Masterarbeit am Institut für Informatik der Freien Universität Berlin

Optimierung eines neuronalen Netzes zur Objekterkennung unter Verwendung evolutionärer Algorithmen

Nils Kornfeld

Matrikelnummer: 4839759

nils.kornfeld@fu-berlin.de

Externer Betreuer: Dr. Karsten Kozempel

Eingereicht bei: Prof. Dr. Raúl Rojas

Berlin, 07.04.2017

Zusammenfassung

In der Bildverarbeitung erweisen sich neuronale Netze seit einigen Jahren als das Mittel der Wahl zur Klassifizierung des Bildinhaltes. Die Netze, die zur Zeit die besten Ergebnisse liefern, sind sehr tiefe residuelle Convolutional Neural Networks. Jedes Jahr werden in unterschiedlichen Vorgehensweisen immer wieder neue Netzwerkstrukturen entwickelt, die auf den vormals aktuellen Architekturen aufbauen. Im Rahmen dieser Arbeit wird die Anwendung eines kanonischen Genetischen Algorithmus zur Optimierung der Strukturen neuronaler Netzwerke untersucht. Als Ausgangspunkt der Optimierung wird auf Netzwerkarchitekturen, die dem aktuellen Stand der Technik entsprechen, aufgebaut. Beim Vergleich der erzeugten Architekturen mit Referenznetzen ist eine Verbesserung der Klassifikationsgüte erkennbar. Die entwickelten Netzstrukturen unterscheiden sich strukturell von anderen, dem Stand der Technik entsprechenden Architekturen, da sie nicht homogen aus sich wiederholenden Elementen, sondern heterogen aus einer Sequenz unterschiedlicher Blöcke bestehen. Die verbesserte Klassifikationsgenauigkeit legt nahe, dass der Einsatz heterogener Strukturen von Vorteil sein kann. Außerdem wird gezeigt, dass die erzeugten neuronalen Netze in weniger Iterationen zu besseren Ergebnissen konvergieren.

abstract

In recent years, neural networks have been widely used as the method of choice for classification tasks in computer vision and image processing. The networks that currently provide the best results are very deep residual Convolutional Neural Networks. New network structures, based on the previously used topologies are developed each year, using different approaches. In this thesis the application of a canonical genetic algorithm to optimize the topology of neural networks is investigated. Current state of the art networks are used as a baseline to start the optimization from. The generated architectures achieve an improved classification accuracy, compared to reference networks. The developed network structures differ from commonly used architectures in their higher level architecture. In contrast to the reference networks they are not homogeneously constructed from repeating elements but heterogeneously from a sequence of different blocks. The improved classification accuracy suggests that the use of heterogeneous structures may be advantageous. Furthermore it is shown that the generated neural networks converge in fewer iterations, producing better results.

0.1 Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder Ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

06.04.2017

Nils Kornfeld

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

NN	Neuronales Netz
KNN	künstliches Neuronales Netz
CNN	Convolutional Neural Net
NEAT	NeuroEvolution of Augmenting Topologies
HyperNEAT	Hypercube based NEAT
kGA	kanonischer Genetischer Algorithmus
EA	evolutionärer Algorithmus
ReLU	Rectified Linear Unit
SVM	Support Vector Machine
ILSVRC	ImageNet Large Scale Visual Recognition Competition
ResNet	Residual Neural Network
WRN	Wide ResNet
BatchNorm	Batch-Normalisierung

1 Einleitung und Motivation der Arbeit

Diese Arbeit wurde beim Deutschen Zentrum für Luft- und Raumfahrt, e. V., im Institut für Verkehrssystemtechnik, in der Arbeitsgruppe Sensorsysteme erstellt.
Im Kontext der Verkehrsforschung fallen große Datenmengen in Form von Bildern und Videomaterial an. Um diese Daten effizient weiter verarbeiten zu können, ist es notwendig, die aufgenommenen Verkehrsobjekte in Modelle zu überführen. Hierzu müssen die relevanten Objekte möglichst genau lokalisiert und verlässlich klassifiziert werden. Da im Rahmen der Verkehrsforschung häufig Echtzeitanforderungen eingehalten werden müssen, ist es wünschenswert, unter der Beibehaltung der Lokalisations- und Klassifikationsgüte die Komplexität und die Laufzeit der eingesetzten Verfahren zu reduzieren.
Zur Klassifizierung der Objekte in Bildern werden heute in großem Maße Technologien aus dem Bereich der künstlichen neuronalen Netze (KNN), insbesondere aus dem Gebiet der Convolutional Neural Networks (CNN), eingesetzt. Diese Technologien liefern Klassifikationsergebnisse, die sich mit der menschlichen Klassifikationsleistung messen können.[1]
Zur Optimierung von Algorithmen und Programmen haben sich evolutionäre Algorithmen als ein probates Mittel erwiesen. Auch zum Training und zur Erzeugung der Struktur künstlicher neuronaler Netze werden solche Verfahren eingesetzt.[2]
Das Ziel dieser Arbeit liegt darin, die Struktur künstlicher neuronaler Netze durch die Anwendung evolutionärer Algorithmen so zu modifizieren, dass ein positiver Effekt auf die Klassifikationsgüte gezeigt werden kann.

2 Theoretische Grundlagen

2.1 Naturanaloge Verfahren und künstliche Intelligenz

Als naturanalog werden diejenigen Verfahren bezeichnet, deren Funktionsweise aus Vorgängen in der Natur durch Beobachtung und Abstraktion abgeleitet sind. Da sich evolutionäre Algorithmen in Anlehnung an die Erkenntnisse zur Funktion der biologischen Evolution entwickelt haben, werden sie diesem Bereich zugeordnet.[3]
Die im Rahmen dieser Arbeit eingesetzten Technologien aus dem Bereich der künstlichen Intelligenz und des maschinellen Lernens basieren auf künstlichen neuronalen Netzen (KNN). Auch diese Algorithmen sind von biologischen Vorbildern abgeleitet und können daher im weitesten Sinne auch als naturanaloge Verfahren angesehen werden.
In der Natur gibt es keine scharfe Trennung zwischen der Evolution, der Neuroevolution und dem Lernen biologischer neuronaler Netze. Alle diese Vorgänge finden gleichzeitig und parallel statt und greifen ineinander. Die Evolution der Nervensysteme biologischer Lebewesen legt die Grundlagen für die Fähigkeit zum Lernen.[4, 5]
Aufgrund dieser in der Natur existierenden Überschneidungen der Vorbilder ist es interessant, ihre Kombination auch in den abgeleiteten Verfahren zu testen und zu bewerten.

2.2 Evolutionäre Algorithmen

Evolutionäre Algorithmen sind naturanaloge Verfahren, mit denen Problemlösungen automatisch generiert werden sollen (siehe [3], S. 33, f.). Vorläufige, nicht optimale Lösungen sollen in Anlehnung an die biologische Evolution durch die Fähigkeit zur Selbstadaption optimiert werden. Die Güte der erzeugten Lösungen wird mithilfe einer dem Problem angepassten Fitness-Funktion bewertet[6]. So kann diese Fitness-Funktion beispielsweise der Zielfunktion des zu Grunde liegenden Optimierungsproblems entsprechen (siehe [3], S. 10, ff.). Der Begriff der Fitness-Funktion leitet sich von dem zur Beschreibung der Evolution oft zitierten Prinzip des *Survival of the Fittest* ab.

Historisch bedingt werden evolutionäre Algorithmen in vier Kategorien eingeteilt[3]:

- Evolutionsstrategien
- Evolutionäre Programmierung
- Genetische Programmierung
- Kanonische Genetische Algorithmen

Die evolutionäre und die genetische Programmierung wurden ursprünglich entwickelt, um gezielt Programme zu erzeugen, welche ihr Ausgabeverhalten stetig verbessern. Sie wurden also explizit für Anwendungsfälle aus dem Bereich der Algorithmik und Informatik entwickelt und nicht, um allgemeine mathematische Optimierungsprobleme zu lösen. Allerdings werden mittlerweile auch Ansätze und Heuristiken aus der evolutionären und genetischen Programmierung in die allgemeineren Verfahren der evolutionären Algorithmen übernommen.
Die Optimierung findet im Suchraum S statt, welcher durch das zugrunde liegende Problem vorgegeben wird. Auf dem Suchraum S wird die Fitnessfunktion $fit : S \to \mathbb{R}$ definiert. Die Zwischenlösungen werden in Anlehnung an die biologische Evolution als Individuen bezeichnet. Die Aggregation aller Individuen zu einem bestimmten Zeitpunkt, bzw. Iterationsschritt t, wird Population genannt. Iterativ werden so nacheinander mehrere Populationen erzeugt und auf diese Weise als verschiedene Generationen voneinander unterschieden. Die Individuen einer Population können durch sogenannte genetische Operatoren verändert, dezimiert oder vermehrt werden. Die Eigenschaften der einzelnen Individuen werden durch jeweils ein zum Individuum zugeordnetes Chromosom kodiert.
Die Chromosomen untergliedern sich wiederum in einzelne Gene, welche je nach eingesetzter Kodierung durch einzelne Bits oder reelle Zahlen repräsentiert werden können. Die Ausprägung eines bestimmten Attributes eines Individuums, welches durch die Gene kodiert wird, bezeichnet man als Allel.[3]

Wenn die Abbildung vom Genom eines Individuums auf seine Allele nicht injektiv ist, wird, genau wie in der Biologie, zwischen Genotypus und Phänotypus unterschieden. Der Genotypus beschreibt die im Genom enthaltene Information, während der Phänotypus die Ausprägung der Allele, also der Eigenschaften des Individuums bezeichnet.[3]

2.2.1 Evolutionsstrategien

Evolutionsstrategien wurden in den sechziger Jahren von Rechenberg und Schwefel für Optimierungsprobleme aus dem technisch-physikalischen Bereich entwickelt und später publiziert (siehe [3], S. 115,[7, 8]). Sie werden genutzt, um reellwertige Zielfunktionen $f : \mathbb{R}^n \to \mathbb{R}$ zu optimieren.
Die Startpopulation $P(0)$ kann entweder zufällig oder bei vorhandenem Vorwissen über das Problem durch entsprechend günstig ausgewählte Stellen innerhalb des Suchraums S erzeugt werden. Alle weiteren Generationen werden durch Mutation erzeugt. Dazu werden λ Nachkommen als Variationen der ursprünglichen Individuen erzeugt. Die Mutation des Chromosoms eines Individuums erfolgt, indem zu den Allelen jedes Gens jeweils eine normalverteilte Zufallszahl mit Erwartungswert $\mu = 0$ und kleiner Streuung σ addiert wird. In diesem Zusammenhang wird die Streuung der Normalverteilung auch als Mutationsschrittweite bezeichnet. Die Mutationsschrittweite σ sollte so klein gewählt werden, dass die erzeugten Kind-Individuen mit großer Wahrscheinlichkeit in der Nähe der Eltern-Individuen bleiben[3]. Bei der Evolutionsstrategie wird normalerweise nach dem Eliteprinzip selektiert. Hierbei werden die μ besten Inividuen in die nachfolgende Generation übernommen. Man unterscheidet bei der Selektion zwischen der Plus- und der Komma-Strategie:
Bei der Plus-Strategie werden die μ besten Chromosomen aus der Vereinigungsmenge der Elternpopulation mit den λ durch Mutation erzeugten Individuen ausgewält. Bei der Komma-Strategie werden die Individuen nur aus der Menge der λ mutierten Individuen gewählt. Bei der Plus-Strategie werden die Individuen, die die bisher besten Lösungen des Optimierungsproblems darstellen immer weiter übernommen. So wird immer von der bisher

besten Lösung weiter optimiert. Dieses Vorgehen birgt allerdings den Nachteil, dass die Gefahr einer vorzeitigen Konvergenz besteht (siehe [3], S. 116). Plus- und Kommastrategie können auch kombiniert eingesetzt werden. Hierzu wird zunächst die Plus-Strategie verwendet. Findet über mehrere Generationen keine Verbesserung des besten Individuums mehr statt, so wird für einige Generationen die Komma-Strategie genutzt, um vorzeitige Konvergenz zu verhindern (siehe [3], S. 117).
Um bessere Ergebnisse zu erzielen, kann die Mutationsschrittweite, bzw. die Varianz σ der Normalverteilung zur Laufzeit angepasst werden. Hierbei unterscheidet man zwischen deterministischer und dynamischer Adaption. Bei der deterministischen Adaption wird der Suchraum anfangs in einem sehr großen Bereich durchsucht, mit einer großen Mutationsschrittweite. Im Laufe der Generationen wird die Varianz schrittweise immer weiter verkleinert, um eine lokale Optimierung zu erreichen. Bei der dynamischen Adaption wird die Mutationsschrittweite σ nicht immer verkleinert, sondern kann auch wieder vergrößert werden, wenn dies sinnvoll erscheint. Die Adaptionsverfahren sind inklusive Rechenbergs 1/5-Regel in [3], S. 117, ff. genauer beschrieben.

2.2.2 Evolutionäre Programmierung

Die evolutionäre Programmierung ähnelt im Vorgehen den fast zeitgleich entwickelten Evolutionsstrategien. Allerdings war das eigentliche Ziel der evolutionären Programme ursprünglich nicht das Lösen von Optimierungsproblemen, sondern das Schaffen problemangepasster endlicher Automaten. Die Startpopulation wird durch gleichverteilte Zufallswerte in einem festgelegten Intervall im Suchraum S erzeugt (siehe [3], S. 125).
Die evolutionäre Programmierung soll an dieser Stelle nur erwähnt und nicht im Detail vorgestellt werden, da sie in dieser Arbeit keine Anwendung findet und in der Interpretation des Evolutionsbegriffes stark von den anderen vorgestellten Teilbereichen abweicht. Die Evolution wird in der evolutionären Programmierung abstrakter als in den anderen Disziplinen betrachtet. Die Anpassung von Individuen an ein gegebenes Problem wird zwar vollzogen, jedoch werden die genauen genetischen Vorgänge, wie z. B. die Rekombinati-

on der Chromosomen, in Anlehnung an die Biologie, in keiner Weise explizit formuliert(siehe [3], S. 125).

2.2.3 Genetische Programmierung

Bei der genetischen Programmierung werden im Gegensatz zu den anderen evolutionären Algorithmen Chromosomen variabler Länge verwendet. Dadurch können z. B. Optimierungsprobleme aus dem Bereich der Regelung gelöst werden. Die Chromosomen können als Programme interpretiert werden, die ihr Ein- und Ausgabeverhalten gewissen Zielen, die durch Rahmenbedingungen gegeben sind, anpassen (siehe [3], S. 127). Da keine Ansätze der evolutinären Programmierung und der genetischen Programme in dieser Arbeit genutzt wurden, wird an dieser Stelle nicht genauer auf diese Verfahren eingegangen. Für weitere Informationen zur genetischen Programmierung siehe [3], S. 127, ff.

2.2.4 Kanonische Genetische Algorithmen

Auch die kanonischen Genetischen Algorithmen (kGA) bilden eine Untergruppe der evolutionären Algorithmen (EA).
Zur Repräsentation der Allele wird ein binäres Alphabet verwendet. Um Teilmengen der reellen Zahlen als Lösungsraum verwenden zu können, muss eine Kodierungs- /Dekodierungsfunktion definiert werden. Dadurch werden reelle Zahlen auf eine Binärdarstellung abgebildet (siehe [3], S. 36, f.).
Bei der Erzeugung der Startpopulation mit der Populationsgröße *popsize* wird, wie bei den Evolutionsstrategien, bereits auf die Einhaltung gewisser Rahmenbedingungen geachtet, um frühzeitig Chromosomen mit einem hohen Fitness-Wert in die Population einzufügen. Dieses Verfahren birgt allerdings das Risiko, in eine vorzeitige Konvergenz gegen ein lokales Optimum zu münden (siehe [3], S. 37). Mithilfe der genetischen Operatoren Crossover und Mutation werden Folgegenerationen erzeugt (siehe [3], S. 35). Zur Reproduktion müssen nicht zwingend beide genetischen Operatoren genutzt werden. Da jedoch der Crossover-Operator zur stärkeren Durchsetzung bereits vergleichbar guter Individuen sorgt und der Mutationsoperator zu einer

breiteren Durchsuchung des Suchraumes führt, ist eine Kombination von Vorteil.
Im folgenden Pseudocode ist der grundsätzliche Ablauf eines kanonischen Algorithmus, aus [3], S. 41 dargestellt. Zu Beginn wird die Zählvariable t mit

```
 1: procedure KGA
 2:     t := 0
 3:     Initialisiere P(0)
 4:     Bewerte P(0)
 5:     Setze P*(t) := ∅
 6:     while ( nicht Abbruchkriterium ) do
 7:         t := t + 1
 8:         for (i = 0; i < popsize ; i++) do
 9:             Auswahl des i-ten Chromosoms für P*(t) aus P(t − 1)
10:             Anwendung genetischer Operatoren auf P*(t)
11:             P(t) := P*(t)
12:             Bewerten von P(t)
13:         end for
14:     end while
15: end procedure
```

dem Wert null initialisiert. Die Startpopulation $P(t = 0)$ wird zufällig oder gezielt, durch Vorwissen über die Struktur des Suchraumes initialisiert. In der dargestellten Schleife werden mit Hilfe genetischer Operatoren so lange Folgegenerationen erzeugt und bewertet, bis ein vorher definiertes Abbruchkriterium erreicht wird.
Wie bereits erwähnt, werden traditionell im kanonischen Genetischen Algorithmus der Crossover- und der Mutationsoperator eingesetzt, um die genetische Struktur der folgenden Generation zu manipulieren (siehe [3], S. 35). Beim Crossover werden durch Austausch von Teilketten zweier schon bestehender Chromosomen neue Chromosomen erzeugt, die im besten Falle eine höhere Fitness als die Elternchromosome aufweisen. Man unterscheidet zwischen One-Point-Crossover, wobei ein Chromosom nur an einem Punkt in zwei Teilketten aufgeteilt wird und dem Multi-Point-Crossover, bei dem das Chromosom stärker partitioniert wird (siehe [3], S. 40). Der One-Point-Crossover ist bildlich in Abbildung 1 auf der nächsten Seite dargestellt. Aus

Abbildung 1 – Cross-Over (nach [3], S. 40)

zwei Elternindividuen der Generation t werden zwei Kindindividuen der Generation $t+1$ erzeugt, die genetische Merkmale beider Eltern enthalten. Bei dieser Form des One-Point-Crossover gehen keine Allele aus der vorhergehenden Generation verloren.
Der Mutationsbegriff wird im Bereich genetischer Algorithmen weniger allgemein benutzt als in der Biologie, so dass hier mit Mutation immer eine Punktmutation gemeint ist. Die Mutation bewirkt bei einem kGA das Umklappen eines zufällig ausgewählten Bit im Chromosom zu seinem Komplement (siehe [3], S.41). Zur Verdeutlichung ist die Mutation in Abbildung 2 schematisch dargestellt. In der Abbildung werden zwei einzelne Gene punktmutiert. Gemäß [3] sollte die Mutationswahrscheinlichkeit eines Gens g_i in Abhängigkeit von der Länge des Chromosoms c gewählt werden:

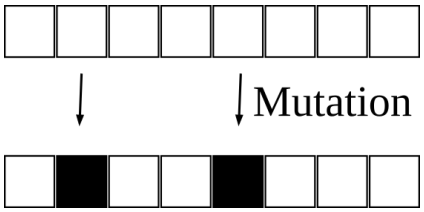
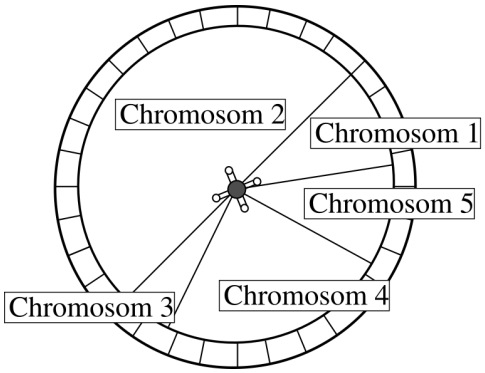


Abbildung 2 – Mutation (nach [3], S. 41)

$$p_{mut}(g_i) = \frac{1}{|c|} \tag{1}$$

So wird gewährleistet, dass der Erwartungswert der Anzahl der mutierten Gene pro Individuum 1 beträgt. Diese Restriktion sorgt dafür, dass bereits vorteilhafte genetische Schemata nur mit einer geringen Wahrscheinlichkeit



Chromosom	*fit(c)*	*p(c)*
1	30	0,1
2	150	0,5
3	15	0,05
4	75	0,25
5	30	0,1
	300	1

Abbildung 3 – Rouletterad mit Überlebenswahrscheinlichkeiten (nach [3], S. 39)

zerstört werden (siehe [3], S. 55).
Die verwendete Fitnessfunktion entspricht der Zielfunktion des zu Grunde liegenden Optimierungsproblems. Aus der Fitnessfunktion ergibt sich gemäß Gleichung (2) die Überlebenswahrscheinlichkeit $p(c_0)$ eines Chromosoms c_0 (siehe [3], S. 38).

$$p(c_0) = \frac{fit(c_0)}{\sum_{c \in P(t)} fit(c)} \tag{2}$$

$P(t)$ ist die gesamte Population zum Zeitpunkt t.
Als Selektionsverfahren wird eine sogenannte Rouletterad-Selektion verwendet. Dazu werden *popsize* Chromosomen gemäß der nach Gleichung (2) berechneten Wahrscheinlichkeiten aus der vorhergehenden Generation $P(t-1)$, deren Populationsgröße mithilfe der beschriebenen genetischen Operatoren bereits erhöht wurde, gezogen. Um die Analogie zu einem Rouletterad nachvollziehen zu können, ist in Abbildung 3 ein graphisches Beispiel einer solchen Selektion dargestellt. In der vorliegenden Population sind fünf Individuen vorhanden, denen jeweils eine Fitness zugeordnet ist. Die Fitness ist der Tabellenspalte $fit(c)$, rechts in der Abbildung zu entnehmen. Die in der nächsten Spalte dargestellten Überlebenswahrscheinlichkeiten werden mit Gleichung (2) berechnet. Auf dem Rouletterad wird der Umfang des gesamten Rades in Bereiche, proportional zur Überlebenswahrscheinlichkeit der

Individuen, aufgeteilt. Die zufällige Auswahl der Individuen für die folgende Generation entpricht nun einer Roulette-Runde auf dem so angepassten Rad.

2.3 Künstliche neuronale Netze

Auch die künstlichen neuronalen Netze zählen wie die evolutionären Algorithmen zu den naturanalogen Verfahren. Außerdem werden sie den Bereichen der künstlichen Intelligenz und des maschinellen Lernens zugerechnet. Im Gegensatz zu explizit programmierten Algorithmen können neuronale Netze durch die Anpassung ihrer inneren Parameter (Gewichte) erlernen, aus Eingabedaten gewünschte Ergebnisse zu erzeugen.
Die Grundzüge der Modellierung künstlicher Neuronen wurden 1943 von McCulloch und Pitts, in [9] entwickelt. Um ein logisches Problem zu lösen wird ein vereinfachtes Modell biologischer neuronaler Netze heran gezogen. In der ursprünglichen Arbeit wurden einfache neuronale Netze zur Erzeugung der boolschen Funktionen **AND**, **OR** und **NOR** verwendet[9]. Die Autoren gehen in dieser sehr frühen Arbeit schon so weit, in der Funktionsweise der neuronalen Netze den Mechanismus zur Bewusstseinskonstitution zu verorten[9]. Daher sind neuronale Netze nicht nur in der angewandten Informatik als relativ universelle Problemlösungsstrategien interessant, sondern es gibt auch Überschneidungen mit anderen Bereichen der künstlichen Intelligenz, mit der Neurobiologie, der Psychologie und der Philosophie. So verorten auch Dawkins und Crick in ihren Werken [4] und [5] die Grundlagen des Bewusstseins in der Funktionsweise der neuronalen Netze.
In künstlichen neuronalen Netzen werden die Nervenzellen durch mathematische Funktionen modelliert. Die Eingangsvariablen werden als die Eingangssignale in den Dendriten dargestellt. Diese Eingangssignale werden gemäß der sogenannten Aktivierungsfunktion im Zellkörper auf die Ausgabe der Funktion, welche dem Axonpotential an den Axonterminalen in der biologischen Nervenzelle entspricht, abgebildet (siehe Abbildung 4 auf der nächsten Seite). Den Synapsen entsprechen die Verkettungen mehrerer mathematischer Funktionen nach diesem Prinzip. Den Synapsen werden anpassbare, innere Parameter des Netzes zugeordnet. Diese werden als Gewichte bezeichnet.

Die anpassbaren Gewichte ermöglichen das Lernen des Systems (siehe [10], S. 23, ff.).

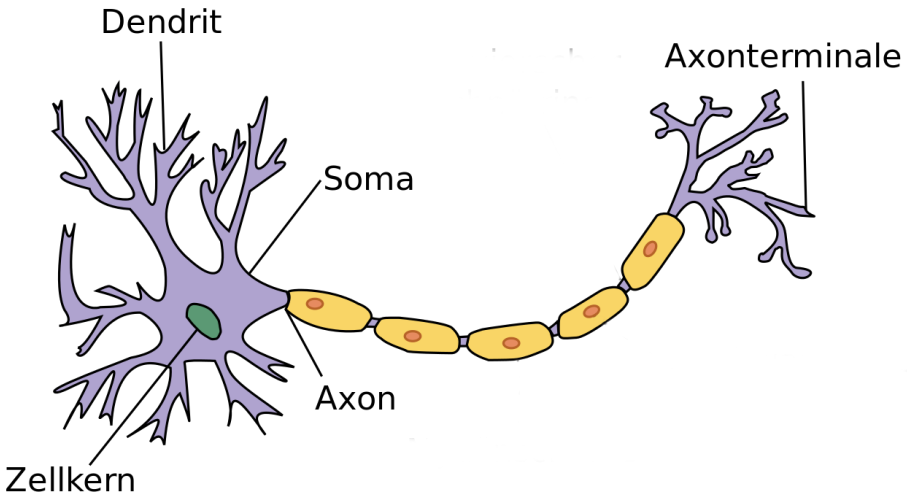


Abbildung 4 – Schemazeichnung eines Neurons[1]

Um die Analogie zwischen den künstlichen neuronalen Netzen und dem biologischen Vorbild zu veranschaulichen ist in Abbildung 4 ein biologisches Neuron und in Abbildung 5 auf der nächsten Seite ein einschichtiges Perzeptron, in Anlehnung an Minsky und Papert in [11], S. 149, ff. sowie Gallant in [12] dargestellt. Die Eingangsdaten $x_1 \dots x_n$ entsprechen den Eingangspotentialen an den Dendriten. Im Zellkörper (Soma) werden die Potentiale aufsummiert und als Eingabeparameter an die Aktivierungsfunktion Θ übergeben. In diesem ersten, einfachen Modell wird die Aktivierungsfunktion durch einen Schwellenwert dargestellt. Das Ausgangssignal o entspricht dem Aktivierungspotential an den Axonterminalen im biologischen Neuron.[10]

2.3.1 Aufbau neuronaler Netze

Der Aufbau künstlicher neuronaler Netze wird verständlicher, wenn man zunächst analytisch die Grundelemente betrachtet. Diese werden zu einer Makrostruktur, dem Netz, zusammengesetzt. Die Struktur der heute einge-

[1] Bildquelle: Quasar Jarosz, in der englischsprachigen Wikipedia, cc-by-sa 3.0, `https://creativecommons.org/licenses/by-sa/3.0/deed.de`

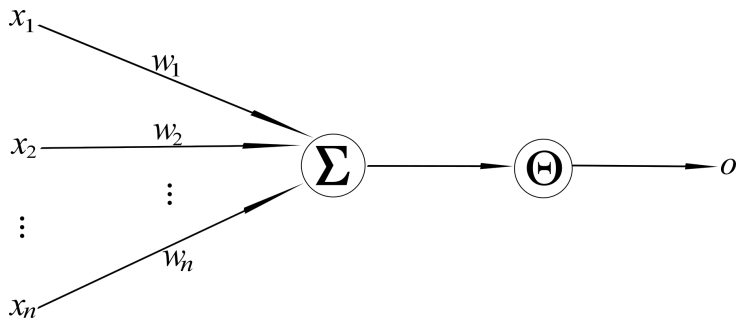


Abbildung 5 – Funktionsweise eines einzelnen Perzeptrons (nach [11, 12, 13])

setzten künstlichen neuronalen Netze basiert auf dem 1958 von David Rosenblatt eingeführten Perzeptron (siehe Abbildung 6, [14]). In dieser Arbeit wird das Perzeptron als eine Annäherung der Bildverarbeitung im menschlichen Gehirn betrachtet. Eingangssignale werden an der Retina erfasst und über lokale Verbindungen zu einem Projektionsbereich weiter geleitet. Die weitere Fortpflanzung der Signale im Netz erfolgt über zufällige, gewichtete Verbindungen zum Bereich A_{II}. Hier werden die Signale aufsummiert und, falls sie den Aktivierungsschwellwert Θ überschreiten, als Ausgangssignale ausgegeben.

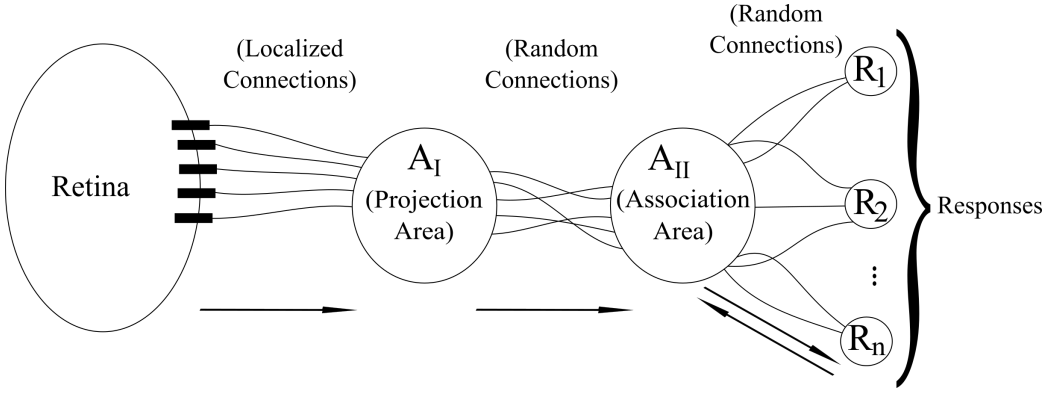


Abbildung 6 – Das klassische Perzeptron (nach [14])

Die grobe Übereinstimmung der Funktionsweise des von Rosenblatt eingeführten Perzeptrons mit dem biologischen Vorbild wurde in [15, 16] experimentell bestätigt. Die genauere Funktionsweise, abgeleitet aus den Ergebnissen in [15], ist in [17] beschrieben.

Die Gewichte der einzelnen Kanten werden zunächst zufällig initialisiert. Damit ein solches Perzeptron als Klassifikator eingesetzt werden kann, muss es lernen können. Rosenblatt schlägt in [13] einen Algorithmus für ein einschichtiges Perzeptron (siehe Abbildung 5 auf der vorherigen Seite) vor, welcher meist einfach als Perzeptron-Lernalgorithmus bezeichnet wird[13]:
Die Gewichte der Eingänge des Perzeptron werden zufällig initialisiert. Wird im Perzeptron der Schwellenwert Θ überschritten, so wird ein Eingangsdatum der positiven Klasse zugeordnet:

$$\sum(x_i \cdot w_i) > \Theta \qquad \Rightarrow y = 1 \tag{3}$$

Wird der Schwellenwert nicht überschritten, so wird das Datum der negativen Klasse zugeordnet:

$$\sum(x_i \cdot w_i) <= \Theta \qquad \Rightarrow y = -1 \tag{4}$$

Wird das Datum korrekt der richtigen Klasse zugeordnet, so wird das nächste Datum klassifiziert. Wird das Datum falsch klassifiziert, werden die Gewichte w_i folgendermaßen angepasst. Wenn ein Element der positiven Klasse falsch klassifiziert wurde, wird der Eingangswert auf das zugehörige Gewicht addiert:

$$w_i := w_i + x_i \tag{5}$$

Wurde ein Element der negativen Klasse falsch klassifiziert, werden die Gewichte durch die Subtraktion des Eingangswertes angepasst:

$$w_i := w_i - x_i \tag{6}$$

Das einschichtige Perzeptron kann, wie in [11] gezeigt wird, die logische Funktion **XOR** nicht abbilden. Hierzu wird bereits ein neuronales Netz mit mindestens zwei Schichten benötigt. Um ein Gradientenabstiegsverfahren bei mehrschichtigen neuronalen Netzen anwenden zu können, wird der Backpropagation-Algorithmus eingesetzt (siehe [10], S. 151).

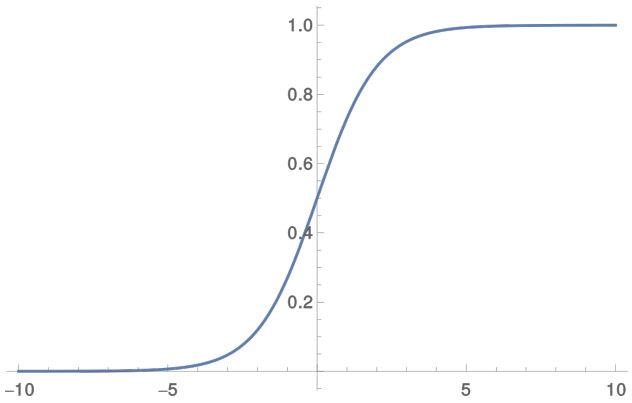


Abbildung 7 – Die Sigmoid-Funktion $s(x)$

2.3.2 Der Backpropagation Algorithmus

Der bereits vorgestellte Perzeptron-Lernalgorithmus kann nicht für mehrschichtige neuronale Netzwerke eingesetzt werden. Um auch tiefere Netze trainieren zu können, wird der Backpropagation-Algorithmus genutzt. 1986 wurde in [18] gezeigt, dass dieser Algorithmus zu sinnvollen Repräsentationen in künstlichen neuronalen Netzen führt. Die Gewichte der Knoten sollen über mehrere Schichten hinweg, in Abhängigkeit von einer Fehlerfunkton E, am Ausgang des Netzes angepasst werden. Diese Fehlerfunktion wird im Allgemeinen auch als Loss-Funktion bezeichnet.
Die Gewichte, die die Fehlerfunktion minimieren, sollen in einem Gradientenabstiegsverfahren gefunden werden. Um den Gradienten an jeder Kante berechnen zu können, ist es notwendig, dass die Aktivierungsfunktionen der einzelnen Neuronen differenzierbar sind (siehe [10], S. 151). Daher wird der Aktivierungsschwellwert Θ durch die Sigmoid-Funktion $s(x)$ ersetzt:

$$s(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

Die Sigmoid-Funktion stellt, wie in Abbildung 7 zu erkennen ist, eine Annäherung an die Stufenfunktion dar.
Um später die Gradienten im Netz bestimmen zu können, wird die Ableitung

der Sigmoid-Funktion benötigt:

$$s(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} \tag{8}$$

$$\begin{aligned} \frac{\mathrm{d}s(x)}{\mathrm{d}x} &= \frac{e^x(1+e^x)-(e^x)^2}{(1+e^x)^2} = \frac{e^x}{1+e^x} - \frac{e^{2x}}{(1+e^x)^2} = \frac{e^x}{1+e^x} \cdot \left(1 - \frac{e^x}{1+e^x}\right) \\ \Rightarrow \frac{\mathrm{d}s(x)}{\mathrm{d}x} &= s(x)(1-s(x)) \end{aligned} \tag{9}$$

Der Backpropagation-Algorithmus ist allerdings auf Netzwerke mit beliebigen, differenzierbaren Aktivierungsfunktionen anwendbar (siehe [10], S. 153). Da die Netzwerkfunktion $\mathcal{F}$, die durch das gesamte Netzwerk erzeugt wird, eine Verkettung von Summen von Aktivierungsfunktionen ist, ist diese auch differenzierbar, wenn alle Aktivierungsfunktionen differenzierbar sind (siehe [10], S. 153).
Das neuronale Netz kann als Graph aufgefasst werden, dessen Knoten Recheneinheiten sind und über dessen Kanten Daten ausgetauscht werden. Die Netzwerkfunktion $\mathcal{F}$ ist also die Verkettung der primitiven Funktionen f_{lk} in den einzelnen Knoten. Das Ausgabeverhalten des Netzwerkes soll so angepasst werden, dass eine Menge von Trainingsdaten $(\vec{x}_1 \dots \vec{x}_n)$ richtig klassifiziert wird. Dazu ist die Trainingsmenge X annotiert, d. h., jedem Trainingsvektor $\vec{x}_i \in X$ ist ein Label t_i (t von engl. *tag*) zugeordnet (siehe [10], S. 105, ff.).
Die Ausgabe o des Netzwerkes soll also so angepasst werden, dass bei der Eingabe von $\vec{x}_i \in X$ sich $o_i = \mathcal{F}(\vec{x}_i)$ möglichst nah an t_i annähert. Um die Abweichung zu beziffern wird eine auf der L^2-Norm basierende Loss-Funktion E eingeführt[18]:

$$E = \frac{1}{2}\sum_{i=1}^{n} \|\mathcal{F}(\vec{x}_i) - t_i\|^2 = \frac{1}{2}\sum_{i=1}^{n} \|o_i - t_i\|^2 \tag{10}$$

Um die Gewichte im Graphen anpassen zu können, wird an jeder Kante, mit den zugehörigen Gewichten w_{lk} der Gradient der Loss-Funktion, bzgl. w_{lk} gebildet. Die Gewichte werden iterativ gemäß einem Gradientenabstieg, mit

der Schrittweite γ angepasst[18]:

$$\boxed{\Delta w_{lk} = -\gamma \frac{\partial E}{\partial w_{lk}}} \tag{11}$$

In jedem Iterationsschritt werden die Gewichte gleichzeitig entlang der negativen Gradientenrichtung angepasst. Hierzu kann der Gradient mithilfe der Kettenregel vom Ausgang des Netzes, über alle versteckten Schichten hinweg, bis zur Eingabeschicht fortgepflanzt werden (siehe [10], S. 164, ff.). Das Verfahren kann an Abbildung 5 auf Seite 12 verdeutlicht werden. Am Ausgang wird der Gesamtfehler bzgl. aller gelabelten Eingabedaten (siehe Gleichung (10)) aggregiert[18]. Hier wird nun außerdem der Gradient bzgl. der Ausgangsdaten o_i berechnet[10]:

$$\frac{\partial E}{\partial o_i} = o_i - t_i \tag{12}$$

Um den Einfluss des Loss E zur Berechnung der Anpassung des Gewichtes w_1 in Abbildung 5 auf Seite 12 durchzuschleifen, wird die Kettenregel angewendet[18]. Der Loss im Perzeptron aus Abbildung 5 auf Seite 12, mit einer Trainingsmenge X aus n Elementen, ergibt sich folgendermaßen:

$$E = \frac{1}{2} \sum_{i=1}^{n} \left\| \left(s\left(x_{i1} \cdot w_1 + x_{i2} \cdot w_2 + \cdots + x_{im} \cdot w_m \right) - t_i \right) \right\|^2 \tag{13}$$

Der Gradient bzgl. des ersten Elementes des Eingabevektors $\vec{x}_i$ kann mithilfe der Kettenregel ausgedrückt werden:

$$\frac{\partial E}{\partial x_{i1}} = \frac{\partial E}{\partial o_i} \cdot \frac{\partial o_i}{\partial x_{i1}} \cdot w_1 \tag{14}$$

Da die Sigmoid-Funktion s als Aktivierungsfunktion genutzt wird, gilt: $o = s(x)$. Mit Gleichung (9) folgt:

$$\frac{\partial o_i}{\partial x_{i1}} = o_i \cdot (1 - o_i) \tag{15}$$

Setzt man nun Gleichung (15) in Gleichung (14) ein, so ergibt sich:

$$\frac{\partial E}{\partial x_{i1}} = (o_i - t_i) \cdot o_i \cdot (1 - o_i) \cdot w_1 \tag{16}$$

Um das Gewicht w_1 anpassen zu können, wird statt nach x_{i1} nach w_1 abgeleitet:

$$\frac{\partial E}{\partial w_1} = (o_i - t_i) \cdot o_i \cdot (1 - o_i) \cdot x_{i1} \tag{17}$$

Wenn das neuronale Netz aus noch weiteren Schichten besteht, kann das Ergebnis von Gleichung (16) verwendet werden, um den Gradienten zu den weiteren Schichten zurück zu propagieren. Dazu wird für jede weitere Schicht nochmals die Kettenregel angewendet ([10], S. 165, f.). Die Anpassung des Gewichtes w_1 ergibt sich mit Gleichung (17) zu:

$$\Delta w_1 = -\gamma \, (o_i - t_i) \, o_i \, (1 - o_i) \cdot x_{i1} \tag{18}$$

Die Gewichte werden iterativ so lange korrigiert, bis ein ausreichend geringer Fehler am Ausgang des Netzes erzielt wird[10].
Um kleinere Gewichte bei der Optimierung zu bevorzugen, kann auf den Loss des Netzwerkes ein Regularisierungsterm addiert werden. Der Regularisierungsterm basiert auf der L^2-Norm. Der modifizierte Loss ist in Gleichung (19) dargestellt.

$$E = \frac{1}{2} \sum \|o_i - t_i\|^2 + \lambda \frac{1}{2} \sum w_j^2 \tag{19}$$

Der Regularisierungsterm wird mit dem Parameter λ multipliziert. Dieser Parameter wird auch als *weight decay* bezeichnet.
Um den Gradientenabstieg mit dem Backpropagation-Algorithmus zu beschleunigen, kann das Verfahren um ein Momentum erweitert werden. Dabei kann der Algorithmus beim Abstieg in Anlehnung an die Trägheit in der Natur einen gewissen Schwung aufnehmen. Dazu wird die Anpassung der Gewichte aus Gleichung (11) folgendermaßen erweitert:

$$\Delta w_{lk}(t) = -\gamma \frac{\partial E}{\partial w_{lk}} + \alpha \Delta w_{lk}(t-1) \tag{20}$$

Gemäß Gleichung (20) wird auf die Anpassung der Gewichte im aktuellen Additionsschritt ein Schritt in die Richtung der vorherigen Anpassung addiert. Dieser Schritt wird mit dem Parameter α gewichtet (siehe [10], S. 186, f.).

2.3.3 Rectified Linear Units (ReLU)

Die in Abbildung 7 auf Seite 14 dargestellte Sigmoid-Funktion birgt für den gerade vorgestellten Backpropagation-Algorithmus den Nachteil, dass der durch sie erzeugte Gradient bei einer hohen Aktivierung gegen null geht. Da der Gradient in einem tiefen Netz über mehrere Schichten hinweg zurück propagiert werden muss, kann bei mehreren aufeinander folgenden, hohen Aktivierungen der Gradient *verschwinden*. Dieses Phänomen wird in der enlischsprachigen Literatur als *vanishing gradients* bezeichnet. Daher können ohne weitere Anpassungen nur sehr flache Netze mit dem Backpropagation-Algorithmus und der Sigmoid-Funktion als Aktivierung trainiert werden. Um dem Problem zu entgehen, wird häufig statt der Sigmoid-Funktion eine ReLU-Schicht verwendet. Die Abkürzung ReLU steht für Rectified Linear Unit. Die durch eine ReLU-Schicht abgebildete Funktion ist in Gleichung (21) und Abbildung 8 auf der nächsten Seite dargestellt.

$$\mathrm{ReLU}(x) = \max(0, x) \tag{21}$$

Obwohl die ReLU-Funktion nicht stetig differenzierbar ist, weist sie doch abschnittsweise einen von 0 verschiedenen, konstanten Gradienten auf. Dadurch kann der Gradient bei aktivierten Neuronen zurückpropagiert werden, während er bei nicht aktivierten Neuronen 0 entspricht. Die ReLU-Funktion ermöglicht es somit, tiefe neuronale Netze zu trainieren.[19]

2.3.4 Convolutional Neural Networks

Für Probleme aus der Bildverarbeitung und dem maschinellen Sehen werden heute vor allem Convolutional Neural Networks (CNN) eingesetzt. Die Grundlagen hierfür wurden 1998 von LeCun, et al. geschaffen[19]. Die Funk-

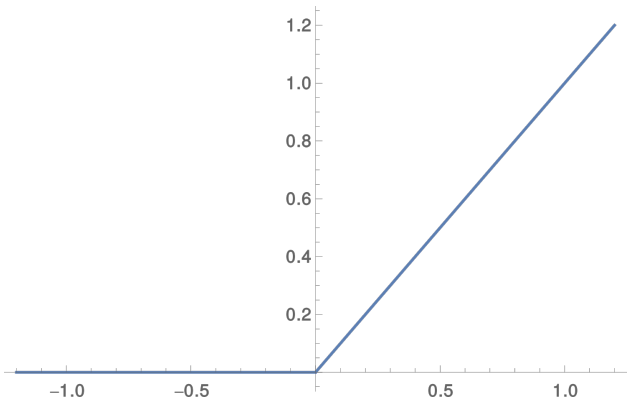


Abbildung 8 – Die ReLU (Rectified Linear Unit)-Funktion

tionsweise der CNNs kann sowohl aus Algorithmen der klassischen Bildverarbeitung, als auch aus den Arbeiten von Yang, Hubel und Wiesel in [16] und [15] abgeleitet werden, die das rezeptive Feld im optischen Kortex von Katzen untersuchen.
Künstliche neuronale Netzwerke werden häufig zur Bildverarbeitung eingesetzt, da sie sich als Verfahren aus dem Bereich des maschinellen Lernens selbst anpassen können und daher recht universell einsetzbar sind. Die in den vorherigen Kapiteln beschriebenen Fully Connected Neural Networks, deren Schichten jeweils das innere Produkt der zugehörigen Gewichte $w_{l,k}$ mit den Aktivierungen der vorhergehenden Schicht $o_{l-1,k}$ beschreiben, unterliegen in ihrer Klassifikationsgüte allerdings häufig traditionellen Bildverarbeitungs-Toolchains, wie sie in Abbildung 9 dargestellt sind. Um auf einen selbst

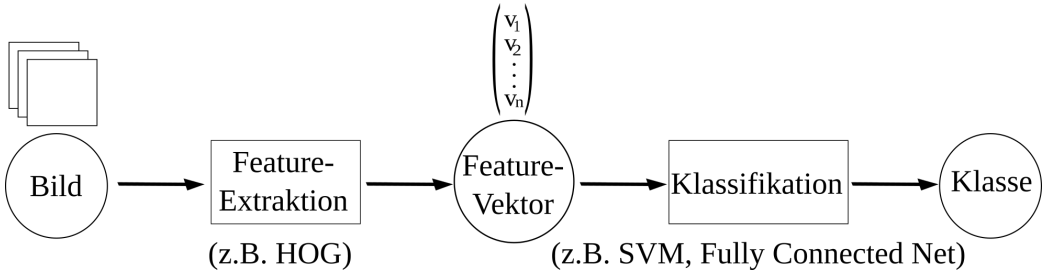


Abbildung 9 – Eine traditionelle Bildverarbeitungs-Toolchain

konstruierten und an das zu Grunde liegende Klassifizierungsproblem angepassten Algorithmus zur Feature-Extraktion verzichten zu können, erscheint

es sinnvoll, auch diesen Teil der Toolchain selbstadaptierend zu entwerfen. Mit einem Convolutional Neural Network (CNN) wird die in Abbildung 10 dargestellte, vereinfachte Toolchain ermöglicht. Würde statt des CNN in

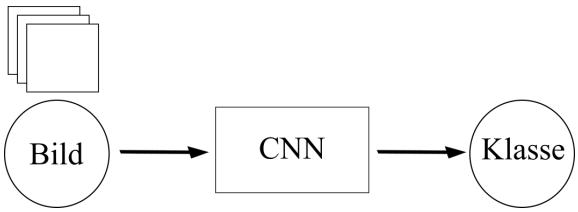


Abbildung 10 – Durch ein CNN vereinfachte Bildverarbeitungs-Toolchain

Abbildung 10 ein Fully Connected-Netzwerk nach der bisher beschriebenen Struktur verwendet, so würde die Anzahl der Gewichte in der ersten Schicht, aufgrund des sehr großen Eingangsvektors, der ein Bild repräsentiert, unangenehm groß werden[20]. Da die Anzahl der zu optimierenden Parameter hierdurch signifikant wachsen würde, wären deutlich größere Trainingsmengen erforderlich[20, 19]. Außerdem haben Fully Connected Networks keine inhärente Translationsinvarianz, welche für die Feature-Extraktion notwendig ist. Hieraus folgt, dass die Trainingsbilder vor der Bearbeitung durch das neuronale Netz so vorprozessiert werden müssten, dass bestimmte Bildeigenschaften an jeder beliebigen Position in der Trainingsmenge auftreten.
Im Gegensatz zu dem Fully Connected Neural Network, das in Abbildung 6 auf Seite 12 eher dem Bereich zwischen A_I und A_{II} entspricht, orientiert sich ein CNN eher an der biologischen Prozessierung von der Retina bis zum assoziativen Bereich[17, 21, 20, 15].
CNNs gehen von lokalen, rezeptiven Feldern am Eingang des Netzwerkes aus[20, 19]. Die Gewichte der Kanten im Netzwerk-Graphen werden über die einzelnen rezeptiven Bereiche hinweg geteilt. Das Teilen der Gewichte, die über ein relativ kleines Feld wiederholt auf das Eingangsbild gefaltet werden, erzeugt im Laufe der Backpropagation-Operationen traditionelle Bildverarbeitungsfilter in der Eingangsschicht[19]. Als Beispiel solcher Filter kann der Sobelfilter betrachtet werden:

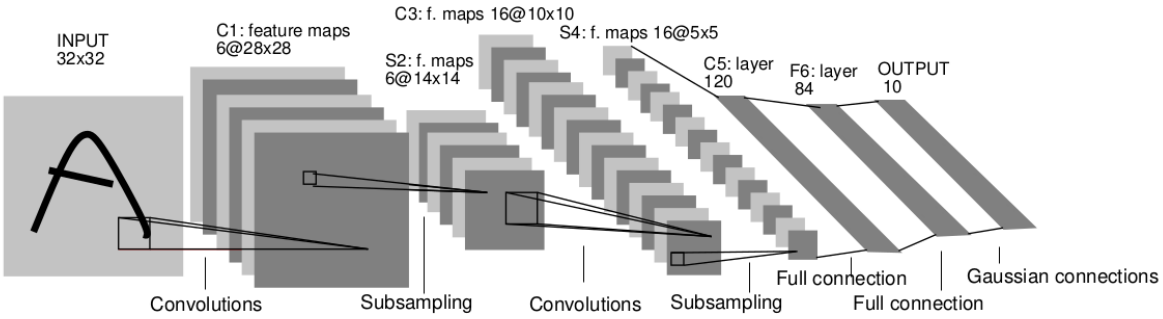


Abbildung 11 – Architektur von LeNet-5. Jede Fläche stellt eine Featuremap dar, die durch Filter mit geteilten Gewichten erzeugt wurden. (aus [19])

Sobelfilter in horizontaler und vertikaler Richtung:

$$S_H = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} S_V = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{22}$$

Die in Gleichung (22) gezeigten Sobelfilter können zur Kantenextraktion genutzt werden. Die Anwendung der Filter auf jedem Pixel des Eingangsbildes sorgt für die Translationsinvarianz der extrahierten Features[19].
Einer Konvolutionsschicht folgt traditionell eine Pooling-Schicht, welche aus einer definierten Anzahl an Eingangssignalen entweder den Durchschnitt[22] oder das Maximum[21] bestimmt und so für die nächste Schicht die Anzahl der Variablen verringert. In Analogie zur traditionellen Bildverarbeitung können die Pooling-Schichten als Tiefpassfilter, die eine Bild/Feature-Pyramide erzeugen, betrachtet werden[23]. Die Bildpyramide sorgt für die benötigte Skalierungsinvarianz der extrahierten Features.
In Abbildung 11 ist die Struktur von LeNet-5, einem CNN zur Handschrifterkennung aus [19] dargestellt. LeNet-5 gilt als Vorbild für viele folgende CNNs. Aus dem Eingangsbild links in Abbildung 11 werden durch eine Konvolutionsschicht sechs Featuremaps extrahiert. Jede einzelne Featuremap wird durch ein Filter, welches mit geteilten Gewichten auf die Pixel im Eingangsbild gefaltet wird, erzeugt. Die sechs Featuremaps entsprechen also sechs verschiedenen Filtern oder Feature-Extraktoren. Die Gewichte

der Filter werden beim Training des Netzwerkes angepasst[19]. Als nächstes folgt eine Pooling-Schicht, die die Ausdehnung der Feature-Maps in Breite und Höhe verringert. Dieses Subsampling findet gemäß der Interpretation der Autoren des GoogLeNet-Papers auch eine Entsprechung im biologischen Vorbild.[20, 19]
Auf die so erzeugten Feature-Maps werden wiederum Faltungsmasken, diesmal allerdings mit 16 verschiedenen Filtern, angewendet. Der Teil des in Abbildung 11 auf der vorherigen Seite dargestellten Netzes bis einschließlich der Schicht S4 ist die CNN-Architektur, die als Feature-Extraktor interpretiert werden kann. Die folgenden Fully-Connected-Layers entsprechen dem in Abbildung 9 auf Seite 19 abgebildeten Klassifikator. Die CNNs erreichen hohe Ergebnisse bei Bild-Klassifikationswettbewerben[2].

2.3.5 Batch-Normalisierung

Damit die durch Bilder erzeugte Aktivierung am Eingang eines CNN nicht beliebig gestreut ist und hinsichtlich der numerischen Stabilität nicht Werte beliebiger Größenordnungen annehmen kann, werden die Eingangsdaten normalisiert[19, 25].
Wie bereits erwähnt, kann die Netzfunktion $\mathcal{F}(x)$ als die Verkettung der Funktionen in den einzelnen Schichten aufgefasst werden:

$$\mathcal{F}(x) = f_L(f_{L-1}(\dots f_2(f_1(x)))) \tag{23}$$

Von der oben angesprochenen Normalisierung der Eingangsdaten sind die Eingangsparameter des Subnetzes $f_L(f_{L-1}(\dots f_2(y)))$ nicht betroffen. Auf dieselbe Weise wie in Gleichung (23) können auch die verbleibenden konstruierbaren Subnetzfunktionen durch Verkettung der einzelnen Schichten zerlegt werden. Auch die auf diese Weise erzeugten Subnetze profitieren nicht direkt von der Normalisierung der Trainingsdaten am Eingang des Gesamtnetzwerkes[25].

[2]Die Ergebnisse und Veröffentlichungen zu bekannten Klassifikations-Challenges können unter `https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html` abgerufen werden[24].

Daher schlagen die Autoren von [25] vor, auch eine Normalisierung der Eingangsdaten aller konstruierbaren Subnetze vorzunehmen. Da die Normierung der Daten innerhalb des Netzes in jeder Schicht bzgl. der Kovarianzmatrix $\Sigma_{X,l}$ der Aktivierung aller Datensätze aus der Trainingsmenge in jedem Iterationsschritt des Gradientenabstiegs zu rechenintensiv wäre, um praktisch einsetzbar zu sein, wird ein alternatives Verfahren vorgeschlagen. Da häufig mit Minibatches trainiert wird, könnten Kovarianz Σ und Mittelwert μ über die Minibatches bestimmt werden und zur Normalisierung der Aktivierungen im Netz genutzt werden. Weil die Möglichkeit besteht, dass die Minibatch-Größe kleiner als die Anzahl der Parameter in der Schicht ist, deren Aktivierungen normalisiert werden sollen, würde hierdurch eine singuläre Kovarianzmatrix erzeugt. Daher wird vorgeschlagen, stattdessen parameterweise die Varianz σ_B^2 zu bilden und so Parameter für Parameter die Aktivierungsvektoren zu normalisieren[25]:

$$\mu_B(X) = \frac{1}{n}\sum_{i=1}^{n}\vec{x}_i \tag{24}$$

$$\sigma_B^2(\vec{x}_i, X) = \frac{1}{n}\sum_{i=1}^{n}(\vec{x}_i - \mu_B)^2 \tag{25}$$

So ergibt sich die normalisierte Aktivierung $\hat{x}_i$:

$$\hat{x}_i = \frac{\vec{x}_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{26}$$

Der zusätzliche Parameter ϵ wird eingeführt, um die numerische Stabilität zu gewährleisten[25].
Da die Normalisierung gemäß [25, 23, 26] vor der Nichtlinearität, bzw. der Aktivierungsfunktion vorgenommen wird, kann sie dazu führen, dass die Eingangsvariablen nur im linearen Teil der Funktion liegen. Um dieses Verhalten zu verhindern, werden zwei weitere lernbare Parameter γ_B und β eingeführt. Im letzten Schritt des Batch-Normalisierungs-Algorithmus wird der bisher berechnete Wert $\hat{x}_i$ um β verschoben und mit γ_B skaliert.[25]

$$\mathrm{BN}_{\gamma_B,\beta}(\vec{x}_i) = \gamma_B \cdot \hat{x}_i + \beta \tag{27}$$

Die Anpassung der Parameter wird nahtlos in den Backpropagation-Algorithmus eingefügt:

$$\Delta\gamma_B = -\gamma \cdot \frac{\partial E}{\partial \gamma_B} \tag{28}$$

$$\Delta\beta = -\gamma \cdot \frac{\partial E}{\partial \beta} \tag{29}$$

Die Batch-Normalisierung ermöglicht das Training der Netze mit deutlich höheren Lernraten und dadurch in weniger Iterationen.[25, 23]

3 Stand der Technik

3.1 CIFAR10

CIFAR10 bezeichnet einen Datensatz, bestehend aus annotierten Bildern, der häufig als Benchmark zur Bewertung von Algorithmen aus dem Bereich des maschinellen Sehens und der Bildverarbeitung eingesetzt wird. Der Datensatz besteht aus einer Untermenge von 60 000 Bildern aus dem *80 million tiny images*-Datensatz. Diese Untermenge ist nochmals in 50 000 Trainingsbilder und 10 000 Testbilder unterteilt. Die Bildgröße beträgt 32×32 Pixel.[27]
In Abbildung 12 auf der nächsten Seite sind je 10 Beispielbilder der 10 annotierten Klassen dargestellt. Die Trainingsbilder bestehen aus 5 000 Bildern pro Klasse. In den Testbildern sind exakt 1 000 Bilder pro Klasse enthalten.[27]
Der CIFAR10-Datensatz ist an dieser Stelle beschrieben, da die dem Stand der Technik entsprechenden, weiter unten aufgeführten Netze, mithilfe dieses Benchmarks bewertet wurden. Auch die im Rahmen dieser Arbeit entwickelten Netzstrukturen sollen mit CIFAR10 bewertet werden.

3.2 GoogLeNet

GoogLeNet ist ein 22 Schichten tiefes CNN, welches 2014 bei der ILSVRC2014 (ImageNet-Challenge) als erstes auf der sogenannten Inception-Architektur basierendes Netz vorgestellt wurde[20].
Um die Möglichkeiten der Parallelisierbarkeit moderner Rechnerarchitekturen (sowohl im CPU, als auch im GPU-Bereich) effizienter ausnutzen zu

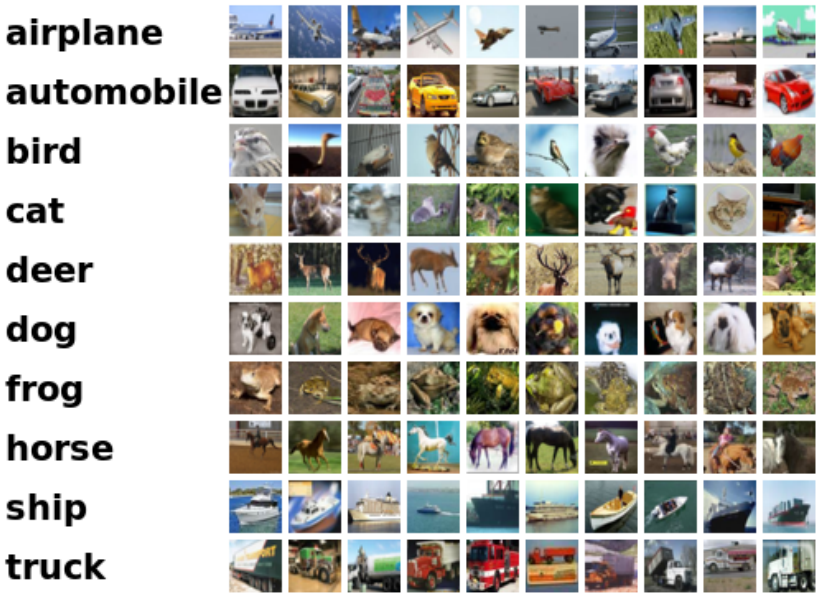


Abbildung 12 – Jeweils 10 Beispielbilder der 10 annotierten Klassen in CIFAR10

können, werden Faltungsmasten verschiedener Größen (1×1, 3×3 und 5×5), sowie Max-Pooling innerhalb einer Schicht genutzt. Das Netz wird also nicht nur in der Tiefe, sondern auch in der Breite erweitert.[20]
Die Max-Pooling-Einheit soll wie bereits in bisherigen CNN-Architekturen die Hebbsche Lernregel umsetzen[20, 21, 17]. Am Ausgang einer solchen, in Abbildung 13 auf der nächsten Seite dargestellten, Inception-Schicht werden die Ausgaben aus den einzelnen Einheiten konkateniert.
Um den durch die recht großen 5×5-Filter entstehenden Rechenaufwand gering zu halten, haben die Autoren entschieden, 1×1-Filtereinheiten einzufügen, welche die Tiefe der bisherigen Feature-Maps vor der Prozessierung durch die 3×3- und 5×5-Filter reduzieren.[20]
Bei der ImageNet-Challenge 2014 setzte das Team die in Abbildung 15 auf Seite 28 dargestellte Struktur, bestehend aus Inception-Blöcken, ein.[20]
Auffällig in Abbildung 15 auf Seite 28 sind rechts, seitlich im Bild, zusätzliche Ausgabeschichten zu erkennen. Genau wie beim Ausgang des gesamten Netzes sind hier zwei Fully-Connected-Layers zu erkennen, welche in einem Softmax-Klassifikator münden.

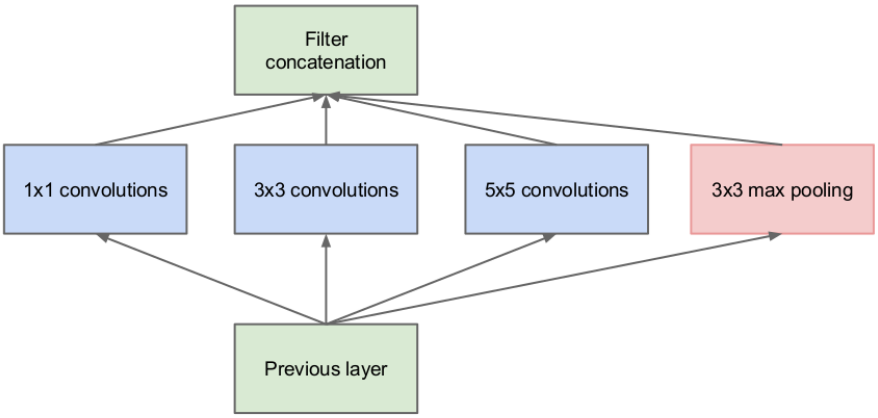


Abbildung 13 – Naive Version eines Inception-Blocks (aus [20])

Team	Year	Place	Error (top-5)	Uses external data
Supervision	2012	1st	16.4%	no
Supervision	2012	1st	15.3%	Imagenet 22k
Clarifai	2013	1st	11.7%	no
Clarifai	2013	1st	11.2%	Imagenet 22k
MSRA	2014	3rd	7.35%	no
VGG	2014	2nd	7.32%	no
GoogLeNet	2014	1st	6.67%	no

Tabelle 1 – Klassifikationsgüte (aus [20])

Die Autoren haben diese zusätzlichen Schichten eingeführt, da sie trotz der ausgiebigen Verwendung von ReLU-Einheiten die Gradienten der Ausgabeschicht im Backpropagation-Schritt nicht durch das bereits sehr tiefe Netz zur Eingabeschicht zurück propagieren können. Die ungewöhnliche Architektur beruht auf der Überlegung, dass sich selbst flachere Netzstrukturen in der Vergangenheit als veritable Klassifikatoren erwiesen haben (siehe beispielsweise LeNet-5 aus [19]). So wird also auch an den zusätzlichen Ausgabeschichten eine Loss-Funktion ausgewertet, deren Gradienten im Backpropagation-Verfahren nur noch über wenige Schichten zurück propagiert werden müssen.[20]
Die Güte der Klassifizierung bei der ImageNet-Challenge 2014 ist in Tabelle 1 dargestellt.

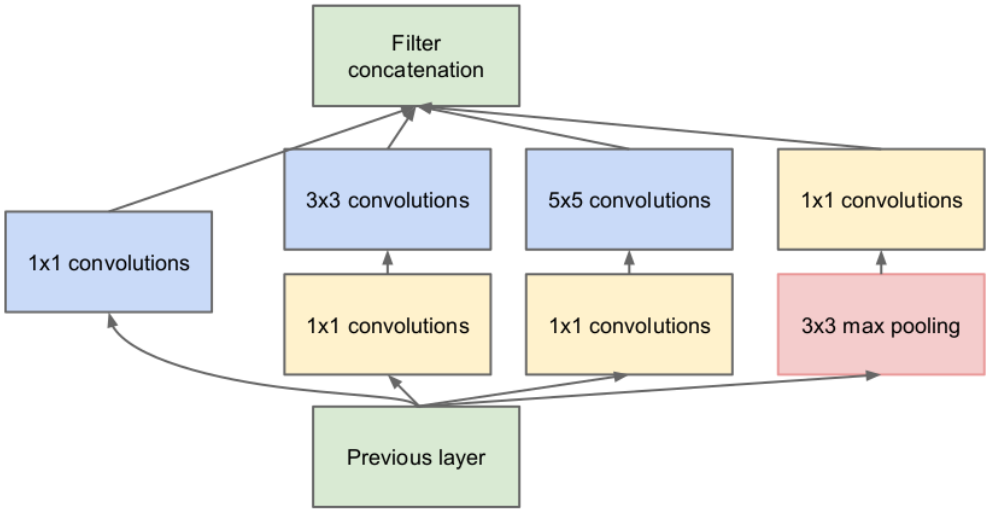


Abbildung 14 – Inception-Block mit Dimensionsreduktion (aus [20])

Die besten Ergebnisse wurden mithilfe einer Vorprozessierung zur Daten-Augmentation auf den Trainingsdaten erzielt (siehe Tabelle 2).

Models	**Crops**	**Cost**	**Top-5 error**	**compared to base**
1	1	1	10.07%	base
1	10	10	9.15%	-0.92%
1	144	144	7.89%	-2.18%
7	1	7	8.09%	-1.98%
7	10	70	7.62%	-2.45%
7	144	1008	6.67%	-3.45%

Tabelle 2 – GoogLeNet-Klassifikationsgüte (aus [20])

Die mit *Crops* überschriebene Tabellenspalte stellt die Ergebnisse mit einer Augmentation dar, in der aus den Trainingsbildern zufällige Bildausschnitte der Größe 28×28 px ausgeschnitten wurden. Die zusätzlichen Kosten bezüglich des Rechenaufwandes durch diese Vorverarbeitung sind in der Spalte *Cost* in der Tabelle dargestellt.

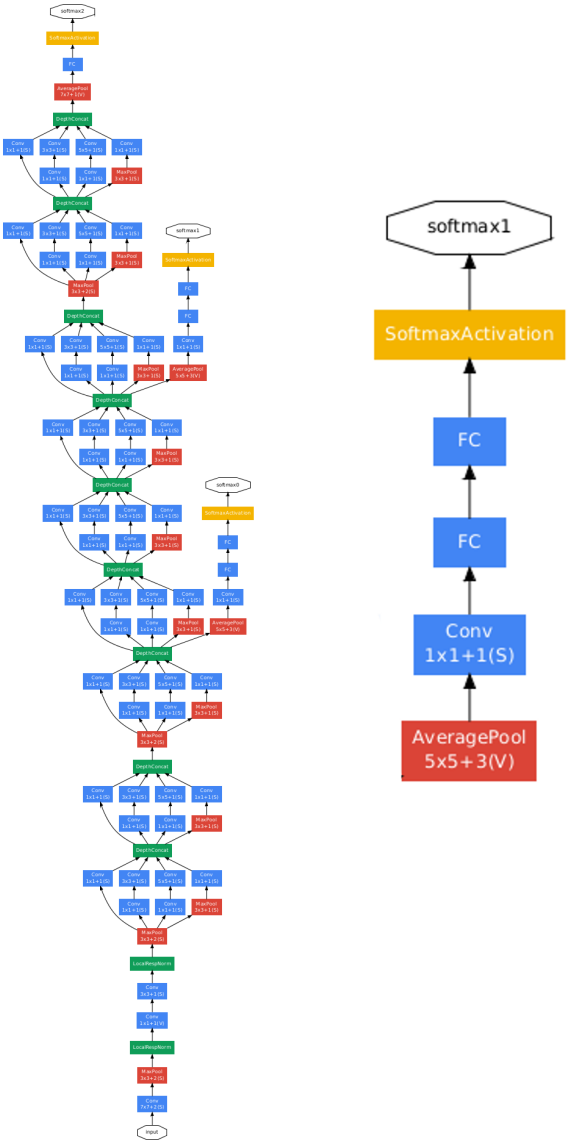


Abbildung 15 – Struktur von GoogLeNet (aus [20])

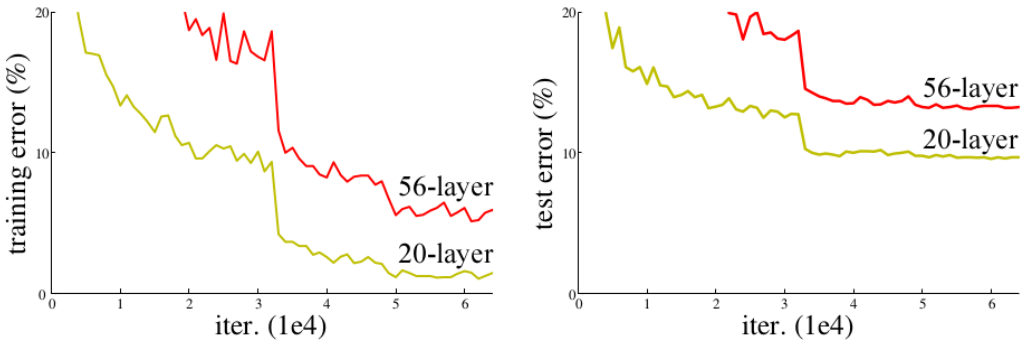


Abbildung 16 – Vergleich der Klassifikationsgüte zweier unterschiedlich tiefer CNNs, bei derselben Anzahl an Iterationen (aus [23])

3.3 Residual Neural Nets

Ein Team von Microsoft Research Asia beschäftigt sich in [23] mit der Beobachtung, dass sehr tiefe Netze mit den verbreiteten Lernverfahren schwerer zu trainieren sind als flachere Netzstrukturen. Die zusätzlichen Parameter der tieferen Netze sollten eigentlich dazu führen, dass komplexere Parameterräume des zu Grunde liegenden Klassifizierungsproblems besser getrennt werden können. Die Autoren beschreiben die Möglichkeit der Konstruktion eines tieferen Modelles aus einem bereits trainierten flacheren Modell[23]:

$$\mathcal{F}(x) = f_n(f_{n-1}(\dots(f_2(f_1(x))))) \tag{30}$$

$$\mathcal{G}(x) = f_{n+1}(f_n(f_{n-1}(\dots(f_2(f_1(x)))))) \tag{31}$$

Wenn die Parameter von f_{n+1} so angepasst werden, dass die Identität erzeugt wird,

$$f_{n+1}(\xi) = \xi \tag{32}$$

dann folgt:

$$\mathcal{F}(x) = \mathcal{G}(x). \tag{33}$$

Hieraus folgt, dass die Klassifikationsgüte eines tieferen Netzes keinesfalls geringer sein sollte, als die eines flacheren[23]. Experimente mit tieferen Netzen zeigen allerdings ein anderes, unerwartetes Ergebnis (siehe Abbildung 16).

Im linken Abschnitt der Abbildung ist der Klassifikationsfehler auf den Daten, welche auch zum Training genutzt werden, zu sehen. Im rechten Abschnitt ist der Fehler auf davon unabhängigen Testbildern dargestellt. Es wird deutlich, dass das tiefere Netz mit 56 Schichten die Bilder schlechter klassifiziert als das flachere. Dadurch, dass sowohl der Trainingsfehler als auch der Testfehler in gleichem Maße von diesem Phänomen betroffen sind, kann ausgeschlossen werden, dass es sich hierbei um einen Effekt handelt, der durch ein übertrainiertes Netz zu erklären ist[23]. Die Autoren folgern aus diesen Ergebnissen, dass es für die Subnetz-Funktionen schwierig ist, die Identität abzubilden. Daher schlagen sie vor, die Abbildung der Identität in die Struktur des Netzes einzubauen[23]. Die erwünschte Funktion des in Ab-

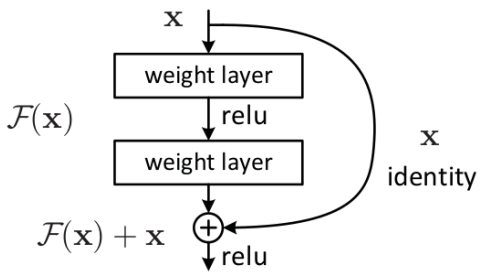


Abbildung 17 – Die Residual-Architektur

bildung 17 dargestellten Blocks sei $\mathcal{H}(x_l)$. Die beiden Schichten innerhalb des Blocks erzeugen die Fuktion $\mathcal{F}(x_l) = \mathcal{H}(x_l) - x_l$. Durch die Addition am Ausgang des Blocks wird also $\mathcal{H}(x_l) = \mathcal{F}(x_l) + x_l$ erzeugt. Das Eingangssignal der ersten Konvolutionsschicht wird hier auf das Ausgangssignal addiert. Nach der Addition folgt als Aktivierungsfunktion eine ReLU-Schicht. Die durch den residuellen Block implementierte Funktion ergibt sich also zu[23]

$$x_{l+1} = f(\mathcal{H}(x_l)) = f(\mathcal{F}(x_l) + x_l) \qquad \text{, mit } f = \text{ReLU} \tag{34}$$

Die Autoren stellen die Hypothese auf, dass es einfacher ist, den residuellen Teil $\mathcal{F}(x_l)$, in Abbildung 17, zu optimieren, als die gesamte Ausgabe des Subnetzwerkes an das gewünschte Mapping $\mathcal{H}(x_l) = \mathcal{F}(x_l) + x_l$ anzupassen. Wenn die zur Optimierung der Klassifikation erwünschte Funktion

also bereits durch ein vorhergehendes Subnetz abgebildet wird, können sich die Gewichte der residuellen Schichten zu 0 verschieben, sodass der gesamte Block die Identität des Eingangssignals x emittiert. Der Pfad in Abbildung 17 auf der vorherigen Seite, der x_l durchschleift, wird auch als Shortcut-Pfad bezeichnet.[23, 26]
Tiefe Netze, zu deren Aufbau die soeben beschriebenen Blöcke genutzt werden, erreichen eine deutlich höhere Klassifikationsgenauigkeit auf verbreiteten Benchmark-Daten, wie z. B. der Imagenet-Challenge (ILSVRC) und CIFAR-10. Da im weiteren Verlauf der Arbeit mit den Ergebnissen auf CIFAR10-Daten verglichen wird, wird an dieser Stelle nur auf die Ergebnisse im CIFAR-10-Setting eingegangen. Die Performance der ResNet-Architektur auf anderen Daten ist in [23] beschrieben. Wie aus Abbildung 18 hervorgeht, sinkt der

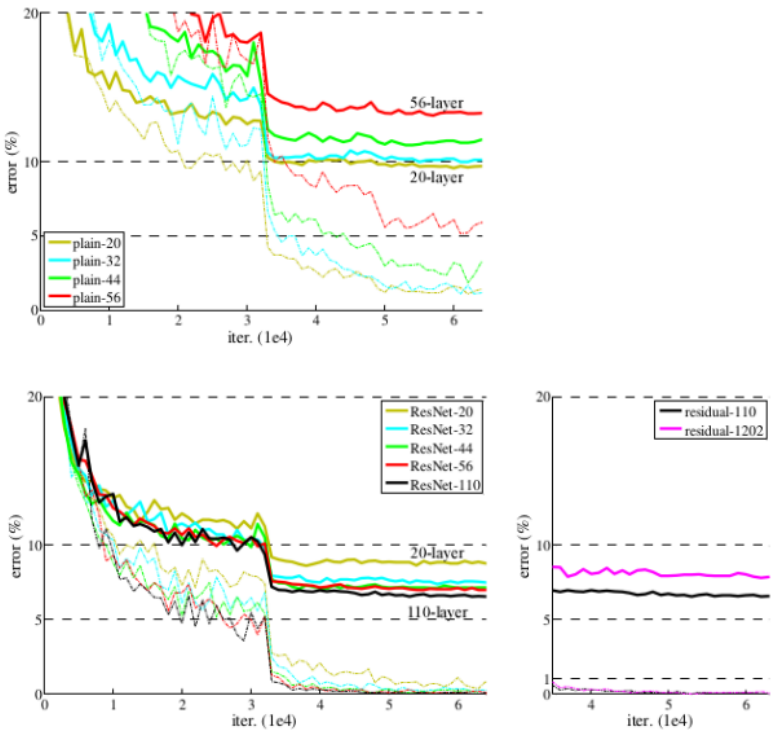


Abbildung 18 – Vergleich der Klassifikationsgüte unterschiedlich tiefer Netze. Oben: ebene Netze (ohne Shortcut). Unten links: Residual-Netze. Unten rechts: ResNets mit 110 bzw. 1202 Schichten. (aus [23])

Klassifikationsfehler bei steigender Tiefe der Netze, wenn die vorgeschlagene

residuelle Architektur eingesetzt wird. Beim Vergleich der ebenen Netze (ohne Shortcut, oben in Abbildung 18 auf der vorherigen Seite) mit den residuellen Netzen (unten links) ist ersichtlich, dass bei den dargestellten Netztiefen die residuelle Implementierung grundsätzlich einen positiven Effekt auf die Klassifikationsgenauigkeit hat.
Unten rechts in Abbildung 18 auf der vorherigen Seite sind die Klassifikationsfehler zweier residueller Netze aus 110 bzw. 1202 Schichten dargestellt. Bei der starken Erhöhung der Tiefe tritt das oben beschriebene Problem des erschwerten Trainings tieferer Netze also wieder zu Tage. Im folgenden Abschnitt wird eine Änderung der Struktur der residuellen Blöcke vorgestellt, die sich dieses Problems annimmt.

3.4 Residual Neural Networks mit Präaktivierung

Inspiriert von einem Blog-Eintrag[3] von Sam Gross und Michael Wilber[28] werden von den Autoren von [23] in einer weiteren Veröffentlichung [26] verschiedene Modifikationen an der Struktur der ResNet-Blöcke untersucht. Vor allem konzentrieren sie sich dabei auf Modifikationen des Shortcut-Pfades. Die Struktur, die den vorteilhaftesten Effekt auf die Klassifikationsgenauigkeit der resultierenden Netze hat, ist die in Abbildung 20 auf Seite 34 dargestellte Prä-Aktivierungsstruktur. Die Autoren machen den vollkommen ungestörten Shortcut-Pfad, auch über mehrere residuelle Blöcke hinweg für die verbesserte Klassifikationsleistung verantwortlich[26]. Die Funktion eines residuellen Blocks aus [23] ist in Gleichung (34) dargestellt. Analog kann auch eine Funktion der Blockstruktur, die in [26] vorgeschlagen wird, hergeleitet werden. Da statt der Anwendung einer Aktivierungsfunktion die Identität I des Ergebnisses der Addition aus Gleichung (34) weiter geleitet wird, ergibt

[3] Blog-Eintrag *Training and investigating Residual Nets*, abrufbar auf: http://torch.ch/blog/2016/02/04/resnets.html

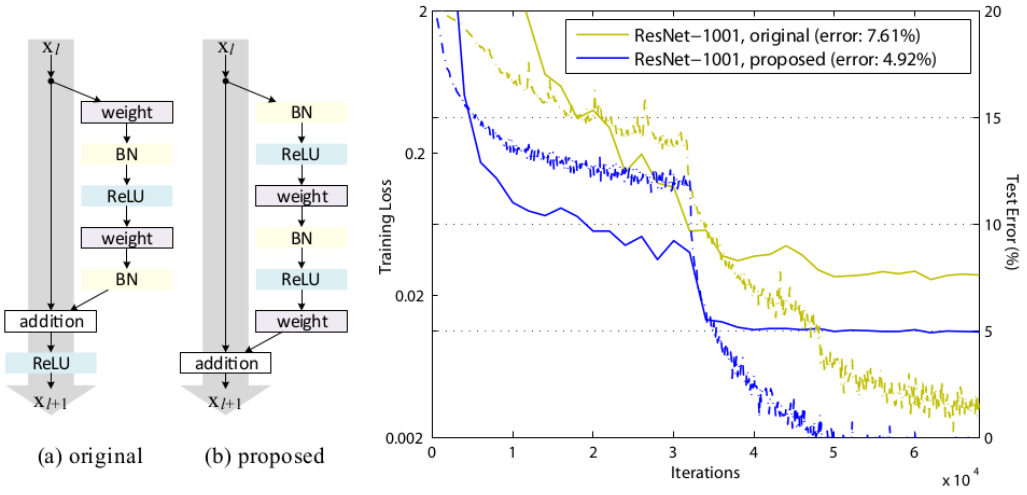


Abbildung 19 – Vergleich der Klassifikationsgüte der ResNet-Struktur aus [23] mit der Präaktivierungsstruktur aus [26]. (Abbildung aus [26])

sich aus Gleichung (34):

$$\begin{aligned} x_{l+1} &= f(\mathcal{F}(x_l) + x_l) \quad \text{, mit } f = \mathrm{I} \\ &= I(\mathcal{F}(x_l) + x_l) \\ &= \mathcal{F}(x_l) + x_l \end{aligned} \tag{35}$$

Für jeden beliebigen, weiteren Block der Tiefe $L > l$ innerhalb des Netzes ergibt sich[26]:

$$x_L = x_l + \sum_{i=l}^{L-1} \mathcal{F}(x_i) \tag{36}$$

Durch diese Modifikation ist es also möglich, auch über mehrere Blöcke hinweg, ohne Störung durch eine Aktivierungsfunktion, die Abbildung einer früheren Schicht des Netzwerkes in Richtung des Ausganges durchzuschleifen. Der Übergang von der ursprünglichen residuellen Struktur zur Präaktivierungs-Architektur ist in Abbildung 20 auf der nächsten Seite abzulesen. Der Übergang von Struktur (a) zu (b) stellt eine Verschiebung der Aktivierung aus dem Shortcut-Pfad in den folgenden residuellen Block dar. Der Übergang zu Struktur (c), rechts im Bild, zeigt, dass (b) equi-

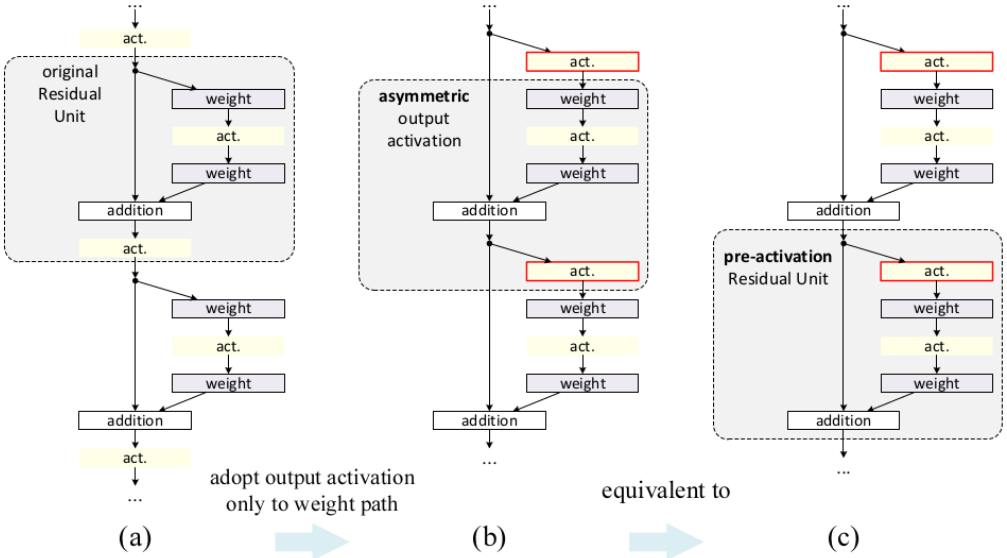


Abbildung 20 – Übergang der in [23] vorgeschlagenen Struktur zur Präaktivierungs-Struktur. (aus [26])

valent zu einer Blockstruktur ist, bei der die Aktivierungsfunktionen vor die Konvolutionsschichten verschoben sind. Aus dieser neuen Positionierung der Aktivierungsfunktion und der BatchNorm-Layers folgt die Bezeichnung „Präaktivierung“[26]. Ausgewählte Ergebnisse aus [26] sind in Tabelle 3 aufgeführt. Bei den untersuchten Netztiefen (110, 164 und 1001 Schichten) sorgt die neue Präaktivierungs-Architektur für eine verbesserte Klassifikationsgenauigkeit. Je tiefer das zugrunde liegende Netz ist, desto deutlicher wird dieser Effekt.

network	**2015 ResNet unit**	**2016 pre-activation unit**
ResNet-110	6.61	6.37
ResNet-164	5.93	5.46
ResNet-1001	7.61	4.92

Tabelle 3 – Ausgewählte Ergebnisse auf CIFAR10 aus [26]

3.5 Wide Residual Neural Networks

Ein weiterer im Jahr 2016 vorgestellter Vorschlag zur Optimierung der Netzarchitektur sind die *Wide Residual Neural Networks* von Sergey Zagoruyko und Nikos Komodakis. Die Struktur leitet sich von der in [23] vorgeschlagenen Architektur ab. Die Autoren schlagen in einer ähnlichen Argumentation wie in [20] vor, die Gegebenheiten der zur Verfügung stehenden Hardware sinnvoller auszunutzen. Die in [23] und [26] vorgestellten Netze beschreiben sie als schmal in dem Sinne, dass pro Netzwerkschicht nur vergleichbar wenige und gleichförmige 3×3-Filter eingesetzt werden. Da die Berechnungen späterer Schichten aufgrund der Verkettung der Funktionen von den Ergebnissen früherer Schichten abhängen, muss ein Anteil der Operationen im neuronalen Netz sequentiell abgearbeitet werden. Die heute zum Trainieren von neuronalen Netzen eingesetzten Grafikkarten bieten durch ihre Architektur die Möglichkeit, sehr effizient Rechenoperationen parallel auszuführen. Um diese Voraussetzung effizient ausnutzen zu können, sollte ein neuronales Netz also möglichst viele Filter pro Konvolutionsschicht trainieren (Breite) und die Tiefe (Anzahl der Schichten) reduzieren. Hierdurch sollte sich das Verhältnis zwischen sequentiellem und parallelisierbarem Anteil der Operationen zu Gunsten der Parallelverarbeitung verschieben. Die Gradienten der Gewichte eines Filters sind nämlich nicht von den Ergebnissen des Vorwärts-Schrittes der anderen Filter innerhalb derselben Schicht abhängig. Die Breite der Schichten wird in [29] durch den Parameter k angegeben und die Anzahl der residuellen Blöcke durch den Parameter d. Die Autoren beabsichtigen, das Verhältnis $\frac{k}{d}$ zu maximieren, unter Beibehaltung oder Verbesserung der Klassifikationsgenauigkeit. In Tabelle 4 ist der Testfehler eines

net	**test error**
ResNet-164	5.46 %
WRN-28-10	4.00 %

Tabelle 4 – Test-Performance von ResNet-164 im Vergleich zu ResNet-164. (aus [29])

ResNet-164 im Vergleich zu dem breiten WRN-28-10-Netz dargestellt. Durch die Verbreiterung der residuellen Blöcke ist also sogar eine weitere Verbesse-

rung der Klassifikationsgenauigkeit möglich. Allerdings müssen für das WRN-28-10-Netz $36,5 \cdot 10^6$ Parameter optimiert werden, wogegen ResNet-164 nur $1,7 \cdot 10^6$ Parameter enthält. Da der Vorteil der Präaktivierung aus [26] nur bei sehr tiefen Netzen zum Tragen kommt und die Tiefe gerade reduziert werden soll, entscheiden sich die Autoren dafür, auf die Präaktivierung zu verzichten und leiten ihre Blöcke von der ursprünglichen residuellen Implementierung von 2015, dargestellt in Abbildung 17 auf Seite 30, ab.

3.6 Residual Inception Nets

Auch die Inception-Architektur kann so angepasst werden, dass durch das Einfügen von Shortcuts die Vorteile, die eine residuelle Implementierung bietet, genutzt werden können. Exemplarisch ist in Abbildung 21 auf der nächsten Seite ein residueller Inception-Block aus [30] dargestellt. Der deutliche Unterschied zu dem in Abbildung 14 auf Seite 27 dargestellten Block ergibt sich dadurch, dass die in [30] vorgeschlagenen residuellen Inception-Architekturen auf modifizierten Strukturen aus [31] basieren, die im Rahmen dieser Arbeit jedoch weder vorgestellt noch verwendet werden. Diese Strukturen sind jedoch wiederum von den ursprünglichen Inception-Schichten in Abbildung 13 auf Seite 26 aus [20] abgeleitet. Die residuelle Inception-Architektur wird nicht auf CIFAR10 sondern nur auf den ImageNet-Daten getestet. Die Autoren erreichen durch die Veränderung der Blockstruktur zwar keine Erhöhung der letztendlich erreichten Klassifikationsgenauigkeit. Allerdings lassen sich die neu vorgeschlagenen Netze in weniger Iterationen bis zu einer vergleichbaren Klassifikationsgüte trainieren.
Da bei den eingesetzten Inception-Block-Strukturen jeweils mehr Filter pro Konvolution eingesetzt werden als bei den vergleichbaren residuellen Netzen aus [23] und [26], können sie auch als eine Variante der breiteren Wide Residual Neural Networks angesehen werden. Auch die residuellen Inception-Blöcke verzichten auf die Präaktivierung aus [26].

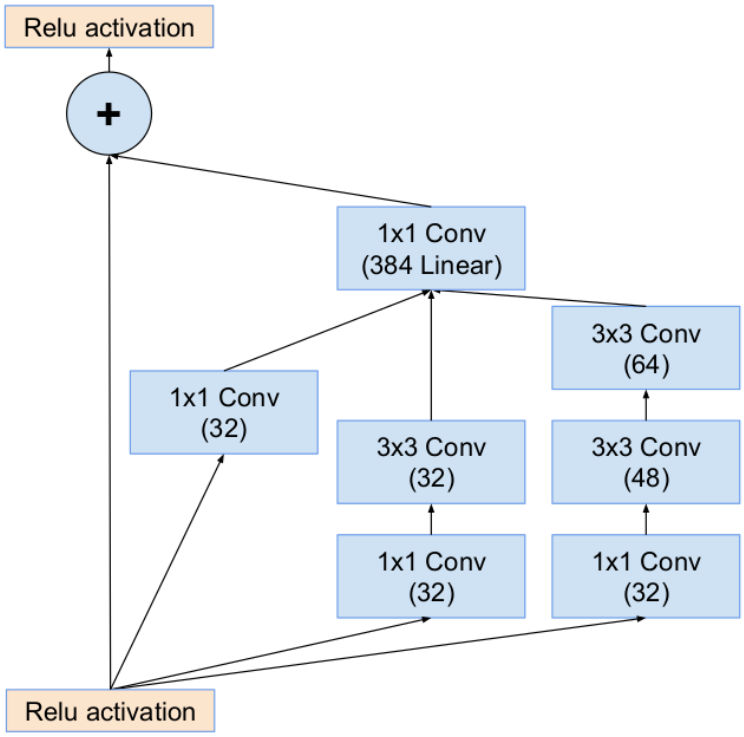


Abbildung 21 – Ein Residual Inception Block aus [30]

3.7 Neuroevolution of Augmenting Topologies

Unter dem Titel NeuroEvolution of Augmenting Topologies (NEAT) wird in [2] ein Verfahren zur Kombination evolutionärer Algorithmen mit künstlichen neuronalen Netzen vorgeschlagen. Bei NEAT werden sowohl die Netzwerktopologie als auch die Gewichte des neuronalen Netzwerkes durch einen genetischen Algorithmus angepasst. Die Startpopulation besteht aus Netzwerken ohne versteckte Schichten. Das bedeutet, dass die Eingangsschicht direkt mit der Ausgangsschicht verbunden wird. Durch Mutation können probabilistisch in jedem Iterationsschritt des Algorithmus neue Knoten und Kanten in den Netzwerkgraphen eingefügt werden (siehe Abbildung 22 auf der nächsten Seite). Nur Individuen, bei denen diese Mutation einen positiven Einfluss auf die Bewertung durch die Fitnessfunktion hat, werden in die nächste Generation übernommen.[2]
Durch die Erzeugung der Startpopulation aus einfachen Strukturen und die

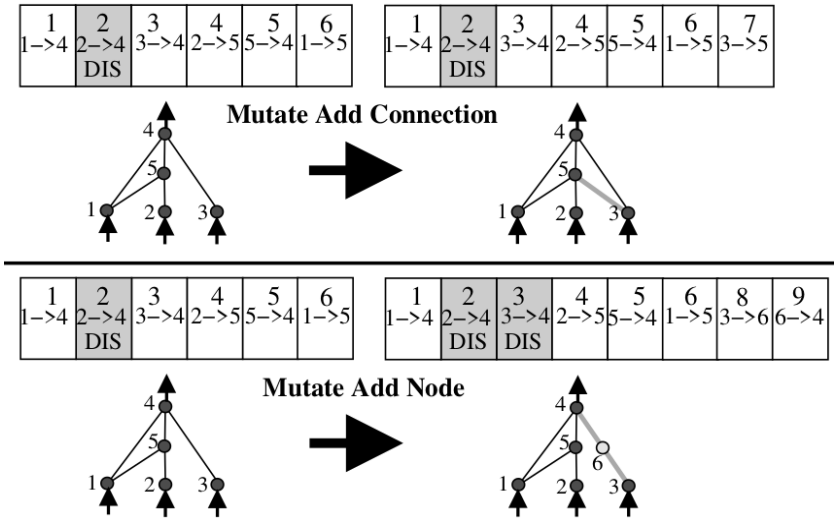


Abbildung 22 – Darstellung der Mutation zum Einfügen zusätzlicher Kanten (oben) und Knoten (unten) in NEAT (aus [2])

iterative Verbesserung durch den genetischen Algorithmus sollen kleine, effiziente Netzstrukturen bevorzugt werden.[2]
Um die Rekombination verwandter Netzstrukturen zu unterstützen, führen die Autoren Speziation zur Unterscheidung der Individuen in weitere Gruppen innerhalb der Populationen ein. Die einzelnen Spezies beanspruchen im biologischen Vorbild unterschiedliche Nischen und konkurrieren daher nicht miteinander. Individuen unterschiedlicher Spezies werden auch nicht gemeinsam zur Rekombination ausgewählt. Um die Spezies voneinander unterscheiden zu können, wird die Historie des Genoms der Individuen festgehalten. Nur Individuen mit Genomen, deren Historie eine ausreichend große Gemeinsamkeit aufweist, werden rekombiniert.[2]
Um größere Netzstrukturen unter geometrischen Randbedingungen, die vom biologischen Vorbild des Säugetiergehirns abgeleitet sind, zu optimieren, wird in [32] die Erweiterung HyperNEAT vorgestellt. HyperNEAT nutzt eine indirekte Kodierung der Netztopologie, die modifiziert wird. Die indirekte Kodierung stellt einen Hyperwürfel in einem höherdimensionalen Raum dar. Von diesem Hyperwürfel ist die Bezeichnung Hypercube based NEAT abgeleitet. Die indirekte Kodierung enthält die oben beschriebenen Randbedingungen in Form von Symmetrie, unvollständiger Symmetrie und Wiederholung von

Strukturen.[32]
Wiederholte, symmetrische Strukturen dominieren auch die oben beschriebenen CNNs und Residual Neural Networks[23, 26, 19].
In [33] wird die Anwendbarkeit von HyperNEAT auf Bildverarbeitungsprobleme untersucht. Hierzu wird eine Modifikation eingeführt, um CNNs zu erzeugen. Die erzeugten Netzwerke erreichen nicht die Performance der CNNs, die durch den Backpropagation-Algorithmus trainiert werden. Sie eignen sich jedoch als Feature-Extraktoren, welche in einer Toolchain, wie sie in Abbildung 9 auf Seite 19 dargestellt ist, eingesetzt werden können.

4 Implementierung des kanonischen Genetischen Algorithmus zur Optimierung der neuronalen Netze

Zur Optimierung der Architektur der neuronalen Netze wird ein kanonischer Genetischen Algorithmus (kGA) ausgewählt, da diese Klasse der evolutionären Algorithmen in der Literatur sehr gut untersucht ist und die Anwendung einer binären Kodierung der Chromosomen unterstützt. Um den in Section 2.2.4 auf Seite 6 vorgestellten Algorithmus anzupassen, wurden die dem folgenden Pseudocode zu entnehmenden Änderungen vorgenommen. Um alle ausgewählten Netzarchitekturen miteinander vergleichen und

```
 1: procedure KGA
 2:     t := 0
 3:     Initialisiere P(0)
 4:     Bewerte P(0)
 5:     Setze P*(t) := ∅
 6:     while (t < T_MAX) do
 7:         t := t + 1
 8:         for (i = 0; i < popsize ; i++) do
 9:             Auswahl des i-ten Chromosoms für P*(t) aus P(t − 1)
10:             Anwendung des Crossover-Operators auf P*(t)
11:             Anwendung des Mutations-Operators auf P*(t)
12:             P(t) := P*(t)
13:             Bewerten von P(t)
14:         end for
15:     end while
16: end procedure
```

noch zwei zufällig generierte Individuen in die initiale Population einfügen zu können, wurde die Populationsgröße auf $popsize = 10$ festgelegt. Aufgrund der Limitierungen der Leistungsfähigkeit der eingesetzten Hardware konnten im Zeitrahmen der Masterarbeit nur $T_{MAX} = 5$ Generationen erzeugt und bewertet werden. Die Wahl der Populationsgröße $popsize = 10$ und der maximalen Anzahl der Iterationsschritte erscheint gegenüber den üblicherweise jeweils genutzten Größenordnungen im Bereich 10^3 sehr gering,

jedoch wird die Auswertung zeigen, dass bereits in diesem eingeschränkten Optimierungs-Setting eine Verbesserung der Klassifikationsgüte durch die Architekturveränderungen möglich ist. Auf eine Speziation, wie sie in [2] vorgeschlagen wird, wird aufgrund der geringen Populationsgröße verzichtet. Bei größeren Populationen kann als Maß der Verwandtschaft der Individuen die Hamming-Distanz der Chromosomen herangezogen werden.
Zum Erhalt der genetischen Varianz und um den Suchraum effizient durchsuchen zu können, werden die beiden genetischen Operatoren, Mutation und Crossover, die bereits in den Grundlagen vorgestellt wurden, verwendet. Zum Crossover werden jeweils zwei Individuen aus der aktuellen Generation in einer zufälligen Kombination zu Paaren gruppiert. Für jedes Paar wird mit einer Crossover-Wahrscheinlichkeit von $\frac{1}{Anzahl\ der\ Paare}$ bestimmt, ob das Paar zum Crossover ausgewählt wird. Bei einem auf diese Weise ausgewählten Paar wird gleichverteilt aus allen möglichen Positionen im Chromosom ein Crossover-Punkt ausgewählt. Zur Rekombination wird der Single-Point-Crossover ausgewählt, da es nahe liegt, dass die Netzstruktur funktional in eine ähnliche Hierarchie, wie die einzelnen Blöcke in der Toolchain, in 9, zerlegt werden kann. Durch das Single-Point-Crossover können auf diese Weise erfolgreiche Feature-Extraktoren mit erfolgreichen Klassifikatoren kombiniert werden.

4.1 Implementierung der genetischen Operatoren

4.1.1 Single Point Crossover

Beim Crossover wird das Chromosom in einzelne Allele, welche jeweils aus drei binären Genen bestehen, unterteilt. Mögliche Crossover-Punkte sind die Allele, die beim Crossover allerdings nicht zerstört werden. Die möglichen Crossover-Punkte werden also an 3-bit-Grenzen ausgerichtet. Wie in Abbildung 1 auf Seite 8 in den theoretischen Grundlagen erkennbar ist, werden aus zwei Elternchromosomen zwei Kindchromosomen erzeugt. Nur die neu erzeugten Kind-Chromosomen werden in die nächste Generation übernommen. Bei den Paaren, die nicht zum Crossover ausgewählt werden, werden die In-

dividuen zunächst einfach unverändert in die folgende Generation kopiert. Die konkrete Implementierung der Mutation kann dem Anhang dieser Arbeit entnommen werden[4].

4.1.2 Punktmutation

Wie dem oben angeführten Pseudocode entnommen werden kann, werden die Chromosomen nach dem Crossover mutiert. Hierzu wird in jedem Chromosom für jedes der 63 Gene, mit einer Mutationswahrscheinlichkeit von $\frac{1}{63}$ bestimmt, ob ein Gen mutiert wird. Da die Gene binär kodiert vorliegen, besteht die Mutation darin, ein Bit umzukehren. Die nun in $P^*(t)$ vorliegenden Chromosomen werden in $P(t)$ eingefügt und die Schleife im Algorithmus wird so lange wieder ausgeführt, bis als Abbruchbedingung die maximale Anzahl der Iterationen T_{MAX} erreicht ist. Die konkrete Implementierung der Mutation ist im Anhang dargestellt[4].

4.2 Auswahl der genutzten Architekturblöcke

Als Ausgangspunkt der Optimierung der Netzwerkarchitekturen werden unterschiedliche dem aktuellen Stand der Technik entsprechende Grundbausteine ausgewählt. Als Grundgerüst der Individuen wird die Struktur der residuellen neuronalen Netze gewählt. Das bedeutet, dass alle eingesetzten Blöcke eine Shortcut-Connection, wie in den theoretischen Grundlagen bereits eingeführt, besitzen. Durch die Auswahl und die festgelegte Anzahl der Blöcke sowie durch die Kodierung ist sichergestellt, dass keine fehlerhaften Netzstrukturen entstehen können. Daher sind keine Reperaturmechanismen, wie sie in [3], S. 87, ff. vorgeschlagen werden, notwendig.
Aufgrund der residuellen Architektur ist es nicht notwendig, wie in [2] empfohlen, mit einer minimalen Netzwerkstruktur zu starten. Zur Lösung des Klassifizierungsproblems nicht benötigte Blöcke können mithilfe der Shortcut-Verbindungen übersprungen werden.
Durch den genetischen Algorithmus wird nur die Architektur der Netze be-

[4]Außerdem ist die Implementierung auf Github verfügbar: `https://github.com/nils489/EvANet`

einflusst. Die Netztopologie wird verändert, da sich die Topologie innerhalb der Blöcke unterscheidet. Die Schnittstellen der Blöcke sind jedoch so gewählt, dass sie frei miteinander ausgetauscht werden können. Der kGA optimiert allerdings im Gegensatz zu NEAT[2] und HyperNEAT[32] nur die Topologie und nicht die Gewichte. Daher wird jedes erzeugte Netz mit dem Backpropagation-Algorithmus trainiert.
Alle Individuen sind vom Aufbau des ResNet-44 aus [23] abgeleitet. ResNet-44 ist eines der Netze, welches zur Klassifizierung des CIFAR10-Datensatzes genutzt wird[23]. Es besteht aus 21 residuellen Blöcken (siehe Abbildung 23), denen eine initiale Faltungsschicht vorausgeht und denen am Ende zur Klassifizierung eine Fully Connected-Schicht folgt. Die einzelnen Blöcke sollen an dieser Stelle im Detail vorgestellt werden.

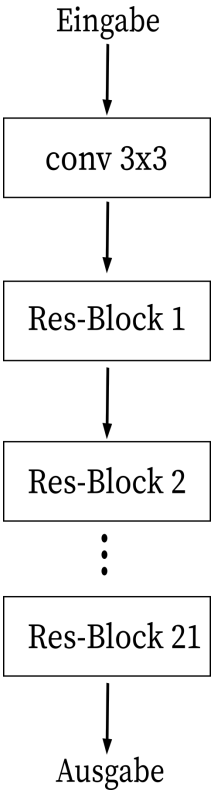


Abbildung 23 – Struktur der Individuen im kGA

Um den Vergleich mit den in [23] vorgestellten Netzen zu ermöglichen und um die Vorteile der vorgeschlagenen Architektur ausnutzen zu können, wurde

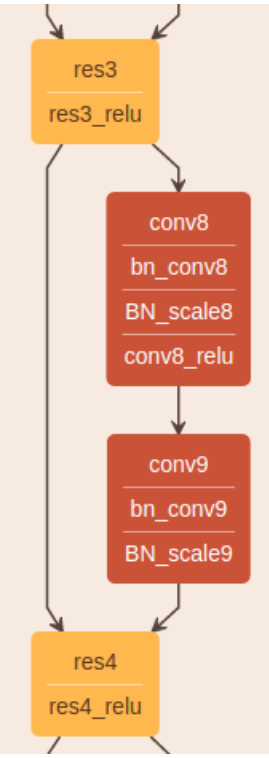


Abbildung 24 – Ein Residual-Block gemäß [23] (000)

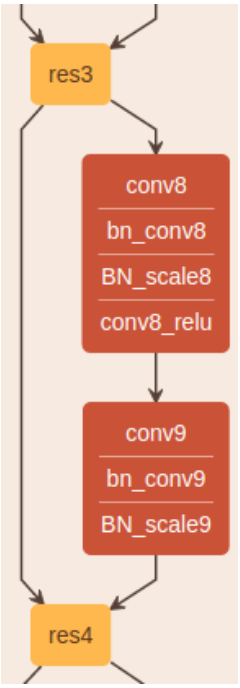


Abbildung 25 – Ein Residual-Block ohne Aktivierungsfunktion nach der zweiten Konvolution [28] (001)

die Entscheidung getroffen, Blöcke gemäß [23] zu implementieren. Die Struktur der implementierten Blöcke kann Abbildung 24 entnommen werden. Aus Abbildung 24 wird außerdem deutlich, dass auf die Batch-Normalisierungsschichten jeweils eine *Scale*-Schicht folgt. Die Nutzung der *Scale*-Schichten wird aufgrund der Implementierung der Batch-Normalisierung des im Rahmen der Arbeit eingesetzten Caffe-Frameworks benötigt. Die anpassbaren Parameter γ_B und β sind die Parameter der *Scale*-Schicht.
Die zweite der ausgewählten Block-Strukturen ist in Abbildung 25 zu sehen. Die dargestellte *No-ReLU*-Konfiguration wurde in dem bereits erwähnten Blog-Eintrag von Sam Gross und Michael Wilber vorgeschlagen[28]. Laut dem Blog-Eintrag konnte eine geringfügig verbesserte Klassifikationsgüte gegenüber der bereits vorgestellten Referenzimplementierung erreicht werden.

Die verbesserte Klassifikationsgüte greifen die Autoren um Kaiming He, von Microsoft Research Asia, auch in [26] auf. Da bei der *No-ReLU*-Konfiguration keine Operation im Shortcut-Pfad besteht, ist es wie bei der in [26] vorgeschlagenen Präaktivierungs-Struktur möglich, auch deutlich tiefere Netze noch effizient zu trainieren.
Die Implementierung der bereits angesprochenen Präaktivierungs-Blöcke aus [26] ist in Abbildung 26 auf der nächsten Seite zu sehen. Im Gegensatz zu Abbildung 24 auf der vorherigen Seite sind die ReLU-, BatchNorm- und Scale-Schichten vor die Konvolutionen verschoben. Dies entspricht der in Abbildung 20 auf Seite 34 vorgestellten Veränderung.
Da die jeweiligen Autoren in [28] und [26] einen positiven Einfluss durch das Auslassen der Aktivierungsfunktion (ReLU) bestätigen, wurde entschieden, hier noch einen Block völlig ohne Scale-Schicht, Batch-Normalisierung und Aktivierungsfunktion nach der zweiten Konvolution zu implementieren. Die hierdurch folgende *No-Act*-Struktur ist in Abbildung 27 auf der nächsten Seite dargestellt. Effektiv wird hierdurch ein äquivalentes Netz erzeugt, in dem nur eine Faltungsschicht pro Residual-Block vorhanden ist. Da zwischen je zwei Faltungsschichten keine Nichtlinearität besteht, können diese gedanklich zusammengefasst werden. Explizit konstruierte residuelle Blöcke mit nur einer Konvolutionsschicht werden in [26] untersucht.
Die Implementierung der Inception-Schichten ist von der in [20] vorgestellten Struktur der Inception-Blöcke abgeleitet. Jeder Block besteht also, wie in Abbildung 28 auf Seite 47 zu erkennen ist, aus 1×1-, 3×3- und 5×5-Filtern. Um die Blöcke in das Schema des ResNet-44 einpassen und auf die in [20] genutzten zusätzlichen Ausgänge des Netzes verzichten zu können, wurden die Inception-Blöcke wie in [30] vorgeschlagen um Shortcut-Verbindungen erweitert. Um einen zu großen Zuwachs der Komplexität des Netzes zu verhindern, wurden, wie in [20] beschrieben, die 3×3- und 5×5-Filter Reduktionsschichten eingeführt.
Wie bereits oben erwähnt, wurden die Residual-Schichten gemäß [23], [26] und [28] implementiert. Die Gewichte wurden gemäß [34] initialisiert. Die Implementierung der Inception-Schichten folgt grob dem in [20] dargestellten Schema. Um den Speicherplatzbedarf in Grenzen zu halten, wurde die

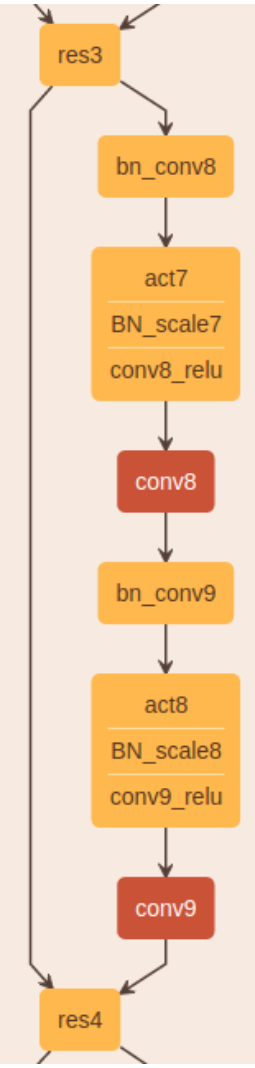


Abbildung 26 – Ein Residual-Block mit Prä-Aktivierung gemäß [26] (101)

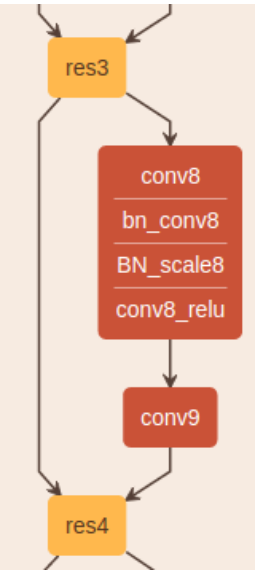


Abbildung 27 – Ein Residual-Block ohne Aktivierungsfunktion und Batch-Normalisierung nach der zweiten Konvolution (100)

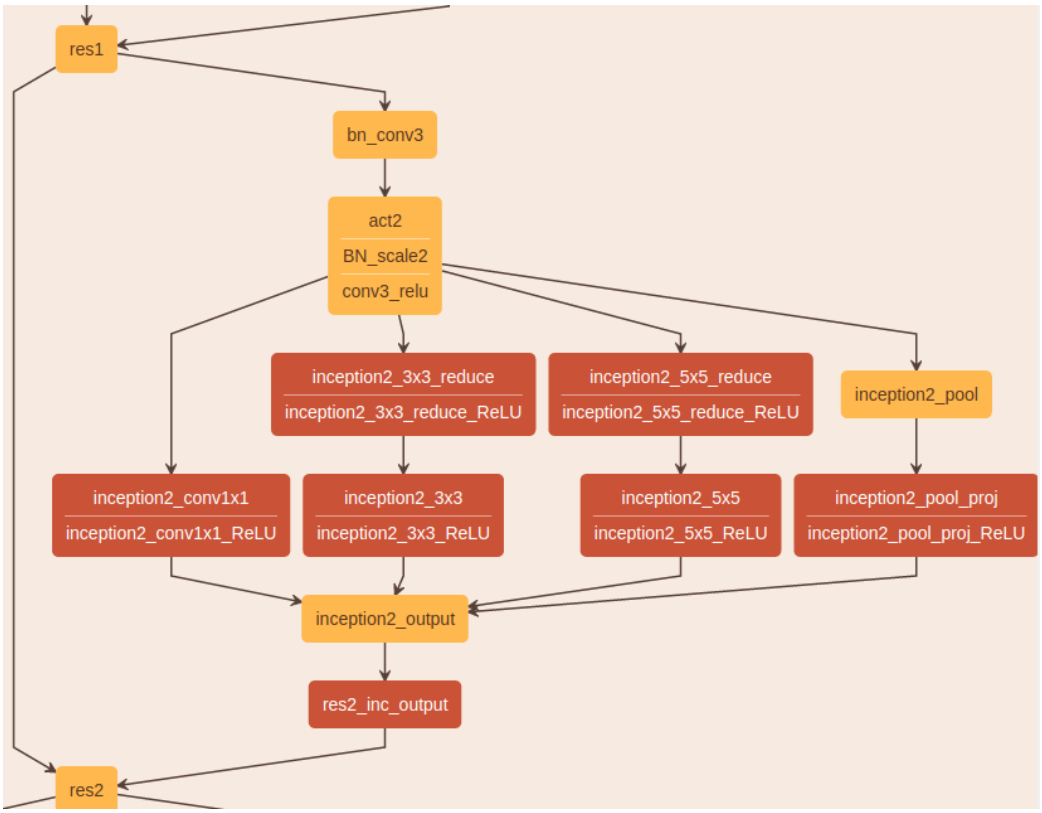


Abbildung 28 – Die an CIFAR10 angepasste Inception-Architektur (011) und (111)

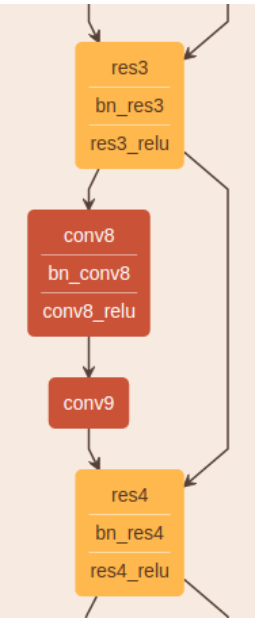


Abbildung 29 – Ein Residual-Block mit Aktivierungsfunktion und Batch-Normalisierung nach der Addition, nach [28] (010)

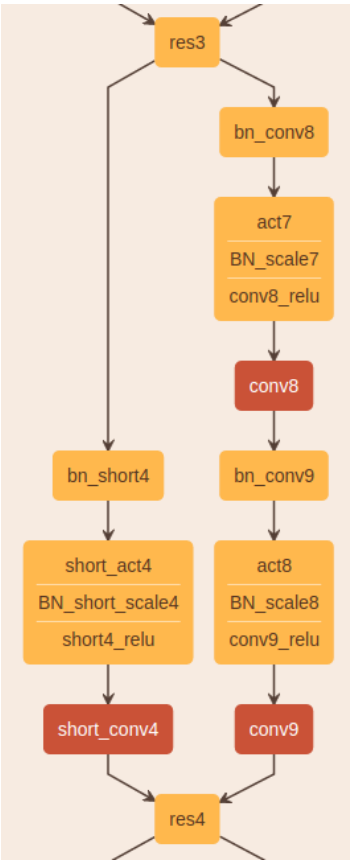


Abbildung 30 – Ein Residual-Block mit einer Konvolution im Residual-Pfad, nach [26] (110)

Tiefe der Filter jeweils verändert. Diese Änderungen erscheinen auf dem einfacher gestalteten Trainingsdatensatz aus CIFAR10 sinnvoll.

Um auf die in [20] genutzten, zusätzlichen Ausgabeschichten während des Trainingsvorganges verzichten zu können, wurden die Inception-Schichten mithilfe eines Shortcuts, ähnlich wie in [30], als residuelle Blöcke implementiert (siehe Abbildung 28 auf der vorherigen Seite).

Betrachtet man die Tiefe der Ausgabe der Convolution-Schichten, so fällt auf, dass die Tiefe der 3×3-Convolution größer ist, als die der 5×5. In [20] wird angedeutet, dass diese Entscheidung getroffen wurde, um die Komple-

xität des Netzes möglichst gering zu halten. Da allerdings die 5×5-Filter kombinatorisch einen größeren Werteraum aufspannen als die 3×3-Kernel, wurden im Rahmen dieser Arbeit zusätzliche Inception-Schichten implementiert, deren Tiefe bei den 5×5-Einheiten größer ist als die der 3×3-Einheiten. Die so entstandene Block-Struktur wird im weiteren Verlauf der Arbeit, in Anlehnung an [29], als *Wide Inception* bezeichnet, da sie pro Block mehr anpassbare Parameter enthält, als die entsprechende Inception-Architektur gemäß [20].
Zur Implementierung und zum Training wurde das Caffe-Deep-Learning-Framework genutzt. Die Netzdefinitionen wurden alle selbst gemäß der Angaben in [23], [26] und [20] nachimplementiert, da bei den auf Github verfügbaren Netzdefinitionen die Initialisierung der Gewichte und die eingesetzten Verfahren zur Data-Augmentation aus [34] fehlen. Außerdem sind auf Github nur Netzstruktur-Definitionen für den ImageNet-Datensatz und nicht für CIFAR10 verfügbar.

4.3 Kodierung

Der verwendete kanonische Genetische Algorithmus benötigt als Eingabedaten binäre Zeichenketten, die die Chromosomen der zu optimierenden Individuen wiederspiegeln. Da acht verschiedene Blockstrukturen als grundsätzliche Bausteine der Individuen ausgewählt wurden, kann jeder Block eindeutig durch 3 bit repräsentiert werden. So ergeben sich bei der in Abbildung 23 angegebenen Struktur $21 \cdot 3$ bit, was zu einer Gesamtlänge des Chromosoms von 63 bit führt.
Die Kodierung der einzelnen Schichten erfolgt nach dem in Tabelle 5 dargestellten Schema. Zur besseren Übersicht ist in den Abbildungen des vorherigen Abschnittes die Kodierung in jeweils 3 bit in der Bildunterschrift angegeben.
Wie bereits in Section 4.1.1 auf Seite 41 erwähnt, arbeitet der Crossover-Operator auf 3 bit-Grenzen, um zusammengehörige Allele nicht zu zerstören. Der Mutationsoperator arbeitet auf allen 63 bit, um eine Veränderung eines Allels zu einem beliebigen anderen zu ermöglichen. Bei der Zuordnung der

Blocktyp	Allel
ResNet-2015	000
fb.no_relu	001
fb.bn_after_add	010
Inception	011
no_act	100
ResNet-2016	101
ResNet-2016-1 × 1-conv	110
Wide-Inception	111

Tabelle 5 – Die Kodierung der einzelnen Blöcke in binären Allelen

Kodierung der einzelnen Allele zu den Blöcken wurde darauf geachtet, dass ähnliche Blöcke nur eine Hamming-Distanz von 1 bit aufweisen. Diese Vorgehensweise lässt sich exemplarisch an der Kodierung für den Inception- (011) und den Wide-Inception-Block (111) erkennen. So kann gewährleistet werden, dass eine einzelne Punktmutation zu einer lokalen Änderung im Suchraum führt, wie in [3], S. 66 empfohlen wird.

4.4 Initialisierung der Startpopulation

Zur Initialisierung der Startpopulation wurden zunächst acht homogene Netze ausgewählt, die jeweils allein aus einem der oben vorgestellten Blocktypen bestehen. Durch die Wahl der Kodierung entstehen paarweise Individuen, welche auf ihren Chromosomen die maximale Hamming-Distanz im Suchraum S von 63 bit aufweisen (siehe Tabelle 6). Durch die Einführung dieser,

Chromosom 1	Chromosom 2	Hamming-Distanz
000×21	111×21	63
001×21	110×21	63
010×21	101×21	63
011×21	100×21	63

Tabelle 6 – Die maximalen Hamming-Distanzen in der Startpopulation

sich maximal unterscheidenden Individuen in die Startpopulation, soll eine möglichst breite Traversion des Suchraumes ermöglicht werden. Um heterogene Individuen in die initiale Generation einzufügen, wurden zwei weitere

Chromosomen aus gleichverteilt generierten, zufälligen Bitstrings erzeugt.

4.5 Warmup und Data Augmentation

Wie in [23] und [26] wurden zur besseren Generalisierung Data Augmentation-Verfahren verwendet. Um die Vergleichbarkeit der Ergebnisse zu gewährleisten, wurden dieselben Techniken, wie in [34] beschrieben, eingesetzt. Die Bilder wurden zunächst zufällig vertikal gespiegelt. Von den 32×32 Pixel großen Bildern wurde ein Unterbild der Größe 28×28 ausgeschnitten und wieder auf die ursprüngliche Größe gepaddet, um die Translationsinvarianz zu erhöhen. Um die Konvergenz der Netze zu gewährleisten, wurde, wie in [26] beschrieben, eine Warmup-Phase von 400 Iterationen mit der Lernrate $\gamma = 0.01$ durchgeführt.

4.6 Reduzierte Batch-Größe

Die im Rahmen dieser Arbeit eingesetzten Netze gehören zur Gruppe der sehr tiefen KNNs. Durch die hohe Anzahl der Parameter ergibt sich ein hoher Speicherbedarf. Um die Arbeit in einem realistischen Zeitrahmen durchzuführen, wurden die Netze auf einer Grafikkarte trainiert. Hierdurch wird der Grafikspeicher zu einem limitierenden Faktor bezüglich der Netzgröße, der Größe der Minibatches und der Größe der Eingangsbilder.
Der Grafikspeicher der eingesetzten Grafikkarten beträgt $4,5$ GiB (*NVidia Tesla K20c*), bzw. $4,0$ GiB (*NVidia GeForce GTX 1050 ti*). Um die Vergleichbarkeit mit den Arbeiten von Kaiming He et al. ([23, 26]) zu erhalten, wurde darauf geachtet, die Minibatch-Größe möglichst groß zu erhalten. Um alle der nach dem oben vorgestellten Schema konstruierbaren Netze auf beiden Karten trainieren zu können, war maximal eine Minibatch-Größe von 80 Bildern möglich.
Da der Zeitrahmen einer Masterarbeit begrenzt ist, war es notwendig, auch die Anzahl der Epochen, in denen die Netze trainiert werden, zu reduzieren. In Abbildung 16 auf Seite 29 kann abgelesen werden, dass die Klassifikationsgüte auch schon lange vor der Verringerung der Schrittweite γ nahezu stagniert. Daher wurde für die in der Arbeit durchgeführten Experimente

zusätzlich zur Reduzierung der Batch-Größe auch die Anzahl der Iterationen um die Hälfte reduziert. Die Anzahl der Iterationen, nach denen eine Verringerung der Schrittweite erfolgt, wurde dementsprechend angepasst.

4.7 Auswahl der Fitnessfunktion

Die Fitnessfunktion bewertet gemäß [3], S. 36 ein Chromosom c und weist diesem einen Wert $fit_c \in \mathbb{R}$ zu, welcher die Güte bezüglich des zu Grunde liegenden Optimierungsproblems beschreibt. Die Netze sollen bezüglich ihrer Klassifikationsgüte bewertet werden. Daher liegt es zunächst einmal nahe, die Fitness eines Chromosoms c durch die Klassifikationsgenauigkeit auf dem Test-Datensatz aus CIFAR10 zu beschreiben. Da das Argument der Fitnessfunktion ein binäres Chromosom der Länge 63 bit ist, besteht die Funktion also aus der Übersetzung in eine konkrete Netstruktur, 32 000 Trainingsiterationen und den entsprechenden Test-Forward-Passes, welche die Klassifikationsgenauigkeit auf den Testdaten ausgeben. Die Fitness-Funktion ist also ein sehr komplexer Prozess, bestehend aus unterschiedlichen Software-Systemen. Statt der Klassifikationsgüte hätte auch eine auf dem Cross-Entropy-Loss basierende Fitnessfunktion gewählt werden können. Da bei der Bewertung eines neuronalen Netzes häufig die Klassifikationsgenauigkeit auf den Testdaten[23, 26, 20] herangezogen wird, wurde, um die Vergleichbarkeit zu erhalten, diese als Grundlage der Fitnessfunktion ausgewählt.
Beim Betrachten der Auswertung fällt auf, dass die Klassifikationsgüte ab der zweiten Generation zwischen den einzelnen Netzen nur geringfügig variiert. Dies schlägt sich auch in der Varianz, welche in Tabelle 8 auf Seite 60 dargestellt ist, nieder. Da sich aus der Fitnessfunktion die Auswahlwahrscheinlichkeit für die folgende Generation ableitet, sollte sich die Fitness der Individuen deutlich genug unterscheiden, um besseres genetisches Material mit einer bedeutend erhöhten Wahrscheinlichkeit auswählen zu können. Um die Fitness dennoch von der Klassifikationsgenauigkeit abzuleiten, kann man folgende Fitnessfunktion einführen:

$$fit_c = Acc_{Test}(c) - (min(Acc_{Test}, t) - 0,1) \tag{37}$$

Von der Klassifikationsgenauigkeit eines einzelnen Individuums $Acc_{Test}(c)$ wird in Gleichung (37) die minimale Genauigkeit, über alle Individuen der Generation $min(Acc_{Test}, t)$ abgezogen. Zusätzlich wird vom Minimum eine Konstante von $0,1$ abgezogen, sodass für die initiale Population gilt:

$$fit_{c,t=0} = Acc_{Test}(c) \tag{38}$$

Die Fitness bestimmt nur die Auswahlwahrscheinlichkeit eines Individuums innerhalb einer Generation. Daher spielt es keine Rolle, wenn die Fitness eines besser performenden Netzes aus einer zukünftigen Generation möglicherweise schlechter ist, als die eines schlechter klassifizierenden aus einer früheren.

5 Auswertung

Zur Auswertung der Ergebnisse wurden zunächst einige der Experimente aus den in den Grundlagen und Stand der Technik beschriebenen Verfahren in die in Tabelle 7 beschriebene Test-Umgebung, welche auch zur Berechnung der Fitness verwendet wird, überführt. Die im unteren Bereich der Tabelle abgesetzt dargestellten Hyperparameter weight_decay und α, welche zur Regularisierung und als Momentum im Gradientenabstieg verwendet werden, kommen in allen Warmup- und Trainingsphasen zur Anwendung. Die Wahl der Hyperparameter ist aus [23] und [26] entnommen.

5.1 Auswertung bisheriger Netzarchitekturen mit CIFAR10

Um zu überprüfen, ob die Vorteile der Residual-Architektur gegenüber Netzen ohne Shortcut-Verbindungen auch in der verwendeten Test-Umgebung deutlich zum Tragen kommen, wurde analog zu den Experimenten in [23] ein ebenes (*plain*) Netz, ohne Shortcuts, derselben Tiefe wie die anderen zu testenden Netze konstruiert. Der Vergleich der Klassifikationsgenauigkeit eines ebenen Netzes mit einem der in [23] dargestellten Struktur in der Test-Umgebung ist in Abbildung 31 auf Seite 55 dargestellt. Statt der Klassifi-

Warmup	400 Iterationen $\gamma = 0,01$
Training lr_0	16 000 Iterationen $\gamma = 0,1$
Training lr_1	8 000 Iterationen $\gamma = 0,01$
Training lr_2	8 000 Iterationen $\gamma = 0,001$
Test	alle 500 Iterationen CIFAR10-Test-Batch 10 000 Bilder
Regularisierungsparameter	weight_decay = 0,0001
Momentum	$\alpha = 0,9$

Tabelle 7 – Kenngrößen der eingesetzten Test- und Bewertungsumgebung

kationsgüte F ist der relative Fehler e auf den Test-Daten dargestellt. Der Fehler e kann allerdings mit der folgenden Formel einfach in die Klassifikationsgenauigkeit F umgerechnet werden:

$$F = 1 - e \tag{39}$$

Der Fehler e soll minimiert werden. Das ebene Netz erreicht einen Fehler von $e_{plain} = 14,7\,\%$, während das residuelle Netz derselben Tiefe einen Fehler von $e_{res} = 11,5\,\%$ erzielt. Dieser deutliche Unterschied in der Performance weist darauf hin, dass die residuelle Architektur auch in der im Gegensatz zur Originalarbeit [23] eingeschränkten Test-Umgebung nachweisbar ist. Beim Vergleich von Abbildung 31 auf der nächsten Seite und Abbildung 18 auf Seite 31 wird deutlich, dass der prinzipielle Verlauf der Kurven des Trainingsfehlers ähnlich ist. Allerdings liegt bei den entsprechenden Architekturen (hellgrün in Abbildung 18 auf Seite 31, links und in der Mitte) nach dem gesamten Training der Fehler in Abbildung 31 auf der nächsten Seite immer noch bedeutend höher.

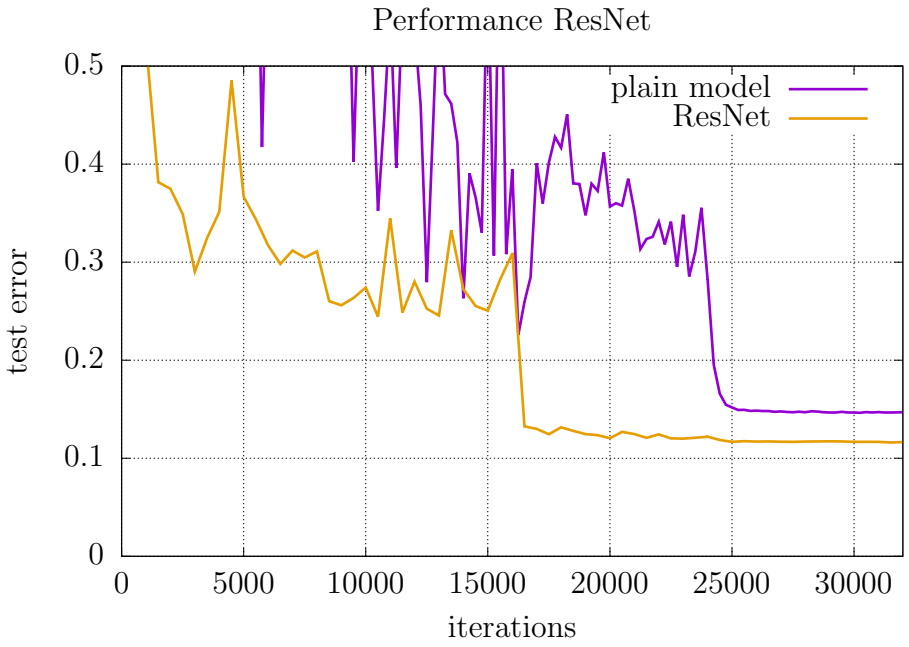


Abbildung 31 – Vergleich der Klassifikationsgüte der beiden ResNet-Architekturen mit dem dazugehörigen nicht residuellen (ebenen) Netz

5.1.1 fb.ResNet-no-relu

In der Startpopulation entspricht eines der Individuen der im FB-Blogeintrag vorgeschlagenen fb.ResNet-no-relu-Struktur. Der Aufbau der einzelnen Blöcke kann Abbildung 25 auf Seite 44 entnommen werden. Wie in [28] und [26] angegeben ist die Struktur ohne eine Störung durch eine Aktivierungsfunktion im Shortcut-Pfad vorteilhaft bezüglich der Klassifikationsgenauigkeit. Ein Vergleich des Testfehlers e mit der Güte des ResNet-44 aus [23] ist in Abbildung 32 auf der nächsten Seite angegeben. Im Einklang mit den Ergebnissen aus [28] und [26] verbessert diese Änderung der Netzstruktur die Testgüte des Netzes nur geringfügig. Die grüne Kurve, welche den Testfehler der fb.ResNet-no-relu-Architektur beschreibt, liegt nur knapp unter der dem ResNet-44 aus [23] entsprechenden Kurve. Die Kurven überschneiden sich auch im Laufe des Trainings häufiger. Eine solche Überschneidung ist auch kurz vor Ende des Trainings zu erkennen. Jedoch erreicht das fb.ResNet-no-relu-Netz am Ende doch einen geringfügig kleineren Fehler von $e = 11,4\ \%$.

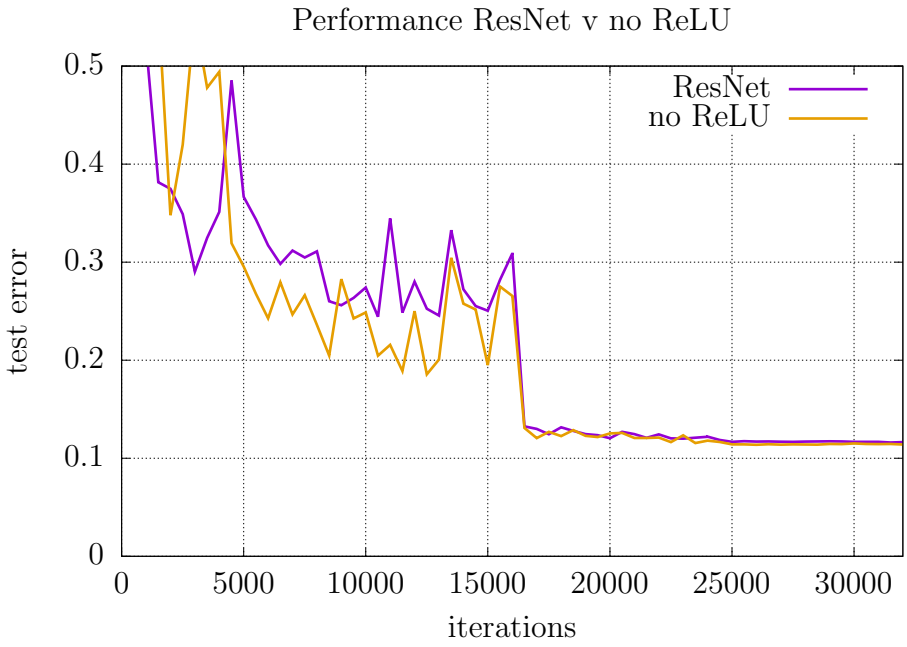


Abbildung 32 – Vergleich des Testfehlers von ResNet-2015 und fb.ResNet-no-relu

5.1.2 fb.ResNet-bn-after-add

Die weitere in [28] untersuchte Architektur wird komplett aus den in Abbildung 29 auf Seite 48 dargestellten Blöcken aufgebaut. Laut [28] weist diese Veränderung der ursprünglichen Netzstruktur Nachteile für die Testgenauigkeit des Netzes auf. Die Ergebnisse eines Tests während des Trainings der ersten Generation im kGA sind in Abbildung 33 auf der nächsten Seite abzulesen. Die Performance in der durch die Fitnessfunktion definierten Testumgebung bestätigt die Ergebnisse aus [28] und [26]. Die Blockstruktur mit einer durch die BatchNorm-Schicht zusätzlich in den Shortcut-Pfad eingefügten Störung weist einen höheren Test-Fehler als die als Referenz genutzte Struktur aus [23] auf. Der Fehler am Ende des Test-Vorganges beträgt: $e = 14,0\ \%$.

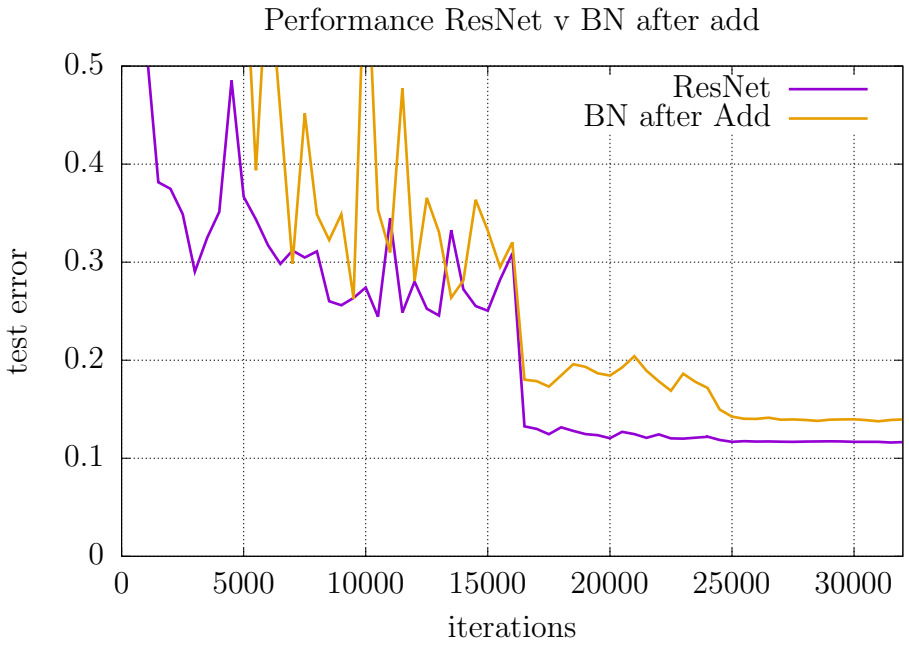


Abbildung 33 – Vergleich des Testfehlers von ResNet-2015 und fb.ResNet-bn-after-add

5.1.3 no-act

Aus den Ideen in [26] und dem FB-Blogeintrag ist auch die Architektur, ohne Aktivierungsfunktion nach der zweiten Konvolution, abgeleitet. Die Block-Struktur ist in Abbildung 27 auf Seite 46 dargestellt. In Übereinstimmung mit den Ergebnissen aus [26] zeigt das aus dieser Struktur abgeleitete, homogene Netz keine Konvergenz, sondern verharrt bei einer Klassifikationsgenauigkeit von $0,1$ % (siehe Abbildung 37 auf Seite 62).

5.1.4 Residual Inception Nets

Die Performance der beiden eingesetzten, auf der Inception-Architektur aus [20] und [30] basierenden Blockstrukturen kann Abbildung 34 auf der nächsten Seite entnommen werden. Obwohl durch diese Architektur komplexere Filter und mehr Parameter pro Block optimiert werden können als bei dem als Referenznetz genutzten ResNet-44 aus [23], ist der in der Testumgebung erreichte Testfehler e deutlich höher.

Abbildung 34 – Klassifikationsgüte der angepassten Inception-Architekturen

Die homogen aus den komplexen Inception-Architekturen aufgebauten Netze gehören also zu den am schlechtesten abschneidenden Individuen der initial getesteten Netze, welche später in dieser Arbeit auch zur Population der ersten Generation des kGA genutzt werden.

5.1.5 Präaktivierung, ResNet-2016

In [26] wird die Präaktivierungsstruktur als Verbesserung gegenüber der Struktur aus [23] vorgeschlagen. Die Ergebnisse der Experimente mit dem CIFAR10-Datensatz sind allerdings nur für bereits sehr tiefe Netze angegeben. Das flachste Netz mit Präaktivierung, welches getestet wird, ist ResNet-110. Die hier getestete Architektur entspricht dem deutlich flacheren ResNet-44. Bei dieser flachen Architektur ist, in Übereinstimmung mit den Ergebnissen aus [29], kein Vorteil durch die Präaktivierung erkennbar. Im Gegenteil zeigen die Ergebnisse in Abbildung 35 auf der nächsten Seite, dass die Präaktivierung sogar einen nachteiligen Effekt auf die Klassifikationsgenauigkeit hat. Der nachteilige Effekt der Präaktivierung geht sogar so weit, dass

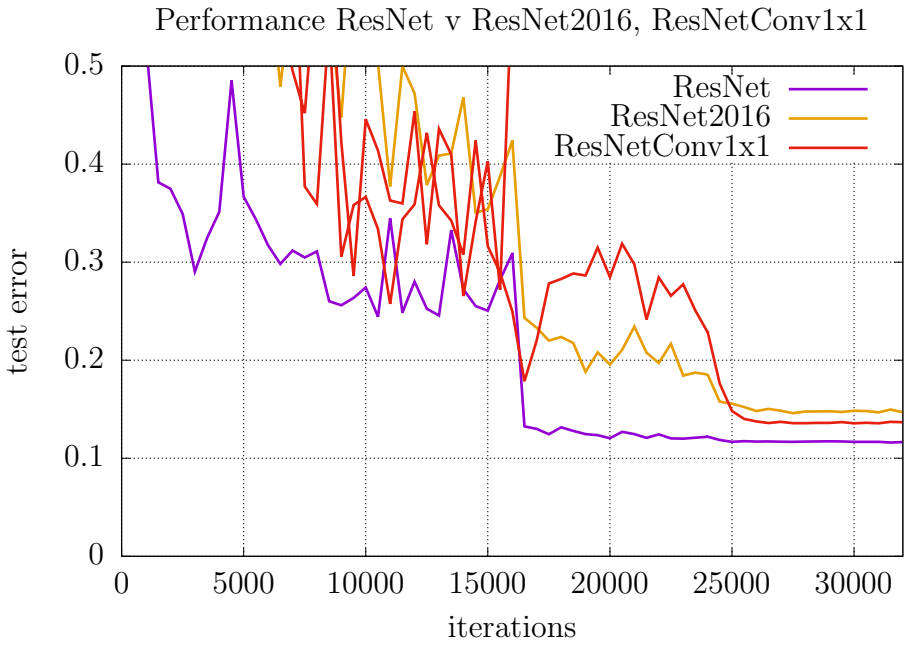


Abbildung 35 – Klassifikationsfehler zweier Architekturen aus [26], im Vergleich zur ursprünglichen ResNet-Struktur aus [23]

selbst die in Abbildung 30 auf Seite 48 dargestellte Struktur, zu einem homogenen Netz aufgebaut, zu einem geringeren Test-Fehler führt. Bei den tieferen Netzen aus [26] kann der umgekehrte Effekt beobachtet werden. Hier führt die Präaktivierung zu besseren Ergebnissen als die Struktur mit einer in den Shortcut-Pfad eingebauten 1×1-Konvolution.

5.2 Ergebnisse des kGA

Das aus der Aufgabenstellung abgeleitete Ziel der Arbeit lautet, durch die Anwendung eines evolutionären Algorithmus die Struktur eines Neuronalen Netzes so anzupassen, dass die Klassifikationsgenauigkeit gegenüber der eines Referenznetzes positiv beeinflusst wird. Als evolutionärer Algorithmus wird ein an das Problem angepasster kanonischer Genetischer Algorithmus verwendet. Dieser Algorithmus ist in Section 4 auf Seite 40 im Detail erklärt. Die erzeugten Netz-Strukturen sind auf Github verfügbar[5].

[5] `https://github.com/nils489/EvANet`

Eine der Auswirkungen der evolutionären Optimierung der Netzstrukturen kann aus Tabelle 8 abgelesen werden. Die Varianz der Klassifikationsgüte des kGA nimmt im Laufe der Iterationen $t < T_{MAX}$ deutlich ab. Dieses Phänomen kann entweder auf eine deutliche Konvergenz in Richtung des Optimums oder eine vorzeitige Konvergenz mit einer deutlich eingeschränkten Durchsuchung des Suchraumes S hindeuten. Um die Breite der Suche in S

Generation	**Varianz**
0	0.0527
1	0.0025
2	$5.05 \cdot 10^{-5}$
3	$7.35 \cdot 10^{-5}$
4	$1.91 \cdot 10^{-6}$

Tabelle 8 – Varianz der Klassifikationsgüte des kGA

zu bewerten, ist es hilfreich, die relativen Häufigkeiten der unterschiedlichen Block-Architekturen in der jeweiligen Generation zu betrachten. Aus Tabelle 9 auf der nächsten Seite ist erkennbar, dass keines der vorkommenden Allele komplett ausstirbt. Sollte eines der Allele jemals aussterben, kann es durch die Mutation allerdings in jeder weiteren Generation wieder in den Genpool eingefügt werden. Die Häufigkeit der ResNet-2015-Architektur, kodiert durch 000, geht zwar deutlich zurück, doch mindestens eine einzelne Instanz dieses Allels bleibt in jeder der betrachteten Generationen erhalten. Die durch 111 kodierte, breite Inception-Architektur erscheint am erfolgreichsten, überflutet jedoch nicht den Genpool, sodass eine deutliche genetische Variabilität auch nach fünf Generationen weiterhin erhalten bleibt.
Der Verlauf der maximalen, minimalen und mittleren Klassifikationsgüte der Generationen ist in Abbildung 36 auf der nächsten Seite zu erkennen. Dadurch, dass das homogen aufgebaute Netz aus residuellen Schichten ohne Aktivierung nach der zweiten Faltung nicht konvergiert ist, sondern bei einer Klassifikationsgüte von etwa $0,1$ verharrte, ist die Varianz in der Startpopulation sehr groß. Verfolgt man die Entwicklung der relativen Häufigkeit, der durch 100 kodierten no-act-Architektur in Tabelle 9 auf der nächsten Seite, wird deutlich, dass die Häufigkeit bereits nach der ersten Generati-

	gen 0	gen 1	gen 2	gen 3	gen 4
000	11,9 %	3,4 %	3,4 %	0,5 %	0,5 %
001	12,9 %	12,9 %	21,9 %	9,1 %	12,9 %
010	12,4 %	12,4 %	13,3 %	20,5 %	10,0 %
011	12,9 %	20,5 %	14,3 %	15,5 %	14,8 %
100	12,4 %	3,4 %	3,4 %	4,6 %	3,8 %
101	13,8 %	4,3 %	8,6 %	4,6 %	2,9 %
110	11,9 %	11,5 %	12,9 %	16,9 %	23,3 %
111	11,9 %	32,1 %	22,4 %	28,3 %	31,9 %

Tabelle 9 – Relative Häufigkeiten der Allele pro Generation im kGA

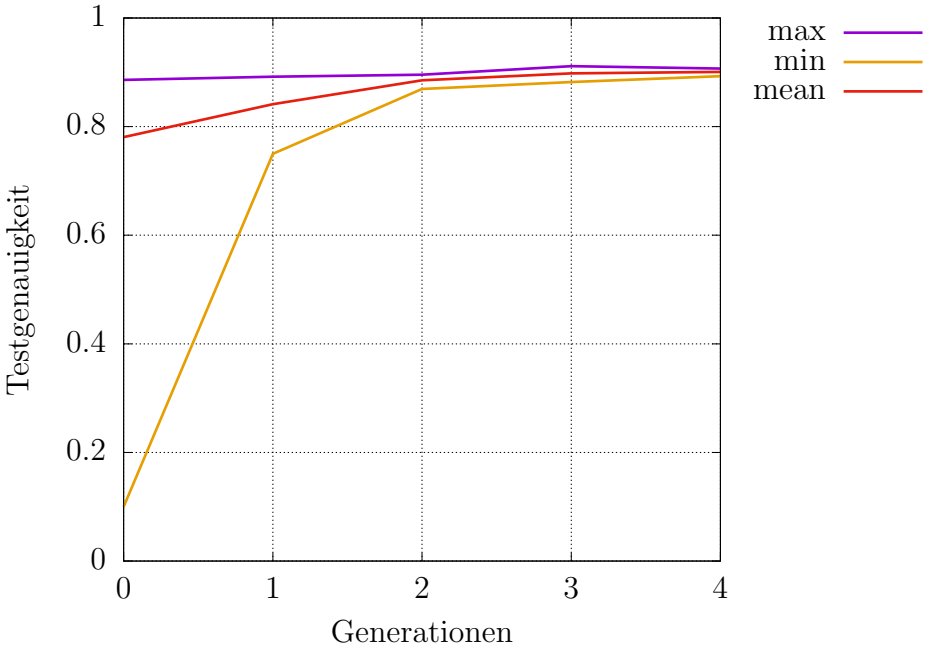


Abbildung 36 – Verlauf der maximalen, minimalen und mittleren Klassifikationsgüte der Generationen

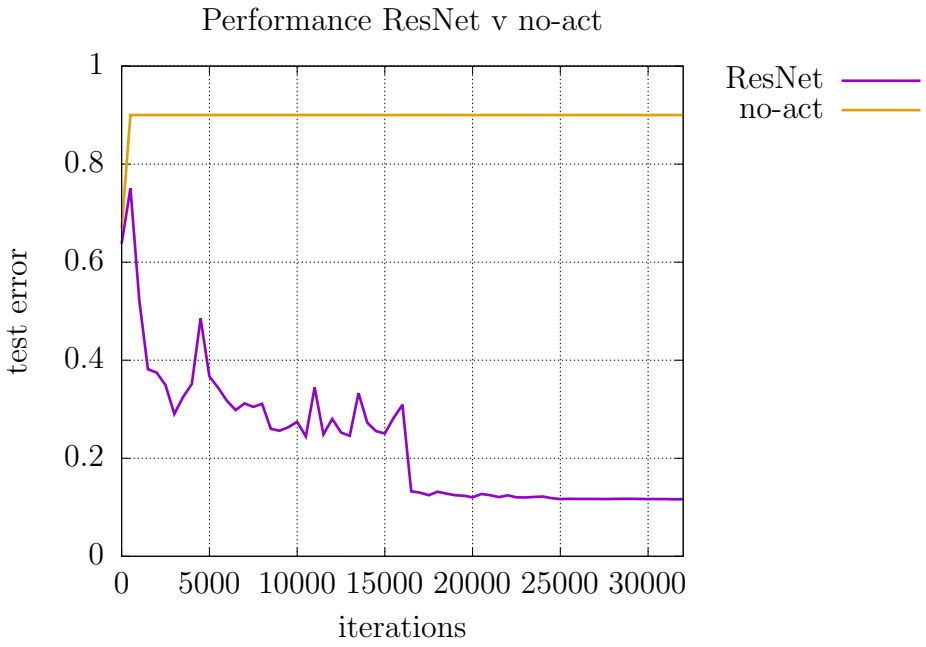


Abbildung 37 – Klassifikationsgüte des no-act-Netzes

on stark absinkt. Dieses Verhalten ist dadurch zu erklären, dass kein Individuum mit der zugehörigen homogenen Architektur in die nächste Generation übernommen wurde. Da die Klassifikationsgüte aller Individuen der folgenden Generationen sehr hoch ist, wird davon ausgegangen, dass einzelne no-act-Blöcke nicht die Konvergenz des gesamten Netzes verhindern. Beim Vergleich der Häufigkeiten von no-act (100) mit der Präaktivierungsstruktur (101) und der ursprünglichen ResNet-Struktur (000) in Tabelle 9 auf der vorherigen Seite fällt sogar auf, dass die no-act-Architektur im letzten Iterationsschritt sogar häufiger vorhanden ist. Es zeichnet sich also ab, dass von der Performance einer Blockstruktur in einem homogenen Netz nicht auf ihre Performance in einem heterogenen Netz geschlossen werden kann. Jedoch werden im Rahmen dieser Arbeit nicht genügend Individuen und Generationen durchlaufen, als dass ein Rückschluss von der Häufigkeit eines Allels auf dessen Performance statistisch belastbar sein könnte.
In Abbildung 37 kann abgelesen werden, dass die Klassifikationsgenauigkeit zu Beginn des Trainings sogar etwas besser ist, als im späteren Verlauf. Dieses

Phänomen lässt sich durch die probabilistische Initialisierung der Gewichte des Netzes erklären. Hierdurch kann eine für die Klassifizierung vorteilhafte Parameterverteilung erfolgen. Der Erwartungswert E der Klassifikationsgenauigkeit bei den randomisiert initialisierten Gewichten liegt allerdings bei 10 Klassen bei $E = 0,1$.
Im weiteren Verlauf des kGA werden die minimale und die mittlere Klassifikationsgüte in jedem Iterationsschritt verbessert. Die Klassifikationsgenauigkeit des besten Individuums aus der vorletzten Generation liegt mit $0,9114$ leicht über der maximalen Klassifikationsgenauigkeit von $0,9070$ der letzten Generation. Aufgrund der probabilistischen Natur der evolutionären Algorithmen ist es allerdings nicht ungewöhnlich, dass die fittesten Individuen einer folgenden Generation eine geringere Güte als die einer vorhergehenden aufweisen können[3].
Der Testfehler e der neuronalen Netze der letzten Generation ist in Abbildung 38 auf der nächsten Seite aufgeführt. Auch hier wird nochmals deutlich, dass die Varianz der Testfehler, im Vergleich zur initialen Population, deutlich abgenommen hat. Zum Vergleich sei an dieser Stelle auf das beste Individuum aus der ersten Generation in Abbildung 32 auf Seite 56 und das schlechteste in Abbildung 37 auf der vorherigen Seite verwiesen. Führt man sich nun nochmal die Häufigkeiten der einzelnen Block-Architekturen aus Tabelle 9 auf Seite 61 vor Augen und vergleicht die Klassifikationsgenauigkeiten der letzten mit der ersten Generation, so wird deutlich, dass der Einsatz von Netzen, die nicht homogen nur aus einer Art von Blöcken bestehen, einen vorteilhaften Effekt auf die Klassifikation zu haben scheint.

5.3 Bewertung des fittesten Individuums

Der Klassifikationsfehler des global fittesten Individuums in der zur Optimierung genutzten Test-Umgebung ist in Abbildung 39 auf Seite 65 dargestellt. Im weiteren Verlauf wird dieses Individuum, da es durch einen evolutionären Algorithmus erzeugt wurde, als EvANet bezeichnet. Das EvANet erreicht in der gegebenen Testsituation einen Fehler $e = 8,86$ %. Das Netz klassifiziert die Testdaten also zu $2,78$ Prozentpunkten besser als das ResNet-44

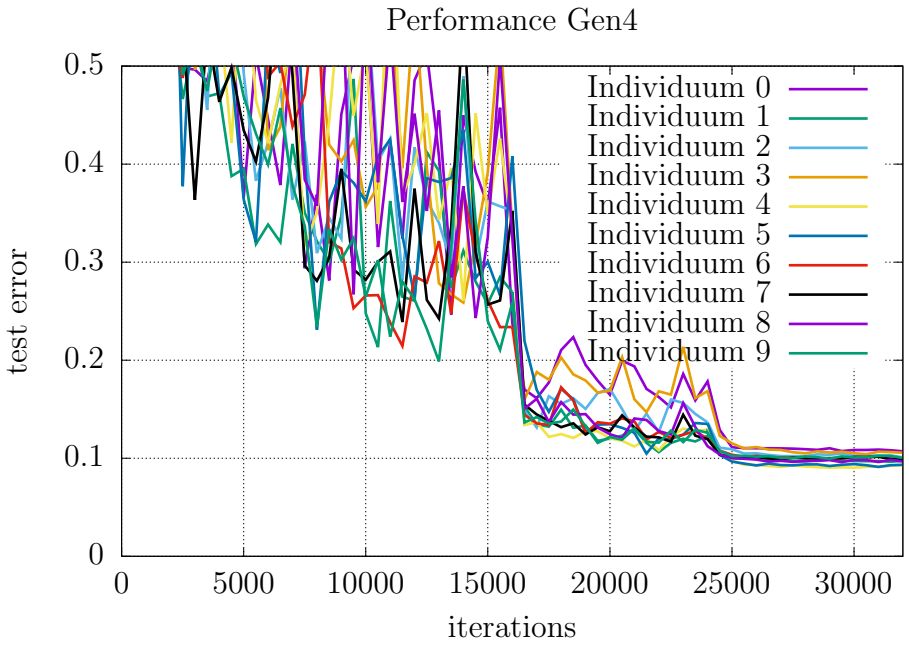


Abbildung 38 – Klassifikationsgüte der letzten Generation

aus [23], welches in dieser Arbeit als Referenz herangezogen wird. Aufgrund der hohen Leistung in diesem Testverfahren wird in Abbildung 40 auf Seite 66 der Aufbau dieses Individuums im Detail vorgestellt. Nach der initialen Faltungsschicht folgt mit den ersten beiden residuellen Blöcken zweimal die im Rahmen dieser Arbeit angepasste Inception-Struktur mit einem Shortcut und tieferen 5×5-Filtern. Daraufhin folgt eine ResNet-2016-Schicht mit Prä-Aktivierung gemäß [26]. Auf die zweite Konvolutions-Schicht in diesem Block folgt aufgrund der Prä-Aktivierung weder eine Batch-Normalisierung, noch eine Aktivierungsfunktion durch eine ReLU-Schicht. Das am Ende des Blockes aufaddierte Signal wird also ohne Nichtlinearität und Normalisierung an die folgende Inception-Schicht mit einem Shortcut weiter geleitet. In diesem Block ist die Tiefe der 5×5-Filter nicht erhöht.
Es folgen drei Inception-Blöcke, mit 5×5-Filtern erhöhter Tiefe und ein residueller Block aus [23] ohne Prä-Aktivierung und mit einer ReLU-Schicht, die den bisher ungehinderten Shortcut-Pfad des Netzes bevölkert. Hiernach folgen 13 Wide-Inception-Blöcke und die aus [26] übernommene Ausgabe-

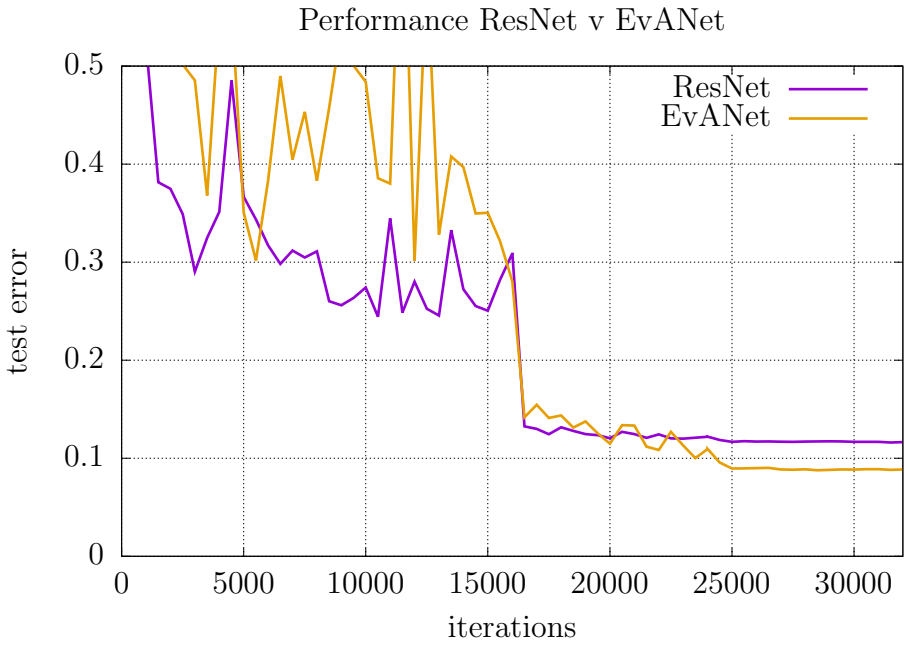


Abbildung 39 – EvANet-Performance in der Testumgebung des kGA

Struktur. Nach dem einzelnen traditionellen ResNet-Block liegt keine Störung im Shortcut-Pfad des Netzes mehr vor. Das Netz kann also ab dem achten Block die Vorteile der in [26] vorgestellten Architektur ausnutzen.
Bei der Betrachtung der in Abbildung 40 auf der nächsten Seite dargestellten EvANet-Struktur erscheint es zunächst durch den massiven Einsatz der Wide Inception-Blöcke so, als sei ganz einfach diese Architektur durch die größere optimierbare Parameteranzahl der Grund für die höhere Klassifikationsgüte. Vergleicht man allerdings nochmals in Abbildung 34 auf Seite 58 die Klassifikationsgüte des homogenen Netzes aus dieser Block-Struktur, so wird deutlich, dass diese nicht allein für die erhöhte Performance verantwortlich gemacht werden kann. Der Grund für das Ergebnis muss also in der Kombination der eingesetzten Blockstrukturen liegen.
Um die Ergebnisse des besten Individuums des kGA mit den Ergebnissen der in Abschnitt 3 vorgestellten Netze vergleichen zu können, wurde das EvANet nochmals in dem in [23], [26] und [29] genutzten Test-Setting trainiert und bewertet. Im Gegensatz zur Testumgebung des kGA wurde die Batch-Größe

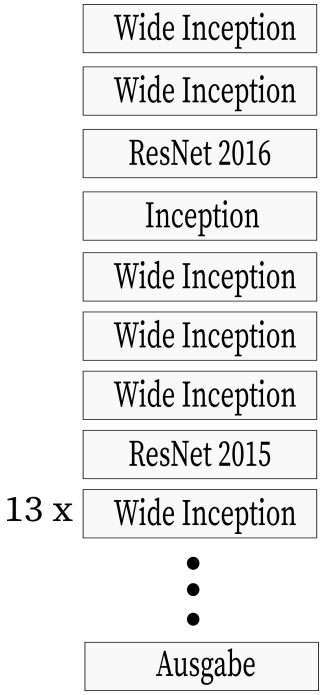


Abbildung 40 – Aufbau des am besten klassifizierenden Individuums (EvA-Net)

auf 128 erhöht und die Anzahl der Iterationen bis zur Anpassung der Schrittweite γ jeweils verdoppelt. Der Verlauf des Testfehlers kann Abbildung 41 auf der nächsten Seite entnommen werden. Der im Gegensatz zum Setting des kGA deutlich geringere Test-Fehler lässt sich nicht allein durch die bereits beschriebenen Anpassungen des Test-Verfahrens erklären. Statt der 10 000 in CIFAR10 enthaltenen Testbilder wurde, wie in [23] beschrieben, die Menge der Trainingsbilder in zwei Untermengen mit jeweils 45 000 und 5 000 Bilder aufgeteilt und die kleinere Untermenge zum Testen genutzt. Eine komplette Cross-Validierung über die gesamten Trainingsdaten war im Zeitrahmen der Masterarbeit leider nicht möglich. Jedoch wurden die meisten der in [23] vorgestellten Netze auch nur jeweils über einem Trainings- und Validierungssplit bewertet.

Der minimal erreichte Klassifikationsfehler $e_{min} = 7,17\,\%$ des ResNet-44 aus [23] ist als waagerechte, gepunktete Linie in Abbildung 41 auf der nächsten Seite zu erkennen. Die Klassifikationsgüte des EvANet in dieser Testumge-

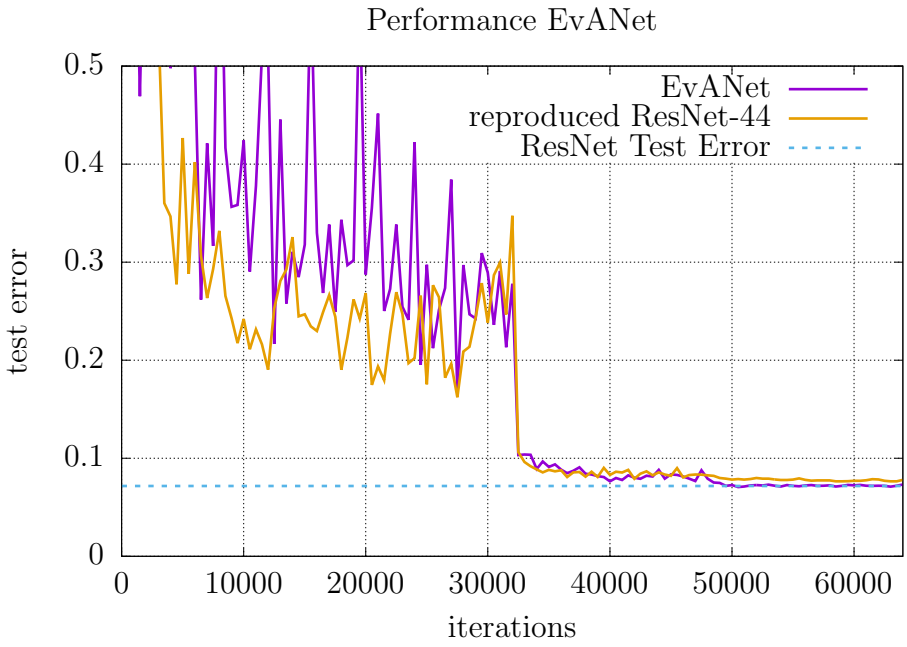


Abbildung 41 – EvANet-Performance in der Testumgebung der Original-Veröffentlichung [23]

bung ist mit einem minimalen Testfehler von $e_{min} = 7,06\,\%$ nur geringfügig besser als die des Referenznetzes. Der Unterschied ist so klein, dass er in Abbildung 41 nicht zu erkennen ist. Da in der Veröffentlichung zum ResNet-44 nicht erkennbar ist, welche Kombination möglicher Untermengen aus CIFAR10 als Trainings- und Testdatensatz verwendet wurden, wurde die Referenzimplementierung des Netzes auf der im Rahmen dieser Arbeit genutzen Kombination getestet. Das Ergebnis ist in Abbildung 41 als reprodiziertes ResNet-44 angegeben. Der minimal erreichte Klassifikationsfehler $e_{min} = 7,54\,\%$ zeigt hier eine geringfügig schlechtere Performance als in der zugehörigen Veröffentlichung[23]. Die Vermutung liegt nahe, dass dieser Unterschied auf die unterschiedlichen Trainings- und Testdaten zurückzuführen ist.

Der Performancegewinn des EvANet gegenüber dem reproduzierten ResNet-44 ist in dem vorgestellten Szenario zwar geringer als im Test-Szenario des kGA, jedoch immer noch deutlich in Abbildung 41 ablesbar. Die deutlich

bessere Klassifikation im kGA legt nahe, dass EvANet in weniger Iterationen trainiert werden kann, bis es eine ausreichende Genauigkeit erreicht als die Referenzarchitektur.

5.4 Untersuchung ausgewählter Gewichte des EvANet

Zur Interpretation der gelernten Features und als Hilfestellung für weitere Optimierungen an der Struktur der im Rahmen dieser Arbeit erstellten Netze ist es wichtig, die am Ende des Trainingsvorganges bestehenden Gewichte zu betrachten. In Abbildung 42 sind exemplarisch die gelernten Gewichte im ersten Wide Inception-Block dargestellt. Bei den links im Bild zu erkennenden vielen Filtern mit nur sehr geringem Kontrast unterscheiden sich die Gewichte nur geringfügig. Diese Filter scheinen also überhaupt nicht wirklich auf das Problem der Klassifizierung trainiert zu sein. Dieses Verhalten tiefer

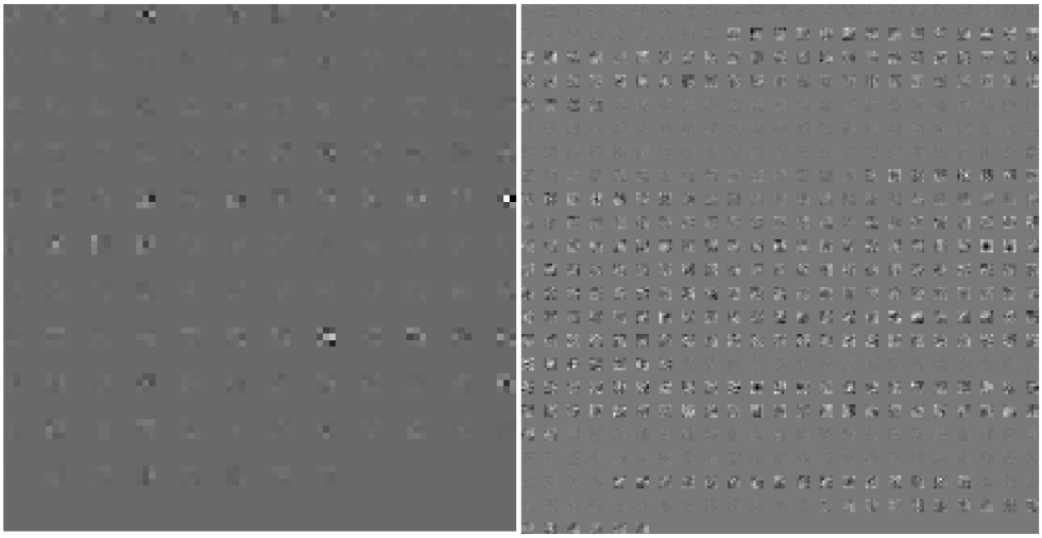


Abbildung 42 – Das 3×3- und 5×5 des ersten Wide Inception-Blocks

Netze wird auch in der Arbeit zu Wide Residual Networks[29] beschrieben. Die geringe Durchdringung der 3×3-Filter ist auch in den weiteren trainierten Schichten der Netze mit Inception- und Wide Inception-Blöcken konsistent zu beobachten. Dieses Ergebnis legt nahe, dass sich für die zu Grunde liegenden Klassifizierungsprobleme die Tiefe der 3×3-Filter weiter reduzieren lässt. Allerdings erreichen die Netz-Architekturen, die nur auf 3×3-Filtern

beruhen, ähnliche Klassifikationsgüten. In [22], [29] und [31] wird ein gegensätzliches Vorgehen empfohlen, sodass keine größeren als die 3×3-Filter verwendet werden sollen.
In Abbildung 43 sind exemplarisch vier trainierte 5×5-Filter aus unterschiedlichen Schichten des EvANet abgebildet. Die ersten beiden Kernel nähern sich einem diagonal orientierten Gabor-Filter und einem negativen diagonal orientierten Gabor-Filter an. Der dritte Kernel entspricht näherungsweise einem negativen Gauß-Filter. Der vierte entspricht einem den ersten beiden Kerneln entgegengesetzten Gabor-Filter höherer Frequenz. Zum Vergeleich sind in Abbildung 44 zwei Gabor-Filter unterschiedlicher Orientierung und Größe aus [17] dargestellt.

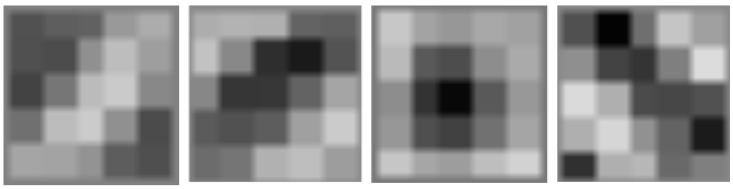


Abbildung 43 – Vier ausgewählte Filter aus EvANet

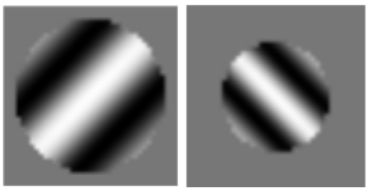


Abbildung 44 – Gabor-Filter unterschiedlicher Größe und Ausrichtung aus [17]

In der Arbeit zu GoogLeNet[20] geben die Autoren an, dass sie sich in der Entwicklung der Inception-Architektur an Arbeiten von Riesenhuber und Poggio [21] über vom visuellen Cortex abgeleitete Bildverarbeitungs-Systeme orientieren. Das in [21] vorgeschlagene System basiert auf dem Einsatz bereits festgelegter Gabor-Filter, in Übereinstimmung mit [17] und den grundlegenden Arbeiten von Hubel und Wiesel in [15]. Die Visualisierung der trainierten Gewichte in den Inception-Blöcken in EvANet zeigt, dass sich die Filter durch die anpassbaren Gewichte den Gabor-Filtern und damit den beobach-

teten Operationen im visuellen Cortex von Säugetieren gemäß [15], [16] und [17] annähern.

6 Zusammenfassung der Ergebnisse

Im Rahmen dieser Arbeit wird die Anwendung eines kanonischen Genetischen Algorithmus (kGA) zur Optimierung der Strukturen neuronaler Netzwerke untersucht. Zur Optimierung wird auf Netzwerkarchitekturen, die dem aktuellen Stand der Technik entsprechen, aufgebaut. Aufgrund der Limitationen durch den Zeitrahmen der Masterarbeit und die eingesetzte Hardware konnte der Algorithmus nur mit wenigen Iterationen und einer geringen Populationsgröße durchgeführt werden.
Innerhalb der angewendeten Iterationen des Algorithmus konnte die Klassifikationsgüte gegenüber den als Referenz genutzten Netzarchitekturen in der verwendeten Test-Umgebung des kGA deutlich verbessert werden. Beim Vergleich mit dem als Referenz verwendeten ResNet-44 in der Testumgebung aus [23] und [26] ist auch eine geringfügige Verbesserung der Klassifikationsgüte erkennbar.
Die innerhalb des kGA entwickelten Netzstrukturen unterscheiden sich strukturell dadurch von den anderen, dem Stand der Technik entsprechenden Architekturen, dass sie nicht homogen aus der Wiederholung immer wieder derselben Grundstruktur aufgebaut sind, sondern heterogen aus unterschiedlichen Blöcken bestehen. Diese Arbeit zeigt also, dass heterogene Netzstrukturen bessere Ergebnisse als ihre homogenen Gegenstücke liefern können. In ausgewählten Test-Szenarien erzielen die heterogenen Netze sogar deutlich bessere Ergebnisse. Außerdem wird deutlich, dass von der Güte homogener Netze einer Blockart nicht in trivialer Weise auf den Effekt der Blöcke in einem heterogenen Netz geschlossen werden kann.
Bei der Betrachtung der Gewichte der trainierten, in dieser Arbeit entwickelten Netzstruktur wird deutlich, dass sich die an biologischen Vorbildern orientierten Inception-Blöcke aus [20] durch die Anpassung der Gewichte den Filter-Funktionen im visuellen Cortex von Säugetieren aus [17], [21], [16] und [15] annähern.
Es wurde gezeigt, dass die Anwendung evolutionärer Algorithmen auf die Struktur neuronaler Netze vorteilhafte Effekte auf die Performance dieser Netze haben kann, bezüglich des Trainingsaufwandes und der Klassifikati-

onsgenauigkeit. Daher sollte diese Kombination von Verfahren in Zukunft genauer untersucht werden.

7 Ausblick

Die Ergebnisse dieser Arbeit legen nahe, dass die mithilfe des kGA entwickelte Netzarchitektur mit weniger Trainingsiterationen auf Datensätze angelernt werden kann als die vergleichbaren Architekturen, welche als Referenz genutzt werden. Möglicherweise können auch noch bessere Klassifikationsraten durch eine maßgeschneiderte Wahl der Trainings-Hyperparameter für das EvANet und einen stärkeren Einsatz der Daten-Augmentation erreicht werden.
Die Auswertung der gewichteten Filter legt nahe, dass die Tiefe der Konvolutionsschichten an einigen Stellen der Netzstruktur weiter verringert werden kann. Durch eine solche Anpassung kann die Komplexität des Trainingsvorganges verringert werden, und die Netze können in kürzerer Zeit und auf weniger potenter Hardware genutzt werden.
Da gezeigt wurde, dass die Anwendung des kGA auf die Netzwerkstrukturen schon in geringem Umfang zu verbesserten Klassifikationsgüten führt, liegt es nahe, in Zukunft mit leistungsfähigerer Hardware die Netzstrukturen nochmals mithilfe eines weiter angelegten kGA zu optimieren. In Zukunft sollten die Anzahl der Generationen, sowie die Populationsgröße erhöht werden. Außerdem sollte das Test-Setting, um die Vergleichbarkeit zu gewährleisten, möglichst nahe den in anderen Veröffentlichungen genutzten angepasst werden. Bei einer groben Durchsicht durch die, auf der in Section 2.3.4 auf Seite 18 erwähnten Website[6] bezüglich CIFAR10 aufgeführten Veröffentlichungen fällt allerdings auf, dass grundsätzlich unterschiedliche Test-Settings genutzt

[6]Die Ergebnisse und Veröffentlichungen zu bekannten Klassifikations-Challenges können unter `https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html` abgerufen werden[24].

werden[24]. Daher ist es ratsam, zur Vergleichbarkeit die Test-Ergebnisse aus anderen Veröffentlichungen nochmals zu reproduzieren.
Verbesserte Klassifikationsergebnisse sind auch eine Voraussetzung für Objekterkennungssysteme, wie z. B. Faster-R-CNN[35]. Gerade im Kontext der Verkehrsforschung wird die im Rahmen dieser Arbeit entwickelte Netzstruktur auch in Objekterkennungssystemen eingesetzt werden.

Anhang

Implementierung des Mutationsoperators

mutate_chromosome.py

```
#!/usr/bin/env python

import numpy as np
import pickle
import sys
import re

filename = sys.argv[1]
allel_size = 3

# helper function to translate indices
def translate_allel_to_index(al):
  return al[0]*4+al[1]*2+al[2]

# function that flips one bit in the chromosome
def mutate_gene(gene_ind, flat_chrom):
  if(flat_chrom[gene_ind] == 1):
    flat_chrom[gene_ind] = 0
  elif(flat_chrom[gene_ind] == 0):
    flat_chrom[gene_ind] = 1
  print("gene mutated at "+str(gene_ind))

gene_file = open(filename, 'rb')
chromosome = pickle.load(gene_file)
gene_file.close()

# initialize flattened chromosome with zeros
flat_chromosome = np.zeros(len(chromosome)*allel_size)

# fill flattened chromosome
```

```
for i in range(0,len(chromosome)):
  for j in range(0,allel_size):
    flat_chromosome[(i*allel_size)+j] = chromosome[i][j]

# determine loci to mutate
mut_prob_arr = np.random.uniform(0.0,1.0,len(
   flat_chromosome))
for i in range(0, len(flat_chromosome)):
  if (mut_prob_arr[i] <= (1.0/len(flat_chromosome))):
    mutate_gene(i, flat_chromosome)

# write flattened chromosome back to original chromosome
for i in range(0,len(chromosome)):
  for j in range(0,allel_size):
    chromosome[i][j] = flat_chromosome[(i*allel_size)+j]


gene_file = open(filename, 'wb')
pickle.dump(chromosome, gene_file)
gene_file.close()
```

Implementierung des Crossover-Operators

crossover_genes.py

```
#!/usr/bin/env python

import numpy as np
import pickle
import sys

#read gene-files
gene1_filename = sys.argv[1]
gene2_filename = sys.argv[2]
gene1_file = open(gene1_filename, 'rb')
gene2_file = open(gene2_filename, 'rb')

gene1 = pickle.load(gene1_file)
gene2 = pickle.load(gene2_file)

gene1_file.close()
gene2_file.close()

# randomly determine crossover point
crossover_point = np.random.randint(len(gene1))
print("crossing over at "+str(crossover_point))

for i in range(crossover_point, len(gene1)):
  gene1[i] = gene2[i]

for i in range(0, crossover_point):
  gene2[i] = gene1[i]


# save new genes
gene1_file = open(gene1_filename, 'wb')
gene2_file = open(gene2_filename, 'wb')
```

```
pickle.dump(gene1, gene1_file)
pickle.dump(gene2, gene2_file)
gene1_file.close()
gene2_file.close()
```

Literatur

[1] Karpathy, Andrej: *Lessons learned from manually classifying CIFAR-10*. https://karpathy.github.io/2011/04/27/manually-classifying-cifar10/. Version: 2011. – abgerufen am 17.02.2017

[2] Stanley, Kenneth O. ; Miikkulainen, Risto: Evolving Neural Networks Through Augmenting Topologies. In: *Evol. Comput.* 10 (2002), Juni, Nr. 2, 99–127. http://dx.doi.org/10.1162/106365602320169811. – DOI 10.1162/106365602320169811. – ISSN 1063–6560

[3] Gerdes, Ingrid ; Klawonn, Frank ; Kruse, Rudolf: *Evolutionäre Algorithmen*. 1. Friedr. Vieweg & Sohn Verlag/GWV Fachverlage GmbH, Wiesbaden, 2004. – ISBN 3–528–05570–7

[4] Dawkins, R: *The Selfish Gene*. Oxford University Press, Oxford, UK, 1976

[5] Crick, Francis: *The astonishing hypothesis*. London : Touchstone Books, 1990 https://cds.cern.ch/record/434377

[6] Holland, John H.: *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA : MIT Press, 1992. – ISBN 0262082136

[7] Schwefel, H. P.: Evolutionsstrategie und numerische optimierung. In: *PhD Thesis* (1975)

[8] Rechenberg, I.: *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart-Bad Cannstatt : Frommann-Holzboog, 1973 (Problemata 15)

[9] McCulloch, Warren S. ; Pitts, Walter H.: A Logical Calculus of the Ideas Immanent in Nervous Activity. In: *Bulletin of Mathematical Biophysics* Bd. 5, 1943, S. 115–133

[10] Rojas, Raúl: *Neural networks: a systematic introduction.* Springer Science & Business Media, 1996

[11] Minsky, Marvin ; Papert, Seymour: *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MA, USA : MIT Press, 1969

[12] Gallant, S. I.: Perceptron-based Learning Algorithms. In: *Trans. Neur. Netw.* 1 (1990), Juni, Nr. 2, 179–191. http://dx.doi.org/10.1109/72.80230. – DOI 10.1109/72.80230. – ISSN 1045–9227

[13] Rosenblatt, Frank: Principles of neurodynamics. perceptrons and the theory of brain mechanisms / DTIC Document. 1961. – Forschungsbericht

[14] Rosenblatt, F.: The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. In: *Psychological Review* (1958), S. 65–386

[15] Hubel, D. H. ; Wiesel, T. N.: Receptive fields of single neurones in the cat's striate cortex. In: *The Journal of Physiology* 148 (1959), Nr. 3, 574–591. http://dx.doi.org/10.1113/jphysiol.1959.sp006308. – DOI 10.1113/jphysiol.1959.sp006308. – ISSN 1469–7793

[16] Stanley, Garrett B. ; Li, Fei F. ; Dan, Yang: Reconstruction of Natural Scenes from Ensemble Responses in the Lateral Geniculate Nucleus. In: *J. Neurosci* 19 (1999), S. 8036–8042

[17] Serre, T. ; Kouh, M. ; Cadieu, C. ; Knoblich, U. ; Kreiman, G. ; Poggio, T.: A theory of object recognition: Computations and circuits in the feedforward path of the ventral stream in primate visual cortex / Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology. 2005. – Forschungsbericht

[18] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Neurocomputing: Foundations of Research. Version: 1988. http://dl.acm.org/citation.cfm?id=65669.104451. Cambridge, MA, USA

: MIT Press, 1988. – ISBN 0–262–01097–6, Kapitel Learning Representations by Back-propagating Errors, 696–699

[19] Lecun, Y. ; Bottou, L. ; Bengio, Y. ; Haffner, P.: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE* 86 (1998), Nov, Nr. 11, S. 2278–2324. http://dx.doi.org/10.1109/5.726791. – DOI 10.1109/5.726791. – ISSN 0018–9219

[20] Szegedy, Christian ; Liu, Wei ; Jia, Yangqing ; Sermanet, Pierre ; Reed, Scott E. ; Anguelov, Dragomir ; Erhan, Dumitru ; Vanhoucke, Vincent ; Rabinovich, Andrew: Going Deeper with Convolutions. In: *CoRR* abs/1409.4842 (2014). http://arxiv.org/abs/1409.4842

[21] Riesenhuber, Maximilian ; Poggio, Tomaso: Hierarchical models of object recognition in cortex. In: *Nature Neuroscience* 2 (1999), S. 1019–1025

[22] Simonyan, Karen ; Zisserman, Andrew: Very Deep Convolutional Networks for Large-Scale Image Recognition. In: *CoRR* abs/1409.1556 (2014). http://arxiv.org/abs/1409.1556

[23] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Deep Residual Learning for Image Recognition. In: *CoRR* abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385

[24] Benenson, Rodrigo: *What is the class of this image?* http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html. – abgerufen am 01.04.2017

[25] Ioffe, Sergey ; Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: *CoRR* abs/1502.03167 (2015). http://arxiv.org/abs/1502.03167

[26] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Identity Mappings in Deep Residual Networks. In: *CoRR* abs/1603.05027 (2016). http://arxiv.org/abs/1603.05027

[27] Krizhevsky, Alex ; Nair, Vinod ; Hinton, Geoffrey: *The CIFAR-10 dataset*. http://torch.ch/blog/2016/02/04/resnets.html. – abgerufen am 03.04.2017

[28] Gross, Sam ; Wilber, Michael: *Lessons learned from manually classifying CIFAR-10*. http://torch.ch/blog/2016/02/04/resnets.html. Version: 2016. – abgerufen am 16.03.2017

[29] Zagoruyko, Sergey ; Komodakis, Nikos: Wide Residual Networks. In: *CoRR* abs/1605.07146 (2016). http://arxiv.org/abs/1605.07146

[30] Szegedy, Christian ; Ioffe, Sergey ; Vanhoucke, Vincent: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In: *CoRR* abs/1602.07261 (2016). http://arxiv.org/abs/1602.07261

[31] Szegedy, Christian ; Vanhoucke, Vincent ; Ioffe, Sergey ; Shlens, Jonathon ; Wojna, Zbigniew: Rethinking the Inception Architecture for Computer Vision. In: *CoRR* abs/1512.00567 (2015). http://arxiv.org/abs/1512.00567

[32] Stanley, Kenneth O. ; D'Ambrosio, David B. ; Gauci, Jason: A Hypercube-based Encoding for Evolving Large-scale Neural Networks. In: *Artif. Life* 15 (2009), April, Nr. 2, 185–212. http://dx.doi.org/10.1162/artl.2009.15.2.15202. – DOI 10.1162/artl.2009.15.2.15202. – ISSN 1064–5462

[33] Verbancsics, Phillip ; Harguess, Josh: Generative NeuroEvolution for Deep Learning. In: *CoRR* abs/1312.5355 (2013). http://arxiv.org/abs/1312.5355

[34] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In: *CoRR* abs/1502.01852 (2015). http://arxiv.org/abs/1502.01852

[35] Ren, Shaoqing ; He, Kaiming ; Girshick, Ross ; Sun, Jian: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In: *Advances in Neural Information Processing Systems (NIPS)*, 2015