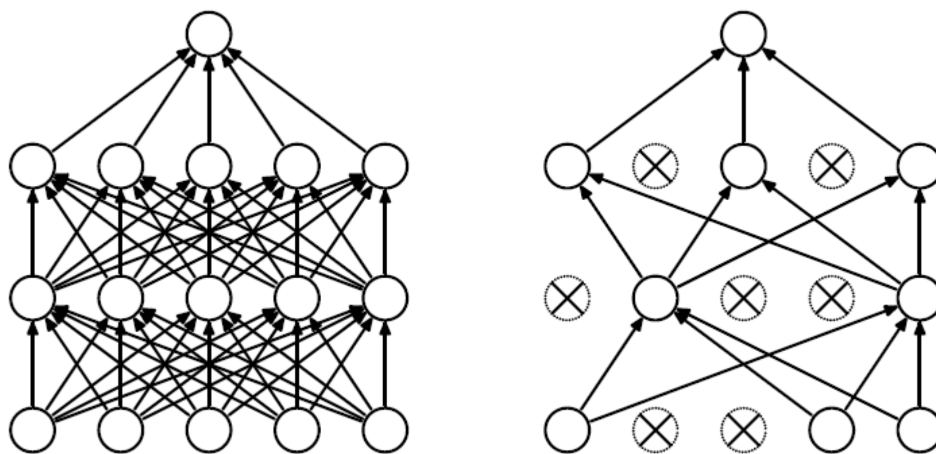

Abbildung 4. Klassifikator: links ohne, rechts – mit Dropout [25].

Darüber hinaus existiert eine Erweiterung von Dropout, bekannt als **DropConnect** [26]. Anstatt die Outputs der Neuronen im Klassifikator auf 0 zu setzen, werden hier die zufälligen Neuronengewichte eliminiert. D.h. ein Neuron bekommt pro Trainingsschritt eine andere Teilmenge seiner Inputs. DropConnect zeigte sich im Vergleich zum Dropout in vielen Fällen als konkurrenzfähig und erreichte teilweise „State-of-the-Art"-Performanz auf Benchmark-Datasets, und zwar mithilfe mehrerer aggregierter Modelle. Allerdings wurde in [26]

festgestellt, dass DropConnect erst dann effektiver ist, wenn die Gewichte pro Bild, anstatt pro mehrere Bilder (wie beim Dropout), geändert werden. Es benötigt deswegen mehr Trainingszeit, weil der Klassifikator aus sehr vielen Gewichten besteht, sowie Speicherplatz, um die Ergebnisse pro Gewichtekombination zu verwalten. Aus diesen Gründen wurde in dieser Arbeit auf DropConnect verzichtet.

Letztendlich empfiehlt es sich, einen Klassifikator aus einigen Full Connection Layers aufzubauen und dazwischen standardmäßig das Dropout zu verwenden. Da nach einem Full Connection Layer normalerweise auch ein Non-Linearity Layer folgt, müssen hier lediglich 2 Hyperparameter pro Layer festgelegt werden – die Anzahl der Outputs und die Variante der Aktivierungsfunktion.

Die richtige Reihenfolge der Layertypen innerhalb einer Feature-Extraction-Phase spielt ebenfalls eine wichtige Rolle beim Aufbau einer guten Faltungsnetzarchitektur. Bei den Untersuchungen in [13] zeichnete sich die Reihenfolge „Filter Bank Layer -> Non-Linearity Layer -> Pooling Layer“ als erfolgreichste Kombination ab, andere Kombinationen (z.B. ohne Non-Linearity Layer) waren deutlich schlechter. Diese Erkenntnis ist deswegen plausibel, weil jeder Layertyp versucht, einen Teil der Sehrinde von Säugetieren zu modellieren, und diese Abfolge von Neuronenschichten in der Natur tatsächlich vorkommt. Trotzdem können in einer Architektur mehrere nacheinander folgende Filter Bank Layers auftauchen, die ihre eigene Feature-Extraction-Phase bilden [15].

Wie viele Feature-Phasen sollten nun verwendet werden? Experimente in [13] zeigen eindeutig, dass eine Phase immer schlechtere Ergebnisse liefert als zwei Phasen, unabhängig von dem Trainingsregime. Die Obergrenze wird allerdings durch Rechen- und Speicherkapazität, aber auch durch die Datasetgröße vorgegeben.

Folglich beträgt die Gesamtzahl der möglichen Varianten nur für die erste Feature-Extraction-Phase $2300 \cdot 4 \cdot 12 = 110400$. In nachfolgenden Phasen steigt die Anzahl der Output-Feature-Maps und somit die Anzahl aller Kombinationen, ganz zu schweigen von diversen Klassifikatoren. Dies macht das Ausprobieren von jeder einzelnen Architektur nahezu unmöglich, da das Training eines Netzwerkes mit einem großen Dataset und GPU-Einsatz mehrere Tage in Anspruch nehmen kann.

Alle oben beschriebenen Hyperparameter eines Faltungsnetzes und ihre möglichen Werte können folgendermaßen zusammengefasst werden, siehe Tabelle 1. Die gegenseitige Auswirkung dieser Hyperparameter ist weitgehend unbekannt und muss systematisch erforscht werden. Zwar existieren Versuche, die Zusammenhänge für ein konkretes Dataset zu ermitteln (z.B. MNIST und NORB in [13], ILSVRC in [15]). Die Autoren merkten aber an, dass sich die Ergebnisse der anderen Forscher nicht ohne weiteres für ein neues Dataset verwenden lassen.

	Hyperparameter				
Filter Bank Layer	Aktivierungsfunktion (ReLU, ELU, Maxout, ELU + Maxout)	--	Filtergröße (3, 5, 7, 9, 11)	Schrittgröße (1, 2, 3, 4, 5)	Anzahl Outputs
Feature Pooling Layer	--	Pooling (Max, Average, Max + Average)	Filtergröße (2, 3)	Schrittgröße (1, 2)	--
Full Connection Layer	Aktivierungsfunktion (ReLU, ELU, Maxout, ELU + Maxout)	--	--	--	Anzahl Outputs

Tabelle 1. Layertypen und ihre Hyperparameter.

2.2 Auswahl der Filter

Es gibt verschiedene Möglichkeiten, wie die initialen Filter bzw. Neuronengewichte eines Faltungsnetzes zu bestimmen sind [13]. Erstens, sie können *manuell festgelegt* werden und bleiben während des Trainings konstant. Standardmäßig werden dafür Gabor-Filter genommen, die bekanntlich das Sehrindemodell der Katzen und Primaten nachahmen, aber auch Filter für Kantendetektion basierend auf SIFT, HOG usw. (für Referenzen siehe [13]). Dazu gehören ebenso rein *zufällige Filter*, die keinerlei erkennbare Merkmale aufweisen. Die manuelle Methode erspart zwar Rechenzeit; andererseits, können vorhandene Kenntnisse über das Dataset auch nicht zum Vorteil genutzt werden, und der Lernerfolg wird möglicherweise dadurch verschlechtert.

Die zweite Vorgehensweise ist das *Unüberwachte Lernen* der Faltungskerne, wobei diese den Gabor-Filtern ähneln, wenn sie mit natürlichen Bildern trainiert werden [27]. Das Lernen ermöglicht es, die Filter aller Feature-Extraction-Phasen präzise und an das Dataset zugeschnitten zu konstruieren. Im Fall des Unüberwachten Lernens muss das Dataset nicht beschriftet sein, was sich bei Millionen von Bildern als großer Vorteil erweist.

Drittens, das *Überwachte Lernen* der Filter mithilfe des Gradientenverfahrens ist die meistverbreitete Methode mit der längsten Tradition (wie ursprünglich in [3]) und wird auch im Rahmen dieser Arbeit ausschließlich verwendet. Zwar erfordert sie viele beschriftete Bilder, um alle Netzgewichte möglichst eindeutig zu bestimmen, aber dazu kann auf diverse Benchmark-Datasets zurückgegriffen werden.

Alle drei genannten Strategien zur Filterauswahl wurden in [13] auf der Grundlage von verschiedenen Datasets miteinander verglichen. Dabei lautete ein erstaunliches Ergebnis – die Differenz zwischen ihren Fehlerraten lag innerhalb von 1-2 %, wenngleich das Überwachte Lernen siegte. Am zweitbesten war die Technik, bei der die Filter mithilfe des Unüberwachten Lernens initialisiert (siehe z.B. „Predictive Sparse Decomposition" bzw. PSD-Algorithmus in [28]) und anschließend mittels Gradientenverfahrens trainiert wurden. Gleichzeitig erwies sich eine Architektur mit konstanten zufälligen Faltungskernen und überwacht trainiertem Klassifikator nicht viel schlechter als die Konkurrenten.

Aus diesem Experiment resultiert die Frage, ob es eine Kohärenz zwischen der Performanz von zufälligen und von trainierten Filtern gibt. Diese Frage wurde experimentell in [13] sowie theoretisch in [17] positiv beantwortet. Dazu wurde in beiden Fällen ein optimaler Input berechnet, d.h. derjenige, bei welchem die höchste Aktivierung nach einer Feature-Extraction-Phase vorliegt. Aus der Tatsache, dass beide optimale Inputs sich sehr ähnlich sind (vgl. Abbildung 5), lässt sich auf die inhärenten Eigenschaften einer Faltungsnetzarchitektur schließen, die von der Filterart unabhängig sind, – Frequenzselektivität und Translationsinvarianz (siehe [17]).

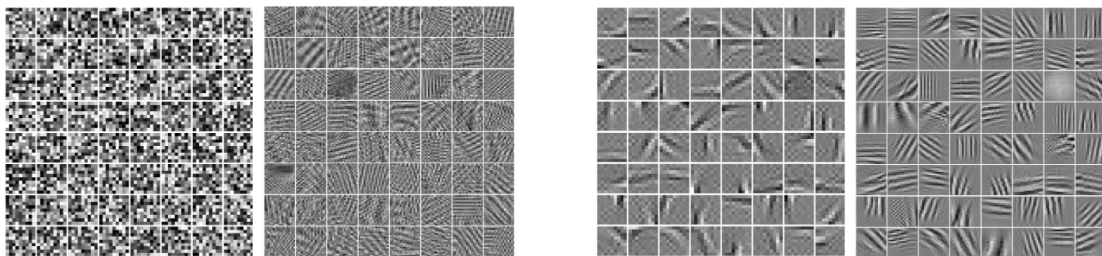

Abbildung 5. Optimaler Input von zufälligen Filtern (links) und von unüberwacht gelernten Filtern (rechts) [13].

Basierend auf diesem Ergebnis kann ein simples heuristisches Verfahren für die Suche nach guten Netzarchitekturen abgeleitet werden [13]: Jede interessante Architektur muss mehrmals zufällig initialisiert und getestet werden, um ihre durchschnittliche Performanz zu ermitteln. Nach allen Testzyklen kann eine Entscheidung getroffen werden, welche Architektur besser ist, und zwar ohne jegliches Training.

Die beschriebene Heuristik hilft also dabei, einige gute Werte der Hyperparameter wie Filtergröße oder Art der Aktivierungsfunktion schnell und ohne großen Aufwand zu bekommen. Sie wurde bereits in [29] eingesetzt und dahingehend verfeinert, dass der Klassifikator der Netzkandidaten immer überwacht trainiert werden soll. Dies hat zweierlei Wirkung – einerseits kann dadurch festgestellt werden, wie gut sich ein Netz an das Trainingsset anpasst, andererseits können die gelernten Netzgewichte des Klassifikators später zum Vorteil für das vollständige Training benutzt werden. Am Ende werden die besten

Architekturen samt Feature-Extraction-Phasen trainiert, um alle Netzgewichte zu bekommen und bestmögliche Erkennungsraten zu erzielen.

Wie genau erfolgt nun die Initialisierung der zufälligen Filter? Sie stellt durchaus eine Herausforderung dar, denn eine gute Initialisierung ist einer der entscheidenden Faktoren für den Trainingserfolg der Deep-Learning-Modelle [30]. Deswegen existieren unterschiedliche *Initialisierungsschemata*, die speziell für Faltungsnetze entworfen worden sind. Sie sollen dafür sorgen, dass keine redundanten oder doppelten Faltungskerne entstehen und dass die erzeugten Gewichte ausgewogen sind. Falls die Gewichte zu groß initialisiert sind, dann werden die Neuronen eventuell schnell gesättigt, bei zu kleinen sind auch die Gradienten sehr klein, und das Training dauert länger [6].

Nach einem solchen Schema werden die **Gauß-initialisierten Filter** erzeugt, dessen Werte aus einer Gauß'schen Verteilung mit dem Mittelwert 0 und Standardabweichung σ genommen werden:

$$gauss(x) = \frac{1}{\sigma \cdot \sqrt{2\pi}} \cdot e^{-\frac{x^2}{2 \cdot \sigma^2}}$$

Der Standardwert für σ ist $0{,}01$, zur besseren Veranschaulichung zeigt Abbildung 6 jedoch einen Gauß-initialisierten Filter mit $\sigma = 0{,}5$, sodass die Kontraste deutlicher sind. Die Effizienz dieses Schemas für überdurchschnittlich große Faltungsnetze wurde in [24] und [31] als nicht ausreichend beurteilt.

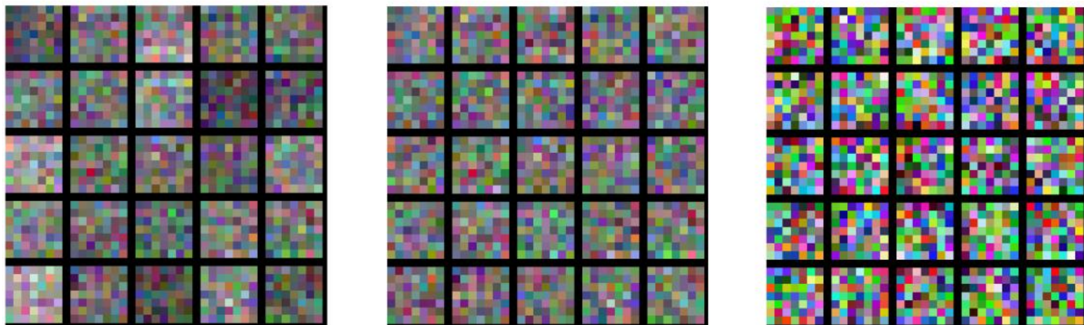

Abbildung 6. Initialisierungsschemata von links nach rechts: Gauß, MSRA, Xavier.

Weitere Schemata beziehen die Anzahl der Inputs n (das sogenannte „Fan-In“) bei der Berechnung der Gewichte ein. Die Motivation dafür ist eine Werteverteilung in gleichem Intervall für alle Neuronen von einer Schicht [6] und der Wunsch, dass Neuronen mit mehr Eingängen auch kleinere Gewichte haben sollen [27].

So wurde in [30] ein Schema **Xavier** vorgeschlagen, das folgendermaßen definiert wird:

$$W_{ij} \sim Uniform\left(-\sqrt{\frac{3}{n}};\ \sqrt{\frac{3}{n}}\right)$$

Hier steht $Uniform(-a;\ a)$ für die gleichmäßig verteilten Werte im Intervall von $-a$ bis a, aus welchen die Gewichte W_{ij} zufällig ausgewählt werden. Obwohl die Ableitung des obigen Intervalls ursprünglich auf der Linearität der Aktivierungsfunktion basiert, funktioniert die Xavier-Initialisierung in der Praxis auch für andersartige Aktivierungsfunktionen sehr gut.

Das neueste Schema aus dieser Kategorie namens **MSRA** stammt aus [31] und verwendet den Gauß-initialisierten Filter mit $\sigma = \sqrt{\frac{2}{n}}$. Es wurde hauptsächlich für ReLU-ähnliche Aktivierungsfunktionen konzipiert und erzielte eine Konvergenz[1] bei 30-lagigen Faltungsnetzen, was mit Xavier-Filtern nicht möglich war. In Abbildung 6 ist sichtbar, dass sich Gauß- und MSRA-initialisierte Filter teilweise ähneln.

2.3 Trainingsparameter

Die meisten klassischen überwachten Lernalgorithmen für neuronale Netzwerke basieren auf Backpropagation, die wiederum eine Variante vom Gradientenverfahren ist [6]. Backpropagation sorgt für die Anpassung von Netzgewichten, sodass das Netz die gewünschten Outputs anhand der gegebenen Input-Bilder berechnet und damit eine möglichst fehlerfreie Objekterkennung gewährleistet. Dafür wird der Netzfehler von der Ausgangsschicht des Netzes zurück bis zu der Eingangsschicht propagiert.

Angenommen, es liegt ein Dataset mit Input-Bildern und den jeweiligen Klassenlabels vor. Dieses wird in ein Trainingsset und Testset unterteilt, um die Lerneigenschaften des Netzes unabhängig voneinander zu prüfen. Dann berechnet ein Faltungsnetz für jedes Trainingsbild dessen Zugehörigkeit zu einer der gegebenen Klassen, und zwar anhand der aktuellen Netzgewichte. Eine sogenannte *Lossfunktion* (dt. Verlustfunktion, Fehlerfunktion) misst den Fehler, d.h. die Diskrepanz zwischen dem berechneten und dem richtigen Klassenlabel. Das Ziel des Lernens kann also als Minimierung der durchschnittlichen Lossfunktion über alle Trainingsbilder formuliert werden [6].

In der Praxis ist es allerdings viel wichtiger, das Fehlverhalten des Netzes für nie zuvor gesehene Muster zu kennen, z.B. für Testbilder. Deswegen wird zusätzlich die Erkennungsrate auf dem Testset gemessen, bekannt als *Accuracy* (dt. Genauigkeit). Die Lücke zwischen Loss und Accuracy sinkt mit steigender Größe des Trainingssets und wird bei vielen

[1] Zur Definition von *Konvergenz* siehe Glossar, Kapitel 8.

Lernalgorithmen geschätzt und zusammen mit der Lossfunktion optimiert (Stichwort „structural risk minimization“, siehe [6]).

Um die Lossfunktion durch die gezielte Gewichteveränderung zu minimieren, wird das stochastische Gradientenverfahren (**SGD**) angewendet, mit der Annahme, dass die Lossfunktion $L(W)$ kontinuierlich in Bezug auf die Netzgewichte W und fast überall differenzierbar ist [6]. Typische Formeln zum Aktualisieren der Gewichte verwenden die Trägheitsmethode, die die Konvergenz des Gradientenverfahrens beschleunigt [32]:

$$V_{i+1} = \mu \cdot V_i - \lambda\big(\delta \cdot W_i + \nabla L(W_i)\big)$$

$$W_{i+1} = W_i + V_{i+1}$$

Wie aus diesen Formeln erkennbar, wird ein iterativ akkumulierter Geschwindigkeitsvektor berechnet, der in die Richtung des persistenten Gradientenabstiegs zeigt. Je größer dieser Vektor ist, desto größer fällt die Gewichteveränderung aus. Dabei ist λ die Lernrate, $\mu \in [0; 1]$ ist der Trägheitskoeffizient (engl. „momentum“), δ steht für „Weight Decay“ (dt. etwa Gewichtesenkung), und $\nabla L(W_i)$ ist der Gradient der Lossfunktion in der Iteration i.

Der Trägheitsterm glättet die Gradientenwerte, die während des Trainings entstehen, sodass die Updates der Gewichte nicht direkt vom Gradienten abhängig sind, der anfällig für Rauschen ist [27]. Der Weight-Decay-Term wird wiederum dazu benutzt, um die Gewichte zu verkleinern, auch wenn der Gradient 0 ist, und bestraft die stetige Gewichtevergrößerung. Es resultiert in einem Netz mit kleineren Gewichten, als bei $\delta = 0$, verbessert aber auch den gesamten Loss [5] und verhindert Overfitting [22]. Typische Werte sind $\mu = 0{,}9$ und $\delta = 0{,}0005$ (siehe z.B. [15], [5]).

Was die Lernrate λ betrifft, so ist sie der allerwichtigste Trainingsparameter eines Faltungsnetzes und soll auf alle Fälle optimiert werden [33]. Bei zu klein gewählter Lernrate konvergiert der Lernalgorithmus sehr langsam, bei zu großer wird er dagegen divergieren [27]. Sie bleibt entweder konstant oder wird im Laufe des Trainings allmählich reduziert. Der Standard-Verlauf der Lernrate (engl. „learning rate decay policy“) ist je nach Bedarf z.B. linear, quadratisch, stufig oder mit der Wurzel fallend. In [15] wurden diese Varianten miteinander verglichen, und die lineare Lernratensenkung hat am schnellsten das gewünschte Performanzniveau erreicht:

$$\lambda = \lambda_0(1 - \gamma i)$$

Offenbar sollte γ so gesetzt werden, dass die Lernrate am Ende des Trainings nahezu 0 ist, etwa $\gamma = \frac{1}{I+1}$ bei der letzten Iterationsnummer I. Die Empfehlung zur Wahl von λ_0 lautet meistens, mit $\lambda_0 = 0{,}01$ anzufangen und viel ausprobieren [27]. Neueste Erkenntnisse geben dennoch eine Vorgehensweise vor, wie ein guter Startwert für die Lernrate schneller zu finden ist, mehr dazu siehe [33] und Kapitel 5.

SGD ermöglicht das Training in Mini-Batches, d.h. in kleineren Teilmengen des Datasets mit fixer Größe, wobei pro Trainingsiteration jeweils ein Mini-Batch verarbeitet wird. Die *Batch-*

Größe ist ein Trade-Off zwischen der Berechnung der Gewichteveränderung pro Bild, was viel Ressourcenverbrauch und ggf. Rauschen bedeutet, und pro gesamtes Dataset, wodurch das Lernen extrem lange dauert und zu schlechteren Erkennungsraten führen könnte [6]. Beispielsweise kann die Batch-Größe aus dem Intervall $[1; 200]$ genommen werden, ein guter Wert ist 32 [27].

Wenn das Trainingsset A Bilder enthält und die Batch-Größe a ist, dann wird das komplette Dataset in A/a Iterationen durch das Netz propagiert – in diesem Fall wird von einer *Epoche* gesprochen [27]. Die Iterationsanzahl I soll demzufolge ein Vielfaches von A/a betragen, sodass mehrere Epochen vergehen – je nach Laufzeiten manchmal bis zu 1000 [11]. Um immer wieder evaluieren zu können, wie gut die Leistung des Faltungsnetzes fortschreitet, kann in gewünschten Abständen das Testen stattfinden. Hierzu wird das ganze Testset mit B Bildern, aufgeteilt in Mini-Batches der Größe b, abgearbeitet, und zwar innerhalb von B/b Testiterationen.

Das fundamentale Problem vom Gradientenverfahren in Deep Learning ist das *Problem des verschwindenden Gradients*. Es wurde bei sättigenden Aktivierungsfunktionen wie Sigmoid und Tangens hyperbolicus entdeckt [1]. Wenn jede Schicht ihr Inputsignal um Faktor k skaliert (z.B. weil sie nicht korrekt initialisiert wurde), dann wird das ursprüngliche Signal nach L Schichten um k^L skaliert, also bei $k < 1$ exponentiell reduziert [34]. Dasselbe gilt für zurückfließende Gradienten (Fehlersignale), weil sie bei den sättigenden Funktionen sehr klein sind. Jede Schicht multipliziert den Gradienten der Nachfolgerschicht mit ihrem eigenen (lt. Kettenregel zur Berechnung des Gradienten der verketteten Funktionen) und propagiert es zurück zur Vorgängerschicht, sodass nach L Schichten der Gradient gleich dem Produkt von L sehr kleinen Zahlen ist. Die Gradienten werden also exponentiell kleiner, je tiefer das Netz ist. Dadurch fallen die Gewichte-Updates in unteren Schichten infinitesimal klein aus, und ein Lernfortschritt findet dort kaum statt. Das Ausmaß des Problems kann dank der Verwendung von Trägheitsmethode, nicht sättigenden Aktivierungsfunktionen und passender Gewichteinitialisierung minimiert werden, siehe [32], [19], [30].

Zu den in dieser Arbeit verwendeten **Trainingsparametern** gehören demzufolge der Typ des Lernalgorithmus (hier: SGD), Anzahl der Trainingsiterationen I, initiale Lernrate λ_0 und γ – Parameter der linearen Lernratensenkung, Trägheitskoeffizient μ, Weight-Decay-Koeffizient δ sowie die Batch-Größen bzw. vollständigen Größen für das Trainings- und Testset (a, A, b, B).

3 Lösungsansätze zur Architektursuche

In diesem Abschnitt wird nun der Stand der Wissenschaft bezüglich der Suche nach guten Architekturen für Faltungsnetze und einige verwandte Netzwerkklassen behandelt. Anschließend werden drei vielversprechende Lösungsansätze detailliert beschrieben, die bisher noch nicht auf Faltungsnetze angewendet worden sind.

Bei der Objektklassifizierung wird das Ziel verfolgt, für ein gegebenes Dataset ein Faltungsnetz zu finden, das diese Daten am besten generalisiert. *Generalisierung* bedeutet per Definition [35], dass das Netz auch solche Input-Bilder richtig klassifiziert, die nicht zum Trainingsset gehören. Mit anderen Worten, es wird das Netz mit dem niedrigsten Testfehler bzw. mit der höchsten Erkennungsrate am Testset gesucht. Die Qualität des Netzes kann zwar auch durch Trainingsdauer, Rechenkomplexität oder Speicherverbrauch gemessen werden, jedoch sind diese Kriterien vielmehr von der Implementierung bzw. vom Werkzeug abhängig.

Im letzten Kapitel wurde verdeutlicht, dass es eine enorme Anzahl an verschiedenen Faltungsnetzen gibt, die für ein Dataset geeignet sind. Dies macht das manuelle Ausprobieren aller Varianten nahezu unmöglich, wenn das Training nur eines Netzwerkes mit einem großen Dataset mehrere Tage, sogar Wochen in Anspruch nimmt. Ein automatisiertes Vorgehen könnte dagegen von Vorteil sein, weil es das lästige Testen aller Kombinationsmöglichkeiten übernimmt und keine Gefahr läuft, einige gute Netze zu übersehen. Dafür sollten so viele Regeln wie möglich aus der vorhandenen Wissensbasis über Faltungsnetze berücksichtigt werden, um die besten Ergebnisse zu erzielen und die von vornherein bekannten Sackgassen zu vermeiden.

Das ultimative automatisierte Verfahren testet alle[2] nach bestimmten Regeln aufgebauten Faltungsnetze systematisch durch, um am Ende ein Netz mit der besten Performanz zu bestimmen – dessen Architektur wird im Rahmen dieser Arbeit als eine für das gegebene Dataset „**optimale**“ Architektur definiert.

Der Forschungsgegenstand, eine optimale Architektur zu entwerfen, kann als ein Suchproblem in einem hochdimensionalen Raum formuliert werden. Dabei wird jede denkbare Netzarchitektur durch einen Punkt in diesem Raum repräsentiert. Wenn das Kriterium für die optimale Architektur vorgegeben ist (wie z.B. die höchste Accuracy), dann kann die Performanz einer Architektur als die Funktion aller Hyperparameter ausgedrückt werden. Der entsprechende Funktionsgraph bildet eine diskrete Oberfläche im Suchraum. Das Finden einer optimalen Netzarchitektur ist also äquivalent zum Finden des höchsten Punktes in diesem „Gebirge“, dessen Eigenschaften sind [36]:

- er ist unter Umständen unendlich groß, da die Anzahl der Neuronenschichten theoretisch unbegrenzt ist
- er ist komplex, mit Rauschen (verursacht durch unbekannte Abhängigkeiten von Daten oder von Trainingsparametern)
- die Zielfunktion ist nicht stetig (ähnliche, naheliegende Netze liefern plötzlich sehr unterschiedliche Ergebnisse) und nicht differenzierbar, da die Änderung solcher Hyperparameter wie Filtergröße oder Layertyp diskret ist und sich nicht zwangsläufig kontinuierlich auf die Performanz auswirkt (Zacken im Gebirge).

In solchen Situationen ist es sehr schwierig, die Auswirkung dieser Parameter auf die Fehlerrate analytisch zu beschreiben, daher muss auf Werkzeuge aus der Optimierungstheorie zurückgegriffen werden. Denn ein Suchproblem mit einer gegebenen Funktion, die Objekte aus dem Suchraum bewertet, um ein Objekt mit maximalem bzw. minimalem Wert zu berechnen, ist gleichzeitig auch ein kombinatorisches Optimierungsproblem, wenn die Grundmenge der Objekte endlich oder abzählbar ist [37]. Da die Hyperparameter der Faltungsnetzarchitektur entweder endliche Werte annehmen (Layertyp) oder als natürliche Zahlen repräsentierbar sind (Filtergröße), ist dies also in diesem Fall gegeben.

Kombinatorische Optimierungsprobleme werden in der Wissenschaft mithilfe exakter oder heuristischer Verfahren gelöst. Exakte Verfahren garantieren eine optimale Lösung innerhalb beschränkter Zeit. Dennoch steigt ihre Rechenkomplexität meist exponentiell, wenn die Problemgröße bzw. Anzahl der Variablen sich erhöht. Aus diesem Grund werden die heuristischen Verfahren den exakten sogar vorgezogen, obwohl sie innerhalb einer absehbaren Zeit lediglich eine sehr gute, nicht unbedingt optimale Lösung berechnen

[2] Vorausgesetzt, die Anzahl solcher Netze ist endlich.

können. Sie finden eine passende Lösung, indem sie das problemspezifische Wissen über den Suchraum anwenden.

Während für manche Problemstellungen längst eine Reihe von Heuristiken existieren, die sie zufriedenstellend lösen (wie z.B. für das Problem des Handlungsreisenden [38]), können sie im Allgemeinen nicht auf andersartige Probleme übertragen werden. Gleichzeitig kann es vorkommen, dass zu viele unterschiedliche, zum Teil nicht theoretisch begründete Erkenntnisse bzw. vielmehr Intuitionen über ein bestimmtes Problem vorliegen. Demzufolge wäre es extrem schwierig, ein spezielles heuristisches Verfahren zu entwerfen, das das komplexe Wissen vereint und noch in der Lage ist, gute Lösungen zu produzieren.

Angesichts dieser Situation besteht die Möglichkeit, das Suchproblem für Faltungsnetzarchitekturen zu lösen, indem ein heuristisches Verfahren gefunden wird, das so allgemein wie möglich funktioniert, sich aber auf diese spezifische Problemstellung übertragen lässt.

Ein Versuch, viele Optimierungsprobleme auf die gleiche Weise zu lösen, stellen die **Metaheuristiken** dar. Eine Metaheuristik ist ein Algorithmus, der dazu entworfen ist, für eine breite Klasse der Optimierungsprobleme eine approximative Lösung zu finden, ohne sich an jede konkrete Problemstellung anpassen zu müssen [39]. Dadurch kann es als ein Verfahren höherer Ordnung angesehen werden, im Gegensatz zu den problemspezifischen Heuristiken, und könnte theoretisch für jedes kombinatorische Optimierungsproblem angewendet werden [40].

Dies macht sie besonders attraktiv in den Bereichen, wo noch keine ausreichend guten problemspezifischen Heuristiken existieren, aber der Suchraum zu groß für die vollständige Suche innerhalb einer sinnvollen Zeit ist [39]. Nichtsdestotrotz sind Metaheuristiken im Allgemeinen sehr rechenintensiv und haben ihre weite Verbreitung (ähnlich wie Faltungsnetze) der gestiegenen Rechenleistung zu verdanken.

Typische Beispiele dafür sind der Ameisenalgorithmus, Genetische und Evolutionäre Algorithmen, Iterative Lokale Suche, Simulated Annealing und Tabu-Suche. Einige Metaheuristiken wurden durch natürliche Prozesse (Schwarmverhalten, Evolution, physikalische Vorgänge) inspiriert, andere basieren auf einfacheren Heuristiken wie dem Greedy-Algorithmus oder Lokaler Suche [38].

Ein Überblick über verschiedene Metaheuristiken kann der Studie [39] entnommen werden. Die Autoren unterteilen diese in 2 Kategorien: Die einen basieren auf einer einzigen Lösung, die anderen auf einer Population von Lösungen. Beide Kategorien versuchen, eine Balance zwischen Diversifikation und Intensivierung zu finden. *Diversifikation* bedeutet das Identifizieren der Bereiche mit sehr guten bzw. nahezu optimalen Lösungen innerhalb des Suchraumes. Bei der *Intensivierung* werden die ausgewählten Bereiche intensiv nach Lösungskandidaten durchsucht, ohne die Zeit auf weniger vielversprechende oder bereits durchsuchte Bereiche zu verschwenden. Metaheuristiken, die auf einer einzigen Lösung

basieren, sind eher auf Intensivierung ausgerichtet, währenddessen die populationsbasierten Verfahren sich mehr für Diversifikation eignen.

Bei der Wahl der Metaheuristik für eine konkrete Probleminstanz, sei es Hyperparametersuche für MLP oder Vehicle Routing, wird meistens auf den „No-Free-Lunch“-Satz verwiesen [39]. Dieser besagt, dass alle allgemein anwendbaren Lösungsalgorithmen die gleiche durchschnittliche Performanz haben, gemessen über alle denkbaren Optimierungsprobleme. Das bedeutet, dass kein Verfahren grundsätzlich besser ist als seine Mitbewerber. Trotzdem kann es sein, dass bestimmte Metaheuristiken einige spezifische Problemklassen mit Erfolg lösen können, liefern bei den anderen jedoch eher unzureichende Ergebnisse [40]. Es erklärt sich unter anderem dadurch, dass die Metaheuristiken selbst einige Parameter enthalten (in dieser Arbeit **Metaparameter** genannt), die ggf. für jede Probleminstanz optimiert werden müssen. Schließlich müssen mehrere, sich gut bewährte Metaheuristiken ausprobiert und entsprechend parametrisiert werden, um den besten Lösungsansatz für eine konkrete Probleminstanz zu bestimmen.

Die Auswahl der Hyperparameter ist das grundsätzliche Problem für viele Optimierungsverfahren, also auch für Verfahren im Bereich Machine Learning. Bei einfacheren Netzwerkklassen wie MLP wurde das beschriebene Suchproblem durch Metaheuristiken bereits erfolgreich gelöst. Am weitesten verbreitet ist darunter der Genetische Algorithmus, siehe [41], [42], [43], [44], gefolgt von Particle Swarm Optimization [43], Simulated Annealing ([45], [46], [47]) und dem Elektromagnetischen Algorithmus [48] sowie Tabusuche.

Bisherige Versuche, ein systematisches Vorgehen bei der Suche nach den besten Architekturen in anderen, komplexeren Netzwerkklassen zu entwickeln, wurden in [49], [50], [51] und [52] beschrieben. Getestet wurden diese Verfahren am Beispiel von Deep Belief Networks (**DBN**) mit ein bis drei verdeckten Schichten und Support Vector Machines (**SVM**), die typischerweise nur zwei Hyperparameter besitzen, sowie auf Standard-MLP (für Definitionen siehe Glossar, Kapitel 8).

Die Motivation dabei war die Tatsache, dass Fortschritte im Bereich Objektklassifizierung ebenso durch Architekturverbesserungen stattfinden können, anstatt nur durch innovative Deep-Learning-Modelle oder Trainingsstrategien [50]. Nach Ansicht der Forscher ist das Hyperparametertuning vielmehr „manuelle Kunst“ als Wissenschaft, sollte aber zur Entwicklung eines Models dazugehören, und zwar in Form von einer automatisierten Außenschleife. Deswegen haben sie Metaheuristiken wie Grid-Suche und Random-Suche als Alternative zur händischen Experten-Suche vorgeschlagen.

Wenn die gewünschten Hyperparameter einer Architektur bekannt sind, können alle denkbaren Kombinationen ihrer Werte generiert werden. Die **Grid-Suche** [49] reduziert diesen Suchraum deutlich, indem sie nur einige, stichprobenartige Werte des jeweiligen Hyperparameters auswählt. Der Abstand zwischen den Werten pro Parameter kann konstant oder logarithmisch sein. Alle Kombinationen der ausgewählten Werte entsprechen also den

Punkten im Suchraum, die eine Art Gitter (bestehend aus Hyperebenen) bilden. Anschließend werden diese Punkte als Lösungskandidaten betrachtet und bewertet, um das beste Ergebnis zu ermitteln. Der Vorteil der Grid-Suche ist ihre einfache Implementierung und leichte Parallelisierung der Trainingsprozesse.

Im Gegensatz dazu findet die **Random-Suche** nur einige zufällige Kombinationen bzw. Punkte im Suchraum und trainiert die entsprechenden Netzarchitekturen. Die zufälligen Punkte können unter Umständen näher an die besten Netze herankommen als Gitter-Punkte, auch wenn man den Abstand zwischen den Gitter-Hyperebenen verringert. Tatsächlich wurde in Experimenten von [51] festgestellt, dass die Random-Suche viel weniger Kandidaten benötigt als Grid-Suche, um ein bestimmtes Niveau von Performanz zu erreichen. Der Nachteil ist hier allerdings, dass die Wahrscheinlichkeit, auf eine sehr gute Lösung zufällig zu stoßen, mit steigender Raumdimensionalität sinkt. Außerdem ist die Random-Suche nicht im Stande, eine gewisse Richtung anzunehmen oder bestimmte Bereiche selektiv zu untersuchen, um zu ermitteln, ob es in diesen Bereichen weitere vielversprechende Lösungen gibt. Sie kann also, genau wie die Grid-Suche, die besten Lösungen theoretisch einfach überspringen.

Was die Faltungsnetze betrifft, haben sie eine spezifische Struktur, sind meistens viel tiefer als DBN und weisen viel mehr Hyperparameter auf als MLP oder SVM. Außerdem beinhaltet das Suchproblem im Fall von Faltungsnetzen einige Nebenbedingungen, was bei MLP nicht zutrifft. Der Grund dafür ist, dass nicht jede beliebige Faltungsnetzarchitektur sinnvoll ist. So wird ein Faltungsnetz, wo z.B. die Schrittgröße in den Filter Bank Layers immer die Filtergröße übersteigt oder mehrere Pooling Layers aufeinanderfolgen, zu viel Bildinformation verlieren und schlechte Erkennungsraten liefern. Deswegen sollen bei Aufbau oder Änderung einer Architektur bestimmte Regeln eingehalten werden, um solche „invalide“ Netze vom Training auszuschließen.

Es wurden über die Jahre nur wenige Lösungsverfahren zur Hyperparametersuche bei Faltungsnetzen veröffentlicht. Beispiele sind die Grid-Suche ([29], [49]), Random-Suche [51] und Bayes’sche Optimierung ([50], [52]). Die letztere konstruiert ein probabilistisches Modell der Zielfunktion auf Basis der bisher ermittelten Werte sowie der Annahme über ihre A-priori-Verteilung. Anschließend verwendet das Modell iterativ eine gewisse Akquirierungsfunktion, um den nächsten interessanten Punkt im Suchraum zu berechnen. Bayes’sche Optimierung kann auf verschiedene Weise implementiert werden, für den Vergleich ihrer Ausprägungen mittels MNIST, CIFAR-10 und CIFAR-100 siehe [53].

Des Weiteren haben die Forscher in [53] ein Verfahren vorgeschlagen, das die Hyperparametersuche beschleunigen kann, indem es bereits während der frühen Trainingsiterationen die Lernkurve eines Netzes vorhersagt. Dies hilft, eine frühzeitige Entscheidung zu treffen, ob es sich lohnt, das gegebene Faltungsnetz weiter zu trainieren. Wenn das gewünschte Performanzniveau mit hoher Wahrscheinlichkeit nicht erreicht wird, dann soll die Suche lieber mit dem nächsten Netz fortgesetzt werden. Das Verfahren hat die Laufzeiten von einer Bayes’schen Optimierung um mehr als die Hälfte verringert.

Trotz dieser Erfolge gaben die Autoren von [51] zu, dass die Suche nach Lösungsansätzen für Hyperparametertuning von Faltungsnetzarchitekturen bei Weitem nicht abgeschlossen ist. Es bedarf also die Anwendung anderer Verfahren, unter anderem Metaheuristiken, die sich für solche speziellen Netzarchitekturen eignen, mit vielen Variablen und Nebenbedingungen klarkommen und den Suchraum auf eine systematische Weise erforschen. Diese könnten in der Lage sein, das Suchproblem eventuell effizienter zu lösen als die Grid-Suche und ggf. bessere Netzarchitekturen zu entdecken.

Dementsprechend basiert der im Rahmen dieser Arbeit entwickelte Lösungsansatz auf der Annahme, dass bestimmte Metaheuristiken auch für komplexere Architekturen der Faltungsnetze anwendbar sind. Eine weitere Annahme besteht darin, dass es ausreichend ist, alle Netzarchitekturen mit zufälligen Filtern zu erzeugen und nur den Klassifikator zu trainieren, um zu ermitteln, welche dieser Netze eine bessere Leistung erbringt (siehe Kapitel 2.2 und [13]). Somit sollten sich die Trainingszeiten sowie die gesamte Laufzeit dieser Algorithmen erwartungsgemäß verkürzen.

Im Folgenden werden einige Metaheuristiken beschrieben, die bereits zur Suche nach MLP-Architekturen benutzt wurden und daher auch für Faltungsnetze infrage kommen können. Diese wurden in Hinsicht auf Diversität ausgewählt (siehe Tabelle 2), da noch viele weitere gleichartige Verfahren existieren. Einer der vorgestellten Ansätze basiert auf Konzepten der Evolutionären Algorithmen, der andere auf Lokaler Suche, der dritte ist ein Hybrid aus beidem. Alle drei sind relativ einfach zu implementieren, sehr gut erforscht und verfügen über Mechanismen, sich aus lokalen Optima zu befreien. Ihre Eigenschaften und Komplexität werden im weiteren Verlauf erklärt.

	Lösungen pro Iteration	Komplexität	Diversifikation	Intensivierung	# spezif. Meta-parameter
Random-Suche (RS)	Eine Lösung	N	Ja	Nein	0
GA	Population	$M \times N$	Ja	Ja	4
SA	Eine Lösung	N	Am Anfang	Ja	5
MA	Population	$M^2 \times N$	Ja	Ja	6

Tabelle 2. Rechenkomplexität der Lösungsansätze (N - Anzahl der Iterationen, M - Populationsgröße).

3.1 Genetischer Algorithmus

Der Genetische Algorithmus (kurz **GA**) wurde in den 70er Jahren vorgestellt [39] und ist seitdem die wohl bekannteste und am meisten verwendete Metaheuristik aus der Menge der Evolutionären Algorithmen. Sie ist populationsbasiert und versucht, Prozesse der natürlichen Evolution zu replizieren [43]. Demnach haben die Individuen mit einer höheren Anpassungsfähigkeit (engl. „fitness") mehr Chancen, zu überleben und sich zu reproduzieren, sodass die vorteilhaften Eigenschaften zu den Nachkommen übergehen, unvorteilhafte eher verschwinden, und die Verbesserung der Population als Ganzes stattfinden kann. Diese Tatsache wurde im sogenannten „Schemasatz" bewiesen, wodurch die Konvergenz des Verfahrens theoretisch begründet ist [54].

Ab Anfang der 90er Jahre wurden viele Beiträge veröffentlicht, die sich mit der Anwendung von GA für die Suche nach optimalen MLP-Architekturen beschäftigen [55]. Vorher wurde dieses Suchproblem durch konstruktive oder destruktive Algorithmen gelöst, die die Netzstruktur (Schichten, Neuronen und Verbindungen) dynamisch veränderten, bis die Performanz nicht mehr gestiegen war. Als allerdings klar wurde, dass solche Algorithmen schnell in lokalen Optima stecken bleiben können, kamen die GA zum Einsatz [42].

In allgemeiner Form funktioniert der GA folgendermaßen. Die Individuen bzw. Lösungskandidaten, in diesem Fall Netzarchitekturen, werden mithilfe binärer Strings repräsentiert. Ein Wert jedes Hyperparameters wird durch ein oder mehrere sogenannte Gene ausgedrückt, sodass pro Architektur ein *Chromosom* aus Genen entsteht. Eine Menge von Chromosomen bildet die aktuelle Population. In jeder Iteration des GA wird die Population geändert, indem bestimmte *genetische Operatoren* auf diese Chromosomen angewendet werden – das sind Selektion, Crossover und Mutation. Zur Messung der Qualität eines Netzes und somit seiner Eignung für die Weiterentwicklung dient die sogenannte *Fitnessfunktion*, die von Trainingsfehler, Testfehler [54] und ggf. Netzgröße [44] abhängen kann und im Laufe von GA maximiert wird.

Der Algorithmus selbst besteht aus mehreren Schritten [41]:

(1) Erste Population mit M zufälligen Individuen erzeugen.

(2) Jedes Individuum mithilfe der Fitnessfunktion evaluieren.

(3) *Selektion*: Jedes Individuum wird je nach seiner Qualität mit einer bestimmten Wahrscheinlichkeit ausgewählt und in die nächste Population übernommen. Restliche Individuen werden verworfen.

(4) *Crossover*: Fittere Individuen aus der neuen Population haben Vorrang, als Eltern ausgewählt zu werden (mit Wahrscheinlichkeit p_c^i) und ihre Gene weiterzugeben. Jeweils zwei solche Individuen produzieren Nachwuchs, bis die Populationsgröße M erreicht ist. Der Nachwuchs bekommt je einen Teil der beiden Eltern-Chromosomen.

(5) *Mutation*: Mit Wahrscheinlichkeit p_m wird ein zufälliges Gen bei mehreren Individuen der Population geändert.

(6) Wiederhole (2) – (5), bis die maximale Anzahl der Iterationen N erreicht oder eine hinreichend gute Lösung gefunden wird.

Die genetischen Operatoren sollen dazu in der Lage sein, solche Eigenschaften wie Diversifikation und Intensivierung zu erfüllen, damit eine globale Suche Erfolg hat. So beabsichtigt Selektion, die durchschnittliche Qualität der Population zu erhöhen und die Suche in Richtung der vielversprechendsten Lösungen fortzusetzen. Crossover basiert auf der Annahme, dass der Nachwuchs der fitteren Elternchromosomen mit höherer Wahrscheinlichkeit auch fitter wird [55], bringt Variation in die Population und sorgt für schnellere Konvergenz. Zufällige Mutationen sollen die Gefahr vom lokalen Optimum ebenfalls vermeiden. Wenn die Population zu homogen wird, dann sinkt die Effektivität des Crossovers, und Mutation bleibt als einziges Werkzeug, um den Suchraum zu erkunden und neues Genmaterial zu beschaffen [56].

Jedoch können diese Operatoren auf unterschiedliche Art und Weise definiert werden, je nachdem, welche Variante erwartungsgemäß zu der Problemstellung besser passt:

- Selektion von K Individuen aus M:
  - Die einfachste Variante der Selektion ist es, die M Chromosomen der Population nach ihrer Fitness zu sortieren und die K besten auszuwählen [57]. Dies entspricht dem Elitismus-Prinzip [43], der garantiert, dass die fittesten Individuen unverändert weiterkommen, damit die beste Lösungsqualität in der neuen Generation mindestens genauso gut ist wie in der alten. Der Nachteil dieser Selektionsmethode besteht darin, dass die Verteilung der Fitnesswerte nicht berücksichtigt wird.
  - Selektion nach dem Roulette-Prinzip [41]: Jedes Individuum wird mit einer Wahrscheinlichkeit ausgewählt, die proportional zu seinem relativen Fitnesswert ist. Hier haben auch schlechte Lösungskandidaten eine Chance zum Überleben. Allerdings kann es je nach Verteilung von Fitnesswerten passieren, dass ihre Diversität in der nächsten Population verloren geht, z.B. wenn wenige gleich fitte Individuen dominant sind.
  - Rangselektion [39] sortiert Individuen zunächst nach Fitness, um ihnen dann einen Rang zuzuweisen: Rang 1 entspricht dem schwächsten, M – dem fittesten Lösungskandidat. Die Selektionswahrscheinlichkeit ist vom Rang linear abhängig und ist demzufolge nie für zwei Individuen gleich. Dadurch werden wenige sehr fitte Individuen nicht so dominant wie beim Roulette-Prinzip.
  - Turnier-Selektion [36] wählt q zufällige Individuen, die in einem Turnier teilnehmen. D.h. sie werden paarweise miteinander verglichen, und jedes Mal bekommt das fittere Individuum von beiden einen Punkt. Der Gewinner des

Turniers wird fest in die nächste Generation übernommen. Nachfolgend werden weitere Turniere mit zufälligen Teilnehmern durchgeführt, bis K erreicht ist. Diese Art der Selektion ist weniger verbreitet, da sie nicht verspricht, dass das fitteste Individuum der Population überlebt.

- Crossover erzeugt ein oder zwei Nachkommen für jedes Chromosomenpaar [39]:
  - n-Punkte-Crossover macht jeweils n gemeinsame zufällige Schnitte bei beiden Elternchromosomen und erzeugt zwei Nachkommen, die abwechselnd aus diesen Teilstücken bestehen.
  - Uniformes Crossover geht jedes Gen eines Kindchromosoms durch und entscheidet per Zufall, von welchem Elternchromosom das Gen an dieser Position übernommen wird.
- Mutation [55]:
  - Die Standard-Variante der Mutation sieht vor, ein Gen an beliebiger Position eines gewählten Chromosoms zu ändern.
  - Andere Varianten ändern eine bestimmte Anzahl an Genen pro Chromosom, die mit dem „Alter“ der Population sinkt.
  - Beim Elitismus sollen die fittesten Individuen von der Mutation ausgeschlossen werden.

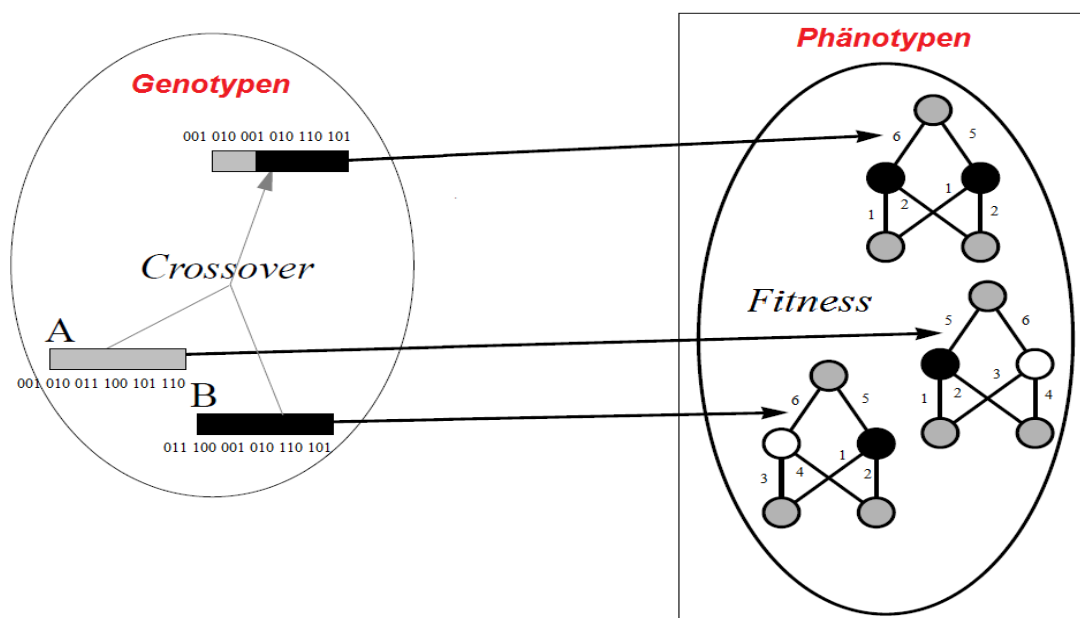

Abbildung 7. Genotyp-Phänotyp-Abbildung (nach [57]).

Es ist außerdem wichtig zu erwähnen, dass der GA mit zweierlei Daten arbeitet [57], siehe Abbildung 7. Einerseits sind es die tatsächlichen Netzarchitekturen (*Phänotypen*) samt allen

trainierbaren Parametern, die nach dem Training mithilfe der Fitnessfunktion bewertet werden müssen. Andererseits agieren die genetischen Operatoren mit den binären Repräsentationen der Netze (*Genotypen*). Folglich muss der Suchraum mit Phänotypen auf einen Raum mit entsprechenden Genotypen abgebildet werden, und umgekehrt.

Um eine optimale Architektur von MLP für ein konkretes Dataset zu entwerfen, wurden die GA auf zwei Weisen eingesetzt. Zum einen haben sie lediglich die Netzstruktur ermittelt, also z.B. die Anzahl der verdeckten Schichten und der jeweiligen Neuronen sowie die Art der Aktivierungsfunktion. Die Neuronengewichte und Trainingsparameter wurden dennoch immer zufällig initialisiert, was zu nicht eindeutigen Werten der Fitnessfunktion führte. So entsteht das Problem der $1{:}N$ Abbildung vom Genotyp zum Phänotyp [42]. Deswegen werden die GA auf eine zweite Weise benutzt – für die Berechnung der optimalen Netzarchitektur sowie der Gewichte. Hierbei kommt es jedoch zum Problem der $N{:}1$ Abbildung vom Genotyp zum Phänotyp, und zwar dann, wenn zwei Netze funktional identisch sind, haben aber zwei verschiedene Repräsentationen (A und B in Abbildung 7). Sollten sich die Netze kreuzen, bekommen ihre Nachkommen eine redundante Struktur und sind deswegen weniger performant. Weitere Folgen dieses Problems sind Reduktion der Populationsdiversität und frühzeitige Konvergenz, was den GA letztendlich ineffizienter macht. Bei der Bestimmung der Genotyp-Phänotyp-Abbildung müssen beide Probleme also bedacht werden [54].

Die Herausforderungen des GA sind folglich die Wahl der passenden genetischen Operatoren und der Fitnessfunktion, ihrer spezifischen Metaparameter sowie der Genotyp-Phänotyp-Abbildung. Trotzdem hat ein GA eine Reihe von Vorteilen [36], die ihn zu einem sehr beliebten Werkzeug zur Architektursuche machen. Zum einen ist der Algorithmus potentiell in der Lage, die globale Suche im Hyperparameterraum durchzuführen und lokale Optima zu vermeiden. Er garantiert die sequentielle Verbesserung der Lösungsqualität, ohne problemspezifische Informationen zu besitzen – vorausgesetzt, diese Informationen wurden in der Genotyp-Phänotyp-Abbildung korrekt einbezogen [57]. Zum anderen setzt der GA nicht voraus, dass die Fitnessfunktion differenzierbar oder stetig sein muss, und ist von der Gradienteninformation unabhängig. Außerdem können Netzwerke mit bevorzugten Charakteristiken generiert werden, wenn diese (z.B. Netzkomplexität oder Testfehler) als ein zusätzlicher Term in der Fitnessfunktion ausgedrückt werden können. Letztendlich sind die GA weniger sensitiv in Bezug auf Initialisierung als andere Verfahren wie Simulated Annealing oder Gradientenverfahren, da sie statt einen einzigen initialen Lösungskandidaten gleich sehr viele betrachten.

3.2 Simulated Annealing

Simulated Annealing (kurz **SA**, dt. „simulierte Abkühlung“) hat seinen Ursprung in den 80ern, als es durch den physikalischen Prozess der Abkühlung des Metalls inspiriert wurde. Und zwar wird in der Metallurgie eine Abkühlungsmethode angewendet, bei der das Metall zuerst

erhitzt, dann aber langsam abgekühlt wird, sodass es in einen stabilen, festen Zustand mit global minimaler Energie übergehen kann [39]. Demzufolge versucht der SA, diesen Prozess auf die Lösung einer Optimierungsaufgabe zu übertragen, indem ein einziger kontrollierbarer Parameter eingeführt wird – die Temperatur T.

Das Verfahren stützt auf den Prinzipien der Lokalen Suche und macht sich die prädefinierte Nachbarschaftsstruktur des Suchraumes zu Nutze [38]. Es hält pro Iteration eine einzige Lösung (anstatt eine Population an Lösungen) und versucht, diese allmählich durch die Suche in der Nachbarschaft zu verbessern. Ein schlechterer Lösungskandidat, der im Laufe des Verfahrens produziert wird, kann mit einer bestimmten Wahrscheinlichkeit trotzdem als Lösung akzeptiert werden, wenngleich diese Wahrscheinlichkeit proportional zur Temperatur sinkt. Das Ziel von SA ist es, eine gegebene Ziel- bzw. Fitnessfunktion f zu maximieren.

Im Wesentlichen kann SA wie folgt beschrieben werden [47]:

(1) Initiale Lösung und Temperatur festlegen.
(2) Laufende Lösung s mithilfe von f evaluieren.
(3) Einen zufälligen Lösungskandidaten s' aus der Nachbarschaft von s auswählen und evaluieren.
(4) *Akzeptanzkriterium*: Wenn s' besser als s ist, dann die laufende Lösung durch s' ersetzen; wenn nicht, dann ersetzen mit der Wahrscheinlichkeit $P(T, f(s), f(s'))$.
(5) *Abkühlungszeitplan*: Temperatur senken.
(6) Wiederhole (2) – (5), bis die maximale Anzahl der Iterationen N erreicht oder eine hinreichend gute Lösung gefunden wird.

Zusätzlich kann die beste bisher gefundene Lösung immer mitgespeichert werden, sonst geht sie durch das Akzeptanzkriterium möglicherweise verloren. Darüber hinaus empfiehlt es sich, in späteren Iterationen eine Liste der abgelehnten Nachbarn zu pflegen [58], denn die meisten Nachbarn werden in dieser Phase schlechter als die aktuelle Lösung und haben kaum eine Chance, akzeptiert zu werden. Eine zufällige Auswahl der Nachbarn würde also die Gefahr erhöhen, dass derselbe Lösungskandidat mehrfach vergeblich evaluiert wird.

Offenbar enthält SA einige Funktionen, die noch definiert werden müssen – Abkühlungszeitplan samt initialer Temperatur und die Wahrscheinlichkeit im Akzeptanzkriterium. Es gibt jedoch keine theoretisch fundierten Richtlinien, um diese auszuwählen, daher sollten sie immer in Bezug auf die konkrete Problemstellung empirisch ermittelt werden.

Zur Berechnung der Akzeptanzwahrscheinlichkeit für den Fall der schlechteren Lösung ($f(s') < f(s)$) wird meistens folgende Gleichung benutzt [46]:

$$P\big(T, f(s), f(s')\big) = e^{\frac{f(s') - f(s)}{T}}$$

Dies bedeutet, dass eine kleinere Abweichung in der Performanz eher akzeptiert wird als eine große, und zwar am Anfang des SA eher als am Ende. In einigen Fällen gelingt es den Wissenschaftlern, diese Wahrscheinlichkeit nach anderen, komplexeren Formeln zu berechnen (siehe z.B. [45]), allerdings wird die Wiederverwendbarkeit dieser Ergebnisse für andere Optimierungsprobleme infrage gestellt.

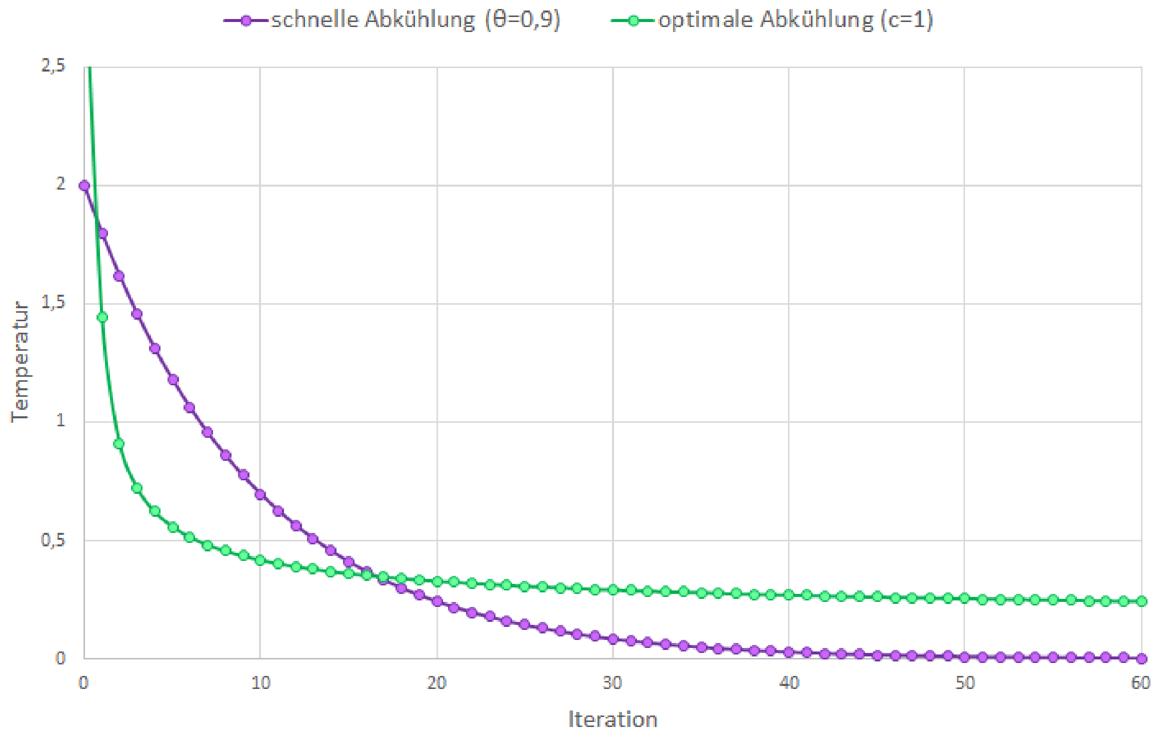

Abbildung 8. Abkühlungsplan von SA.

Was den Abkühlungsplan betrifft, wurde es bereits bewiesen, dass er und die Suchraumtopologie entscheidend für die Konvergenz des SA sind [58]. Bei einer sehr langsamen Abkühlung der Art $T_i = {}^{c}/_{\log(1+i)}$ konvergiert SA zum globalen Optimum genau dann, wenn c größer oder gleich ist als das zweitbeste lokale Optimum. Im Fall von Netzarchitekturen mit einer Zielfunktion, die die Erkennungsrate des Netzes misst, wäre ein möglicher Wert $c = 1$. Es gibt hingegen nur approximative Schätzungen, wie viele Iterationen für die Konvergenz zum globalen Optimum nötig sind, und zwar mindestens so viele wie die Größe des Suchraumes. Das gilt auch für die abweichenden Akzeptanzkriterien, die nach [58] keine nachweisbaren signifikanten Verbesserungen der Lösungsqualität bewirkten. Eine schnellere Abkühlung nach der ursprünglichen Formel kann das globale Optimum zwar nicht garantieren, dafür aber die Laufzeit deutlich verkürzen [58], vgl. Abbildung 8:

$$T_i = (\theta^i) \cdot T_0, \text{ wobei } 0{,}8 \leq \theta \leq 0{,}99$$

Die Temperatur soll letztendlich das Suchverhalten kontrollieren, indem es die Sprünge aus dem lokalen Optimum zu schlechteren Lösungen erstmals zulässt. Somit eignet sich SA am Anfang der Suche für Diversifikation und insgesamt für Intensivierung, die durch den lokalen Charakter der Suche gegeben ist [38].

Im Vergleich zu den anderen Metaheuristiken hat SA einige Vorzüge. Einerseits enthält er keine spezifischen Operatoren, wie z.B. bei GA, und ist sehr einfach zu implementieren. Dadurch kann das Verfahren für diejenigen Optimierungsprobleme angewendet werden, wo der Crossover nicht sinnvoll definiert werden kann oder per se schlechtere Lösungskandidaten generiert [36]. Andererseits hängt seine Zeitkomplexität linear von der Iterationszahl ab, was weniger Aufwand für die Evaluierung der möglichen Lösungen pro Iteration bedeutet. Schließlich ist SA besonders erfolgreich, wenn die Fitnessfunktion im Suchraum eine „glattere" Topologie aufweist bzw. wenn die lokalen Optima nicht überall gestreut sind [58].

3.3 Memetischer Algorithmus

Für einige Probleminstanzen wurde festgestellt, dass Evolutionäre Algorithmen wie GA für feingranulare Suche nicht wirklich gut geeignet sind [59]. Obwohl sie gut in Bezug auf Diversifikation funktionieren, ist ihr Mechanismus, die Suchraumbereiche mit hochqualitativen Lösungen effektiv zu durchsuchen, eher mangelhaft. Im Gegensatz zu globaler Suche, die in GA und anderen Metaheuristiken stattfindet, ist **Lokale Suche** darauf ausgelegt, in der Nähe von einer initialen Lösung schnell eine bessere zu finden. Daraus folgten viele Versuche zur Hybridisation des GA mit anderen Heuristiken, die Lokale Suche in irgendeiner Form enthalten und dadurch die Schwächen von GA bezüglich der Intensivierung eliminieren sollten [40]. Heutzutage sind solche Hybriden weit verbreitet und in vielen Domänen zunehmend konkurrenzfähig [39].

Um die besten Eigenschaften der GA mit Lokaler Suche zu kombinieren, wurden memetische Algorithmen (**MA**) entwickelt, auch hybride GA genannt. Der Name stammt vom englischen Begriff „meme", d.h. eine Informationseinheit, die sich durch den Ideenaustausch zwischen Menschen verbreitet [60]. Als Inspiration hierzu dienten die Modelle der Anpassung in natürlichen Systemen, wo nicht nur evolutionäre Anpassung, sondern auch lebenslanges Lernen jedes Individuums eine wichtige Rolle spielt [59].

MA haben die gleichen Schlüsselkomponenten wie GA, nur alle Chromosomen werden nach der Anwendung von genetischen Operatoren zusätzlich mithilfe Lokaler Suche verbessert [61]. Demzufolge besteht jede Population ausschließlich aus lokalen Optima, die durch den strategischen Austausch mit der Nachbarschaft entstanden sind. Das Verfahren wurde bereits für viele Problemstellungen mit Erfolg verwendet [62], unter anderem für das

Problem des Handlungsreisenden, Vehicle Routing, das Stundenplanproblem und sogar für die Merkmalselektion von SVM.

Die Grundidee von MA ist folgende [61]:

(1) Erste Population mit M zufälligen Individuen generieren.
(2) Jedes Individuum mithilfe der Fitnessfunktion evaluieren.
(3) *Lokale Suche*: Eine Untermenge der Population auswählen und jedes Chromosom darin durch das beste Ergebnis der jeweiligen Lokalen Suche ersetzen. Dafür müssen mehrere Nachbarn des Chromosoms evaluiert werden.
(4) *Selektion, Crossover, Mutation* – wie bei GA.
(5) Wiederhole (2) – (4), bis die maximale Anzahl der Iterationen N erreicht oder eine hinreichend gute Lösung gefunden wird.

Lokale Suche im Schritt (3) wird meistens basierend auf domänenspezifischem Wissen durchgeführt und kann grob in zwei Kategorien unterteilt werden [61]: die Suche bis zur ersten Verbesserung und Greedy-Suche, die durch alle Nachbarn iteriert. Demnach steigt die Anzahl der nötigen Evaluierungen im schlimmsten Fall quadratisch mit wachsender Populationsgröße M. Aufgrund dieser Tatsache besteht die Population meistens aus wenigen, dafür aber erwartungsgemäß hochwertigen Individuen im Gegensatz zu GA, typischerweise im Bereich von 10 bis 40 [60]. Die Kehrseite dieser Reduktion ist eine mögliche vorzeitige Konvergenz des Verfahrens, der wiederum durch das dynamische Erhöhen der Mutationsrate entgegengesteuert werden kann.

Trotz der Rechenkomplexität können viele Problemstellungen von MA profitieren. Erstens, enthält der Algorithmus nur zwei Metaparameter mehr als GA, und zwar die Anzahl der Nachbarn und den Nachbarschaftsradius für Schritt (3). Zweitens, dank Lokaler Suche handelt es sich in jeder Iteration um bessere Lösungen, was letztendlich zu kürzerer Laufzeit und schnellerer Konvergenz führen soll. Drittens, sei die Erweiterung des klassischen GA um Lokale Suche besonderes dort vorteilhaft, wo der Verlauf der Fitnessfunktion im Suchraum sehr komplex ist [59], was für Faltungsnetzarchitekturen erwartungsgemäß zutreffen müsste.

4 Implementierung

Um die in Kapitel 3 genannten drei Lösungsansätze zur Architektursuche zu implementieren, bedarf es ein Werkzeug, das stets auf dem neustem Stand bzgl. Layertypen, Trainingsparameter und optimierten Laufzeiten ist. Des Weiteren sollte das Werkzeug über verschiedene Schnittstellen verfügen, die die Erzeugung von diversen Architekturen programmiertechnisch erleichtern, sowie die Möglichkeit bieten, das Training nur im Klassifikator des Faltungsnetzes durchführen zu können.

Für diese Zwecke eignete sich das Tool „Caffe“ [63], ein in C++ implementiertes Open-Source-Framework für Machine Learning. Hierbei handelt es sich um eines der am weitesten verbreiteten Tools zum Trainieren von Faltungsnetzen, das sowohl in der Forschung als auch in der Industrie sehr beliebt ist. „Caffe“ hat Schnittstellen zu Python und MATLAB, läuft auf diversen Betriebssystemen, hat gute Geschwindigkeiten beim Training auf CUDA-fähigen Grafikkarten und kann Bilderdaten in verschiedenen Formaten verarbeiten. Zusätzlich bietet es alle „State-of-the-Art“-Trainingsstrategien, Layertypen und Netzparameter. Es ermöglicht außerdem, den bestehenden Code zu erweitern, um neue Layer oder Trainingsmethoden zu entwickeln.

Die endgültige Arbeitsumgebung ist eine Workstation mit CPU Intel® Core™ i7-6700K mit 4 Kerne / 8 Threads (4.00 GHz), sowie 16 GB RAM, Grafikkarte NVIDIA GeForce 1080 mit 8 GB VRAM und dem Betriebssystem Ubuntu 14.04.. Darauf sind „Caffe“, Python 2.7. und die entsprechenden Abhängigkeiten installiert, sowie CUDA-Software inklusive Treiber und CuDNN zur beschleunigten Berechnung auf der Grafikkarte.

Das Training eines Faltungsnetzes verläuft in „Caffe“ wie folgt. Das Netz wird in ein Trainingsnetz und ein Testnetz aufgesplittet, die in der jeweiligen Datei im Format `<…>.prototxt` gespeichert werden. Der Grund dafür ist, dass beide Netze die Bilder aus unterschiedlichen Quellen beziehen – also aus dem Trainingsset bzw. Testset. Außerdem hat das Testnetz unter Umständen nicht die gleiche Architektur wie das Trainingsnetz, weil das

so gewollt ist oder z.B. weil dort kein Dropout und keine Backpropagation stattfinden. Letztere Unterschiede werden jedoch erst zur Laufzeit bemerkbar, und abgesehen von der Input-Schicht enthalten beide Dateien in den meisten Fällen die gleiche Information: Jede definiert die Reihenfolge der Schichten sowie ihre Hyperparameter. Ferner muss eine weitere `.prototxt`-Datei existieren – der sogenannte Solver, wo die Trainingsparameter wie Lernrate, Iterationszahl, Batch-Größen usw. gespeichert werden. Zwecks Training kann „Caffe" aus der Befehlszeile mit dem Solver als Argument gestartet werden, daher muss dieser die Pfade zu den Netzdateien kennen. Beispiele für diese drei Dateien können den Anhängen (Kapitel 12) entnommen werden.

Alle Lösungsansätze sind in Python modular implementiert, d.h. jedes Verfahren ist ein separates Skript, der das gemeinsame Modul mit den Methoden zum Erzeugen und Trainieren der Faltungsnetze mittels der Python-Schnittstelle von „Caffe" („pycaffe" genannt) verwendet. Somit ist sichergestellt, dass alle Verfahren die gleiche Wissensbasis haben, aber unabhängig voneinander funktionieren. Die festgelegten Trainingsparameter sowie die Metaparameter des jeweiligen Algorithmus sind in den entsprechenden Konfigurationsdateien gespeichert, die zur Laufzeit ausgelesen werden. Weitere Skripte enthalten Funktionen zur Visualisierung der Ergebnisse, sei es Architektur (siehe Abbildung 10), Filter oder Grafiken vom Verlauf der Erkennungsrate (Accuracy und Loss aus Kapitel 2.3) und vieles mehr. Zur Speicherung der Architekturen und ihrer Performanz für jeden Testlauf wird eine SQLite-Datenbank benutzt. Die verwendeten Datasets MNIST und CIFAR-10 liegen im LMDB-Format vor.

Der typische Ablauf eines Lösungsverfahrens sieht folgendermaßen aus:

(1) Meta- und Trainingsparameter müssen in den Dateien Config_<…>.ini festgelegt werden.

(2) Das Skript mit dem jeweiligen Algorithmus wird gestartet.

(3) *Initialisierung* – eine oder mehrere Netzarchitekturen nach bestimmten Regeln generieren, validieren und in der Datenbank speichern, und zwar in Form von einer Liste der Schichten (siehe Kapitel 4.1).

(4) *Konvertierung* – Jede Netzarchitektur an das Netzmodul übergeben, der für das Training zuständig ist. Er wandelt die Liste von Schichten per „pycaffe" in ein Trainingsnetz-, Testnetz- sowie ein Solver-Objekt um und speichert sie als `.prototxt`-Dateien ab.

(5) *Training und Testing* – Aufruf von „pycaffe", Sammeln und zurückgeben der Performanzdaten pro Iteration für die nachfolgende Auswertung.

(6) *Algorithmus* – Erzeugung des nächsten Faltungsnetzes bzw. Population anhand der errechneten Performanzdaten.

(7) Wiederhole (4) – (6), bis das Abbruchskriterium zutrifft.

(8) *Auswertung* – Veränderung der Zielfunktion im Laufe des Algorithmus, Grafiken des Trainingsverlaufs sowie das beste gefundene Netz ausgeben.

Wie in Kapitel 2.2 bereits erwähnt, werden alle Faltungsnetze mit zufälligen Filtern initialisiert, die während des Trainings konstant bleiben sollen. Jedoch können sich bei der mehrmaligen Initialisierung eines Netzes theoretisch immer unterschiedliche Filter ergeben, was die Reproduzierbarkeit einer einmal erreichten Netzperformanz nahezu ausschließt. Aus diesem Grund mussten die Neuronengewichte bei der Hyperparametersuche für MLP entweder zusammen mit den Architekturparametern gesucht werden, oder eine mittlere Performanz über viele Initialisierungen musste ermittelt werden. Beides führte zu einem längeren Suchprozess, da entweder der Suchraum deutlich größer wurde oder mehrere Trainingszyklen nötig waren.

Glücklicherweise gibt es eine Möglichkeit, die Filterinitialisierung sowohl zufällig als auch reproduzierbar zu gestalten. Sie besteht darin, ein sogenanntes *Random-Seed* zu hinterlegen, das als Startwert für den Zufallszahlengenerator dient. Wenn das Random-Seed für alle Netze und Verfahren den gleichen Wert hat (z.B. `0xCAFFE`), dann wird jedes Netz immer deterministisch initialisiert. In „Caffe" wird dies durch den Solver-Parameter `random_seed` gewährleistet, der auch über „pycaffe" gesetzt werden kann: `s.random_seed = 0xCAFFE`, wobei `s` das Solver-Objekt ist. Außerdem werden in Faltungsnetzen bestimmte Initialisierungsschemata (Xavier oder MSRA) verwendet, bei denen die Gewichtevarianz lediglich von den eingehenden Verbindungen abhängig ist. Dank diesen wird ein und dasselbe Faltungsnetz immer gleich initialisiert, was mehrere Trainingseinheiten überflüssig macht.

Damit die Filter beim Training konstant bleiben, muss die Backpropagation in allen Schichten außer dem Klassifikator ausgeschaltet werden. Dies geschieht mittels des Flags `propagate_down = False`, der jedem Layertyp zur Verfügung steht und muss in der Layerdefinition ein Mal pro Vorgängerschicht vorkommen. Zusätzlich empfiehlt es sich, in den Schichten Convolution und InnerProduct den Lernrate- und Weight-Decay-Faktor auf 0 zu setzen, und zwar sowohl für die Gewichte als auch für Bias. Dafür wird ein weiterer Parameter in die Layerdefinition aufgenommen: `param = [dict(lr_mult=0, decay_mult=0)] * 2`.

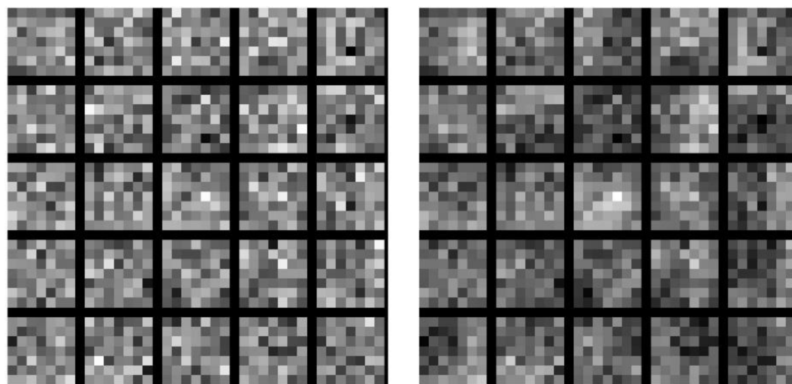

Abbildung 9. Gauß-initialisierte Filter vor und nach dem Training.

Nachdem ein Lösungsverfahren das beste, *„finale"* Netz zurückgegeben hat, muss dieses vollständig trainiert werden, um die maximal mögliche Performanz zu erzielen. Daher wird vom Netz ein Snapshot gemacht, wo der bisher erreichte Zustand gesichert ist, d.h. sämtliche Netzgewichte und Bias. Das Snapshot kann dann als Startpunkt für den neuen Trainingszyklus benutzt werden, bei welchem nun das Lernen der Filter stattfindet. Dieser letzte Schritt ist im jeweiligen Algorithmus integriert und kann bei Bedarf mitausgeführt werden. Hierbei ist es besonders interessant, die ausgegebenen Filter vor und nach dem vollständigen Training zu vergleichen (siehe Abbildung 9). Es wird sichtbar, wie gut die vom besten Netz erkannten Merkmale sich an die Objekte aus dem Dataset angepasst haben, denn bis zu diesem Zeitpunkt waren die Filter immer konstant.

Im Laufe der Experimente wurden einige Herausforderungen überwunden. Zum einen wurde festgestellt, dass das Random-Seed lediglich beim Training auf CPU behilflich ist, um reproduzierbare Ergebnisse zu bekommen. Beim GPU-Training dagegen reicht es nicht aus, weil dort andere Quellen für Nicht-Determinismus existieren, z.B. durch CuDNN[3]. Um diese zu beseitigen, muss der Netzparameter `engine` mitberücksichtigt werden. Er spielt lediglich in bestimmten Layertypen eine Rolle (insbesondere bei Convolution, Pooling und ReLU), und sein Wert wird dementsprechend durch die „pycaffe"-Anweisung `engine = caffe.params.<layertype>.CAFFE` gesetzt.

Zum anderen kam es bei zu hohen Batch-Größen zu Memory Leaks, sodass das Training immer langsamer wurde und am Ende wochenlang dauerte. Zu diesem Problem wird kontrovers diskutiert[4], ob es am Treiber der Grafikkarte liegen könnte. Die einzige derzeit bekannte Lösung ist, die Batch-Größe zu reduzieren. Demzufolge wurde bei weiteren Versuchen die Batch-Größe vom Trainingsset auf 32 heruntergesetzt, vom Testset – auf 10.

4.1 Kodierung der Architektur

Um Evolutionäre Algorithmen für Faltungsnetze nutzen zu können, muss eine entsprechende Repräsentation ihrer speziellen Architektur gefunden werden. So kann die Netzarchitektur „LeNet-5" aus Abbildung 1 beispielsweise durch folgende symbolische Sequenz kodiert werden:

(conv, 5, 1, 6) -> (pool, 2, 2, max) -> (conv, 5, 1, 16) -> (pool, 2, 2, max) -> (fc, 120) -> (fc, 84) -> (fc, 10)

Dabei bedeutet „(conv, 5, 1, 6)" einen Filter Bank Layer mit Filtergröße 5, Schritt 1 und 6 Outputs. Analog „(pool, 2, 2, max)" steht für Max Pooling mit Filtergröße 2 und Schritt 2. Der Full Connection Layer hat hier lediglich einen Hyperparameter – Anzahl der Outputs. Anstelle von „(fc, 120)" kann auch die äquivalente Schicht „(conv, 1, 1, 120)" verwendet werden. Nach

[3] https://github.com/BVLC/caffe/issues/3168

[4] https://github.com/BVLC/caffe/issues/4026

dem Filter Bank Layer und dem Full Connection Layer könnte theoretisch auch je ein Non-Linearity Layer hinzugefügt werden, z.B. mit Tangens hyperbolicus, wie ursprünglich in [6], oder ein ReLU. Deswegen muss die erste, intuitive Repräsentation noch erweitert werden, siehe auch Tabelle 1:

(conv, 5, 1, 6, relu) -> (pool, 2, 2, max) -> (conv, 5, 1, 16, relu) -> (pool, 2, 2, max) ->

-> (fc, 120, relu) -> (fc, 84, relu) -> (fc, 10, -)

Somit ergibt sich, dass jeder Layertyp aus 6 Platzhaltern für Hyperparameter besteht: Name des Layertyps, Non-Leariry-Typ, Pooling-Typ, Filtergröße, Schrittgröße, Anzahl Outputs. Offensichtlich werden nicht alle Platzhalter für die Beschreibung des jeweiligen Layertyps gebraucht. Dennoch können sie zu einem Chromosom fester Länge umgewandelt werden, und zwar nach folgenden Regeln:

- ✓ Name des Layertyps → conv = **00**, pool = **01**, fc = **10**
- ✓ Non-Linearity-Typ → keins = **00**, ReLU = **01**, ELU = **10**, Maxout = **11**
- ✓ Pooling-Typ → keins = **00** (nur für conv und fc), max = **01**, avg = **10**, max+avg = **11**
- ✓ Filtergröße → binäre Repräsentation der Zahl mit fester Länge
- ✓ Schrittgröße → binäre Repräsentation der Zahl mit fester Länge
- ✓ Anzahl Outputs → binäre Repräsentation der Zahl mit fester Länge

So wird z.B. aus dem Layer „(conv, 5, 1, 6, relu)“ eine Genabfolge „00.01.00.00101.00001.0000000110“. Die maximale Länge des jeweiligen Platzhalters kann durch bekannte Kennzahlen (Metaparameter) des Datasets ermittelt werden. Wenn das Netz mit MNIST trainiert wird, sind die Input-Bilder 28×28 Pixel. Nach den in Kapitel 2.1 erwähnten Faustregeln können Filter- und Schrittgröße maximal 28 sein und ist daher mit 5 Bits kodierbar. Die maximale Anzahl der Outputs wird dann erreicht, wenn jeder Pixel des Inputs als ein separates Merkmal betrachtet wird, also sind es insgesamt $28 \cdot 28 = 784 < 2^{10}$ Outputs und 10 Bits. In diesem Fall wäre die Länge eines Layers dementsprechend 26 Bit bzw. Gene. Dasselbe gilt übrigens für CIFAR-10, wo die Bilder 32×32 Pixel groß sind. Die Berechnung der maximalen Chromosomlänge anhand der Input-Dimensionen erfolgt in den Lösungsverfahren automatisch.

Nun stellt sich die Frage, aus wie vielen Schichten ein Chromosom bestehen kann. Da ein klassisches GA mit Chromosomen konstanter Länge arbeitet, wurde die potentiell gute Anzahl der Layer auf 6 bis 8 eingeschätzt. Der Grund dafür ist, dass einzelne 6- oder 7-lagige Architekturen mit MNIST die niedrigsten bekannten Fehlerraten erreichten (≈0,47 % [64] bzw. 0,8 % bei „LeNet-5“ [6]). Es wird also vermutet, dass ein Optimum in Bezug auf Hyperparameter bei den 7-lagigen Architekturen liegt. Für spätere Verfeinerungen des GA könnten aber auch Architekturen bzw. Chromosomen variabler Länge in Betracht gezogen werden.

Allerdings ergeben nicht alle zufällig generierte oder durch genetische Operatoren berechnete Chromosomen Sinn. Aus diesem Grund ist es notwendig, jedes Chromosom zu

evaluieren, bevor es in die Population übernommen wird. Dafür sollen folgende Bedingungen für eine **valide** Architektur geprüft werden, die zu einer allgemeingültigen Wissensbasis für Faltungsnetze gehören (siehe Kapitel 2.1):

- Der erste Layer ist „conv“; der letzte ist „(fc, C, -)“, wo C Anzahl der Klassen ist
- Nach einem „fc“ Layer kann kein „conv“ oder „pool“ folgen
- Mehrere „pool“ Layers nacheinander machen keinen Sinn
- Filtergröße bei „conv“ ist ungerade, liegt zwischen 3 und 11 bzw. der Hälfte der Input-Dimension
- Schrittgröße bei „conv“ liegt zwischen 1 und 5 bzw. der Filtergröße
- Filtergröße bei „pool“ ist 2 oder 3 bei der Schrittgröße 1 oder 2
- Anzahl der Output-Feature-Maps ist positiv, gerade, zwischen 16 und 128 in der ersten Schicht, monoton steigend in den mittleren Schichten und kann nur im Klassifikator wieder sinken.

Beispiel einer validen Architektur ist in Abbildung 10 dargestellt. Obwohl sie offenbar einige „krumme“ Zahlen enthält, ist sie als Architektur plausibel und kann qualitativ sogar besser sein als einige „geraden“ Kontrahenten. Standardmäßig wird jede Architektur am Ende um den Softmax-Layer mit Loss ergänzt, der nicht mitgezählt und zur Erfassung von Loss auf dem Trainingsset benötigt wird.

Die meisten von oben erwähnten Hyperparameter lassen sich mit „Caffe“ leicht umsetzen. Zwei Ausnahmen bilden die Summe von Max und Average Pooling sowie Maxout. Diese müssen aus vorhandenen Layertypen zusammengesetzt werden, wie in Abbildung 11 veranschaulicht wird. Der Layertyp `Eltwise` berechnet elementenweise die Summe, das Produkt oder auch das Maximum von seinen Inputs. Der `Slice`-Layer splittet die Menge der Input-Feature-Maps in die gegebene Anzahl der Teile auf (vollständige Teilung vorausgesetzt!), sodass diese Teilmengen separat bearbeitet werden können. So kommt die obige Bedingung zustande, dass nur eine gerade Anzahl der Outputs valide ist.

Letztendlich wird in den Lösungsverfahren mit beiden Arten der Architekturrepräsentation gearbeitet – mit symbolischer und binärer. Die erste sorgt für menschliche Lesbarkeit, die zweite erleichtert enorm die Durchführung von Crossover und Mutation und nähert diese Operatoren an die Bitmanipulation auf Prozessor-Ebene [36]. Beide Repräsentationen erfordern allerdings Methoden zur gegenseitigen Konvertierung.

Eine binäre Kodierung stößt nicht an die Grenzen der Hardware-Präzision und ist leicht erweiterbar, enthält im Idealfall aber keine überflüssigen Bits, die die Evolution verhindern könnten. Ihre Vorteile erstrecken sich sogar auf SA, wo im Allgemeinen keine derartige Kodierung nötig ist. Und zwar hilft sie dabei, die Distanz zwischen zwei Architekturen zu messen und dadurch den Nachbarschaftsraum von einem Netz zu definieren.

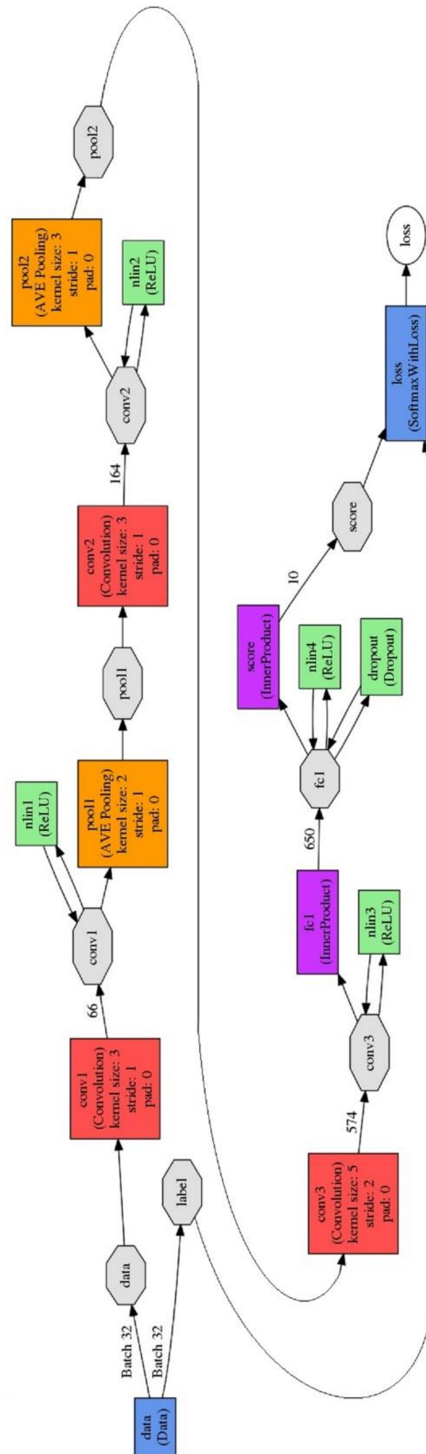

Abbildung 10. Ein Faltungsnetz mit valider Architektur aus 7 Schichten.

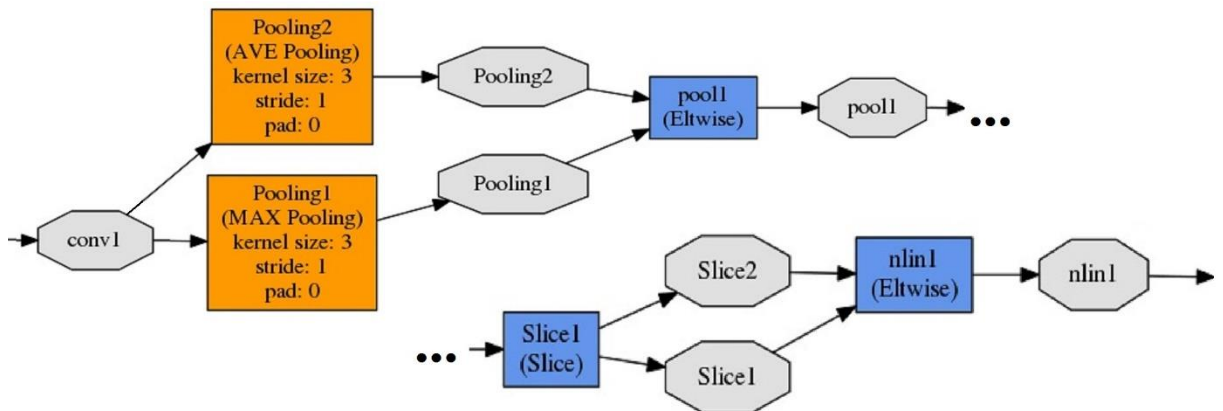

Abbildung 11. Umsetzung der Summe von Max- und Average-Pooling (links) und des Maxouts (rechts) in "Caffe".

Bei der beschriebenen Umwandlung eines Faltungsnetzes in ein binäres Chromosom kann das Problem der N:1 Abbildung vom Genotyp zu Phänotyp nicht vorkommen. D.h. aus zwei funktional gleichen Netzen (bis auf Permutation) lassen sich keine zwei verschiedenen Repräsentationen ableiten, wie das bei MLP geschildert wurde. Denn die Faltungsnetzstruktur ist linear, ohne Verzweigungen, und die Permutation der Bits würde immer ein anderes Netz erzeugen. Es ist ebenfalls nicht möglich, dass ein binärer String in zwei unterschiedliche Architekturen resultiert; vorausgesetzt, die weiteren Architekturparameter (Padding, Dropout, Anzahl der Teile in Slice-Layer bei Maxout etc.) bleiben konstant für alle Netze. Daraus folgt, dass die ausgewählte Abbildung vom Genotyp zum Phänotyp im Versuchsaufbau mit Faltungsnetzen eineindeutig ist.

4.2 Genetischer Algorithmus

Im Folgenden wird beschrieben, wie ein GA an die Optimierung der Faltungsnetzarchitektur angepasst wurde. Da die obige binäre Repräsentation bereits gute Eigenschaften hat, müssen noch die sinnvollen genetischen Operatoren passend definiert werden. Erfahrungsgemäß muss bei der Wahl ihrer konkreten Ausprägung immer die gegebene Problemstellung beachtet werden.

Die in Kapitel 3.1 vorgestellten Varianten von Selektion unterscheiden sich vor allem darin, wie stark sie die Verteilung der Fitnesswerte in der aktuellen Population berücksichtigen. Während das Elitismus-Prinzip diese Verteilung ignoriert und nur die besten Chromosomen favorisiert, sind die anderen Selektionsarten dazu konzipiert, Individuen mit möglichst diverser Fitness (d.h. auch die schlechten) in die nächste Population mitzunehmen.

Jedoch ist die Populationsdiversität aus der Sicht der Architektursuche entscheidender für den Erfolg des GA als die Fitnessdiversität. Denn zwei Netzarchitekturen mit ähnlichen Fitnesswerten müssen nicht zwangsläufig strukturell ähnlich sein (vgl. zwei entfernte Punkte im Suchraum), und die Populationsentwicklung würde eher davon profitieren, wenn beide Netze überleben und ihre guten Eigenschaften weitergeben. Auf diese Weise können zwei ähnlich vielversprechende Regionen abgesucht werden, ohne dass eine verfrühte Entscheidung getroffen werden muss. Andererseits können zwei ähnliche Netze stark abweichende Fitnesswerte haben, und die Selektion beider Netze würde die Evolution womöglich nicht voranbringen. In dieser Hinsicht hat Elitismus mehr Vorteile als andere Selektionsvarianten, da es sich nicht auf die Fitnesswerteverteilung verlässt und die Erhaltung der besten bisher gefundenen Lösungsqualität sichert.

Dementsprechend wurde die elitistische Variante der Selektion den anderen vorgezogen. Diese Entscheidung wirkt sich ebenso auf Mutation aus, denn die selektierten Individuen dürfen nicht verändert werden, um die beste Lösungsqualität für die Zukunft zu bewahren.

Was den Crossover betrifft, muss diejenige Variante gewählt werden, die größtenteils zulässige und gute Architekturen mit wenig Aufwand erzeugt. Beides wird z.B. bei uniformen Crossover schwierig, weil Elternchromosomen beliebige Teile der kodierten Hyperparameter austauschen. Dadurch werden die Werte durchgemischt, die für sich genommen alle Nebenbedingungen erfüllt haben, aber in Kombination kein gutes Faltungsnetz ergeben. n-Punkte-Crossover arbeitet dagegen mehr sequentiell, was zur linearen Layerabfolge eher passen würde. Allerdings kann er für $n \geq 3$ ebenso unvorteilhaft werden wie der uniforme Crossover, vor allem für größere n. Auch für $n = 2$ wird das mittlere Stück des Chromosoms aus dem Kontext gerissen, in dem es früher gut funktioniert hat, und in das andere Netz geschoben, sodass er dort unter Umständen ineffizienter wird.

Aus diesen Gründen fiel die Wahl auf 1-Punkt-Crossover, bei welchem ein gemeinsamer Schnittpunkt bei den Elternchromosomen gesucht wird, sodass das Kindchromosom bis zum Schnittpunkt aus dem ersten (fitteren) Elternteil und ab dem Schnittpunkt aus dem zweiten zusammengesetzt wird. Dies wird solange durchgeführt, bis das Kindchromosom eine valide Architektur gebildet hat. Sollte dieser Fall für alle Schnittpunkte nie eintreten, wird das nächste Elternpaar genommen. Somit ist sichergestellt, dass alle Hyperparameter im Kontext bleiben, die guten Netzeigenschaften weitergereicht werden und trotzdem neue Kombinationen mit viel Potential entstehen können.

Der Grundgedanke von Mutation ist es wiederum, Populationsdiversität zu verstärken, ohne die Suche durch große Sprünge in zufällige Richtungen ständig zu verstreuen. Die Mutationswahrscheinlichkeit von einem Gen soll sehr klein sein und beträgt typischerweise 0,01 [39]. Mit anderen Worten, jedes Individuum (hier – bis auf die fittesten) nimmt an der Mutation mit der Wahrscheinlichkeit p_m teil. Wenn es also für die Mutation ausgewählt wird, soll bei ihm ein zufälliges Gen unbedingt verändert werden. D.h. die Position des zu kippenden Bits wird solange gewürfelt, bis eine zulässige Architektur entsteht, die sich von der ursprünglichen unterscheidet, oder bis alle Positionen ausgeschöpft sind.

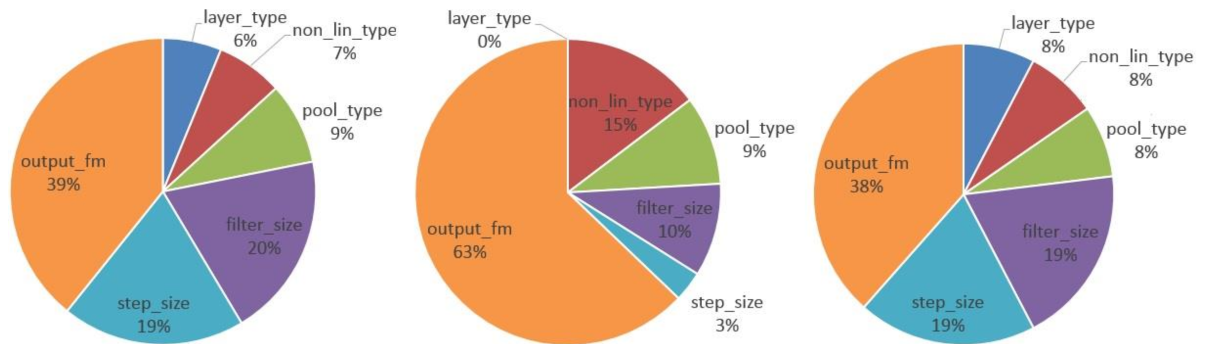

Abbildung 12. Änderungshäufigkeit pro Hyperparameter bei Crossover (links) und bei Mutation (Mitte), Anteil der entsprechenden Gene an einem Chromosom (rechts).

Die Statistik über die Häufigkeit, welche Hyperparameter wie oft bei Crossover und Mutation im Rahmen der Experimente verändert wurden, kann der Abbildung 12 entnommen werden. Es fällt dabei auf, dass der valide Crossover jeden Hyperparameter fast gleichermaßen verändern kann (vgl. die Anteile in den Kuchendiagrammen links und rechts). Bei Mutation ist es anders: Der Layertyp wurde nie durch das Bitkippen gewechselt, in den meisten Fällen wird aber die Anzahl der Outputs getroffen. Dieses Ergebnis ist plausibel, denn dank der wohldefinierten Abfolge der Netzschichten ereignet sich bei Crossover der Austausch der Chromosomenteile, die höchstwahrscheinlich zueinanderpassen werden, egal wo der Schnitt war. Im Gegensatz dazu kann die Layertypveränderung bei Mutation nie stattfinden, sonst würden solche unzulässigen Kombinationen wie „Pooling -> Pooling" erzeugt werden. Die Schrittgröße hat mit 3 % ebenfalls einen kleinen Anteil an validen Mutationen, weil sie sehr stark von der Filtergröße abhängig ist, vor allem bei Pooling Layer. Die potentiell veränderte Filtergröße müsste wiederum viele Nebenbedingungen erfüllen, um für eine valide Mutation infrage zu kommen, daher ist auch ihr Anteil relativ klein. Den größten Spielraum (63 %) bekommt hier wie erwartet die Anzahl der Outputs.

Bei der Generierung der ersten Population ist es äußerst wichtig, von Anfang an valide, gute Architekturen zu erzeugen, sonst wird der GA zu schlecht initialisiert und läuft in die Gefahr eines lokalen Optimums. Die Eigenschaften einer **guten** Architektur sind:

- Die erste Schicht ist Convolution, die letzte – „(fc, 10, -)"
- Der Klassifikator besteht aus einem oder zwei „fc" Layers (von außen einmalig konfigurierbar, so wie die gesamte Anzahl der Layers)
- Eine Feature-Extraction-Phase besteht aus „conv" oder „conv + pool" (beides gleich wahrscheinlich)
- Jede Art der Non-Linearity ist gleich wahrscheinlich: ReLU, ELU, Maxout oder die Kombination „ELU nach Convolution + Maxout im Klassifikator" (wie in [15])

- Der Pooling-Typ ist in allen Pooling Layers derselbe und gleich wahrscheinlich
- Filter- und Schrittgröße nach den Regeln einer validen Architektur
- Anzahl der Outputs ist pro Schicht zufällig:
    - Für die erste Convolution-Schicht liegt sie zwischen 16 und 128, für weitere Schichten – zwischen der Anzahl von Outputs des Vorgängers und der Hälfte des verbliebenen Spielraumes (beschränkt durch die maximale Anzahl 1023)
    - In den Full Connection Layers wird sie wieder abgesenkt

Der resultierende, angepasste GA kann so beschrieben werden:

(1) Erste Population mit M zufälligen Individuen erzeugen (Faltungsnetze mit konstanten zufälligen Filtern).
(2) Jedes Individuum trainieren, um den Wert der Fitnessfunktion zu ermitteln.
(3) *Selektion*: Die K fittesten Individuen werden ausgewählt und in die nächste Population übernommen. Restliche Individuen werden verworfen.
(4) *Crossover*: Nach der Sortierung von Individuen nach Fitnesswerten müssen die Paare gebildet werden. Jeweils ein solches Chromosomenpaar produziert genau 1 Nachwuchs, bis die Populationsgröße erreicht ist.
(5) *Mutation*: Mit der Wahrscheinlichkeit p_m wird bei einem Kindchromosom ein zufälliges Gen geändert.
(6) Wiederhole (2) – (5), bis die maximale Anzahl der Populationen N erreicht oder eine hinreichend gute Lösung gefunden wird.

Beispielsweise sei $p_c = K/M = 0{,}5$ der Anteil der in Schritt (3) ausgewählten Architekturen. Dann werden bei einer Population mit 30 Netzen die 15 fittesten ausgewählt, miteinander gekreuzt und produzieren 15 weitere Kindchromosomen. Diese werden nun mutiert, sodass bei $p_m = 0{,}75$ insgesamt $0{,}75 \cdot 15 \approx 11$ Gene verändert werden. Das gibt den Kindchromosomen die Chance, ohne große Veränderungen in die nächste Population überzugehen und fitter zu sein als ihre Vorfahren. Letzten Endes sind die spezifischen Metaparameter von GA folgende: $N,\ M,\ p_c,\ p_m$.

4.3 Simulated Annealing

SA enthält ebenfalls einige spezifische Funktionen und optimierbare Metaparameter, die für jede konkrete Problemstellung sorgfältig ausgewählt werden müssen. Wie in Kapitel 3.2 bereits erwähnt, sind es die Formeln für den Abkühlungszeitplan und die Wahrscheinlichkeit im Akzeptanzkriterium. Bei ersterem wurde versucht, seine Kurve an die von einer optimalen Abkühlung anzunähern, jedoch so, dass die Temperatur innerhalb von 100 Iterationen schneller gegen 0 geht. Dies gelingt mit folgender Formel:

$$T_i = (\theta^i) \cdot T_0, \text{ wobei } \theta = 0{,}9 \text{ und } T_0 = 2$$

Der Verlauf der Temperatur nach dieser Formel und der Vergleich zur optimalen Abkühlung wurden in Abbildung 8 bereits dargestellt. Die Temperaturwerte haben direkte Auswirkung auf die Wahrscheinlichkeit davon, wie oft die schlechteren Lösungskandidaten akzeptiert werden. Deswegen muss die Temperatur eventuell skaliert werden, damit die Wahrscheinlichkeit nicht dauerhaft zu hoch ist:

$$P\big(T, f(s), f(s')\big) = e^{\frac{f(s')-f(s)}{d \cdot T}}$$

Bei $d = 1$ wird diese Gleichung zur klassischen Formel für die Akzeptanzwahrscheinlichkeit, hat aber in Bezug auf den gewählten Abkühlungsplan den Nachteil, dass die Senkung der Fitness um 10 % bis zum Ende des Algorithmus toleriert wird. Der Wert $d = 0{,}05$ dagegen fördert das gewünschte Verhalten, wie in Abbildung 13 verdeutlicht wird. Eine relativ große Änderung der Fitnesswerte ($\Delta = 0{,}1 \sim 10\%$) wird nur am Anfang, aber nicht einmal in der Hälfte aller Fälle akzeptiert; eine sehr kleine Änderung ($\Delta = 0{,}0001 \sim 0{,}01\%$) ist in den späteren Iterationen immer noch möglich. Auf diese Weise kann SA den lokalen Optima auch in Endstadien entkommen, ohne dass die Lösungsqualität drastisch sinkt.

Des Weiteren wird bei SA eine Nachbarschaftsstruktur im Suchraum benötigt, die letztendlich auf dem Abstand zwischen zwei Faltungsnetzarchitekturen beruht. Eine prädefinierte Distanz existiert in dem Fall nicht, sie könnte aber beispielsweise anhand der binären Repräsentation leicht berechnet werden. Demnach ist die Distanz zwischen zwei Architekturen nichts anderes als die Anzahl der Bits, in denen sich die entsprechenden Chromosomen unterscheiden. Mit anderen Worten, je mehr Bits zwei Chromosomen gemeinsam haben, desto näher im Suchraum (und auch ähnlicher) sind sich die Architekturen. So wird die Nachbarschaft eines Netzes durch den Radius R vorgegeben: Alle Netze innerhalb des Kreises mit Radius R gehören zu seinen Nachbarn. Da der Radius von der Chromosomlänge L abhängig ist, muss er als Faktor der Länge vorgegeben werden. D.h. wenn $R = 0{,}15$ und $L = 26 \cdot 7 = 182$, dann haben die Nachbarn eines 7-lagigen Netzes höchstens $R \cdot L = 0{,}15 \cdot 182 \approx 28$ Bits mit ihm nicht gemeinsam. Um einen zufälligen Nachbarn auszuwählen, muss die Distanz r zwischen 1 und R zufällig bestimmt werden. Dann wird versucht, $r \cdot L$ Gene beim ursprünglichen Netz zu ändern, bis eine valide Architektur entsteht. Diese wird als nächster Lösungskandidat genommen. Die endgültigen spezifischen Metaparameter von SA sind: $N,\ \theta,\ T_0,\ d,\ R$.

Ansonsten bleibt der Ablauf von SA wie in Kapitel 3.2 beschrieben. Zur Generierung von guten initialen Architekturen müssen die gleichen Regeln eingehalten werden wie bei GA. Die richtigen Regeln sind bei SA sogar viel wichtiger als bei GA, weil hier eine einzige Initiallösung existiert, und diese muss so gut wie möglich sein, damit das Verfahren Erfolg hat. Der andere Weg, eine gute Initiallösung zu bekommen, besteht darin, diese durch weitere Heuristiken berechnen zu lassen (unter anderem durch GA).

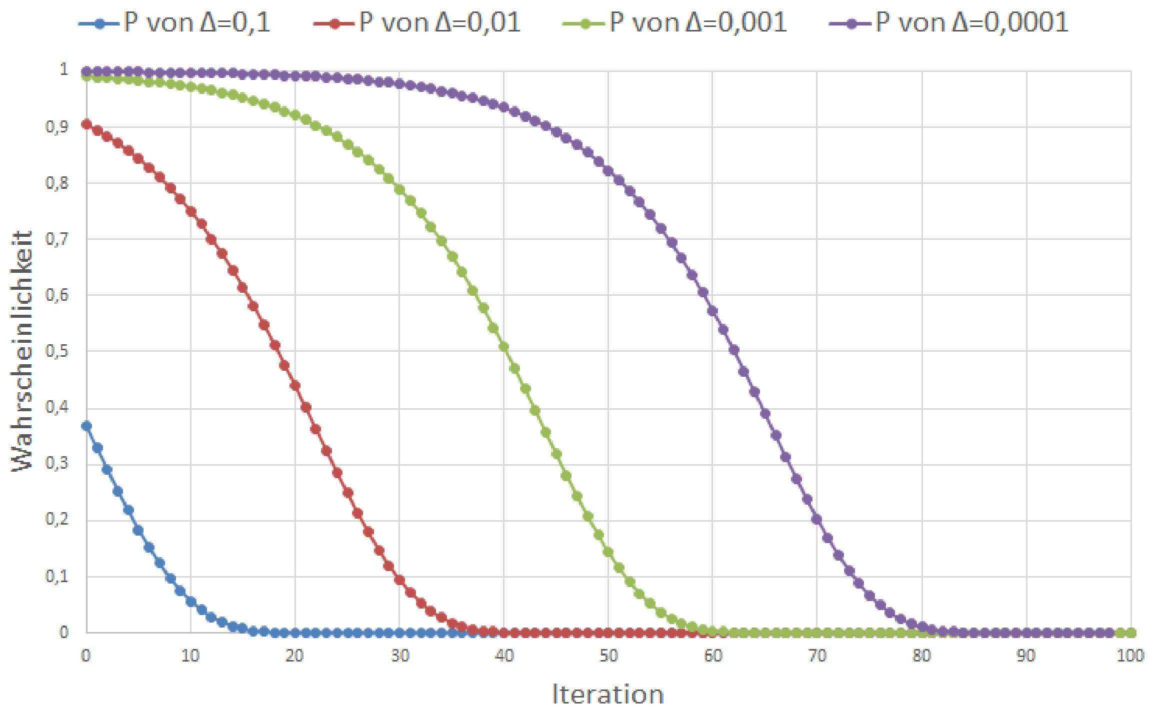

Abbildung 13. Akzeptanzwahrscheinlichkeit für unterschiedlich große Änderungen der Fitnesswerte (Δ).

4.4 Memetischer Algorithmus

Als Vorlage zur Implementierung von MA diente GA. Die Erweiterung sollte darin bestehen, am Anfang jeder Schleife Lokale Suche für einen Teil der Population durchlaufen zu lassen. Um den Vorteil von MA gegenüber dem GA vollständig zu nutzen, wurde jedoch die Entscheidung getroffen, Lokale Suche für die gesamte Population zu erlauben. Dies stellt keinen Widerspruch zum Elitismus-Prinzip dar, denn die fittesten Individuen der Population werden nur durch noch fittere ersetzt.

Demzufolge wird für jedes Individuum S Nachbarn aus dem Umkreis mit Radius R zufällig generiert (analog SA) und mittels Fitnessfunktion evaluiert. Sollte eine Nachbararchitektur einen höheren Fitnesswert haben als die ursprüngliche, so wird die letztere entsprechend ersetzt. Die spezifischen Metaparameter von MA sind also N, M, p_c, p_m, S, R.

Da MA mit langen Ausführungszeiten verbunden ist, was wiederum die Gefahr eines Absturzes aufgrund des Memory Leaks erhöht, wurde eine Möglichkeit zum Speichern des Status der Lokalen Suche geschaffen. Genauer gesagt, wird für jede Architektur ein Flag in der Datenbank gesetzt, ob für sie bereits Lokale Suche durchgeführt wurde. Damit ist es nicht mehr nötig, nach dem Absturz die Lokale Suche für diejenigen Individuen der Population zu wiederholen, für die es zuvor geschah.

4.5 Datasets

Um die vorgestellten Lösungsansätze miteinander und mit anderen „State-of-the-Art"-Deep-Learning-Modellen evaluieren zu können, bedarf es Datasets, die verschiedene Aspekte der Algorithmen zum Vorschein bringen, z.B. Skalierbarkeit, Handhabung von Farbinformationen, Merkmalskomplexität usw.

Im Bereich Objektklassifizierung gibt es heutzutage ein breites Spektrum an Benchmark-Datasets, die von vielen Wissenschaftlern verwendet werden, um ihre Modelle objektiv miteinander vergleichen zu können. Jedes Dataset hat seinen eigenen Zweck, sei es Geschlechtserkennung, Verkehrsschildererkennung (GTSRB, [29]) oder Hausnummererkennung (SVHN, [20]), und sie variieren in Größe, Bilddimensionen, Anzahl der Bilder pro Klasse und Farbschema.

Name des Datasets	# Bilder	# Klassen	# Bilder pro Klasse	Farbschema
Caltech-101	9.145	102	31 – 800	Farbe
PASCAL VOC	11.540	20	303 – 4.087	Farbe
Caltech-256	30.607	257	80 – 827	Farbe
NORB	58.320	6	9.720	Graustufen
CIFAR	60.000	10 / 100	6.000 / 600	Farbe
MNIST	70.000	10	7.000	Graustufen
SVHN	630.420	10	-	Farbe
ILSVRC	1.331.167	1.000	782 – 1.350	Farbe
ImageNet	14.197.122	21.841	Ø650	Farbe

Tabelle 3. Datasets im Bereich Objektklassifizierung.

Das größte und dafür umso mehr begehrte Dataset ist ImageNet [4] mit über 14 Millionen Bildern von natürlichen Objekten aus knapp 22.000 Klassen. Manche Klassen sind feingranular (z.B. Hunderassen oder Vogelarten), andere grobgranular (Küchenutensilien); sie enthalten durchschnittlich 650 Bilder in der Auflösung 224 × 224 Pixel. Eine Teilmenge von ImageNet bestehend aus 1,3 Mio. Bildern und 1.000 Kategorien wird jährlich für den ILSVRC-Wettbewerb festgelegt.

Das Training eines Netzes auf dem ILSVRC-Dataset nimmt sogar mit dem Einsatz moderner GPUs mehrere Tage in Anspruch. Deshalb wurde es noch nie für die automatisierte

Architekturoptimierung verwendet – ein Testlauf eines solchen Algorithmus würde Monate oder sogar Jahre dauern (auf der in dieser Arbeit verwendeten Hardware). Allerdings gibt es Studien, die eine manuelle und selektive Grid-Suche nach guten Architekturen durchgeführt haben [15]. Dabei wurde die Bildauflösung halbiert, was das vollständige Training eines 10-lagigen Netzes noch innerhalb von 24 Stunden ermöglichte.

Eine Alternative bieten zahlreiche kleinere Datasets, die ähnlich wie ImageNet aufgebaut sind: Caltech-101, Caltech-256, PASCAL VOC und CIFAR, siehe Tabelle 3. So sind beispielsweise natürliche Objekte bei **CIFAR-10** und CIFAR-100 auf 32×32 Pixel herunterskaliert. Die beiden Datasets sind durch eine Internetsuche gesammelt worden und enthalten 60.000 Bilder, davon gehören 50.000 zum Trainingsset und 10.000 zum Testset. Die dargestellten Objekte sind bei CIFAR-10 Flugzeuge, Autos, LKWs, Schiffe, Vögel, Katzen, Reintiere, Hunde, Frösche und Pferde (vgl. Abbildung 14 links). Die Kategorien von CIFAR-100 sind mehr feingranular und umfassen unter anderem Reptilien (Krokodile, Dinosaurier, …), Blumen (Rosen, Tulpen, …), Möbel, Haushaltsgeräte etc. [11].

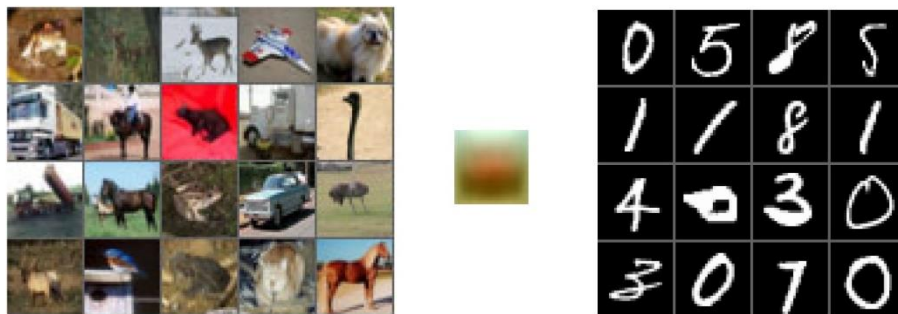

Abbildung 14. Von links nach rechts: Beispielbilder von CIFAR-10; Mean-Bild von CIFAR-10; Beispielbilder von MNIST [22].

Eine andere Art von Objekten wird vom Dataset **MNIST** behandelt – handschriftliche Ziffern in Graustufen (Abbildung 14 rechts). Hierzu liegen 60.000 Trainingsbilder und 10.000 Testbilder vor. Diese wurden durch das Scannen von Handschriften bei der US-amerikanischen Volkszählungsbehörde, aber auch von Studenten gesammelt [6].

Trotz der Tatsache, dass den modernen Datasets mehr Bilder zur Verfügung stehen als je zuvor, neigen viele, vor allem größere Faltungsnetze zum Overfitting. Eine Regularisierungstechnik, die das Problem von Overfitting auf Datasetebene zu lösen versucht, nennt sich *Datenaugmentation* [22]. Sie besteht darin, die Input-Bilder auf verschiedene Weise zu transformieren, um einerseits die Datenmenge anzureichern, andererseits aber die Robustheit der Objekterkennung zu erhöhen. Mögliche Transformationen sind das Zuschneiden des Bildes auf eine beliebige Region (Cropping [5]), Spiegelung, zufällige Verschiebung [34] usw.

Auf der anderen Seite können die Bilder bei Bedarf vorverarbeitet werden (engl. *Preprocessing*), damit das Faltungsnetz sich auf wesentliche Merkmale konzentriert. Klassischerweise wird vom kompletten Dataset ein Mean-Bild berechnet, zusammengesetzt aus Mittelwerten von gleichpositionierten Pixeln über alle Input-Bilder – siehe Abbildung 14, Mitte. Dieses Mean-Bild wird anschließend von jedem Input-Bild subtrahiert, bevor dieses an das Netz weitergegeben wird. Dadurch, dass das Mean von den transformierten Bildern nun 0 ist, wird die Datenverteilung zentriert [11]. Weitere Techniken von Preprocessing sind lokale Kontrastnormalisierung [22], die vor allem bei sättigenden Aktivierungsfunktionen wie Tangens hyperbolicus nötig ist [11], oder globale Kontrastnormalisierung mit „ZCA whitening" [20].

Im Rahmen der eigenen Experimente mit Metaheuristiken wurden MNIST und CIFAR-10 als Datasets ausgewählt. Zum einen sind sie gute Repräsentanten für Problemstellungen aus der realen Welt, die mit natürlichen Objekten sowie abstrakten Zeichen, und zwar in Farbe oder in Graustufen, zu tun haben. Zum anderen sind sie in der Forschung extrem beliebt, und viele Fortschritte in den Erkennungsraten wurden bislang dokumentiert. Darüber hinaus enthalten sie genügend Daten, erweisen dennoch gute Laufzeiten beim GPU-Einsatz, sodass die Metaheuristiken innerhalb einiger Tage gute Ergebnisse abliefern können.

Bei den Tests wurden Graustufen von MNIST-Bildern um den Faktor $1/256$ skaliert, bei CIFAR-10 erfolgte die Subtraktion des Mean-Bildes. Es ist also wissenswert, wie hoch die beste bekannte Performanz bei Deep-Learning-Modellen ist, die dieselben Vorverarbeitungsschritte verwenden. So liegt für CIFAR-10 die Fehlerrate bei 15,13 % [22], wobei das Modell mit Local Response Normalization aus [11] stochastisches Pooling verwendet. Ein anderer Ansatz mit einer ausgiebigen Datenaugmentation führte zur noch nicht übertroffenen Fehlerrate 3,47 %, mehr dazu siehe [34].

Für MNIST galt über Jahre die Fehlerrate 0,95 % von „LeNet-5" [6] als die beste Performanz ohne Augmentation und Preprocessing, bis sie schließlich auf 0,53 % verbessert wurde [13], allerdings mithilfe unüberwacht gelernter Filter. Ein weiteres, 6-lagiges Netz mit DropConnect [26] erreichte 0,52 % ohne Augmentation und 0,21 % mit, was ein bisher nicht geschlagener Rekord für MNIST ist.

5 Qualitätssicherung

Während der Implementierung wurden verschiedene qualitätssichernde Maßnahmen umgesetzt, auf die in diesem Kapitel eingegangen wird. Diese sollen dazu dienen, die langen Trainingszeiten vieler nacheinander folgender Faltungsnetze störungsfrei zu ermöglichen und dadurch die begrenzte Rechenkapazität am effektivsten zu nutzen.

Da es sich bei jedem Lösungsansatz um eine Software handelt, die bestimmte Anforderungen erfüllen muss, ist ein Software-Test hier unumgänglich. Zu den Anforderungen gehören z.B. die Regeln zum Aufbau einer validen Faltungsnetzarchitektur, Funktionsweise der genetischen Operatoren, zuverlässige Ausgabe der Ergebnisse usw. Die aus den Anforderungen abgeleiteten Testfälle wurden mit dem standardmäßigen Unit-Test-Framework von Python 2.7. namens `unittest` umgesetzt.

Beim Testen wurde ein risikobasierter Ansatz verfolgt, der besagt, dass diejenigen Software-Komponenten mit der höchsten Priorität getestet werden müssen, bei den das Auftreten von Fehlern das höchste Risiko darstellt. Dafür wurden die Methoden identifiziert, die essentiell für die fehlerfreie Funktionsweise von den implementierten Algorithmen sind und deswegen einen besonderen Fokus bekommen sollen. Das sind in erster Linie die Methoden zur Prüfung der Architekturen auf Validität (`is_valid`) und zur Umwandlung zwischen ihrer binären und symbolischen Repräsentation (`chrom_to_arch` und `arch_to_chrom`) aus dem Modul `Utility_Arch`. Diese Methoden haben den Vorteil, dass sie deterministisch sind und bei der gleichen Eingabe immer die gleiche Ausgabe liefern.

Im Gegensatz dazu sind Methoden wie `generate_init_arch` (im Modul `Utility_Arch`) nicht deterministisch, weil sie die zufälligen Architekturen generieren sollen, insofern können sie nicht für alle möglichen Architekturen vollständig getestet werden. Dies gilt ebenfalls für Methoden zum Trainieren der Faltungsnetze (`create_and_train`) im Netzmodul, denn je nach Trainings- oder Hyperparameter kann es zur Divergenz im Trainingslauf kommen, was

nicht vorhersagbar ist. In diesen Methoden sollen die Tests eher die Ausnahmefälle abdecken, ohne Anspruch auf Vollständigkeit.

Aufgrund der dynamischen Typisierung in Python ist es außerdem wichtig, zu prüfen, dass die an eine Methode übergebenen Argumente vom richtigen Typ sind. Auf diese Weise wurde die Schnittstelle zwischen zwei wesentlichen Komponenten jedes Lösungsverfahrens – dem Algorithmusmodul und dem Netzmodul – getestet (`create_and_train`). Auch die Methoden zur Visualisierung der Ergebnisse erforderten Tests auf die richtige Typisierung und Dimensionalität der Argumente, denn der Abbruch des Lösungsverfahrens während der Generierung von den finalen Statistiken aufgrund z.B. der nicht passenden Länge einer Liste wäre fatal.

Ein weiteres Risiko stellen Faltungsnetzarchitekturen dar, die entweder sehr schlechte Performanz liefern oder eine ineffiziente Struktur haben. Im Worst-Case-Szenario kommt es dazu, dass die Lossfunktion divergiert, übersteigt den maximal in „Caffe“ vorgesehenen Wert von ca. $87{,}3365$ und wird zu NaN (engl. „Not a Number“). Nachdem NaN sich schnell durch das Netz propagiert hat, sendet „Caffe“ ein Abort-Signal (SIGABRT), das den kompletten Prozess unterbricht. Es ist also umso wichtiger, auf die möglichen Fallstricke zur Laufzeit zu achten und die Ursachen schnell zu erkennen.

Im Fall von schlechter Performanz hilft der **frühzeitige Abbruch** des Trainings, um keine Zeit mehr für weitere Iterationen zu verschwenden und noch vor der Unterbrechung auszusteigen. Es wird deswegen im Netzmodul zur Laufzeit geprüft, ob die Erkennungsrate ab der Iteration 500 auf einem Niveau unter 0,1 stagniert; falls dies zutrifft, wird das Training sofort abgebrochen, und das nächste Netz wird trainiert. Sollte ein Absturz des Programms trotz dieser Vorkehrungen passieren, ist die manuell gestartete Fortsetzung der Ausführung ab der letzten Population bzw. Iteration in jedem Verfahren vorgesehen (Variable `next_run = False` in der Konfigurationsdatei).

Eine ineffiziente Struktur könnte wiederum durch *tote Neuronen* verursacht werden. Das sind Neuronen, dessen Gewichte und Bias im Laufe des Trainings so gelernt werden, dass die Neuronen nie aktiviert werden und immer 0 ausgeben [65]. Diese 0 wird also jedes Mal weiterpropagiert, aber der durch das tote Neuron zurückfließende Gradient wird ebenfalls 0, da keine Änderung des Signals vorlag. D.h. das Lernen der Gewichte und Bias hört hiermit auf. Als Folge hat das lernende Netz „blinde Flecken“, die keinen Nutzen mehr haben, und ein Lernfortschritt wird nur vorgetäuscht. Besonders begünstigend für tote Neuronen sind solche Aktivierungsfunktionen wie ReLU, weil die untere Grenze ihres Wertebereichs 0 ist und alle negativen Eingangssignale in 0 verwandelt werden (siehe Kapitel 2.1.2). Dieser Effekt von ReLUs wird durch das Dropout nicht verringert [20].

Dementsprechend wurde ein **Test auf tote Neuronen** implementiert, der zur Laufzeit ermittelt, welchen Anteil die toten Neuronen in der jeweiligen Schicht haben. Die Testergebnisse bestätigen die Annahme, dass das Problem nur in den Faltungsnetzen mit ReLUs vorkommt, vgl. Abbildung 15. Die toten Neuronen in Netzen, wo nach jeder

Convolution- und Full-Connection-Schicht ein ReLU folgt, bilden bis zu 40 % der gesamten Neuronenzahl. Ihr Anteil steigt von Layer zu Layer, in den letzten Schichten beträgt er sogar beinahe 99 %, was sehr alarmierend ist. Die Architekturen mit ELU oder Maxout haben dagegen keine toten Neuronen und zeigen stets bessere Fitnesswerte. Sie werden daher bei allen Metaheuristiken über die ReLU-Netze bevorzugt und dominieren ferner in jeder Population.

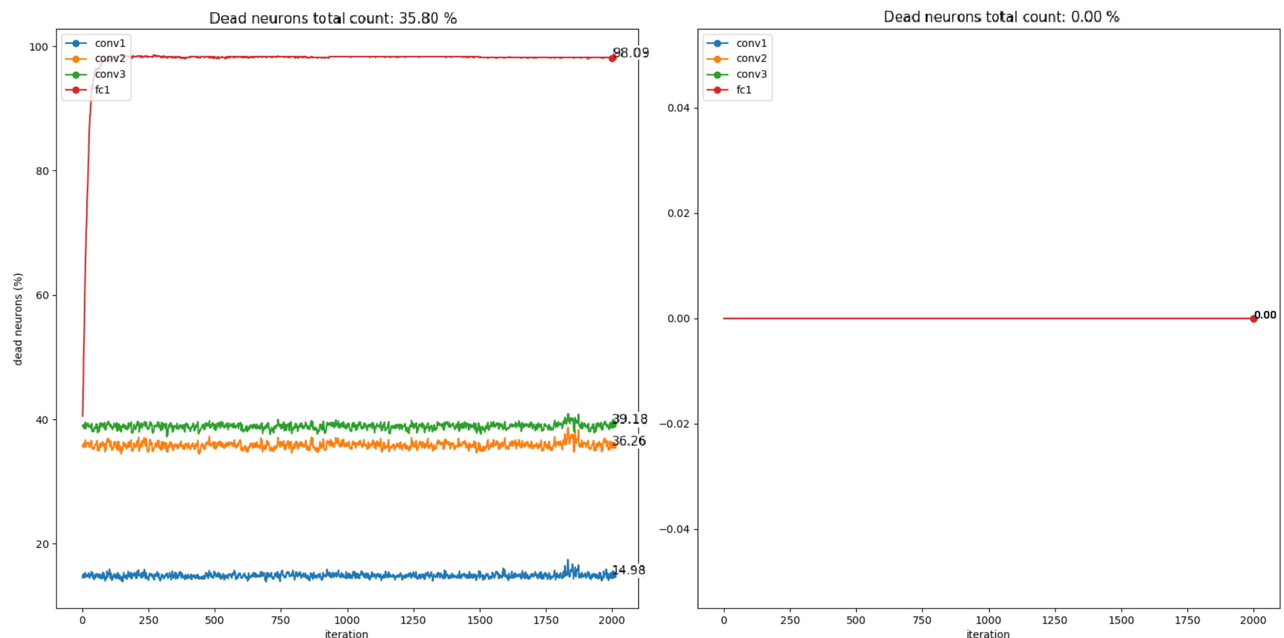

Abbildung 15. Tote Neuronen bei einem Netz mit ReLU (links) und mit ELU (rechts), getestet auf MNIST.

Zusätzlich zu den oben genannten Tests muss sichergestellt werden, dass die verwendeten Trainingsparameter sinnvoll definiert sind und das Training eines „durchschnittlichen" Faltungsnetzes effizient stattfinden kann. Wie im Kapitel 2.3 bereits angesprochen, ist der allerwichtigste von diesen Parametern die initiale Lernrate, die unbedingt optimiert werden muss.

Vor kurzem wurde eine simple Methode vorgeschlagen, wie ein guter Startwert für die Lernrate eines jeden Netzes zu ermitteln ist. Sie nennt sich „**LR range test**" [33] und besteht darin, zunächst eine minimale und eine maximale Lernrate festzulegen, z.B. $\lambda_{min} = 0{,}0001$ und $\lambda_{max} = 0{,}01$. Dann wird das gegebene Netz einige Epochen lang trainiert, und zwar mit einer von λ_{min} bis λ_{max} linear steigenden Lernrate. Bei der Analyse der resultierenden Lernkurve, die die Abhängigkeit der Erkennungsrate von der Lernrate darstellt (wie in Abbildung 16), wird schließlich sichtbar, dass die Erkennungsrate ab einem bestimmten

Punkt nicht mehr steigt, sondern gezackt aussieht bzw. wieder sinkt. Dieser Punkt soll dementsprechend als Startwert für die Lernrate genommen werden (in der Abbildung ist $\lambda \approx 0{,}0005$).

Ein solcher Test wurde separat implementiert und für 30 zufällig generierte Netze je 5 Epochen lang durchgeführt, und zwar noch vor dem Start der Lösungsverfahren. Die endgültige Lernrate wurde als das ungefähre Minimum über die Testergebnisse festgehalten: $\lambda_0 = 0{,}000125$ für MNIST und $\lambda_0 = 0{,}0001$ für CIFAR-10.

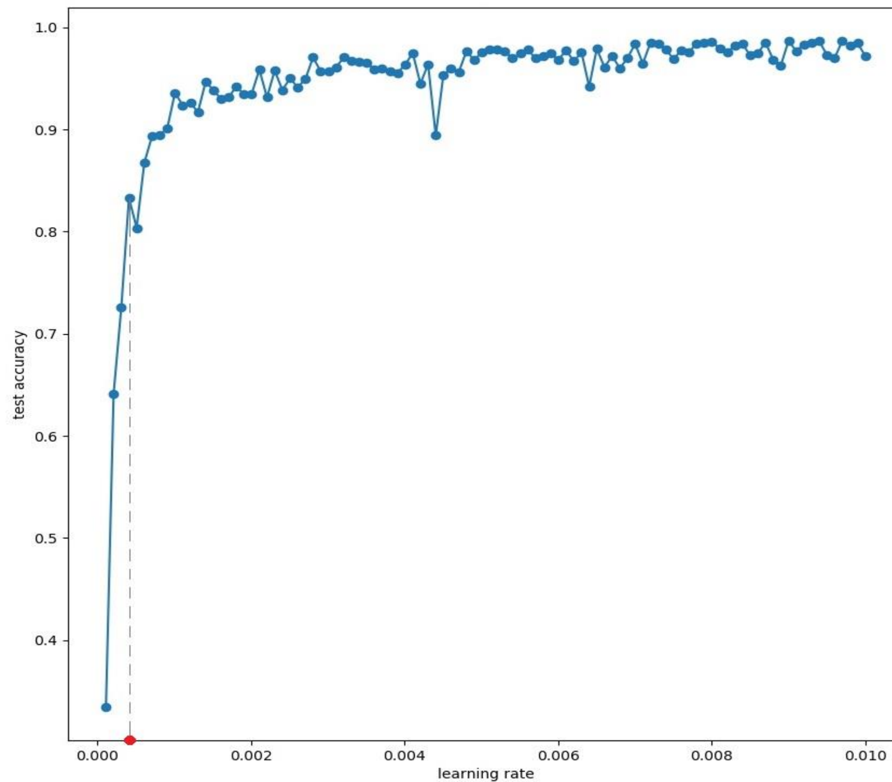

Abbildung 16. "LR range test" für MNIST.

6 Ergebnisse

Nachdem die Implementierung und die Qualitätssicherung der vorgestellten Lösungsansätze abgeschlossen waren, wurde eine Reihe von Testläufen für jedes Verfahren durchgeführt, um sie miteinander vergleichen zu können. Im Folgenden wird die Vorgehensweise bei allen Experimenten sowie die Auswertung der jeweiligen Ergebnisse beschrieben. Im Anschluss wird der beste Lösungsansatz zur Architektursuche bei Faltungsnetzen ermittelt.

6.1 Versuchsaufbau

Wie aus den vorherigen Kapiteln ersichtlich, besteht sowohl jeder Lösungsansatz als auch der Trainingsprozess von einem Faltungsnetz aus vielen variablen Werten, die im Voraus definiert werden müssen. Dabei können diese in fünf Kategorien unterteilt werden – zur Übersicht siehe Tabelle 4, zur Implementierung siehe Anhänge (Kapitel 12).

Einige Parameter sind demzufolge für alle Faltungsnetze fixiert worden, um den Suchraum ausschließlich auf Architekturvariationen zu reduzieren. Zu dieser Kategorie gehören vor allem die Trainingsparameter, weil hier vermutet wird, dass für generierte „durchschnittliche“ Architekturen fester Größe nicht viele entsprechende Variationen notwendig sind. Sobald eine „überdurchschnittliche“ Architektur gefunden wird, können die passenden Trainingsparameter nachträglich konfiguriert werden, anstatt sie für jede einzelne, meist schlechte Architektur zu konfigurieren.

Andere (Meta-)Parameter würden dagegen erwartungsgemäß eine größere Rolle bei der Lösungsqualität der Algorithmen spielen – deren mögliche Werte wurden zunächst manuell getestet. Falls die Wirkung bei diesen Testwerten eindeutig war (wie bei Filtern und verfahrensspezifischen Metaparametern), wurde der beste Wert für alle nachfolgenden Experimente übernommen. Die restlichen Metaparameter (Anzahl der Schichten und Koeffizienten der Fitnessfunktion) wurden bewusst variiert, um ihre Auswirkung auf das Ergebnis festzustellen.

Kategorie	Gültigkeit	Parameter	Wert (MNIST / CIFAR-10)
Metaparameter (für Dataset)	Alle Netze	Dimensionen eines Input-Bildes	28x28 / 32x32
		Anzahl der Farbkanäle	1 / 3
Trainings-parameter	Alle Netze	Random-Seed	0xCAFFE
		Trainingssetgröße A	60.000 / 50.000
		Testsetgröße B	10.000 / 10.000
		Batch-Größe für Trainingsset a	32
		Batch-Größe für Testset b	10
		Anzahl der Trainingsiterationen I	4.000 / 16.000
		Anzahl der Testiterationen	1.000
		Initiale Lernrate λ_0	0,000125 / 0,0001
		Parameter der linearen Lernratensenkung	$\gamma = \frac{1}{I+1}$
		Trägheitskoeffizient μ	0,9
		Weight-Decay-Koeffizient δ	0,0005
		Dropout-Wahrscheinlichkeit	0,5
Metaparameter (für Architektur)	Alle Netze	Anzahl der Schichten insgesamt	**Variabel** (6-8) / 7
		Anzahl der Full Connection Layer	2 / 1
		Filterinitialisierung	**manuelle Auswahl** (Gauß, Xavier, MSRA)
Hyperparameter	Pro Netz	Abfolge von Schichten	-
		Pro Schicht: Layertyp, Non-Linearity-Typ, Pooling-Typ, Filter- und Schrittgröße, Anzahl der Outputs	-
Metaparameter (für Verfahren)	global	Koeffizienten der Fitnessfunktion	**Variabel**: $\alpha_1 = 0{,}25$ und $\beta_1 = 0{,}75$, oder $\alpha_2 = 0$ und $\beta_2 = 1$
		GA und MA, Populationsgröße M	30
		GA und MA, Anzahl Iterationen N	10
		SA, Anzahl Iterationen N	100
		Weitere spezifische Metaparameter	**manuelle Auswahl**

Tabelle 4. Übersicht aller variablen Parameter.

Die Netzgewichte werden mithilfe von SGD aktualisiert, wobei das Trainingsregime für alle Netze konstant bleibt. Das Training erfolgt in insgesamt I Iterationen, und zwar mit einem Testzyklus je 100 bzw. 200 Trainingsiterationen, währenddessen das komplette Testset einmal verarbeitet wird. Nachdem eine oder mehrere Gewinner-Architekturen vom Verfahren geliefert worden sind, erfolgt das **finale Training** der Gewinner mit 20.000 Iterationen (10 Epochen bei MNIST) bzw. 160.000 Iterationen (102 Epochen bei CIFAR-10), währenddessen die bisher konstanten Faltungskerne mittrainiert werden. Die zehnfache Anzahl der Epochen von CIFAR-10 erklärt sich dadurch, dass es ein schwierigeres Dataset als MNIST ist und daher mehr Trainingsaufwand benötigt.

Die passende Fitnessfunktion ist von enormer Bedeutung, denn sie spiegelt die Qualität des Netzes wieder [54]. Sie sollte deswegen sowohl den Trainingsfehler (Loss: $Q_{loss} \in [0; 87{,}3365]$) als auch die Erkennungsrate am Testset (Accuracy: $Q_{acc} \in [0; 1]$) einbeziehen. Dies kann durch folgende Formel ausgedrückt werden:

$$Q_{fit} = -\alpha \cdot Q_{loss} + \beta \cdot Q_{acc}$$

Wie in Kapitel 2.3 besprochen, ist Accuracy ein Qualitätsmerkmal für den späteren Produktivbetrieb des Faltungsnetzes. Sie ergänzt zwar den Loss, ist jedoch aus Benutzersicht viel wichtiger. Daraus folgt, dass $0 \leq \alpha < \beta$ und $\alpha + \beta = 1$. Da es kontinuierlich viele unterschiedliche Kombinationen von α und β gibt, wurden nur die wesentlichen davon betrachtet: $\alpha_1 = 0{,}25$ und $\beta_1 = 0{,}75$ (Loss spielt eine kleine Rolle beim Fitnesswert) sowie $\alpha_2 = 0$ und $\beta_2 = 1$ (Loss spielt keine Rolle). Diese Koeffizienten der Fitnessfunktion werden im Folgenden als *„fitness_1“* und *„fitness_2“* bezeichnet. Der Fitnesswert bei „fitness_1“ liegt im Intervall $[-21{,}834125; 0{,}75]$, bei „fitness_2“ – im Intervall $[0; 1]$.

Sofern nicht explizit angegeben, handelt es sich bei den weiteren Ergebnissen um diejenigen, die sich *mithilfe von MNIST* ergaben. Am Ende des Kapitels wird auf die Tests mit CIFAR-10 und die gelegentlichen Unterschiede zu MNIST eingegangen.

6.2 Auswahl der Metaparameter

Am Anfang der Experimente wurde die Auswirkung der **Filterinitialisierung** auf das Training untersucht. Dazu wurden mehrere Testläufe von GA gestartet, bei den die Filter jeweils mit Gauß-Schema ($\sigma = 0{,}01$), Xavier oder MSRA initialisiert wurden, aber diverse weitere Parameter (Anzahl der Iterationen, Lernrate) variierten. Alle betroffenen Architekturen bestanden aus 7 Schichten, und die Fitnessfunktion war „fitness_1“. Daraus ergab sich, dass alle Testläufe mit Gauß-initialisierten Filtern am schlechtesten abgeschnitten und am längsten gedauert haben: Die Accuracy stieg z.B. von 0,5267 auf 0,7354 nach 10 Iterationen, was eine Woche Rechenzeit in Anspruch nahm. Im Vergleich dazu fingen die Startwerte der Erkennungsraten für Xavier und MSRA bei 0,9 an. Nach 10 Iterationen stiegen sie auf über 0,95 bzw. 0,96, wobei das Training von Xavier-initialisierten Netzen ungefähr doppelt so lange dauerte wie von MSRA (etwa zwei Tage statt einen Tag).

Anhand dieser Erkenntnisse wurde bei weiteren Experimenten auf Gauß- und Xavier-Schemata verzichtet und die Faltungskerne nur mithilfe von MSRA initialisiert. Diese Erfahrung ist im Einklang mit der Tatsache, dass die MSRA-Filter für die Arten der Aktivierungsfunktion konzipiert wurden, die im Versuchsaufbau vorkommen – ReLU, ELU und eventuell auch Maxout. Allerdings kann MSRA zur Divergenz beim Training bestimmter Arten von Faltungsnetzarchitekturen führen, wo Xavier wiederum für Konvergenz sorgt [34]. Es lohnt sich also, in jedem konkreten Fall zu überprüfen, welche Filterauswahl besser zu der jeweiligen Problemstellung passt.

Weitere Untersuchungen widmeten sich den **spezifischen Metaparametern** der Algorithmen**,** die das beste Suchverhalten erzielen würden.

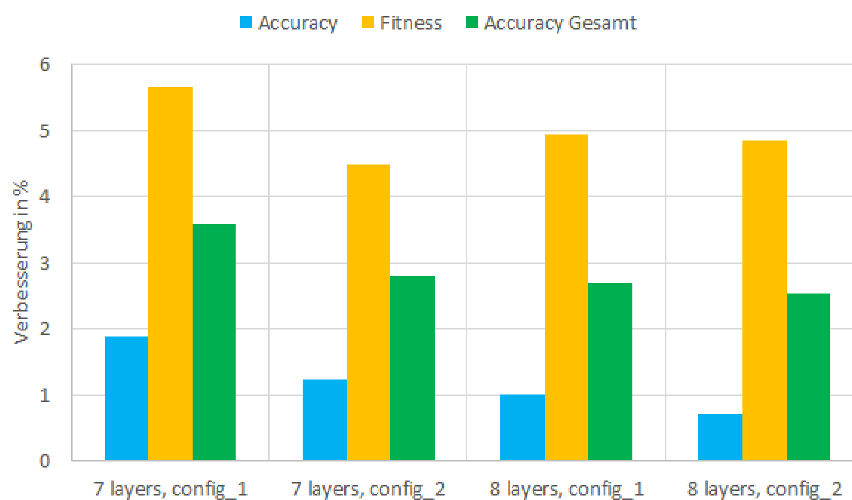

Abbildung 17. GA, Lösungsverbesserung je nach Metaparameterkonfiguration und Anzahl der Schichten.

Im Fall von GA wurden die Metaparameter $p_c = 0{,}5$, $p_m = 0{,}75$ („config_1“) sowie $p_c = 0{,}25$, $p_m = 0{,}5$ („config_2“) erprobt, und zwar für 7- und 8-lagige Architekturen. GA mit „config_2“ selektiert fitteste Individuen, die ein Viertel der Gesamtpopulation bilden, und mutiert anschließend die Hälfte ihrer Nachkommen. D.h. die wenigen dominanten Individuen produzieren mehr Nachwuchs in der nächsten Population. Bei „config_1“ dagegen wird die Hälfte der Individuen selektiert, aber auch die Mutationsrate ist größer, um mehr neues Genmaterial zu erzeugen. Die Tests zeigen den eindeutigen Vorsprung von „config_1“ unabhängig von der Schichtenzahl, siehe Abbildung 17. Sowohl Accuracy als auch Fitness (hier: „fitness_1“) haben bei „config_1“ von der ersten bis zur letzten Iteration einen stärkeren Anstieg als bei „config_2“. Die Verbesserung der Accuracy insgesamt, also das Verhältnis zwischen der initialen populationsbesten Accuracy und der besten Accuracy vom finalen Training, folgt ebenfalls diesem Muster. Tatsächlich generiert GA mit „config_1“

Populationen mit mehr Diversität, wo auch die weniger fitten Individuen eine Überlebenschance haben und wo durch viel Mutation potentiell noch bessere Lösungen entstehen.

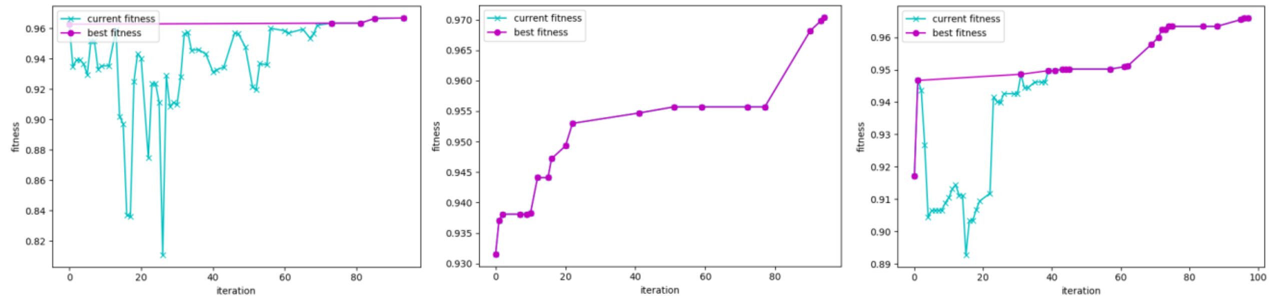

Abbildung 18. SA, Fitnessverlauf für d = 1 (links), d = 0,001 (Mitte) und R = 0,05 (rechts).

Bei SA andererseits musste nur der Radius R und Skalarkoeffizient d variiert werden, denn der im Kapitel 3.2 ausgewählte Abkühlungsplan näherte sich bereits der optimalen Abkühlung bei $\theta = 0{,}9$ und $T_0 = 2$. Wie im gleichen Kapitel ebenfalls erwähnt, sind die Werte $d = 0{,}05$ und $R = 0{,}15$ vermutlich optimal für das gewünschte Suchverhalten von SA. Dennoch wurden auch andere Werte überprüft (vgl. Beispiele in Abbildung 18 und Abbildung 21, rechts):

- ✓ $d = 1$, wie in der ursprünglichen Formel [39], führte zur Verschlechterung der laufenden Performanz, weil die ungünstigeren Architekturen mit sehr hoher Wahrscheinlichkeit bis zu den letzten Iterationen akzeptiert wurden. Dies steigerte den initialen Fitnesswert lediglich um 0,4 %.
- ✓ $d = 0{,}001$ bewirkte, dass gar keine weiteren Architekturen als laufende Lösung akzeptiert wurden und die Suche den Umkreis der initialen Lösung nicht verließ. D.h. SA mit einer solch kleinen Akzeptanzwahrscheinlichkeit agiert wie reine Lokale Suche.
- ✓ $R = 0{,}5$ hätte bis zur Hälfte der Gene in einer Start-Architektur verändern müssen, um ihren Nachbarn zu finden. Dies kann allerdings mehrere Stunden Wartezeit bedeuten oder sogar endlos laufen, da sehr viele entfernte Nachbar-Architekturen nicht valide sind.
- ✓ $R = 0{,}05$ machte den Nachbarschaftskreis sehr klein (z.B. maximal 8 geänderte Gene in einer 6-lagigen Architektur), sodass SA viele kürzere Sprünge im Suchraum gemacht hat. Dies nähert den Charakter von SA wieder zu Lokaler Suche und führt potentiell zu einer eher geringen Fitnessverbesserung.

Basierend auf den obigen Erkenntnissen wurde bei GA die „config_1“, bei SA $d = 0{,}05$ und $R = 0{,}15$ ausgewählt. Die Metaparameter für MA wurden entsprechend der gleichnamigen Metaparameter von GA und SA festgelegt: $N = 10$, $M = 30$, $p_c = 0{,}5$, $p_m = 0{,}75$, $R = 0{,}15$. Der einzige noch nicht belegte Metaparameter von MA ist die Anzahl der Nachbarn, die bei Lokaler Suche evaluiert werden müssen. Es wurde entschieden, diese auf $S = 5$ zu setzen, damit die Verfahrenskomplexität nicht zu sehr im Verhältnis zu GA steigt, aber Lokale Suche eine gewisse Breite hat.

Nun stellt sich die Frage – wie sehen die Netzarchitekturen am Anfang und am Ende des jeweiligen Lösungsverfahrens aus? Wie verändern die Algorithmen mit oben festgelegten Metaparametern eine gegebene Netzarchitektur?

Bei der Betrachtung jeder einzelnen Population von GA kann folgendes bemerkt werden. Je weiter die Entwicklung voranschreitet, desto ähnlicher werden sich die Netzarchitekturen. So wurden bei der ersten Population beispielsweise Filtergrößen von 3 bis 11 beobachtet, die Anzahl der Outputs lag im Intervall von 16 bis 1020, und es kamen sogar die (durchaus legitimen) Layerabfolgen „conv + conv + pool + conv“ vor:

```
[['c',11, 1,114, 'e'], ['p', 3, 1, 'max'], ['c', 7, 3, 520, 'e'], ['p', 2, 1, 'max'],   ['f', 650, 'm'], ['f', 10, '-']]
[['c', 9, 1, 82, 'e'], ['p', 3, 1, 'm+a'], ['c', 3, 1, 276, 'e'], ['p', 3, 2, 'm+a'],   ['f', 682, 'e'], ['f', 10, '-']]
[['c', 3, 1, 16, 'e'], ['c', 7, 1,450,'e'],['p', 2, 2, 'avg'],    ['c', 5, 5,696, 'e'], ['f', 788, 'e'], ['f', 10, '-']]
[['c',11, 2, 22, 'm'], ['p', 2, 1, 'max'], ['c', 3, 1, 358, 'm'], ['c', 3, 3,592, 'm'], ['f', 912, 'm'], ['f', 10, '-']]
[['c', 3, 1,124, 'm'], ['c',11, 2,180,'m'],['p', 3, 1, 'm+a'],    ['c', 3, 3,480, 'm'], ['f', 572, 'm'], ['f', 10, '-']]
[['c', 3, 1, 58, 'e'], ['p', 3, 1, 'max'], ['c', 11, 1, 346, 'e'],['c', 5, 4,472, 'e'], ['f', 476, 'm'], ['f', 10, '-']]
[['c',11, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 3, 1, 390, 'e'], ['p', 3, 2, 'max'],   ['f', 634, 'm'], ['f', 10, '-']]
[['c', 3, 2, 88, 'r'], ['p', 2, 2, 'm+a'], ['c', 3, 1, 148, 'r'], ['p', 2, 1, 'm+a'],   ['f', 332, 'r'], ['f', 10, '-']]
[['c', 5, 2, 78, 'e'], ['p', 2, 1, 'max'], ['c', 5, 2, 174, 'e'], ['p', 2, 2, 'max'],   ['f', 480, 'm'], ['f', 10, '-']]
[['c',11, 1, 38, 'r'], ['p', 3, 1, 'max'], ['c', 3, 1, 504, 'r'], ['p', 2, 2, 'max'],   ['f', 530, 'r'], ['f', 10, '-']]
[['c', 3, 1, 78, 'e'], ['p', 3, 1, 'm+a'], ['c', 7, 5, 512, 'e'], ['p', 2, 2, 'm+a'],   ['f', 918, 'e'], ['f', 10, '-']]
[['c', 7, 2, 38, 'r'], ['p', 2, 1, 'avg'], ['c', 3, 1, 200, 'r'], ['c', 3, 1,364, 'r'], ['f', 884, 'r'], ['f', 10, '-']]
[['c', 3, 1, 98, 'e'], ['p', 2, 2, 'max'], ['c', 3, 2, 386, 'e'], ['p', 3, 2, 'max'],   ['f', 922, 'e'], ['f', 10, '-']]
[['c', 5, 2, 94, 'e'], ['p', 2, 1, 'avg'], ['c', 3, 2, 368, 'e'], ['p', 2, 1, 'avg'],   ['f', 924, 'e'], ['f', 10, '-']]
[['c', 7, 1, 60, 'e'], ['p', 2, 2, 'avg'], ['c', 3, 1, 142, 'e'], ['p', 3, 1, 'avg'],   ['f', 620, 'e'], ['f', 10, '-']]
[['c', 7, 1, 42, 'e'], ['c', 9, 1,436,'e'],['p', 3, 1, 'm+a'],    ['c', 3, 2,500, 'e'], ['f', 982, 'm'], ['f', 10, '-']]
[['c', 9, 1, 72, 'e'], ['p', 2, 1, 'm+a'], ['c', 3, 1, 516, 'e'], ['p', 2, 2, 'm+a'],   ['f', 834, 'm'], ['f', 10, '-']]
[['c', 7, 1,108, 'e'], ['p', 2, 2, 'avg'], ['c', 5, 1, 266, 'e'], ['p', 3, 1, 'avg'],   ['f', 430, 'e'], ['f', 10, '-']]
[['c', 3, 1,106, 'e'], ['p', 2, 1, 'avg'], ['c', 9, 2, 148, 'e'], ['c', 3, 3,206, 'e'], ['f', 736, 'm'], ['f', 10, '-']]
[['c', 7, 1, 78, 'e'], ['p', 2, 2, 'max'], ['c', 3, 1, 486, 'e'], ['p', 3, 1, 'max'],   ['f', 578, 'e'], ['f', 10, '-']]
[['c', 5, 2, 20, 'e'], ['p', 3, 1, 'm+a'], ['c', 5, 1, 190, 'e'], ['p', 3, 2, 'm+a'],   ['f',1020, 'm'], ['f', 10, '-']]
[['c', 3, 2, 70, 'm'], ['p', 2, 1, 'max'], ['c', 3, 1, 462, 'm'], ['p', 3, 1, 'max'],   ['f', 710, 'm'], ['f', 10, '-']]
[['c', 9, 1, 20, 'r'], ['p', 2, 1, 'max'], ['c', 9, 3, 452, 'r'], ['p', 2, 1, 'max'],   ['f', 810, 'r'], ['f', 10, '-']]
[['c', 5, 1, 60, 'm'], ['p', 3, 1, 'm+a'], ['c', 9, 1, 280, 'm'], ['p', 2, 2, 'm+a'],   ['f', 942, 'm'], ['f', 10, '-']]
[['c', 7, 1, 18, 'e'], ['p', 2, 1, 'avg'], ['c', 7, 1, 106, 'e'], ['p', 3, 2, 'avg'],   ['f', 304, 'm'], ['f', 10, '-']]
[['c', 7, 1, 26, 'r'], ['c', 3, 3,226,'r'],['p', 2, 1, 'avg'],    ['c', 3, 1,618, 'r'], ['f',1016, 'r'], ['f', 10, '-']]
[['c', 5, 1, 16, 'r'], ['p', 3, 1, 'avg'], ['c', 9, 2, 404, 'r'], ['p', 2, 2, 'avg'],   ['f', 632, 'r'], ['f', 10, '-']]
[['c', 3, 1, 86, 'r'], ['p', 3, 1, 'avg'], ['c', 5, 2, 362, 'r'], ['p', 3, 1, 'avg'],   ['f', 670, 'r'], ['f', 10, '-']]
[['c', 7, 2, 32, 'e'], ['p', 2, 1, 'max'], ['c', 3, 2, 468, 'e'], ['p', 2, 1, 'max'],   ['f', 738, 'm'], ['f', 10, '-']]
[['c', 5, 1, 40, 'r'], ['p', 3, 1, 'm+a'], ['c', 7, 1, 242, 'r'], ['p', 2, 1, 'm+a'],   ['f', 488, 'r'], ['f', 10, '-']]
```

Diese Diversität wurde dann Schritt für Schritt eliminiert, sodass die erste Feature-Extraction-Phase in der Iteration 10 bei fast allen Netzen gleich war; die Layerabfolge war nun fest und es kamen lediglich wenige Änderungen in der Anzahl der Outputs zustande:

```
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 570, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 568, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 608, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 610, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 462, 'e'], ['p', 3, 1, 'max'], ['f', 912, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 462, 'e'], ['p', 3, 1, 'max'], ['f', 916, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 398, 'e'], ['p', 3, 2, 'max'], ['f', 634, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 638, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 624, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 640, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 706, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 636, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 642, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 630, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 708, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 824, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 386, 'e'], ['p', 3, 2, 'max'], ['f', 608, 'm'], ['f', 10, '-']]
[['c', 3, 1, 54, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 618, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 3, 462, 'e'], ['p', 3, 1, 'max'], ['f', 918, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 656, 'm'], ['f', 10, '-']]
[['c',11, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 398, 'e'], ['p', 3, 2, 'max'], ['f', 634, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 572, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 762, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 634, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 392, 'e'], ['p', 3, 2, 'max'], ['f', 634, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 626, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 393, 'e'], ['p', 3, 2, 'max'], ['f', 634, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 890, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 710, 'm'], ['f', 10, '-']]
[['c', 3, 1, 52, 'e'], ['p', 3, 1, 'max'], ['c', 7, 1, 390, 'e'], ['p', 3, 2, 'max'], ['f', 576, 'm'], ['f', 10, '-']]
```

D.h. der Suchbereich wird immer kleiner, bis die Architekturveränderung nur noch durch Mutation möglich ist und die Fitnessfunktion ein Plateau erreicht. Auch die Kurven von Accuracy verdeutlichen diesen Trend (siehe Abbildung 19): Während der ersten Iteration von GA sind sie zerstreut, am Ende dagegen sehr dicht beieinander.

Interessanterweise findet sich diese Tendenz bei den Architekturen im Verlauf von SA wieder. Anhand der folgenden Liste der akzeptierten Architekturen wird deutlich, dass in den ersten Iterationen (siehe Anfang der Liste) mehr Varianz stattfindet, da größere Sprünge im Suchraum möglich sind. Später (Ende der Liste) werden nur kleinere Änderungen am Chromosom akzeptiert, die den Fitnesswert tatsächlich verbessern:

```
[['c',11, 1, 94, 'e'], ['p', 2, 1, 'm+a'], ['c', 7, 1, 498, 'e'], ['p', 3, 2, 'm+a'], ['f', 630, 'e'], ['f', 10, '-']]
[['c',11, 1, 58, 'e'], ['p', 2, 1, 'm+a'], ['c', 7, 1, 496, 'e'], ['p', 3, 2, 'm+a'], ['f', 628, '-'], ['f', 10, '-']]
[['c', 7, 1,122, 'e'], ['p', 3, 1, 'avg'], ['c', 3, 3, 496, 'm'], ['p', 3, 2, 'm+a'], ['f', 574, '-'], ['f', 10, '-']]
[['c', 7, 1,106, 'm'], ['p', 3, 1, 'm+a'], ['c', 3, 3, 500, 'm'], ['p', 3, 2, 'm+a'], ['f', 564, '-'], ['f', 10, '-']]
[['c', 7, 1,106, 'm'], ['p', 3, 1, 'avg'], ['c', 3, 3, 180, 'm'], ['p', 3, 2, 'm+a'], ['f', 560, '-'], ['f', 10, '-']]
[['c', 7, 1,104, 'e'], ['p', 2, 1, 'm+a'], ['c', 3, 3, 180, 'm'], ['p', 3, 2, 'm+a'], ['f', 560, '-'], ['f', 10, '-']]
[['c', 7, 1,104, 'e'], ['p', 2, 1, 'm+a'], ['c', 3, 3, 180, 'e'], ['p', 3, 2, 'm+a'], ['f', 816, '-'], ['f', 10, '-']]
[['c', 7, 1,104, 'e'], ['p', 2, 1, 'm+a'], ['c', 3, 3, 308, 'e'], ['p', 3, 2, 'm+a'], ['f',1012, '-'], ['f', 10, '-']]
[['c', 7, 1, 40, 'e'], ['p', 2, 1, 'max'], ['c', 3, 3, 816, 'e'], ['p', 3, 2, 'm+a'], ['f', 868, '-'], ['f', 10, '-']]
[['c', 7, 1,104, 'e'], ['p', 3, 1, 'max'], ['c', 3, 3, 784, 'e'], ['p', 3, 2, 'm+a'], ['f', 876, '-'], ['f', 10, '-']]
[['c', 3, 1, 96, '-'], ['p', 3, 1, 'max'], ['c', 7, 2, 272, 'e'], ['p', 3, 2, 'm+a'], ['f', 996, '-'], ['f', 10, '-']]
[['c', 3, 1,112, '-'], ['p', 3, 1, 'max'], ['c', 7, 2, 272, 'e'], ['p', 3, 2, 'm+a'], ['f', 484, '-'], ['f', 10, '-']]
[['c', 3, 1, 48, 'e'], ['p', 3, 1, 'max'], ['c', 7, 2, 400, 'e'], ['p', 3, 2, 'm+a'], ['f', 486, '-'], ['f', 10, '-']]
[['c', 3, 1, 50, 'e'], ['p', 3, 1, 'max'], ['c', 7, 2, 384, 'm'], ['p', 3, 2, 'm+a'], ['f', 486, '-'], ['f', 10, '-']]
[['c', 3, 1, 48, 'e'], ['p', 3, 1, 'max'], ['c', 3, 2, 384, 'm'], ['p', 3, 2, 'max'], ['f', 996, '-'], ['f', 10, '-']]
[['c', 3, 1, 48, 'e'], ['p', 3, 1, 'm+a'], ['c', 3, 2, 384, 'm'], ['p', 3, 2, 'max'], ['f', 996, '-'], ['f', 10, '-']]
[['c', 3, 1, 48, 'e'], ['p', 3, 1, 'm+a'], ['c', 3, 2, 256, 'm'], ['p', 3, 2, 'max'], ['f', 996, '-'], ['f', 10, '-']]
[['c', 3, 1, 16, 'e'], ['p', 3, 1, 'max'], ['c', 3, 2, 624, 'm'], ['p', 3, 2, 'max'], ['f', 996, '-'], ['f', 10, '-']]
[['c', 3, 1, 16, 'e'], ['p', 3, 1, 'max'], ['c', 3, 2, 624, 'r'], ['p', 3, 2, 'max'], ['f', 996, '-'], ['f', 10, '-']]
```

```
[['c', 7, 1, 16, '-'], ['p', 2, 1, 'max'], ['c', 3, 2, 624, 'm'], ['p', 3, 2, 'max'], ['f', 996, '-'], ['f', 10, '-']]
[['c', 7, 1, 16, '-'], ['p', 2, 1, 'max'], ['c', 3, 2, 624, 'e'], ['p', 3, 2, 'max'], ['f', 996, '-'], ['f', 10, '-']]
[['c', 7, 1, 16, '-'], ['p', 2, 1, 'max'], ['c', 3, 2, 760, 'e'], ['p', 3, 2, 'max'], ['f', 996, '-'], ['f', 10, '-']]
[['c', 7, 1, 16, '-'], ['p', 2, 1, 'm+a'], ['c', 3, 2, 760, 'e'], ['p', 3, 2, 'max'], ['f',1012, '-'], ['f', 10, '-']]
[['c', 7, 1, 16, '-'], ['p', 2, 1, 'm+a'], ['c', 3, 2, 760, 'e'], ['p', 3, 2, 'max'], ['f',1014, '-'], ['f', 10, '-']]
[['c', 7, 1, 16, '-'], ['p', 2, 1, 'm+a'], ['c', 3, 2, 760, 'e'], ['p', 3, 2, 'max'], ['f',1016, '-'], ['f', 10, '-']]
```

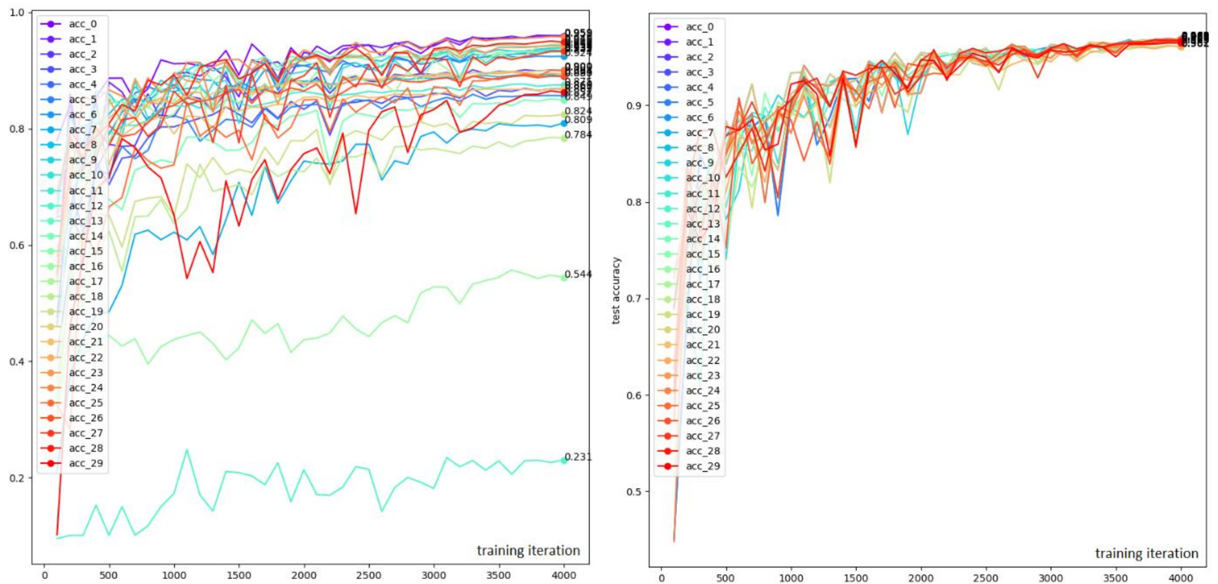

Abbildung 19. GA, Verlauf der Accuracy pro Faltungsnetz in der Iteration 1 (links) und 10 (rechts).

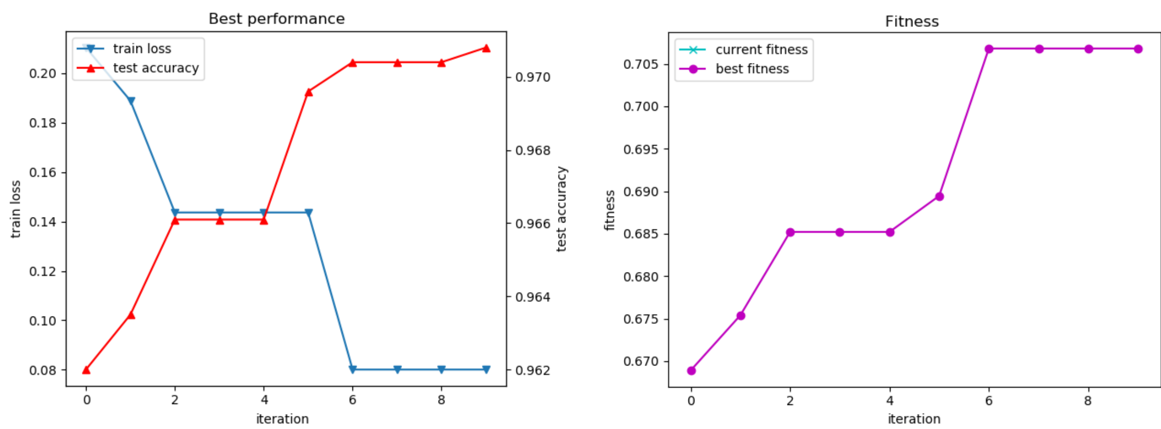

Abbildung 20. GA, beste Performanz in jeder Iteration (links), Verlauf der populationsbesten Fitness (rechts).

Für alle Lösungsansätze wäre es außerdem wissenswert, wie sich die **beste bisher gefundene Lösung** im Laufe der Zeit entwickelt. Im Fall von GA ist sie das fitteste Individuum der Population, der dank dem Elitismus-Prinzip immer in die nächste Iteration übernommen wird, sodass die Lösungsqualität nicht mehr sinken kann, wie in Abbildung 20 dargestellt.

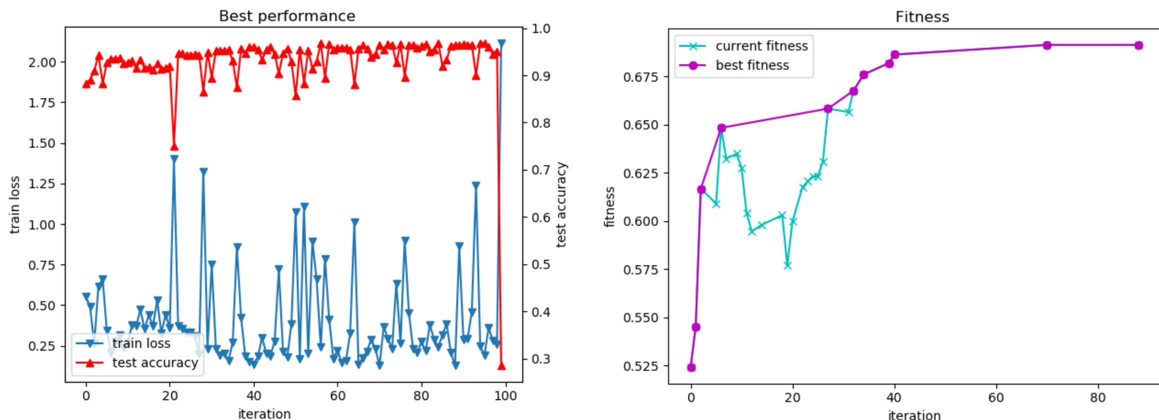

Abbildung 21. SA, Performanz jedes trainierten Netzes (links), Verlauf der aktuellen und besten Fitness (rechts).

Für SA muss sowohl die beste bisher gefundene Lösung, als auch die aktuelle Lösung verfolgt werden, siehe Abbildung 21 rechts. Letztere kann zwar minderwertig sein, ist aber die Ausgangsbasis für die nächsten Lösungen, die sie qualitativ übertreffen können. In Abbildung 21 links wird sichtbar, wie viel Diversität bei Fitnesswerten von SA möglich ist, obwohl es mit einer einzigen Lösung pro Iteration statt mit einer Lösungsmenge arbeitet.

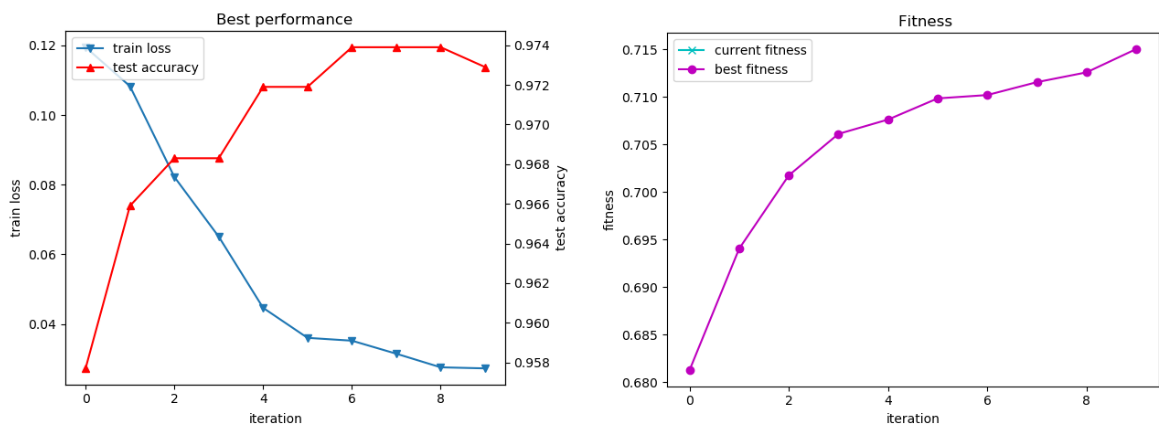

Abbildung 22. MA, beste Performanz in jeder Iteration (links), Verlauf der populationsbesten Fitness (rechts).

Während die populationsbeste Fitness bei GA oft Plateaus bildet (wie in Abbildung 20 rechts), sodass keine Fitnessverbesserung im Laufe mehrerer Iterationen vorliegt, zeigt MA einen streng monotonen Fitnessanstieg, siehe Abbildung 22 rechts. Lokale Suche ist offensichtlich in der Lage, das fitteste Individuum aus der letzten Population durch einen noch fitteren zu ersetzen. Aus der linken Grafik der Abbildung 22 ist ersichtlich, dass dank „fitness_1" die Verbesserung zum Teil durch die Senkung von Loss bei gleichbleibender Accuracy erreicht wird, z.B. bei Iteration 6 bis 8.

Eine weitere experimentell bestätigte Eigenschaft von MA ist die Populationsdiversität, die stärker als bei GA ausgeprägt ist und sich auf die Verteilung von Accuracy und Loss innerhalb einer Population auswirkt. So wird beispielsweise das Intervall zwischen der kleinsten und der größten Accuracy der Population im Laufe des Algorithmus zwar schrumpfen, weil die Architekturen in bestimmten Suchrichtungen bevorzugt werden, dennoch bleibt bis zur letzten Iteration eine bestimmte Breite des Intervalls erhalten (vgl. Abbildung 19 und Abbildung 23).

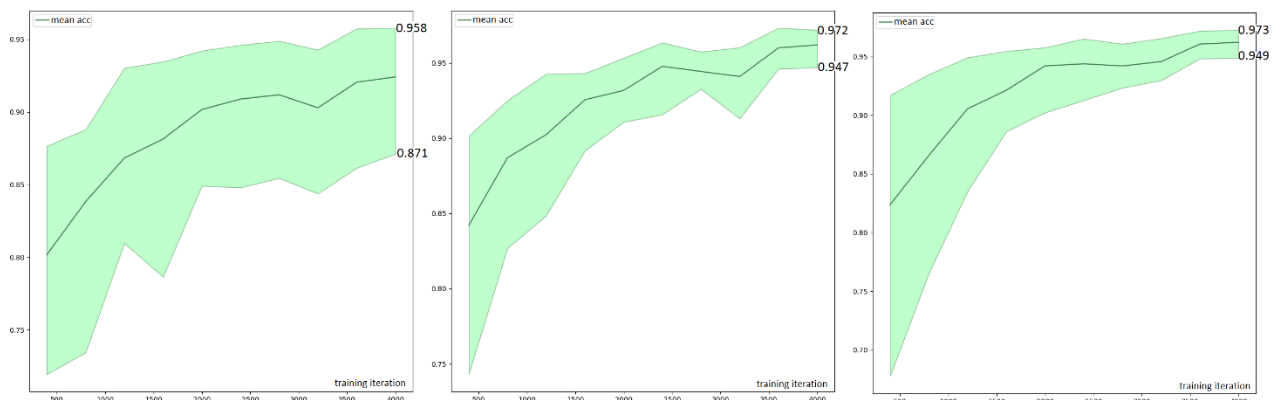

Abbildung 23. MA, Verteilung der Accuracy innerhalb der Population in der Iteration 1 (links), 5 (Mitte) und 10 (rechts).

Letztendlich stellt sich die Frage, ob mehr Iterationen N eine merkliche Verbesserung der finalen Lösung bringen würden. Die Tests geben eine eindeutig negative Antwort: Beim Versuch, die drei Verfahren länger laufen zu lassen, hat es in allen Fällen ein Plateau im Verlauf der besten Fitness gegeben. Das deutet darauf hin, dass die ausgewählten Metaparameter zu den Werten von N und M aus Tabelle 4 passend konfiguriert sind, damit das jeweilige Verfahren konvergieren kann. Hierzu sollte bedacht werden, dass die Lösungsoptimierung von GA und MA problemlos ab der letzten Population weiterlaufen kann. Bei SA dagegen erreicht die Temperatur am Ende der Ausführung den Wert 0, was weitere Abkühlung unmöglich macht. Um SA demnach mit einer größeren Anzahl Iterationen laufen zu lassen, muss der Abkühlungsplan vorher angepasst werden.

Dank des Einsatzes einer GPU haben die Lösungsverfahren insgesamt gute Laufzeiten gezeigt. Die mittlere Trainingszeit eines Faltungsnetzes betrug ca. 10 Minuten auf MNIST und 20 Minuten auf CIFAR-10. So lief ein Testlauf von GA mit MNIST bis zu zwei Tage (300 Netze), von SA – unter 24 Stunden (100 Netze), und von MA – über eine Woche (1500 Netze).

6.3 Experimente mit MNIST und Verfahrensvergleich

Um zu beurteilen, wie gut ein Lösungsansatz seinen Zweck erfüllt, gibt es mehrere **Kennzahlen**. Der allerwichtigste von denen ist die finale Erkennungsrate (Accuracy), die von der jeweiligen Gewinner-Architektur nach dem finalen Training geleistet wird. Diese Kennzahl ist für den zukünftigen produktiven Betrieb einer Netzarchitektur von großer Bedeutung. Da die Leistungsfähigkeit eines Verfahrens jedoch stark von der initialen Lösung bzw. Population abhängig ist, muss zudem die Fähigkeit zu deren Verbesserung eingeschätzt werden. In diesem Zuge kann als zweite Kennzahl die Accuracy-Verbesserung von der ursprünglichen Lösung bzw. besten Lösung der Population im Vergleich zur finalen Accuracy genommen werden. Andererseits, arbeitet jedes Lösungsverfahren statt mit Accuracy ausschließlich mit der Fitnessfunktion, die eventuell auch Loss berücksichtigt (wie „fitness_1"), deswegen ist die Fitnessverbesserung ebenfalls als Kennzahl aussagekräftig.

Am Ende zählt auch die Laufzeit zu den möglichen Kennzahlen eines Lösungsverfahrens, obwohl sie abhängig von Trainingszeiten und Größe jedes einzelnen Faltungsnetzes ist. So gilt ein Netz als „klein", wenn es aus wenig Schichten besteht, große Filter hat (welche die Feature-Maps der kleineren Dimensionen erzeugen) und stets eine niedrige Anzahl von Output-Feature-Maps pro Schicht aufweist. Ein „kleines" Netz mit 6 Schichten und oben genannten Trainingsparametern kann z.B. innerhalb von 1,5 Minuten trainiert werden. Der Grund dafür ist die niedrige Anzahl der trainierbaren Parameter, was das Training beschleunigt. Wenn SA mit einem „kleinen" Netz startet, wird er bereits in ca. 2,5 Stunden erfolgreich beenden, weil sich in der Nachbarschaft des ursprünglichen Netzes ebenfalls „kleine" Netze befinden. Ein Lauf von GA dagegen, bei welchem die Population aus „großen" 6-lagigen Netzen (mit vielen Output-Feature-Maps und geringen Filtergrößen) besteht, würde 15 Stunden in Anspruch nehmen. Aufgrund dieser Diskrepanz kann die absolute Laufzeit der Algorithmen nicht als objektive Kennzahl gewählt werden.

Aus der Sicht des Anwenders spielt die Trainingszeit eines Netzes ohnehin eine untergeordnete Rolle, vielmehr ist die Komplexität des jeweiligen Algorithmus von Interesse. Komplexität wird hier als Anzahl der Fitnessevaluierungen definiert, d.h. die gesamte Anzahl der Netze, die trainiert werden müssen. Diese ist in Abhängigkeit zur Anzahl der Iterationen N und ggf. Populationsgröße M. Wie in Tabelle 2 bereits aufgezählt, hat SA die kleinste Komplexität – sie wächst linear zu N. MA dagegen ist der komplexeste von den drei Algorithmen, weil hier pro Iteration nicht nur jedes Individuum aus der Population, sondern auch seine Nachbarn (im schlimmsten Fall M davon) evaluiert werden müssen.

Aus diesen Überlegungen werden nun die drei implementierten Lösungsansätze mithilfe folgender Kriterien bzw. Kennzahlen verglichen:

- Beste finale Accuracy (nach dem finalen Training)
- Verbesserung (%) der initialen Accuracy auf die finale Accuracy
- Verbesserung (%) von Fitness im Laufe des Verfahrens (vgl. erste und letzte Iteration)

Alle Lösungsverfahren haben zwei Metaparameter gemeinsam – Koeffizienten der Fitnessfunktion und Anzahl der Schichten in einer Netzarchitektur. Es muss also untersucht werden, wie sich die Verfahren je nach Wahl dieser Metaparameter verhalten und ob diese die resultierenden Kennzahlen der Algorithmen eventuell beeinflussen.

So fällt anhand Abbildung 22 (letzte Iteration) auf, dass ein Faltungsnetz mit der höchsten Erkennungsrate nicht unbedingt das fitteste ist. Im Allgemeinen kann es sogar passieren, dass das Netz mit der besten Accuracy nicht in die nächste Population bzw. Iteration übernommen wird, weil es einen großen Loss-Wert hat. Deswegen musste überprüft werden, welche Koeffizientenwerte in der Fitnessfunktion zu den besseren Lösungen führen – „fitness_1" oder „fitness_2". Der zweite Metaparameter, die Anzahl der Schichten, die im Versuchsaufbau für MNIST von 6 bis 8 beträgt, könnte ebenso eine Auswirkung auf das finale Ergebnis haben, weil das Optimum sich in Bezug auf das gegebene Dataset zwischen solchen Architekturen befinden kann. Somit ergaben sich jeweils 6 Kombinationen dieser zwei Metaparameter, die als Grundlage für weitere Experimente dienten.

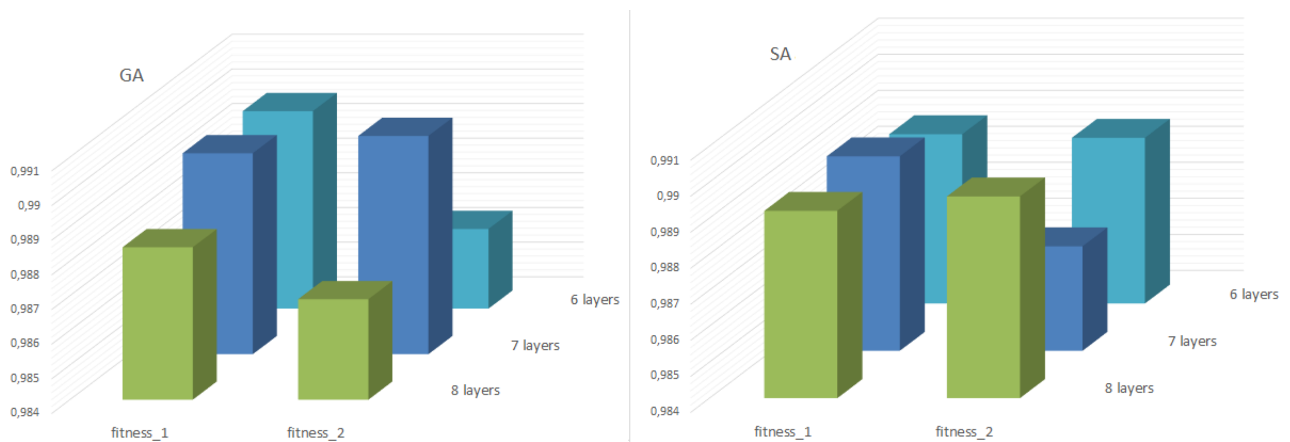

Abbildung 24. Beste finale Accuracy bei GA und SA.

In Abbildung 24 ist zu sehen, welche **finale Accuracy** sich je nach Layeranzahl und gewählter Fitnessfunktion ergeben hat. Hier handelt es sich um das beste Ergebnis über alle durchgeführten Testläufe. Dank „fitness_1" werden sowohl bei GA als auch bei SA stets bessere Architekturen geliefert, was dafür spricht, dass Loss ein ebenso wichtiges

Qualitätsmerkmal für ein Faltungsnetz ist wie Accuracy. Eine Ausnahme wird von einer 7-lagigen Architektur verursacht mit einer Accuracy von 0,9903 (=99,03 %, also Fehlerrate 0,97 %), die von GA berechnet wurde und sein bestes Ergebnis darstellt. Die zweite Ausnahme ist eine 8-lagige Architektur mit der Accuracy 0,9899, das beste Ergebnis von SA.

Mit anderen Worten, trotz einer gewissen Wahrscheinlichkeit, eine optimale Architektur mithilfe von „fitness_2“ zu finden, sind die meisten durch „fitness_1“ gelieferten Architekturen besser. Es ist dagegen kein Zusammenhang zwischen der Schichtenanzahl und der besten finalen Accuracy zu erkennen.

Die durchschnittliche finale Accuracy von GA und SA beträgt ca. 0,9886, und die Unterschiede liegen weit im Nachkommastellenbereich. Dies spricht dafür, dass beide Verfahren sehr gute Architekturen liefern und dass weitere Kennzahlen für den Vergleich nötig sind.

Abbildung 25. Verbesserung von Fitness in % bei GA und SA.

Das nächste Vergleichskriterium ist die Fähigkeit des Algorithmus, die **Fitness** der initialen Lösung bzw. Menge der Lösungen **zu verbessern**, siehe Abbildung 25. Auch hier gilt die Tatsache, dass „fitness_1“ gegenüber „fitness_2“ bevorzugt werden muss: In allen Fällen brachte „fitness_1“ einen größeren Anstieg der Fitness mit sich. Die Fitnessverbesserung durch reine Accuracy fällt erwartungsgemäß kleiner aus als durch Accuracy und Loss, weil das Intervall der möglichen Loss-Werte und damit auch das Verbesserungspotential von Loss größer ist.

Bei GA liegen die Verbesserungspotentiale unter der Verwendung derselben Fitnessfunktion dicht beieinander, d.h. GA liefert eine stabile Fitnessverbesserung unabhängig von der Schichtenanzahl. Bei SA (vgl. Abbildung 25 rechts) liegt ein anderer Trend vor: Je mehr Schichten ein Faltungsnetz hat, desto unwahrscheinlicher ist es, dass SA ein deutlich besseres Netz findet. Ein möglicher Grund dafür besteht darin, dass der Suchraum bei 8-lagigen Architekturen größer ist, und die lokalen Optima vermutlich breiter gestreut sind. Daher ist

es weniger wahrscheinlich, dass sich in der Nachbarschaft eines Netzes genügend bessere Netze befinden, auf die SA stoßen kann. GA dagegen sucht nicht nur die Nachbarschaft eines Netzes durch (Mutation), sondern macht beliebige Sprünge anhand der inneren Netzstruktur (Crossover), was sich in jedem Suchraum als hilfreich erweist.

Eine weitere Auffälligkeit betrifft den Zusammenhang zwischen der Fitnessverbesserung und dem initialen Fitnesswert. So enthielt die erste Population von GA immer mindestens eine Lösung, deren Accuracy über 0,9 lag. Dadurch bekommt GA stets eine gute Basislösung, die nur noch verbessert werden muss. Dies ist für SA nicht zwangsläufig der Fall – es gab Testläufe, wo SA mit einer Accuracy von 0,7 anfing. Umso erstaunlicher ist es, dass auch in solchen Fällen SA im Stande ist, die schlechte Lösung deutlich zu verbessern, gelegentlich um bis zu 32 %.

Aus diesen Beobachtungen folgt, dass SA in kleineren Suchräumen mehr Fitnessverbesserung bewirken kann als GA, in größeren ist dagegen GA erfolgreicher. Außerdem ist SA mehr von der initialen Lösung abhängig und weist deswegen keine stabile Fitnessverbesserung auf. Diese Tatsache kann durch folgende *Allegorie* illustriert werden: Der Verlauf einer Fitnessfunktion bildet eine „Fitnesslandschaft" im Suchraum, die aus mehreren „Hügeln" diverser Höhe besteht. Wenn SA an einem beliebigen Punkt im Tal startet, wo es in der Nähe einen großen Hügel gibt, dann kann SA seine Position radikal verbessern. Sollte SA jedoch auf der Flanke eines kleineren Hügels anfangen, bringt der Aufstieg nicht so viel. Bei einem zufälligen Punkt kann SA dementsprechend keine Garantie über die Größe des Aufstiegs geben. Im Gegensatz dazu wählt GA gleichzeitig viele Punkte auf der Landschaft, und mindestens einer davon wird sich wahrscheinlich in der Nähe oder sogar auf einem mittelgroßen Hügel befinden. Das macht GA von den Startpunkten weniger abhängig als SA.

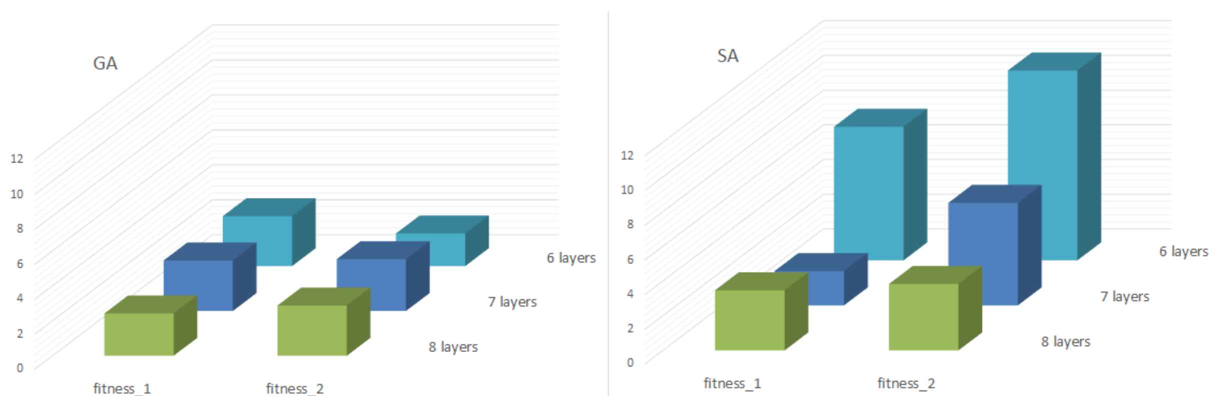

Abbildung 26. Verbesserung von Accuracy in % bei GA und SA.

Das letzte Vergleichskriterium ist das Potential des Lösungsansatzes, die initiale Accuracy (erste Iteration) bis zur finalen **Accuracy** (finales Training nach der letzten Iteration) **zu steigern**. In Abbildung 26 wird wieder verdeutlicht, dass bei GA die Unterschiede in dieser

Hinsicht geringer ausfallen. Durchschnittlich verbessert GA die ursprüngliche Accuracy (wie oben erwähnt, etwa ~0,9) um 2,6 %, und zwar bei jeder Fitnessfunktion.

SA zeigt wiederum größere Diskrepanzen je nach Schichtenanzahl, und zwar nochmals aufgrund der Suchraumgröße. Im Vergleich zur Fitnessverbesserung fällt bei SA außerdem auf, dass „fitness_2“ die höhere Accuracy-Verbesserung verursacht. Tatsächlich ist der Anstieg von Accuracy bei „fitness_1“ geringer, weil er sich teilweise durch Losssenkung begründet, die von „fitness_2“ ignoriert würde.

SA empfiehlt sich also als Verfahren der Wahl, falls eine große Verbesserung von Accuracy oder Fitness in einem kleinen Suchraum erwünscht ist. Dabei bewirkt „fitness_1“ bei SA stets einen höheren Anstieg der Fitness, „fitness_2“ – einen höheren Anstieg der Accuracy. Dennoch ist SA gegenüber der Initiallösung sehr empfindlich. GA erfüllt sein Zweck recht stabil und von der Suchraumgröße unabhängig, startet mit einem breiten Spektrum an Lösungen, unter denen sich bereits gute Lösungen befinden.

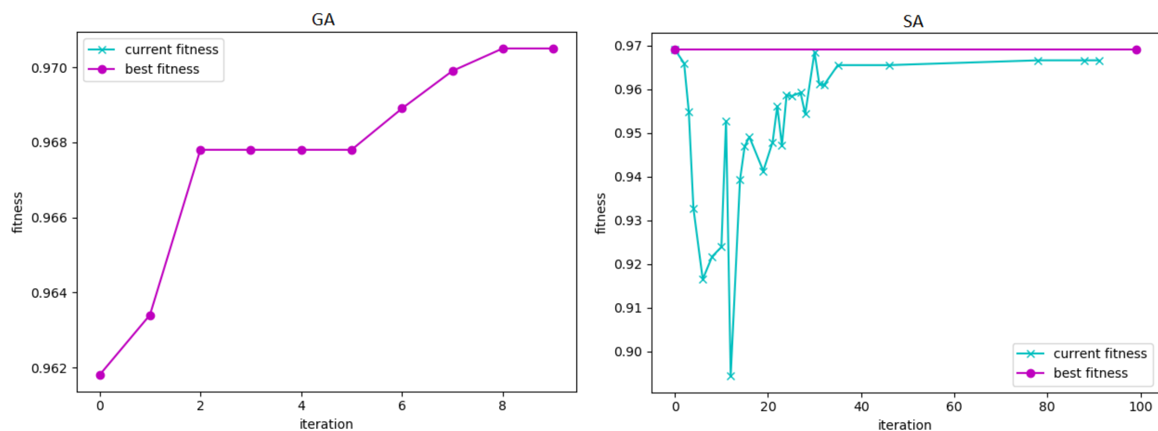

Abbildung 27. Fitnessverlauf von GA (links) mit anschließender Anwendung von SA (rechts).

Diese Empfehlung wirft die Frage auf, ob die Vorteile beider Verfahren miteinander kombiniert werden können, indem die beste von GA gelieferte Architektur als Input für SA benutzt wird. Dies wurde z.B. für eine der besten bisher gefundenen Architekturen mit 7 Schichten und der finalen Accuracy von 0,9903 getestet (vgl. Tabelle 5).

Der Fitnessverlauf vom zuerst ausgeführten GA ist in Abbildung 27 links dargestellt. Die Gewinner-Architektur erreichte in der letzten Population den Fitnesswert 0,9691, der im Laufe von SA nicht gesteigert werden konnte (vgl. Abbildung 27 rechts). Mit anderen Worten, SA war nicht in der Lage, in der Nachbarschaft von der gegebenen Architektur eine bessere zu finden. Da sowohl die Fitnessfunktion als auch der Suchraum durch die vorgegebenen Regeln für den Architekturaufbau beschränkt sind, könnte die nicht gefundene bessere

Lösung ein Indiz dafür sein, dass die Lösung von GA eventuell ein lokales Optimum ist. Ein Hinweis auf das globale Optimum liegt allerdings nicht vor, obwohl dessen Existenz durch die genannte Beschränktheit begründet ist.

Eine weitere Art des Vergleichs mehrerer Algorithmen besteht darin, für sie die gleichen Vorbedingungen zu schaffen. In diesem Fall sind es die gleichen Populationen bzw. Lösungen, mit denen die Verfahren initialisiert werden. Zu diesem Zweck wurde eine zufällige Population generiert und an GA und MA als initiale Population übergeben. Da beide Verfahren die Suchrichtung eines besten Individuums der Population annehmen, wurde es ausgewählt und als die Startlösung für SA benutzt.

Als Referenzverfahren für diesen letzten Vergleich, welches die in der Wissenschaft typische Vorgehendweise bei Hyperparameteroptimierung für Faltungsnetze darstellt, wurde zusätzlich die Random-Suche (**RS**) implementiert. Sie verfügt über keine spezifischen Metaparameter außer Iterationszahl N (die wie bei SA auf 100 festgelegt wurde) und hat die gleiche Komplexität wie SA.

Die Ergebnisse des Vergleichs mit gleichen Vorbedingungen sind in Abbildung 28 präsentiert. Die Anzahl der Schichten betrug hier 7, als Fitnessfunktion diente „fitness_1“. Die besten Ergebnisse wurden demnach in allen Fällen durch MA erreicht. Seine beste Architektur erreichte eine finale Accuracy von 0,9897 und war um etwa 15 % fitter als die beste Architektur in der initialen Population. Das Ergebnis von GA fiel erwartungsgemäß bescheidener aus: Trotz der Fitnessverbesserung von etwa 7,8 % lieferte er lediglich die zweitbeste Architektur mit Accuracy von 0,9865. Die Lösung von SA ist nur knapp schlechter als die von GA, obwohl hier eine höhere Fitnessverbesserung vorliegt (8,4 %). Alle drei Verfahren gewinnen eindeutig gegen RS, bei welchem alle Kennzahlen verhältnismäßig schlechter sind.

Abbildung 28. Vergleich von RS, SA, GA, und MA mit der gleichen Initialisierung.

Aus allen durchgeführten Experimenten können folgende Schlüsse abgeleitet werden:

- Für jedes Dataset kann mithilfe der Hyperparameteroptimierung eine nach bestimmten Regeln aufgebaute Faltungsnetzarchitektur gefunden werden, die auf diesem Dataset überdurchschnittlich gute Performanz liefert.
- Die Regeln können sich jedoch von Dataset zu Dataset unterscheiden (vgl. Kapitel 6.4).
- Die Auswahl der Faltungskerne darf nicht unterschätzt werden, weil sie ebenso wie die Trainingsparameter eine direkte Auswirkung auf die Laufzeiten und Erkennungsraten haben.
- Gute Lösungsansätze zur Hyperparameteroptimierung sind GA, SA und MA.
- Passende Werte der wenigen spezifischen Metaparameter sollen für jeden Ansatz separat gefunden werden.
- Wenn der Suchraum der Hyperparameter relativ klein ist, liefert SA die besten Ergebnisse. Er hat die niedrigste Komplexität, muss aber idealerweise mit einer bereits sehr guten Architektur gestartet werden.
- GA und MA bringen stabile Lösungsverbesserungen, und zwar unabhängig von der Suchraumgröße und initialen Population. MA hat einen Leistungsvorsprung gegenüber GA und SA, jedoch auch kubische Komplexität.
- Alle Lösungsansätze sollen zunächst mit der kleinsten eingeschätzten Anzahl an Schichten durchgeführt werden. Wenn das gewünschte Performanzniveau nicht erreicht wurde, kann die Schichtenzahl allmählich erhöht werden. So wird die kleinste Architektur mit der bestmöglichen Erkennungsrate gefunden.
- Viel Wert soll auf die Auswahl der Fitnessfunktion gelegt werden. Im Idealfall sollte diese nicht nur von Accuracy, sondern auch zu einem gewissen Grad von Loss abhängig sein.

Lösungsansatz	Anzahl der Schichten	Erkennungsrate in %	Architektur
SA	6	98,99	[['c', 3, 1, 40, 'e'], ['p', 2, 1, 'm+a'], ['c', 9, 2, 106, 'e'], ['p', 3, 1, 'm+a'], ['f', 642, 'm'], ['f', 10, '-']]
MA	6	99,02	[['c', 3, 1, 76, 'e'], ['p', 2, 1, 'm+a'], ['c', 7, 3, 482, '-'], ['p', 3, 2, 'max'], ['f', 790, '-'], ['f', 10, '-']]
GA	7	99,03	[['c', 5, 1, 118, 'm'], ['p', 3, 1, 'm+a'], ['c', 7, 2, 520, 'e'], ['p', 3, 1, 'max'], ['c', 3, 1, 530, 'm'], ['f', 914, 'm'], ['f', 10, '-']]
SA mit 20.000 Trainingsiterationen	8	99,04	[['c', 3, 1, 80, '-'], ['c', 5, 1, 314, 'e'], ['p', 2, 1, 'm+a'], ['c', 5, 2, 468, 'e'], ['p', 3, 1, 'max'], ['c', 3, 1, 816, 'e'], ['f', 822, 'm'], ['f', 10, '-']]

Tabelle 5. Die höchsten erreichten Erkennungsraten auf MNIST.

Zu guter Letzt gibt Tabelle 5 einen Überblick über die besten erreichten Erkennungsraten auf MNIST. Die höchste Accuracy aller Experimente wurde mithilfe von SA erzielt, wobei in jenem Testlauf alle Faltungsnetze ausnahmsweise mit 20.000 Iterationen statt 4.000 trainiert wurden, und die Dauer der Suche stieg von 1 auf fast 5 Tage. Dieser Versuch beweist, dass eine weitere Steigerung der Erkennungsraten bei einer höheren Anzahl an Trainingsiterationen durchaus möglich wäre, wenngleich auf Kosten der Laufzeit.

6.4 Experimente mit CIFAR-10

Der Vergleich der Lösungsansätze auf dem CIFAR-10 Dataset ist ähnlich ausgefallen wie bei MNIST, aus diesem Grund werden hier nur die Besonderheiten erwähnt.

Es wurde unter anderem festgestellt, dass die Erkennungsraten von vielen validen Architekturen, die nach den obigen Regeln aufgebaut wurden, beim Training rasch gegen 0,1 tendierten. D.h. solche Architekturen lernen keine Zusammenhänge zwischen den Objektmerkmalen, sondern raten die Zugehörigkeit zu einer der 10 Klassen per Zufall. Dabei werden die gelernten Filter in allen Schichten komplett weiß. Dieses extrem schlechte Ergebnis wurde bei Architekturen identifiziert, die Max Pooling (vor allem im Klassifikator), Maxout oder die Summe von Max und Average Pooling verwendeten.

Interessanterweise zeigte sich Average Pooling auf MNIST lediglich um etwa 0,1 schlechter in Accuracy als Max Pooling, auch wenn die absolute Accuracy dabei sehr selten unter 0,2 lag. Die meisten Architekturen mit Average Pooling schieden bei MNIST aus dem Optimierungsverlauf aus, und nur wenige wurden als Gewinner-Architektur ausgewählt. Der Unterschied zwischen MNIST und CIFAR-10 liegt jedoch an den abgebildeten Gegenständen – farbige natürliche Objekte erfordern womöglich eine andere Handhabung als schwarzweiße Ziffern. Andererseits, legten die Experimente mit natürlichen Bildern aus ImageNet in [15] dar, dass die Objekterkennung sehr wohl von Max Pooling profitiert. Es gibt also keine klare Antwort auf die Frage, welche Art von Pooling für ein konkretes Dataset vorteilhafter ist.

Die schwache Performanz von Max Pooling und Maxout auf CIFAR-10 wurde in einigen Forschungsbeiträgen ebenfalls erkannt. So ist beim Vergleich in [22] Average Pooling leistungsfähiger als Max Pooling und weniger anfällig für Overfitting. Maxout ist wiederum nur dann effizienter als ReLU, wenn es zusammen mit Batch Normalization zum Einsatz kommt, wie am Beispiel einer sehr speziellen Faltungsnetzarchitektur gezeigt wurde [66]. Andererseits verwenden die meisten Forscher bestimmte Arten von Preprocessing und Datenaugmentation auf CIFAR-10, wodurch sich die Divergenz beim Training eines Netzes mit Max Pooling offenbar vermeiden lässt (vgl. [20], [53]).

Die Verbesserung der Erkennungsrate durch entsprechende Manipulation des Datasets liegt allerdings außerhalb der Zielsetzung dieser Arbeit. Deswegen wurde der Suchraum für anschließende Tests mit CIFAR-10 durch weitere Regeln beschränkt, um solche von

vorneherein schlechte Architekturen mit Max Pooling und Maxout von der Optimierung auszuschließen. Des Weiteren half der implementierte frühzeitige Abbruch des Trainings für die Fälle, wenn nach einer bestimmten Anzahl Iterationen die Erkennungsrate unter 0,1 blieb, sodass keine Zeit für die schlechten Architekturen verschwendet wurde.

7 Zusammenfassung

7.1 Fazit

In dieser Arbeit wurde die Suche nach einer für das gegebene Dataset optimalen Faltungsnetzarchitektur als ein Optimierungsproblem formuliert, analysiert und mithilfe von drei heuristischen Ansätzen gelöst. Einer der Ansätze basiert auf dem Genetischen Algorithmus, der zweite – auf Simulated Annealing (eine Variante der Lokalen Suche), der dritte ist der Memetische Algorithmus, ein Hybrid von den ersten beiden.

Es wurde demonstriert, dass diese weit verbreiteten Metaheuristiken, die bisher nur zur Suche von einer optimalen Architektur bei MLP eingesetzt wurden, auch auf den Fall von Faltungsnetzen übertragbar sind. Diese Verfahren enthalten bedeutend weniger Konfigurationsparameter als die Faltungsnetze und können anhand der vorgestellten Kodierung der Netzarchitektur den Hyperparametersuchraum effizient erkunden.

Des Weiteren wurde aufgezeigt, wie die wenigen Metaparameter von den genannten Metaheuristiken auszuwählen sind, damit die Suche am erfolgreichsten ist. Somit bietet die Evolutionäre Optimierung kombiniert mit Lokaler Suche ein mächtiges Werkzeug für tiefe Erforschungen von Faltungsnetzarchitekturen, welches sich nicht auf das spezifische Wissen über die Netze verlässt. In dieser Hinsicht gehören die Metaheuristiken zu den Black-Box-Optimierungsverfahren. Insbesondere diese Eigenschaft ist ihr großer Vorteil, weil das Regelwerk zur Erzeugung der Architekturen je nach Dataset problemlos ausgetauscht werden kann. Es ermöglicht die Anwendung der genannten Lösungsverfahren auf weitere Arten von Modellen im Bereich Deep Learning, wie z.B. Network in Network [66], Residual Networks [8] oder Maxout Netze [21], wo noch keine spezifischen Heuristiken zur Hyperparametersuche entworfen wurden.

Darüber hinaus wurden die drei Lösungsansätze erfolgreich implementiert und getestet, um das beste automatisierte Verfahren zur Architektursuche bei Faltungsnetzen zu ermitteln.

Die Experimente legten nahe, dass Evolutionäre Optimierung eine stabile Lösungsverbesserung liefert, Lokale Suche dagegen stark von der initialen Lösung abhängig ist. Wenn die Laufzeit keine Rolle spielt, soll die Kombination der beiden Verfahren, beispielsweise der Memetische Algorithmus in Betracht gezogen werden.

Außerdem wurde untersucht, unter welchen Bedingungen die Lösungsansätze zu den besten Ergebnissen führen. So kann Simulated Annealing besonders gut in kleineren Suchräumen eingesetzt werden und hat zudem die niedrigste Komplexität. Es kann dafür verwendet werden, um zu prüfen, ob in der Nähe einer guten Lösung sich noch bessere Lösungen befinden. Die beiden anderen Verfahren sind deutlich komplexer, dafür aber in jedem Suchraum einsetzbar. Dank der geschickt ausgewählten genetischen Operatoren ist der Genetische Algorithmus in der Lage, die guten Eigenschaften eines Faltungsnetzes weiterzuentwickeln, und die Verwendung einer Population ermöglicht ihm das Durchsuchen mehrerer Suchregionen gleichzeitig. Der Memetische Algorithmus weist noch bessere Eigenschaften bei der Suche auf, denn er kombiniert die Vorteile von seinen Kontrahenten. Alle drei Verfahren verfügen über Techniken, lokale Optima im Suchraum zu vermeiden.

Zusammenfassend können die implementieren Lösungsansätze als enorme Hilfe angesehen werden, wenn eine große Menge von gewünschten Architekturen systematisch durchsucht werden muss. Nichtsdestotrotz sind die Regeln bzw. die Art von Architekturen entscheidend für den Sucherfolg, denn sie bestimmen die Ober- und Untergrenze der möglichen Netzperformanz. Mit anderen Worten, die Metaheuristiken sind nur so gut wie die zugrundeliegenden Faltungsnetze, die den gewählten Rahmenbedingungen entsprechen.

7.2 Ausblick

Trotz den annehmbaren Laufzeiten auf MNIST und CIFAR-10 können die implementierten Lösungsansätze von der Verwendung mehrerer GPUs und Parallelisierung enorm profitieren. Bei größeren Datasets wie ImageNet und tiefen Architekturen mit mehr als 20 Schichten wäre dies der einzige Weg, die Optimierung der Hyperparameter innerhalb eines menschlichen Lebens durchzuführen. Unter der Berücksichtigung der Erkenntnisse aus [53], wo die Schätzung der zukünftigen Erkennungsrate mittels eines probabilistischen Modells vorgeschlagen wurde, könnte das überflüssige Training vermieden und die Laufzeit weiter gesenkt werden.

Die Erhöhung der Erkennungsrate kann neben der Optimierung der Architekturparameter durch zwei weitere Faktoren erreicht werden. Einer davon ist das Dataset: Durch Preprocessing der Bilder und geschickte Datenaugmentation können deutliche Fortschritte für jede Architektur erzielt werden. Falls solche Techniken benutzt werden, ändert sich für die Lösungsverfahren lediglich der Verlauf der Fitnessfunktion im Suchraum – z.B. durch ein höheres globales Optimum.

Der zweite Faktor ist das Training, wobei die Trainingsparameter in der Theorie immer für ein konkretes Faltungsnetz optimiert werden müssen. Dementsprechend können die Trainingsparameter zusammen mit den Architekturparametern in einem Chromosom mitkodiert werden, um im Laufe eines Lösungsverfahrens gemeinsam beachtet zu werden. In diesem Fall muss bedacht werden, wie die genetischen Operatoren bzw. der Nachbarschaftskreis auf eine Architektur mit bestimmten Trainingsparametern sinnvoll angepasst werden müssen. Es bleibt offen, ob diese Erweiterung Vorteile für die Optimierung mit sich bringt.

Wenn nur die Architekturverbesserung erwünscht ist, könnte außerdem geprüft werden, inwiefern das „Ensemble Learning" [9] von mehreren Gewinner-Architekturen die Erkennungsrate steigern kann. Auch fortgeschrittene Techniken wie DropConnect oder Batch Normalization können zusätzlich implementiert werden.

Sollten die genannten Erweiterungen umgesetzt werden, könnte es dazu beitragen, dass sich das Potential der in dieser Arbeit vorgestellten Metaheuristiken noch breiter entfalten kann und dadurch den Forschern im Bereich Deep Learning hoffentlich viel Arbeit abgenommen wird.

8 Glossar

Abkürzung	Langform	Bedeutung
CIFAR-10	-	Ein Dataset bestehend aus 50.000 Trainings- und 10.000 Testbilder mit kleiner Auflösung (32×32), die natürliche Objekte aus 10 Klassen in Farbe darstellen
CIFAR-100	-	Ein Dataset wie CIFAR-10, nur mit 100 Objektklassen
DBN	Deep Belief Network	Eine Klasse von tiefen neuronalen Netzwerken, die aus aufeinander gestapelten RBMs (Restricted Boltzmann Machines) mit jeweils einer einzigen verdeckten Schicht bestehen [1]
ELU	Exponential Linear Unit	Ein Neuron mit der Aktivierungsfunktion, die für positive Inputs linear verläuft, für negative – exponentiell
GA	Genetischer Algorithmus	Eine Metaheuristik aus der Menge der Evolutionären Algorithmen, die zur Lösung von Optimierungs- und Suchproblemen den Prozess der natürlichen Selektion nachahmen
ILSVRC	ImageNet Large Scale Visual Recognition Challenge	Ein jährlicher Wettbewerb für Objektklassifizierung und -segmentierung sowie das dazugehörige Dataset, welches aus über 1,3 Mio. Bilder und 1.000 Klassen von ImageNet zusammengesetzt ist
ImageNet	-	Ein großes Dataset bestehend aus über 14 Mio. Bilder von natürlichen Objekten, die zu knapp 22.000 Klassen gehören

Konvergenz	-	Konvergenz eines Verfahrens ist gegeben, wenn nach einer bestimmten Anzahl an Iterationen seine Zielfunktion (z.B. Fitnessfunktion oder Lossfunktion) sich an eine eindeutige Zahl annähert, die in dem Fall als ihr Grenzwert gilt
Lokale Suche	-	Eine Metaheuristik zur Lösung von Optimierungs- und Suchproblemen, die mit einer initialen Lösung startet und iterativ die lokalen Änderungen der aktuellen Lösung vornimmt, bis ein lokales Optimum in Bezug auf die Zielfunktion gefunden wird [39]
MA	Memetischer Algorithmus	Eine Metaheuristik, die die Evolutionären Algorithmen mit Lokaler Suche kombiniert
MLP	Multilayer Perzeptron	Eine Klasse von künstlichen neuronalen Feedforward-Netzwerken, die aus mehreren verdeckten Schichten bestehen
MNIST	-	Ein Dataset mit 60.000 Trainings- und 10.000 Testbilder von handschriftlichen Ziffern in Graustufen
NORB	-	Ein Dataset bestehend aus über 50.000 Bilder von 50 Spielzeigen aus 5 Klassen, die auf verschiedene Weise (Blickwinkel, Beleuchtung, Hintergrund usw.) in Graustufen fotografiert wurden
ReLU	Rectified Linear Unit	Ein Neuron mit der Aktivierungsfunktion, die für positive Inputs linear verläuft, die negativen aber in 0 verwandelt
SA	Simulated Annealing	Eine Metaheuristik zur Lösung von Optimierungs- und Suchproblemen basierend auf Lokaler Suche
SGD	Stochastic Gradient Descent	Eine stochastische Approximation des Gradientenverfahrens zur Lösung von Optimierungsproblemen sowie zum Überwachten Lernen bei künstlichen neuronalen Netzwerken (Stichwort Backpropagation) [1]
SVM	Support Vector Machine	Ein nicht neuronaler, überwachter Lernalgorithmus zur Mustererkennung aus dem Bereich Machine Learning [1]

Wichtige Begriffe:

Trainierbare Parameter – die Parameter eines Faltungsnetzes, die im Laufe des Trainings gelernt werden (Gewicht und Bias der Neuronen).

Trainingsparameter – die für das Training zuständigen Parameter eines Faltungsnetzes, also die Bestandteile eines Trainingsregimes. Einige davon bleiben konstant, andere werden beim Training automatisch angepasst.

Hyperparameter – die Parameter eines Faltungsnetzes, die im Voraus festgelegt werden müssen und während des Trainings nicht gelernt werden. Im Rahmen dieser Arbeit sind damit ausschließlich die Architekturparameter gemeint.

Metaparameter – die Parameter eines Optimierungsverfahrens, die sowohl für das Verfahren selbst, als auch für die von ihm generierten Lösungen festgelegt werden müssen.

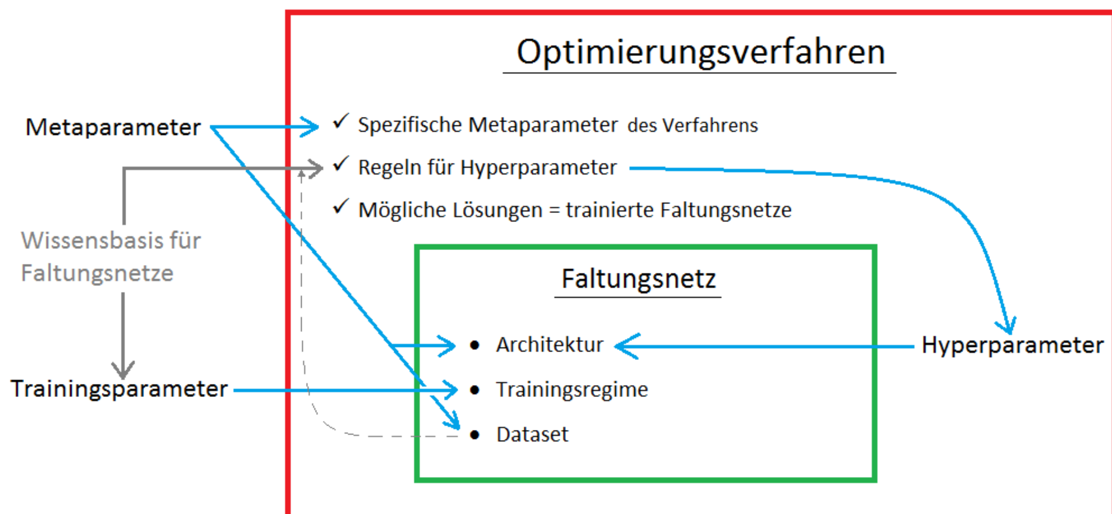

Abbildung 29. Im Rahmen dieser Arbeit verwendete Arten von Parametern.

9 Abbildungsverzeichnis

Abbildung 1. Architektur von „LeNet-5“ [6]. 11

Abbildung 2. Funktionsweise des Filter Bank Layers (nach [16]). 12

Abbildung 3. Aktivierungsfunktionen. 16

Abbildung 4. Klassifikator: links ohne, rechts – mit Dropout [25]. 19

Abbildung 5. Optimaler Input von zufälligen Filtern (links) und von unüberwacht gelernten Filtern (rechts) [13]. 22

Abbildung 6. Initialisierungsschemata von links nach rechts: Gauß, MSRA, Xavier. 23

Abbildung 7. Genotyp-Phänotyp-Abbildung (nach [57]). 35

Abbildung 8. Abkühlungsplan von SA. 38

Abbildung 9. Gauß-initialisierte Filter vor und nach dem Training. 43

Abbildung 10. Ein Faltungsnetz mit valider Architektur aus 7 Schichten. 47

Abbildung 11. Umsetzung der Summe von Max- und Average-Pooling (links) und des Maxouts (rechts) in "Caffe". 48

Abbildung 12. Änderungshäufigkeit pro Hyperparameter bei Crossover (links) und bei Mutation (Mitte), Anteil der entsprechenden Gene an einem Chromosom (rechts). 50

Abbildung 13. Akzeptanzwahrscheinlichkeit für unterschiedlich große Änderungen der Fitnesswerte (Δ). 53

Abbildung 14. Von links nach rechts: Beispielbilder von CIFAR-10; Mean-Bild von CIFAR-10; Beispielbilder von MNIST [22]. 55

Abbildung 15. Tote Neuronen bei einem Netz mit ReLU (links) und mit ELU (rechts), getestet auf MNIST. 59

Abbildung 16. "LR range test" für MNIST. 60

Abbildung 17. GA, Lösungsverbesserung je nach Metaparameterkonfiguration und Anzahl der Schichten. 64

Abbildung 18. SA, Fitnessverlauf für d = 1 (links), d = 0,001 (Mitte) und R = 0,05 (rechts). 65

Abbildung 19. GA, Verlauf der Accuracy pro Faltungsnetz in der Iteration 1 (links) und 10 (rechts). 68

Abbildung 20. GA, beste Performanz in jeder Iteration (links), Verlauf der populationsbesten Fitness (rechts). 68

Abbildung 21. SA, Performanz jedes trainierten Netzes (links), Verlauf der aktuellen und besten Fitness (rechts). 69

Abbildung 22. MA, beste Performanz in jeder Iteration (links), Verlauf der populationsbesten Fitness (rechts). 69

Abbildung 23. MA, Verteilung der Accuracy innerhalb der Population in der Iteration 1 (links), 5 (Mitte) und 10 (rechts). 70

Abbildung 24. Beste finale Accuracy bei GA und SA. 72

Abbildung 25. Verbesserung von Fitness in % bei GA und SA. 73

Abbildung 26. Verbesserung von Accuracy in % bei GA und SA. 74

Abbildung 27. Fitnessverlauf von GA (links) mit anschließender Anwendung von SA (rechts). 75

Abbildung 28. Vergleich von RS, SA, GA, und MA mit der gleichen Initialisierung. 76

Abbildung 29. Im Rahmen dieser Arbeit verwendete Arten von Parametern. 85

10 Tabellenverzeichnis

Tabelle 1. Layertypen und ihre Hyperparameter. 21

Tabelle 2. Rechenkomplexität der Lösungsansätze (N - Anzahl der Iterationen, M - Populationsgröße). 32

Tabelle 3. Datasets im Bereich Objektklassifizierung. 54

Tabelle 4. Übersicht aller variablen Parameter. 62

Tabelle 5. Die höchsten erreichten Erkennungsraten auf MNIST. 77

Tabelle 6. Variablen in `Config_<algo_name>.ini` 97

Tabelle 7. Variablen in `Config_<dataset_name>_Net.ini` 99

11 Literaturverzeichnis

[1] J. Schmidhuber, „Deep learning in neural networks: An overview," *Neural Networks,* Bd. 61, pp. 85-117, Januar 2015.

[2] L. Deng und D. Yu, „Deep Learning: Methods and Applications," in *Foundations and Trends in Signal Processing*, Bd. 7, NOW Publishers, 2014, pp. 197-387.

[3] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard und L. D. Jackel, „Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Comput.,* Bd. 1, Nr. 4, pp. 541-551, 1989.

[4] O. Russakovsky, 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, „ImageNet Large Scale Visual Recognition Challenge," *CoRR,* Bd. abs/1409.0575, 2014.

[5] A. Krizhevsky, I. Sutskever und G. E. Hinton, „ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou und K. Weinberger, Hrsg., Curran Associates, Inc., 2012, pp. 1097-1105.

[6] Y. LeCun, L. Bottou, Y. Bengio und P. Haffner, „Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* Bd. 86, Nr. 11, pp. 2278-2324, November 1998.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke und A. Rabinovich, „Going Deeper with Convolutions," *CoRR,* Bd. abs/1409.4842, 17 September 2014.

[8] K. He, X. Zhang, S. Ren und J. Sun, „Deep Residual Learning for Image Recognition,“ in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[9] L. Deng und J. Platt, „Ensemble Deep Learning for Speech Recognition,“ Proc. Interspeech, 2014.

[10] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally und K. Keutzer, „SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 1MB model size,“ *CoRR,* Bd. abs/1602.07360, 4 November 2016.

[11] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever und R. Salakhutdinov, „Improving neural networks by preventing co-adaptation of feature detectors,“ *CoRR,* Bd. abs/1207.0580, 2012.

[12] J. Ponce, T. Berg, M. Everingham, D. Forsyth, M. Hebert, S. Lazebnik, M. Marszalek, C. Schmid, B. Russell, A. Torralba, C. Williams, J. Zhang und A. Zisserman, „Dataset Issues in Object Recognition,“ in *Lecture Notes in Computer Science*, Bd. 4170, J. Ponce, M. Hebert, C. Schmid und A. Zisserman, Hrsg., Springer Berlin Heidelberg, 2006, pp. 29-48.

[13] K. Jarrett, K. Kavukcuoglu, M. Ranzato und Y. LeCun, „What is the Best Multi-Stage Architecture for Object Recognition?,“ in *Computer Vision, 2009 IEEE 12th International Conference on*, Kyoto, 2009.

[14] M. D. Zeiler und R. Fergus, „Visualizing and Understanding Convolutional Networks,“ *CoRR,* Bd. abs/1311.2901, 2013.

[15] D. Mishkin, N. Sergievskiy und J. Matas, „Systematic evaluation of CNN advances on the ImageNet,“ *CoRR,* Bd. abs/1606.02228, 13 Juni 2016.

[16] A. Meisel, *Faltung,* Vorlesungsskript ”Robot Vision", 2012.

[17] A. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh und A. Y. Ng, „On random weights and unsupervised feature learning,“ in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011.

[18] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus und Y. LeCun, „Overfeat: Integrated recognition, localization and detection using convolutional networks,“ *International Conference on Learning Representations (ICLR 2014),* p. 16, 2013.

[19] D.-A. Clevert, T. Unterthiner und S. Hochreiter, „Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),“ *CoRR,* Bd. abs/1511.07289, 2015.

[20] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville und Y. Bengio, „Maxout Networks,“ in *Proceedings of the 30th International Conference on International Conference on Machine Learning*, Atlanta, GA, USA, 2013.

[21] P. Swietojanski, J. Li und J. T. Huang, „Investigation of maxout networks for speech recognition," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Florence, Italy, 2014.

[22] M. D. Zeiler und R. Fergus, „Stochastic Pooling for Regularization of Deep Convolutional Neural Networks," *CoRR,* Bd. abs/1301.3557, 2013.

[23] C.-Y. Lee, P. W. Gallagher und Z. Tu, „Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree," in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, Cadiz, Spain, 2016.

[24] K. Simonyan und A. Zisserman, „Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR,* Bd. abs/1409.1556, 4 September 2014.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever und R. Salakhutdinov, „Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research,* Bd. 15, Nr. 1, pp. 1929-1958, Januar 2014.

[26] L. Wan, M. Zeiler, S. Zhang, Y. LeCun und R. Fergus, „Regularization of Neural Networks Using Dropconnect," in *Proceedings of the 30th International Conference on International Conference on Machine Learning*, Atlanta, GA, USA, 2013.

[27] Y. Bengio, „Practical Recommendations for Gradient-Based Training of Deep Architectures," in *Neural Networks: Tricks of the Trade: Second Edition*, 2 Hrsg., Berlin, Heidelberg, Springer Berlin Heidelberg, 2012, pp. 437-478.

[28] K. Kavukcuoglu, M. Ranzato und Y. LeCun, „Fast inference in sparse coding algorithms with applications to object recognition," Computational and Biological Learning Laboratory, Courant Institute, NYU, New-York, 2008.

[29] P. Sermanet und Y. LeCun, „Traffic sign recognition with multi-scale Convolutional Networks," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, San Jose, CA, 2011.

[30] X. Glorot und Y. Bengio, „Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Chia Laguna Resort, Sardinia, Italy, 2010.

[31] K. He, X. Zhang, S. Ren und J. Sun, „Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, Washington, DC, USA, 2015.

[32] I. Sutskever, J. Martens, G. Dahl und G. Hinton, „On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning*, Atlanta, Georgia, USA, 2013.

[33] L. N. Smith, „Cyclical Learning Rates for Training Neural Networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, Santa Rosa, CA, USA, USA, 2017.

[34] D. Mishkin und J. Matas, „All you need is a good init," *CoRR,* Bd. abs/1511.06422, 2015.

[35] S. Haykin, Neural networks and learning machines, 3. Hrsg., Upper Saddle River, NJ, USA: Pearson Education, 2009.

[36] X. Yao, „Evolving artificial neural networks," *Proceedings of the IEEE,* Bd. 87, Nr. 9, pp. 1423-1447, September 1999.

[37] G. Walz, Lexikon der Mathematik: Moo bis Sch, Bd. 4, Springer Berlin Heidelberg, 2017.

[38] L. Bianchi, M. Dorigo, L. M. Gambardella und W. J. Gutjahr, „A survey on metaheuristics for stochastic combinatorial optimization," *Natural Computing,* Bd. 8, Nr. 2, pp. 239-287, Juni 2009.

[39] I. Boussaïd, J. Lepagnot und P. Siarry, „A survey on optimization metaheuristics," *Information Sciences,* Bd. 237, pp. 82-117, Juli 2013.

[40] C. Blum, J. Puchinger, G. R. Raidl und A. Roli, „Hybrid metaheuristics in combinatorial optimization: A survey," *Applied Soft Computing,* Bd. 11, Nr. 6, pp. 4135-4151, September 2011.

[41] Q. Shen, C.-j. Liu, H.-l. Zou, S.-s. Zhou und T.-t. Chen, „A Method of Image Classification with Optimized BP Neural Network by Genetic Algorithm," in *Intelligent Networking and Collaborative Systems (INCOS), 2015 International Conference on*, 2015.

[42] T.-C. Lu, G.-R. Yu und J.-C. Juang, „Quantum-Based Algorithm for Optimizing Artificial Neural Networks," *Neural Networks and Learning Systems, IEEE Transactions on,* Bd. 24, Nr. 8, pp. 1266-1278, August 2013.

[43] B. Correa und A. Gonzalez, „Evolutionary Algorithms for Selecting the Architecture of a MLP Neural Network: A Credit Scoring Case," in *Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on*, 2011.

[44] T. H. Oong und N. A. M. Isa, „Adaptive Evolutionary Artificial Neural Networks for Pattern Classification," *Neural Networks, IEEE Transactions on,* Bd. 22, Nr. 11, pp. 1823-1836, November 2011.

[45] S.-W. Lin, Z.-J. Lee, S.-C. Chen und T.-Y. Tseng, „Parameter determination of support vector machine and feature selection using simulated annealing approach,“ *Applied Soft Computing,* Bd. 8, Nr. 4, pp. 1505-1512, September 2008.

[46] L. S. Shu, S. Y. Ho und H. S. J., „Tuning the structure and parameters of a neural network using an orthogonal simulated annealing algorithm,“ in *2009 Joint Conferences on Pervasive Computing (JCPC)*, Tamsui, Taipei, Taiwan, 2009.

[47] J. S. Sartakhti, H. Afrabandpey und M. Saraee, „Simulated annealing least squares twin support vector machine (SA-LSTSVM) for pattern classification,“ *Soft Computing,* pp. 1-13, 2016.

[48] A. M. Turky, S. Abdullah und N. R. Sabar, „Electromagnetic algorithm for tuning the structure and parameters of neural networks,“ in *2014 IEEE Congress on Evolutionary Computation (CEC)*, Beijing, China, 2014.

[49] H. Larochelle, D. Erhan, A. Courville, J. Bergstra und Y. Bengio, „An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation,“ in *Proceedings of the 24th International Conference on Machine Learning*, Corvalis, Oregon, USA, 2007.

[50] J. S. Bergstra, R. Bardenet, Y. Bengio und B. Kégl, „Algorithms for Hyper-Parameter Optimization,“ in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira und K. Q. Weinberger, Hrsg., Curran Associates, Inc., 2011, pp. 2546-2554.

[51] J. Bergstra und Y. Bengio, „Random Search for Hyper-parameter Optimization,“ *J. Mach. Learn. Res.,* Bd. 13, Nr. 1, pp. 281-305, January 2012.

[52] J. Snoek, H. Larochelle und R. P. Adams, „Practical Bayesian Optimization of Machine Learning Algorithms,“ in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, Lake Tahoe, Nevada, 2012.

[53] T. Domhan, J. T. Springenberg und F. Hutter, „Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves,“ in *Proceedings of the 24th International Conference on Artificial Intelligence*, Buenos Aires, Argentina, 2015.

[54] P. P. Palmes, T. Hayasaka und S. Usui, „Mutation-based genetic neural network,“ *IEEE Transactions on Neural Networks,* Bd. 16, Nr. 3, pp. 587-600, Mai 2005.

[55] F. H. F. Leung, H. K. Lam, S. H. Ling und P. K. S. Tam, „Tuning of the structure and parameters of a neural network using an improved genetic algorithm,“ *IEEE Transactions on Neural Networks,* Bd. 14, Nr. 1, pp. 79-88, Januar 2003.

[56] A. S. Shirazi und T. Seyedena, „Design of MLP using Evolutionary Strategy with Variable Length Chromosomes,“ in *2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, Phuket, Thailand, 2008.

[57] P. J. Angeline, G. M. Saunders und J. B. Pollack, „An Evolutionary Algorithm That Constructs Recurrent Neural Networks,“ *Trans. Neur. Netw.,* Bd. 5, Nr. 1, pp. 54-65, Januar 1994.

[58] R. Eglese, „Simulated annealing: A tool for operational research,“ *European Journal of Operational Research,* Bd. 46, Nr. 3, pp. 271-281, 1990.

[59] N. Krasnogor und J. Smith, „A tutorial for competent memetic algorithms: model, taxonomy, and design issues,“ *IEEE Transactions on Evolutionary Computation,* Bd. 9, Nr. 5, pp. 474-488, Oktober 2005.

[60] P. Merz und B. Freisleben, „A comparison of memetic algorithms, tabu search, and ant colonies for the quadratic assignment problem,“ in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, Washington, DC, USA, 1999.

[61] Y. Chang, Y. Wang, K. Ricanek und C. Chen, „Feature selection for improved automatic gender classification,“ in *2011 IEEE Workshop on Computational Intelligence in Biometrics and Identity Management (CIBIM)*, Paris, France, 2011.

[62] D. Betül Gümüş, E. Ozcan und J. Atkin, „An investigation of tuning a memetic algorithm for cross-domain search,“ in *2016 IEEE Congress on Evolutionary Computation (CEC)*, Vancouver, BC, Canada, 2016.

[63] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama und T. Darrell, „Caffe: Convolutional Architecture for Fast Feature Embedding,“ Berkeley Vision and Learning Center, 2014. [Online]. Available: http://caffe.berkeleyvision.org/. [Zugriff am 30 März 2017].

[64] D. C. Ciresan, U. Meier und J. Schmidhuber, „Multi-column Deep Neural Networks for Image Classification,“ *CoRR,* Bd. abs/1202.2745, 13 Februar 2012.

[65] S. Han, J. Pool, J. Tran und W. Dally, „Learning both Weights and Connections for Efficient Neural Network,“ in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama und R. Garnett, Hrsg., Curran Associates, Inc., 2015, pp. 1135-1143.

[66] Z. Liao und G. Carneiro, „On the importance of normalisation layers in deep learning with piecewise linear activation units,“ in *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, Lake Placid, NY, USA, 2016.

12 Anhänge

Implementierung der Parameter:

Parameterart	Name	Bedeutung
Parameter eines Testlaufes	env	Name der Umgebung; steht z.B. für den absoluten Pfad zu „Caffe“ und Skripten, falls in zwei unterschiedlichen Umgebungen gearbeitet wird
	algo_name	Legt den Ordnernamen fest, wo die Ergebnisse gespeichert werden
	dataset	Bestimmt den Namen des Config-Abschnittes, wo die Parameter des Datasets aufgeführt sind
	with_output_architecture	True, wenn der Graph der Netzarchitektur bzw. aller Architekturen der Population auszugegeben ist; die Ausgabe erfolgt durch den Aufruf von „Caffe“-internen Modul `draw_net.py`
	with_output_compare	True, wenn die Lernkurven jedes Netzes in einem Plot auszugeben sind; bei GA und MA wird ein Bild pro Population erzeugt, bei SA und RS – eins für alle Netze

	next_run	True, wenn der nächste Lauf des Algorithmus gestartet werden soll; False, wenn z.B. der vorherige Lauf abgebrochen wurde und fortgesetzt werden soll
	only_result	True, wenn nur der letzte Schritt (das finale Training von der Gewinner-Architektur) erwünscht ist
	final_train_from_weights	True, wenn das Laden des Snapshots für den letzten Schritt (s.o.) erforderlich ist
Parameter des Datasets	image_dim_x, image_dim_y	Breite und -höhe des Input-Bildes
	image_channels	1, wenn das Input-Bild schwarzweiß ist, sonst 3
	class_count	Anzahl der Objektklassen im Dataset
	dataset_name	Legt den Ordnernamen fest, wo die Ergebnisse gespeichert werden (Unterordner von `algo_name`)
Metaparameter des Algorithmus	layer_count	Wie viele Layers sollen es insgesamt geben?
	fc_count	Wie viele davon sollen Full Connection Layers sein?
	max_iter	Wie viele Iterationen lang soll der Algorithmus laufen?
	alpha, beta	Koeffizienten der Fitnessfunktion
	…	Spezifische Metaparameter

Tabelle 6. Variablen in `Config_<algo_name>.ini`

Art des Parameters	Name	Bedeutung
Parameter eines Testlaufes	env	Name der Umgebung; steht z.B. für den absoluten Pfad zu „Caffe“ und Skripten, falls in zwei unterschiedlichen Umgebungen gearbeitet wird

	algo_name	Legt den Ordnernamen fest, wo die Ergebnisse gespeichert werden
	with_output_filter	True, wenn pro Netz die Filter aller Schichten ausgegeben werden sollen, und zwar nur bei der ersten und letzten Trainingsiteration
	with_output_architecture	True, wenn der Graph der Netzarchitektur auszugegeben ist; die Ausgabe erfolgt durch den Aufruf von „Caffe"-internen Modul `draw_net.py`
	with_output_performance	True, wenn pro Netz die Kurven von Accuracy und Loss in einem Plot auszugeben sind
Parameter des Datasets	image_dim_x, image_dim_y	Breite und –höhe des Input-Bildes
	image_channels	1, wenn das Input-Bild schwarzweiß ist, sonst 3
	class_count	Anzahl der Objektklassen im Dataset
	dataset_name	Legt den Ordnernamen fest, wo die Ergebnisse gespeichert werden (Unterordner von `algo_name`)
Trainingsparameter	device	CPU oder GPU (für das Training)
	with_seed	True, wenn das Random-Seed verwendet werden soll
	with_debug	Setzt Debug-Modus für „Caffe", sodass beim Training mehr Informationen angezeigt werden
	train_classifier	True, wenn der Klassifikator trainiert werden soll
	use_dropout	True, wenn Dropout im Klassifikator verwendet werden soll
	dropout_ratio	Gibt die Wahrscheinlichkeit des Dropouts vor

	train_size	Trainingssetgröße
	test_size	Testsetgröße
	data_batch_size_train	Batch-Größe für das Trainingsset
	data_batch_size_test	Batch-Größe für das Testset
	niter	Anzahl der Trainingsiterationen
	noutput	Anzahl der Testzyklen (also der Ermittlungen der Accuracy)
	snapshot	Nach wie vielen Iterationen soll ein Snapshot gespeichert werden?
	solver_type	Typ des Lernalgorithmus (SGD, Nesterov, …)
	base_lr	Initiale Lernrate
	momentum	Trägheitskoeffizient
	weight_decay	Weight-Decay-Koeffizient
	lr_policy	Art der Lernratensenkung (‚inv‘ soll bei linearem Fall benutzt werden)
	gamma, power	Variablen der Lernratensenkung (gamma wird im Skript automatisch gesetzt, power=-1)
	display	Anzeige des Lernerfolges in der Konsole durch „Caffe“ erfolgt jede „display“ Iterationen
	filler_name	Initialisierungsschema für Faltungskerne (msra, xavier, gauss)
	std	Standardabweichung für Gauß-initialisierte Filter

Tabelle 7. Variablen in `Config_<dataset_name>_Net.ini`

Ein Faltungsnetz in "Caffe":

Solver.prototxt:

```
train_net: "/home/user/Documents/caffe/Victoria_Workspace/Generate/RS/MNIST/Nets/custom_0_0_train.prototxt"
test_net: "/home/user/Documents/caffe/Victoria_Workspace/Generate/RS/MNIST/Nets/custom_0_0_test.prototxt"
test_iter: 1000
test_interval: 400
base_lr: 0.000125
display: 400
max_iter: 4000
lr_policy: "inv"
gamma: -0.00025
power: -1.0
momentum: 0.9
weight_decay: 0.0005
snapshot: 40000
snapshot_prefix: "/home/user/Documents/caffe/Victoria_Workspace/Generate/RS/MNIST/Nets/custom_0_0"
solver_mode: GPU
random_seed: 831486
type: "SGD"
```

Train_net.prototxt:	*Test_net.prototxt:*

layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  transform_param {
    scale: 0.00392156862745
  }
  data_param {
    source:
"/home/user/Documents/caffe/examples/mnist/mnist_train_lmdb"
    batch_size: 32
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  propagate_down: false
	layer {
	  name: "data"
	  type: "Data"
	  top: "data"
	  top: "label"
	  transform_param {
	    scale: 0.00392156862745
	  }
	  data_param {
	    source:
	"/home/user/Documents/caffe/examples/mnist/mnist_test_lmdb"
	    batch_size: 10
	    backend: LMDB
	  }
	}
	layer {
	  name: "conv1"
	  type: "Convolution"
	  bottom: "data"
	  top: "conv1"
	  param {
	    lr_mult: 0
	    decay_mult: 0
	  }
	  param {
	    lr_mult: 0
	    decay_mult: 0
	  }
	  propagate_down: false

```
  convolution_param {
    num_output: 48
    pad: 0
    kernel_size: 3
    group: 1
    stride: 1
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
    engine: CAFFE
  }
}
layer {
  name: "Slice1"
  type: "Slice"
  bottom: "conv1"
  top: "Slice1"
  top: "Slice2"
  propagate_down: false
  slice_param {
    slice_point: 24
    axis: 1
  }
}
layer {
  name: "nlin1"
  type: "Eltwise"
  bottom: "Slice1"
  bottom: "Slice2"
  top: "nlin1"
  propagate_down: false
  propagate_down: false
  eltwise_param {
    operation: MAX
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "nlin1"
  top: "conv2"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  propagate_down: false
  convolution_param {
    num_output: 258
    pad: 0
    kernel_size: 9
    group: 1
    stride: 1
    weight_filler {
```

```
  convolution_param {
    num_output: 48
    pad: 0
    kernel_size: 3
    group: 1
    stride: 1
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
    engine: CAFFE
  }
}
layer {
  name: "Slice1"
  type: "Slice"
  bottom: "conv1"
  top: "Slice1"
  top: "Slice2"
  propagate_down: false
  slice_param {
    slice_point: 24
    axis: 1
  }
}
layer {
  name: "nlin1"
  type: "Eltwise"
  bottom: "Slice1"
  bottom: "Slice2"
  top: "nlin1"
  propagate_down: false
  propagate_down: false
  eltwise_param {
    operation: MAX
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "nlin1"
  top: "conv2"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  propagate_down: false
  convolution_param {
    num_output: 258
    pad: 0
    kernel_size: 9
    group: 1
    stride: 1
    weight_filler {
```

```
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
    engine: CAFFE
  }
}
layer {
  name: "Slice3"
  type: "Slice"
  bottom: "conv2"
  top: "Slice3"
  top: "Slice4"
  propagate_down: false
  slice_param {
    slice_point: 129
    axis: 1
  }
}
layer {
  name: "nlin2"
  type: "Eltwise"
  bottom: "Slice3"
  bottom: "Slice4"
  top: "nlin2"
  propagate_down: false
  propagate_down: false
  eltwise_param {
    operation: MAX
  }
}
layer {
  name: "Pooling1"
  type: "Pooling"
  bottom: "nlin2"
  top: "Pooling1"
  propagate_down: false
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 1
    pad: 0
    engine: CAFFE
  }
}
layer {
  name: "Pooling2"
  type: "Pooling"
  bottom: "nlin2"
  top: "Pooling2"
  propagate_down: false
  pooling_param {
    pool: AVE
    kernel_size: 2
    stride: 1
    pad: 0
    engine: CAFFE
  }
}
```

```
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
    engine: CAFFE
  }
}
layer {
  name: "Slice3"
  type: "Slice"
  bottom: "conv2"
  top: "Slice3"
  top: "Slice4"
  propagate_down: false
  slice_param {
    slice_point: 129
    axis: 1
  }
}
layer {
  name: "nlin2"
  type: "Eltwise"
  bottom: "Slice3"
  bottom: "Slice4"
  top: "nlin2"
  propagate_down: false
  propagate_down: false
  eltwise_param {
    operation: MAX
  }
}
layer {
  name: "Pooling1"
  type: "Pooling"
  bottom: "nlin2"
  top: "Pooling1"
  propagate_down: false
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 1
    pad: 0
    engine: CAFFE
  }
}
layer {
  name: "Pooling2"
  type: "Pooling"
  bottom: "nlin2"
  top: "Pooling2"
  propagate_down: false
  pooling_param {
    pool: AVE
    kernel_size: 2
    stride: 1
    pad: 0
    engine: CAFFE
  }
}
```

```
layer {
  name: "pool1"
  type: "Eltwise"
  bottom: "Pooling1"
  bottom: "Pooling2"
  top: "pool1"
  propagate_down: false
  propagate_down: false
  eltwise_param {
    operation: SUM
  }
}
layer {
  name: "conv3"
  type: "Convolution"
  bottom: "pool1"
  top: "conv3"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  propagate_down: false
  convolution_param {
    num_output: 294
    pad: 0
    kernel_size: 3
    group: 1
    stride: 2
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
    engine: CAFFE
  }
}
layer {
  name: "Slice5"
  type: "Slice"
  bottom: "conv3"
  top: "Slice5"
  top: "Slice6"
  propagate_down: false
  slice_param {
    slice_point: 147
    axis: 1
  }
}
layer {
  name: "nlin3"
  type: "Eltwise"
  bottom: "Slice5"
  bottom: "Slice6"
  top: "nlin3"
  propagate_down: false
```

```
layer {
  name: "pool1"
  type: "Eltwise"
  bottom: "Pooling1"
  bottom: "Pooling2"
  top: "pool1"
  propagate_down: false
  propagate_down: false
  eltwise_param {
    operation: SUM
  }
}
layer {
  name: "conv3"
  type: "Convolution"
  bottom: "pool1"
  top: "conv3"
  param {
    lr_mult: 0
    decay_mult: 0
  }
  param {
    lr_mult: 0
    decay_mult: 0
  }
  propagate_down: false
  convolution_param {
    num_output: 294
    pad: 0
    kernel_size: 3
    group: 1
    stride: 2
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
    engine: CAFFE
  }
}
layer {
  name: "Slice5"
  type: "Slice"
  bottom: "conv3"
  top: "Slice5"
  top: "Slice6"
  propagate_down: false
  slice_param {
    slice_point: 147
    axis: 1
  }
}
layer {
  name: "nlin3"
  type: "Eltwise"
  bottom: "Slice5"
  bottom: "Slice6"
  top: "nlin3"
  propagate_down: false
```

```
  propagate_down: false
  eltwise_param {
    operation: MAX
  }
}
layer {
  name: "Pooling3"
  type: "Pooling"
  bottom: "nlin3"
  top: "Pooling3"
  propagate_down: false
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
    pad: 0
    engine: CAFFE
  }
}
layer {
  name: "Pooling4"
  type: "Pooling"
  bottom: "nlin3"
  top: "Pooling4"
  propagate_down: false
  pooling_param {
    pool: AVE
    kernel_size: 2
    stride: 2
    pad: 0
    engine: CAFFE
  }
}
layer {
  name: "pool2"
  type: "Eltwise"
  bottom: "Pooling3"
  bottom: "Pooling4"
  top: "pool2"
  propagate_down: false
  propagate_down: false
  eltwise_param {
    operation: SUM
  }
}
layer {
  name: "fc1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "fc1"
  inner_product_param {
    num_output: 950
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
  }
}
```

```
  propagate_down: false
  eltwise_param {
    operation: MAX
  }
}
layer {
  name: "Pooling3"
  type: "Pooling"
  bottom: "nlin3"
  top: "Pooling3"
  propagate_down: false
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
    pad: 0
    engine: CAFFE
  }
}
layer {
  name: "Pooling4"
  type: "Pooling"
  bottom: "nlin3"
  top: "Pooling4"
  propagate_down: false
  pooling_param {
    pool: AVE
    kernel_size: 2
    stride: 2
    pad: 0
    engine: CAFFE
  }
}
layer {
  name: "pool2"
  type: "Eltwise"
  bottom: "Pooling3"
  bottom: "Pooling4"
  top: "pool2"
  propagate_down: false
  propagate_down: false
  eltwise_param {
    operation: SUM
  }
}
layer {
  name: "fc1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "fc1"
  inner_product_param {
    num_output: 950
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
  }
}
```

```
layer {
  name: "Slice7"
  type: "Slice"
  bottom: "fc1"
  top: "Slice7"
  top: "Slice8"
  slice_param {
    slice_point: 475
    axis: 1
  }
}
layer {
  name: "nlin4"
  type: "Eltwise"
  bottom: "Slice7"
  bottom: "Slice8"
  top: "nlin4"
  eltwise_param {
    operation: MAX
  }
}
layer {
  name: "score"
  type: "InnerProduct"
  bottom: "nlin4"
  top: "score"
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "score"
  bottom: "label"
  top: "loss"
}
```

```
layer {
  name: "Slice7"
  type: "Slice"
  bottom: "fc1"
  top: "Slice7"
  top: "Slice8"
  slice_param {
    slice_point: 475
    axis: 1
  }
}
layer {
  name: "nlin4"
  type: "Eltwise"
  bottom: "Slice7"
  bottom: "Slice8"
  top: "nlin4"
  eltwise_param {
    operation: MAX
  }
}
layer {
  name: "score"
  type: "InnerProduct"
  bottom: "nlin4"
  top: "score"
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "msra"
    }
    bias_filler {
      type: "msra"
    }
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "score"
  bottom: "label"
  top: "loss"
}
```

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _________________ ______________________________