Bachelorarbeit

Marten Boessenkool

Erkennung von Handgesten mit Faltungsnetzwerken

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marten Boessenkool

Erkennung von Handgesten mit Faltungsnetzwerken

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter: Prof. Dr. Stephan Pareigis

Eingereicht am: 01. Oktober 2014

Marten Boessenkool

Thema der Arbeit
Erkennung von Handgesten mit Faltungsnetzwerken

Stichworte
Faltungsnetze, Neuronale Netze, maschinelles Lernen, Handgesten

Kurzzusammenfassung
In dieser Arbeit werden Faltungsnetze zur Erkennung von Handgesten verwendet. Diese Gesten beschränken sich dabei auf die Anzahl gezeigten Fingern einer Hand. Unter Verwendung unterschiedlicher Netzkonfigurationen soll festgestellt werden, wo die Möglichkeiten und Grenzen des Lernprozesses eines Computers in dieser Frage liegen. Um ein aussagekräftigen Ergebnisses zu erreichen, werden Trainings- und Testdatensätze unter unterschiedlichen Verhältnissen erstellt. Auch werden diese in einigen Experimenten vermischt, um realistischere Bedingungen zu simulieren. Dabei hat sich herausgestellt, dass für eine gute Erkennungsrate alle möglichen Bedingungen wie unterschiedliche Kontraste explizit trainiert werden müssen.
Marten Boessenkool

Title of the paper
Hand gesture recognition with convolutional networks

Keywords
Convolutional Networks, CNN, neural networks, machine learning, hand gestures

Abstract
In this thesis, convolutional networks are used for hand gesture recognition. These gestures are limited to the amount of fingers shown with one hand. Possibilities and limits of the computer learning process are determined using different network configurations. Training and test data are recorded in various simplified conditions and enhanced by mixed experiments to simulate realistic conditions. It has been exposed that every single possibility like different contrast has to be trained explicitely to get good detection rates.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Alternativen, Interaktivität und Mobilität sind Stichworte, die im Zusammenhang mit Zukunftstechnologien immer wieder auftauchen. Sei es privat oder in der Industrie, überall gibt es technologische Errungenschaften, die uns bei unseren täglichen Arbeiten helfen. Dies funktioniert natürlich nicht ohne Interaktion. Schließlich können Maschinen noch keine Gedanken lesen, auch wenn dies in Zukunft vielleicht irgendwann möglich sein wird.
Maus und Tastatur zur Interaktion kennt sicherlich jeder vom heimischen Computer. Bei Tablets und Smartphones haben sich Touchscreens etabliert. In der Industrie sind Schalter, Knöpfe und Sensoren weit verbreitet.

Doch wie steuert man jetzt zum Beispiel eine große Industrieanlage? Fest verbaute Steuerungspulte sind für große Anlagen, wo an vielen Stellen gearbeitet wird, nicht flexibel genug. Fernbedienungen gehen schnell verloren, sind anfällig für mechanische Defekte und bedingt durch ihren Batterieverbrauch keine gute Alternative. Smartphones oder Tablets sind mit einer passenden Steuerungsapp wegen ihrer weiten Verbreitung und ihre Mobilität durch Funkverbindungen wie WLAN und Bluetooth prinzipiell geeignet für solche Aufgaben. Aufgrund ihres hohen Stromverbrauchs, die geringen Akkukapazitäten und die unterschiedlichen Betriebssysteme sind sie für diese Aufgabe dennoch keine gute Lösung. Einen Ansatz für gute Lösung, welche die Anforderungen an Mobilität und Verfügbarkeit erfüllt, bietet hier die Unterhaltungsindustrie: Die Verwendung unseres Körpers zur Interaktion mit Industrieanlagen. Zur Steuerung reicht dabei eine fest montierte Kamera, welche den Arbeitsbereich abdeckt, sowie einen daran angeschlossen Computer, der die Bilder auswertet und die Steuerungsbefehle interpretiert.

Da unser Körper komplex ist und Arbeiter eine Anlage auch ohne aufwendige Ausbildung steuern sollen, ist eine Begrenzung auf die Hand sinnvoll. Da sich jeder Mensch individuell,

aber dennoch ähnlich bewegt, ist eine Anlernung mittels maschinellen Lernens sinnvoll, um flexibel auf neue Personen zu reagieren.

1.2. Zielsetzung

Ziel dieser Bachelorarbeit ist es zu ermitteln, ob sich Faltungsnetze zum Erkennen von Handgesten eignen. Dazu soll mit unterschiedlichen Netzkonfigurationen und Umgebungen getestet werden, ob ein möglichst aussagekräftiges Ergebnis erzielt werden kann. Das Netz soll dabei so effizient arbeiten, dass auch auf schwache Hardware, wie z. B. ein System-on-a-Chip (SoC) eine gute Performance erzielt wird. Primär geht es um die Erkennung von einfachen Gesten wie die Anzahl der gezeigten Finger.

1.3. Gliederung

Im zweiten Kapitel dieser Arbeit werden die Grundlagen von neuronalen Netzwerken und die Besonderheit von Faltungsnetzen vorgestellt. Die verwendete Programmiersprache und dessen Vorteile für diese Arbeit werden im dritten Kapitel angesprochen. Außerdem wird hier auf Probleme bei der Erkennung von Handgesten eingegangen, sowie einige Lösungen dazu beschrieben. Anschließend werden im vierten Kapitel die unterschiedlichen verwendeten Netzkonfigurationen analysiert und verglichen. Abschließend folgt eine Zusammenfassung und ein Ausblick über mögliche Anwendungsfelder und weitere Forschungen im Bereich Handgestenerkennung mit Faltungsnetzen.

2. Theoretische Grundlagen

2.1. Bisherige Arbeiten

Computer Vision ist ein aktiver Forschungsbereich, welche sich mit der Erkennung von Objekten, Formen und Schriftzeichen in Bildern befasst. Auch die Erkennung von Gesten zur Interaktion mit Robotern und Maschinen gehört dazu. Viele verschiedene Ansätze sind dazu zu finden. So beschreiben z. B. Platt und Nowlan (1995) die Unterscheidung von geschlossener und offener Hand in einem kontinuierlichen Bildstrom. Konda u. a. (2012) zeigen hingegen Gesten zur Steuerung eines Roboters. Einen etwas anderen Weg gehen Kim u. a. (2008), welche die Erkennung von bewegten Gesten vorstellen.

Zwei Arbeiten, welche sich ebenfalls mit der Erkennung der Anzahl gezeigten Fingern befassen sind Nagi u. a. (2011) und Nagi u. a. (2012), wobei letzteres auf die Erste aufbaut. Beide Experimente verwenden sogenannte Footbots zur Aufnahme der Bilder, das sind kleine mobile Roboter. Letzteres nutzt eine Kombination aus mehreren solchen Footbots zur Bildaufnahme aus mehreren Winkeln. Damit die Hand in den Farbbildern erkannt werden kann, wird dazu ein farbiger Handschuh getragen.

Der Ansatz setzt eine gewisse Vorverarbeitung des Inputs voraus, welche Rechenzeit kostet. Der Versuch in dieser Arbeit soll ohne große Vorverarbeitung auskommen. Auch werden die Bilder im Graustufen-Modus aufgenommen. Zwar beschreiben Nagi u. a. (2011), dass man das Farbmodell des Handschuhs durch Farbmodelle von Hautfarben ersetzen kann, allerdings hilft dies bei einer industriellen Anwendung nicht. Dort werden aus Sicherheitsgründen meistens Arbeitshandschuhe getragen. Da diese die unterschiedlichsten Farben aufweisen können und die Möglichkeiten durch Verschmutzung noch steigen, ergibt eine Farberkennung keinen Sinn.

2.2. Grundlagen neuronale Netzwerke

Es gibt im Internet unzählige Seiten, die sich mit neuronalen Netzen befassen. Die Seite von Rey und Beck erklärt die Grundlagen sehr anschaulich und umfangreich. Daher bezieht sich dieses Kapitel weitestgehend auf diese Seite.
Neuronale Netze stellen eine abstrakte Struktur des Nervensystems im menschlichen Gehirn dar. In der Wissenschaft und Informatik im Forschungsbereich künstliche Intelligenz versucht man diese mittels künstliche neuronale Netze zu simulieren bzw. nachzubilden. Dabei steht die Abstraktion von Informationsverarbeitung im Vordergrund, nicht die Nachbildung biologischer neuronaler Netze.

Neuronale Netze bestehen aus mehrere vernetzte Neuronen, auch Knoten oder Units genannt. Diese können Informationen von anderen Neuronen oder von der Umwelt aufnehmen und auch weitergeben. Dabei werden drei Arten von Neuronen unterschieden:

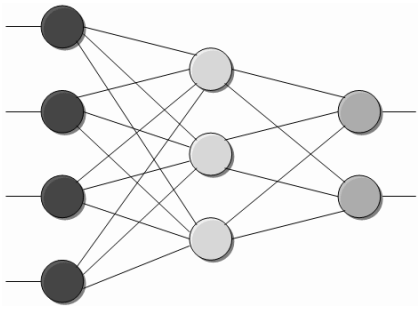


Abbildung 2.1.: Schematische Darstellung eines neuronalen Netzes, Quelle: Rey und Beck

- Links -> Input-Units: Diese Units nehmen Signale von der Umwelt auf. Dabei dienen sie als Eingabe/Input des neuronalen Netzes.
- Mitte -> Hidden-Units: Diese Units befinden sich zwischen den Input-Units und den Output-Units und stellen eine interne Repräsentation der Umwelt dar.
- Rechts -> Output-Units: Diese Units geben Signale an die Umwelt weiter.

Übereinander angeordnete Neuronen werden als Schicht oder Layer bezeichnet. Analog zu den Units werden diese Input-Layer, Hidden-Layer und Output-Layer genannt. Dabei sind Input- und Output-Layer in jedem neuronalen Netz genau einmal vorhanden, während es keine, eine, oder mehrere Hidden-Layer geben kann.

Die Units sind mit Kanten miteinander verbunden. Diese Kanten werden durch ein sogenanntes Gewicht definiert, welche die Stärke der Verbindung repräsentiert. Dabei kann das Gewicht positiv, negativ, oder null sein. Bei einem positiven Gewicht wirkt ein Neuron anregend auf ein verbundenes Neuron. Ein negatives Gewicht wirkt hemmend, während ein Gewicht von null besagt, dass ein Neuron derzeit nicht auf ein verbundenes Neuron wirkt. Diese Kanten lassen sich auch als Matrix darstellen, wodurch sich diese Informationen mit mathematischen Matrizenoperationen relativ einfach bearbeiten lassen.

Über die Kanten bekommt ein Neuron Eingaben, auch Input genannt, von anderen Neuronen. Wie hoch der Input ist, hängt vom Output des sendenden Neurons ab, sowie vom Gewicht der Kante, welche die beiden Neuronen verbindet. Diese Werte werden dabei miteinander multipliziert. Die gesamte Eingabe eines Neurons wird Netzinput genannt. Dieser setzt sich aus der Summe der einzelnen Inputs zusammen. Dies lässt sich als Formel wie folgt darstellen:

Input Unit i: $input_{ij} = a_j w_{ij}$
Netzinput Unit i: $Netzinput_i = \sum_j input_{ij} = \sum_j a_j w_{ij}$

a_j = Aktivitätslevel der sendenden Unit j
w_{ij} = Gewicht zwischen den Units i und j

Der Zusammenhang zwischen Netzinput und Aktivitätslevel eines Neurons wird mittels einer Aktivitätsfunktion, auch Aktivierungsfunktion oder Transferfunktion genannt, dargestellt. Diese Funktion bekommt den Netzinput als Eingabe und hat den Aktivitätslevel als Ausgabewert. Der Aktivitätslevel wird dann mittels einer Ausgabefunktion in den Output transferiert. Hierzu wird normalerweise die Identitätsfunktion verwendet, wodurch der Output dem Aktivitätslevel entspricht.
Man unterscheidet zwischen folgende Aktivitätsfunktionen:

- lineare Aktivitätsfunktion
- lineare Aktivitätsfunktion mit Schwelle: Erst nach Erreichen einer Schwelle wird der Zusammenhang zwischen Netzinput und Aktivitätslevel linear. Dies eignet sich z.B. um Rauschen zu eliminieren.
- binäre Aktivitätsfunktion
- sigmoide Aktivitätsfunktion: Diese Funktion wird bei der Simulation von kognitiven Prozessen verwendet. Man unterscheidet hier die logistische Funktion (Wertebereich 0

bis 1) und die Tangens-Hyperbolicus-Funktion (Wertebereich -1 bis 1). Beide Funktionen weisen ein ähnliches Verhalten auf. Bei einem niedrigen Netzinput steigt der Aktivitätslevel zunächst langsam an, steigt dann fast linear an, um sich bei einem hohen Netzinput der Obergrenze asymptotisch anzunähern. Durch den begrenzten Wertebereich wird das Überschwappen von Fehlern verhindert. Außerdem ist eine solche Funktion an allen Stellen differenzierbar, was für einige Lernregeln Voraussetzung ist.

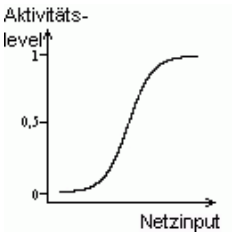


Abbildung 2.2.: Sigmoide Aktivitätsfunktion, Quelle: Rey und Beck

Damit ein neuronales Netz Aufgaben erledigen kann, muss dieses erst lernen bzw. trainiert werden. Das Lernen wird dabei durch die Veränderung der Kantengewichte definiert. Wie genau sich die Gewichte verändern, hängt von der verwendeten Lernregel ab. Man unterscheidet im Groben zwischen überwachtes Lernen (supervised learning), unüberwachtes Lernen (unsupervised learning) und bestärkendes Lernen (reinforcement learning). Beim unüberwachten Lernen klassifiziert das Netz den Input selbstständig anhand von Ähnlichkeiten der Eingabemuster. Beim bestärkenden Lernen wird anhand einer Belohnung - diese kann sowohl positiv, als auch negativ sein - gelernt. Dies wird häufig bei autonom handelnden Robotern verwendet, da hier eine vollständige Modellierung der Umgebung oftmals nicht möglich ist. Beim überwachten Lernen, welches auch in diesem Projekt verwendet wird, wird in der Lernphase zu jedem Input die gewünschte Ausgabe mitgegeben. Anhand dieser Kombination werden die Kantengewichte so angepasst, dass am Ende im Idealfall das Netz zu jeder Eingabe die korrekte Ausgabe ermittelt.

Auf die Trainingsphase folgt die Testphase, in welcher der Lernfortschritt des neuronalen Netzes geprüft wird. Dabei kann man entweder die Trainingsdaten oder Teile davon als Eingabe verwenden. Besser ist es aber neue Eingabedaten zu verwenden, um zu kontrollieren, wie gut diese erkannt werden. Auch hierzu wird wieder die gewünschte Ausgabe mitgeteilt, anhand dessen man anschließend die Fehlerrate ermitteln kann. Je kleiner die Fehlerrate, umso besser hat das neuronale Netz gelernt.

Ein häufig verwendetes Lernverfahren ist Backpropagation mit dem Gradientenabstiegsverfahren. Dieses basiert darauf, dass es für jedes mögliche Kantengewicht eine Fehlergröße gibt. Ziel ist es, diese zu minimieren. Dazu wird ein Netz mit zufälligen Gewichten initialisiert. Dann wird an dieser Stelle der Gradient berechnet, welche die Richtung des steilsten Abstiegs angibt. Anhand der Lernrate und des Gradienten wird die Änderung an den Gewichten bestimmt. Bei einem sehr steilen Abstieg erfolgen größere Änderungen als bei weniger steilen Abstiegen. Abbildung 2.3 zeigt zwei vereinfachte Beispiele. E ist die Fehlergröße und w_{ij} ein Kantengewicht. Mit mehreren Gewichten im n-dimensionalen Raum ergibt dies eine Hyperebene, welche sich allerdings zur Veranschaulichung nicht so gut eignet.

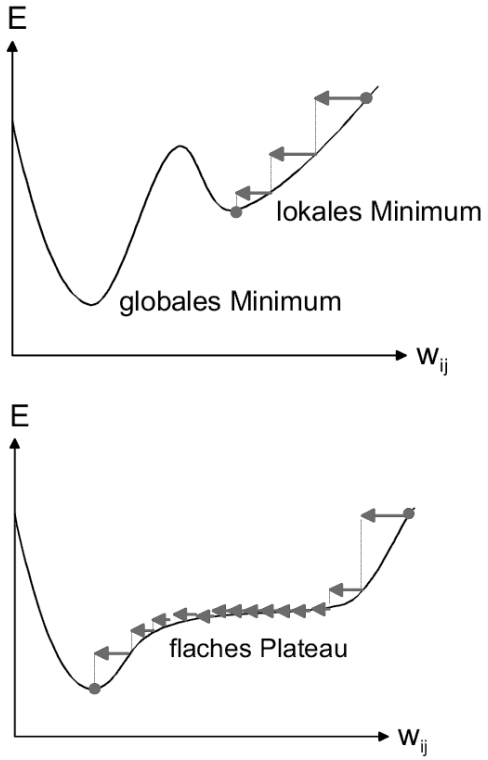


Abbildung 2.3.: Gradientenabstiegsverfahren, Quelle: Meisel (2012)

Dieses Verfahren stellt sicher, dass ein Minimum gefunden wird. Ob es sich hierbei um ein lokales oder ein globales Minimum handelt, lässt sich nicht feststellen. Dieses würde voraussetzen, dass alle nahezu unendlich viele Möglichkeiten ausprobiert werden müssten.
Zwar braucht das Gradientenabstiegsverfahren nur seine lokale Position zu kennen, nicht die ganze Hyperebene, das bringt allerdings einige Probleme mit sich, wie in Abbildung 2.4 zu sehen ist. Zum Einen kann es in einem Tal mit zwei steilen Seiten zur Oszillation kommen. Das

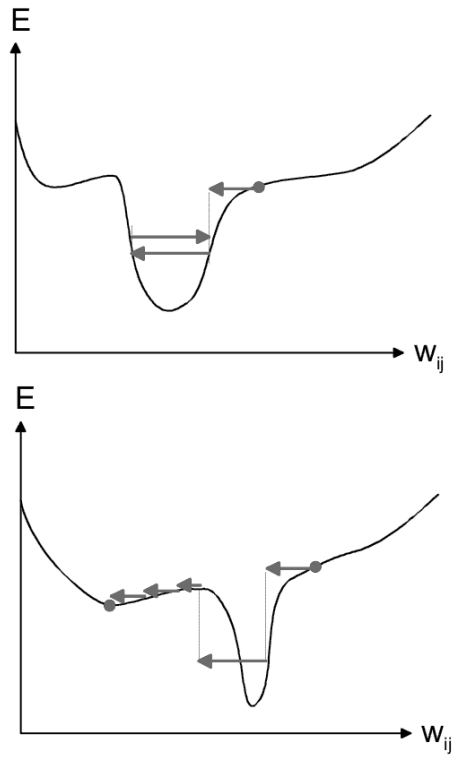


Abbildung 2.4.: Probleme beim Gradientenabstiegsverfahren, Quelle: Meisel (2012)

heißt, dass von einer Seite zur anderen Seite hin und her gesprungen wird, ohne dass jemals eine bessere Position erreicht wird. Dieses Problem lässt sich durch modifizierte Verfahren leicht umgehen, wie z. B. mit dem Momentum-Term. Dieser sorgt dafür, dass an flachen Stellen langsam größere Schritte verwendet werden, damit diese relativ schnell übersprungen werden können. Bei einem Richtungswechsel wird die Schrittweite reduziert, wodurch das Problem der Oszillation nicht auftreten kann.

Ein weiteres Problem ist das Überspringen von schmalen, tiefen Minima. Hierzu gibt es keine universelle Lösung. Lediglich ein Verringern der Lernrate kann dieses Problem eindämmen, aber nicht vollständig beheben. Dieses führt außerdem dazu, dass langsamer gelernt wird. Somit würde eine sehr kleine Lernrate zwar zu einer hohen Wahrscheinlichkeit führen solche Minima zu finden, allerdings auf Kosten der Lerngeschwindigkeit.

2.3. Besonderheiten von Faltungsnetzwerken

Faltungsnetzwerke sind eine Kombination von neuronalen Netzen und Faltungsmasken, welche sich gut zur Erkennung von Mustern in Bildern eignen.

2.3.1. Faltungsmasken

Faltungsmasken sind Filter, welche in der digitalen Bildverarbeitung verwendet werden. Sie werden auch Faltungskern, Faltungsmatrix, Filterkern, oder im Englischen convolutional kernel genannt. Hierbei handelt es sich meistens um quadratische Matrizen mit ungerader Dimension, welche verschiedene Funktionen übernehmen können. Gängig sind hier Weichzeichner, Filter zur Bildschärfung und Filter zur Kantendetektion.

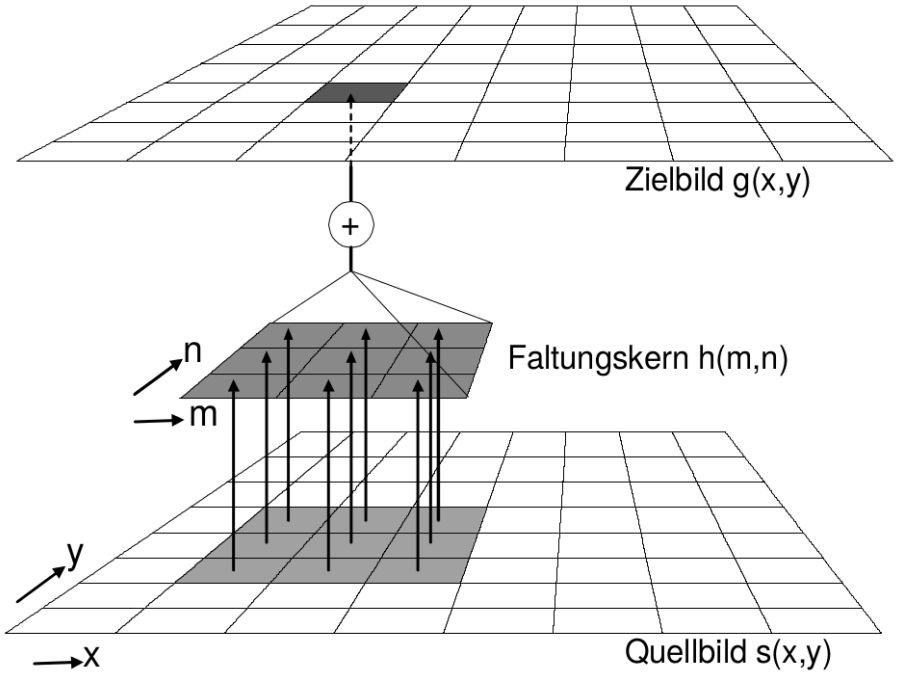


Abbildung 2.5.: Anwendungsbeispiel einer Faltungsmaske, Quelle:Meisel (2012)

Abbildung 2.5 zeigt die Anwendung einer Faltungsmaske. Diese wird im Quellbild an einer bestimmtem Stelle gelegt. Dort werden die Werte der Faltungsmaske mit den Farbwerten der darunter liegenden Pixel multipliziert und anschließend addiert. Als Formel lässt sich dies wie folgt darstellen:

$$g(x,y) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} h(m,n) * s(x-m, y-n)$$

In einigen Fällen wird das Ergebnis noch mit einem Faktor multipliziert, um die Werte innerhalb des normalen Farbbereichs zu halten. Dies erfolgt z. B. bei einem Weichzeichner. Im einfachsten Fall sind dort alle Werte der Maske mit eins belegt, was einer Addition aller Farbwerte bedeutet. Dieser wird durch die Größe der Faltungsmaske dividiert, was eine Mittelwertbildung benachbarter Pixel entspricht.

2.3.2. Funktionsweise von Faltungsnetzwerken

Wie normale neuronale Netze, bestehen auch Faltungsnetze aus mehreren Layern, welche unterschiedliche Aufgaben übernehmen.

- C-Layer: Dies sind die Faltungslayer, in der die Faltungsmasken auf die Eingabe angewendet werden. Das C steht dabei für convolution.
- S-Layer: In den Subsampling-Layern wird die Auflösung des Inputs herunter gerechnet. Dies dient dazu das Faltungsnetz stabiler gegenüber kleinere Störungen zu machen.
- F-Layer: Die Fully-Connected-Layer sind vollständig verknüpfte Layer, welche unmittelbar vor dem Output verwendet werden. Sie übernehmen die Aufgabe als linearer Klassifikator.

Die genaue Funktionsweise lässt sich anhand eines Papers von LeCun u. a. (2010) gut erklärt. Dieses erklärt anschaulich, wie sich Faltungsnetze in Stufen unterteilen lassen, sowie die Aufgaben in diesen Stufen. Aus diesem Paper stammt auch die Abbildung 2.6, welches den allgemeinen Aufbau eines Faltungsnetzes darstellt.
Faltungsnetze sind Multi-Layer Netze, bei der jeder Layer aus mehreren Ebenen besteht. Der Output dieser Ebenen werden auch Feature-Maps genannt. Ihre Aufgabe ist es mittels lokale rezeptive Felder sogenannte Features in mehreren Stufen zu extrahieren. Das können im Falle von Bildern z. B. Kanten, Endpunkte oder Ecken sein. Dabei werden diese in nachfolgende Layer kombiniert, um Features höherer Ordnung der Vorgängerlayer zu erkennen.
Alle Units einer Feature-Map teilen sich die Gewichte. Dadurch werden die Units einer Map dazu gezwungen dieses Feature im ganzen Bild zu erkennen. Dies führt dazu, dass Faltungsnetze zu einem bestimmten Grad Positions- und Translationsinvariant sind. Sobald das Netz ein Feature gelernt hat, ist die genaue Position im Bild nicht mehr so wichtig.
Man kann Faltungsnetze in mehrere Stufen unterteilen. Jede dieser Stufen besteht dabei aus

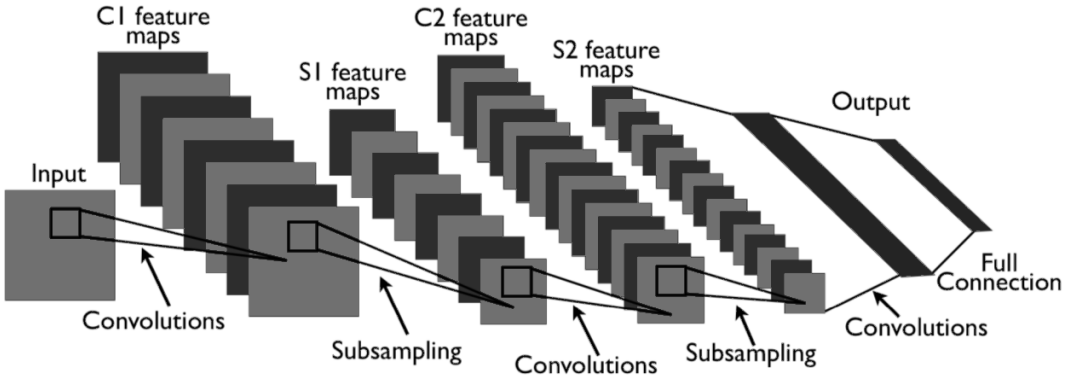


Abbildung 2.6.: Allgemeine Struktur eines Faltungsnetzes, Quelle: LeCun u. a. (2010)

drei Layern: Ein Filter Bank Layer, ein Non-Linearity Layer und ein Feature Pooling Layer. Gängigen Implementierungen bestehen aus einen, zwei, oder drei solcher Stufen.

Ein **Filter Bank Layer** bekommt ein 3D Array als Input. Die Größe entspricht die Anzahl der 2D Feature-Maps mit einer Auflösung von $in_1 \times in_2$, wobei sich jedes Element mit ijk adressieren lässt. Die Ausgabe entspricht ebenfalls ein 3D Array mit einer $out_1 \times out_2$ Auflösung der Feature-Maps. Diese ergibt sich aus der Anwendung einer $f_1 \times f_2$ Faltungsmaske und der Addition eines trainierbaren Bias-Wertes. Als Formel lässt sich dies wie folgt darstellen: $y_j = b_j + \sum_i k_{ij} * x_i$. Dabei entspricht x_i der Input einer Feature-Map, k_{ij} die trainierbare Faltungsmaske und $*$ der Faltungsoperator.

Auf einem Filter Bank Layer folgt ein **Non-Linearity Layer**. In herkömmlichen Faltungsnetzen besteht dieser lediglich aus einer punktweisen $tanh()$ Sigmoidfunktion, welche auf jedes Element ijk angewendet wird. In neueren Implementierungen werden dazu auch komplexere Funktionen verwendet, wie z. B. die bereinigte Sigmoidfunktion. Auf diese folgt manchmal noch eine Kontrastnormalisierung, um die Feature-Erkennung zu verbessern.

Nachfolgend wird ein **Feature Pooling Layer** verwendet. Dieser entspricht das Subsampling. Dazu werden die Werte benachbarter Einheiten addiert und mit einem Koeffizienten multipliziert. Anschließend wird ein Bias dazu addiert. Durch diese Verringerung der Auflösung werden die Positionen der Features weniger wichtig, wodurch das Netz robuster wird. In herkömmlichen Netzen wird abschließend eine punktweise $tanh()$ Sigmoidfunktion auf alle Elemente angewendet. In neueren Implementierungen wird diese teilweise auch weggelassen, oder gar ganz auf diesen Layer verzichtet. In diesem Fall wird das Subsampling schon im Filter

Bank Layer durchgeführt, indem für die Abtastrate Schrittweiten größer als eins verwendet werden.

Um diese Funktionsweise weiter zu verdeutlichen, eignet sich der Artikel von LeCun u. a. (1998) sehr gut. Dort wird zum Einen auf Faltungsnetze allgemein eingegangen und zum Anderen die spezielle Implementierung von "LeNet5" beschrieben. Dieses wurde in diesem Beispiel für die Erkennung von handschriftlich geschriebenen Zahlen verwendet.

Der Filter Bank Layer und der Non-Linearity Layer entsprechen dem Faltungslayer in LeNet5. Da als nicht-Linearitätsfunktion nur eine einfache $tanh()$ Sigmoidfunktion verwendet wird, werden diese beiden Layer als einen implementiert.
Der erste Faltungslayer besitzt sechs Feature-Maps mit einer Eingabe von 32*32 Pixeln, einer Faltungsmaskengröße von 5*5 und einer Ausgabe von 28*28 Pixeln. Da jeder Wert der Faltungsmasken ein trainierbarer Parameter entspricht, sind das in diesem Layer 6*5*5=150 für die Masken. Hinzu kommt ein trainierbarer Bias für jede Feature-Map, was einer gesamten Anzahl von 156 trainierbare Parameter ergibt. Aufgrund der gemeinsam genutzten Gewichten sind die Anzahl Verbindungen wesentlich höher. Diese ergeben sich aus der Größe der Faltungsmasken, multipliziert mit der Anzahl Feature-Maps, multipliziert mit der Auflösung des Outputs. Hinzu kommen die Verbindungen jedes Outputpixels mit dem Bias-Wert. Das entsprechen in diesem Layer 122 304 Verbindungen.

Der Feature Pooling Layer mit dem Subsampling, gefolgt von der punktweisen $tanh()$ Sigmoidfunktion, ist genauso in LeNet5 zu finden.
Layer zwei in LeNet5 ist so ein Subsampling-Layer. Die Anzahl trainierbaren Parameter ergeben sich aus jeweils einen Bias und einen Koeffizienten pro Feature-Map. Im Beispiel entsprechen das 12 trainierbare Parameter. Die Anzahl der Verbindungen ergeben sich aus der Subsampingrate, multipliziert mit der Anzahl Feature-Maps, multipliziert mit der Auflösung des Outputs. Auch hier kommt eine Verbindung jedes Outputpixels mit dem Bias hinzu. Für diesen Layer summiert sich das auf 5880 Verbindungen.

Der dritte Layer in LeNet5 ist wieder ein Faltungslayer. Dieser besitzt sechs Feature-Maps als Input, sowie 16 als Output. Es werden 60 Faltungskerne mit einer Größe von 5*5 verwendet. Es gibt dabei zwei Gründe, wieso dieser Layer nicht vollständig verknüpft ist. Zum Einen verringert das die Anzahl Verbindungen und trainierbare Parameter. Zum Anderen bricht dies die Symmetrie des Netzes, wodurch jede Feature-Map unterschiedliche Features lernen kann,

da nicht jede Feature-Map den gleichen Input bekommt.

Trotz vollständiger Verknüpfung gilt der fünfte Layer mit 120 Feature-Maps als Output als ein Faltungslayer. Die Masken sind genauso groß wie der Input aus dem vorherigen S-Layer, was in einem 3D Array von n*1*1 als Output resultiert, wobei n die Anzahl Feature-Maps entspricht. In diesem spezifischen Beispiel wurde ein weiterer vollständig verknüpfter Layer mit 84 Units als Output verwendet, welche in der Standardimplementation von LeNet5 so nicht zu finden ist. Diese besitzt die gleiche Anzahl Layer wie in Abbildung 2.6 dargestellt.

3. Implementierung

3.1. Programmiersprache Lush

Lush steht für **L**isp **U**niversal **SH**ell. Hierbei handelt es sich um eine objekt-orientierte Programmiersprache mit einer einfach zu erlernenden, Lisp-ähnlichen Syntax. Lush läuft auf GNU/Linux, Solaris, Irix, Windows unter Cygwin und auf MacOS unter MacPorts (ehem. DarwinPorts) oder Fink.

Die Entwicklung begann in 1987 durch Yann LeCun und Leon Bottou, damals noch unter dem Namen SN als Front-end eines Neuronale Netze Simulators. Im Laufe der Zeit wurde SN immer weiter entwickelt unter Beteiligung weiterer Nutzer. 2001 enstand Attlush aus dem SN3.2 Compiler und dem TL3 Interpreter, eine Weiterentwicklung aus einem SN fork. 2002 wurde der Attlush Code neu geschrieben und als Lush unter der GPL [1] veröffentlicht. Außerdem wurde eine etwa 650 Seiten umfassende Dokumentation [2] dazu angefertigt, welche auf der Lush Homepage in verschiedene Formate zum Download bereit steht.

Entwickelt wurde Lush um die Vorteile einer flexiblen, schwach typisierten, interpretierten Sprache mit der Effizienz einer stark typisierten, nativ kompilierten Sprache zu verbinden. Der Lush Compiler erzeugt aus dem Lush-Code sehr effizienten C-Code, welche dann mit dem besten auf dem System verfügbaren C-Compiler kompiliert wird.

Ein besonderes Feature von Lush ist die Möglichkeit, C-Code in Lush zu verwenden. Dies ist sogar innerhalb von einzelnen Funktionen möglich. Es gibt Fälle, wo es unpraktisch oder ineffizient ist eine bestimmte Funktion oder Teile einer Funktion in Lush zu schreiben. In diesen Fällen kann die direkte Verwendung von C-Code zu einer einfacheren und kompakteren Implementierung sowie zu einer signifikanten Beschleunigung der Berechnungen führen.

[1] GNU General Public License, `http://www.gnu.org/licenses/licenses.html`
[2] `http://lush.sourceforge.net/doc.html`

Lush besitzt etwa 1000 in C geschriebene Funktionen, welche beim Start definiert werden. Dazu bestehen etwa 1300 Funktionen in der in Lush geschriebenen Bibliothek, sowie etwa 12000 Funktionen in den mitinstallierten Paketen im ”Packages” Verzeichnis von Lush. Außerdem bietet Lush eine große Anzahl Schnittstellen zu externen Bibliotheken. Hierzu gehören die wissenschaftlichen Bibliotheken GSL (GNU Scientific Library), LAPACK (Linear Algebra PACKage), BLAS (Basic Linear Algebra Subprograms), die graphischen Bibilotheken OpenGL, GLU (OpenGL Utility Library), GLUT (OpenGL Utility Toolkit), die Multi-Media Bibliothek SDL (Simple DirectMedia Layer), die Sound-Schnittstelle ALSA (Advanced Linux Sound Architecture) und die Video-Schnittstelle V4L (Video4Linux).

3.2. Gründe für die Verwendung von Lush

Seine Stärken spielt Lush in der Verwendung bei umfangreichen numerischen und graphischen Anwendungen aus. Als Anwendungsgebiete nennt die Lush Homepage Forschung und Entwicklung in den Bereichen Signalverarbeitung, Bildverarbeitung, maschinelles Lernen, Computer Vision, Bioinformatik, Data-Mining, Statistiken, Simulationen, Optimierungen und künstliche Intelligenz. Dank seiner mächtigen Engine für Matrix-, Vektor- und Tensor-Operationen, sowie dank des speziell auf maschinelles Lernen ausgelegten Paketes ”gblearn2” ist Lush geradezu prädestiniert für eine Aufgabe wie die Handgestenerkennung. Mittels der V4L Schnittstelle lassen sich die benötigten Bilddaten direkt in Lush ohne Umwege mit einer von V4L unterstützten Kamera erfassen. Auch die anschließende Aufbereitung ist problemlos und schnell in Lush möglich.

In dem ”gblearn2” Paket ist eine komplette Implementierung des Faltungsnetzes ”LeNet5” enthalten. Dieses wurde schon für die Erkennung von handschriftlich geschriebene Zahlen aus der MNIST Datenbank verwendet und hat dort geringe Fehlerraten erzielt. Für die Erkennung von Handgesten kann dieses direkt verwendet und bei Bedarf angepasst werden. Es entfällt somit eine komplette Neuimplementierung eines Faltungsnetzes.

3.3. Probleme bei der Erkennung

Das Hauptproblem bei der Erkennung der Handgesten besteht darin, dass es unmöglich ist, eine 100-prozentige Erkennung zu erreichen. Hierfür gibt es mehrere Gründe, von denen einige anhand der Abbildung 3.1 vorgestellt werden.
Im Beispiel 3.1a ist die Interpretation der Anzahl gezeigter Finger problematisch: sowohl drei

(a) (b) (c)

Abbildung 3.1.: Interpretationsprobleme bei der Erkennung 1

oder vier wären hier als Deutung möglich und zulässig. Beispiel 3.1b ist noch problematischer, da in diesem Bild sowohl null, eins, als auch zwei Finger als Deutung möglich sind. Daumen und Ringfinger können beide sowohl als gezeigt, als auch als nicht gezeigt interpretiert werden. Zusätzlich dazu sind die sehr großen Datenmengen, die zum Trainieren und Testen des Netzes verwendet werden ein Problem. So kann es sein, dass man sich bei der Klassifikation mal für die Eine und mal für die andere Interpretation entscheidet. Auch kann es vorkommen, dass bei einer Interpretation von vier Fingern an einer Stelle ein Finger weniger groß oder weniger ausgestreckt zu sehen ist, als bei einer Interpretation von drei Fingern an anderer Stelle. Dies könnte man dadurch versuchen zu verhindern, dass man alle interpretationsfähigen Bilder sammelt und durch genaues Nachmessen eine konsistente Interpretation erreicht. Bei den Zehntausend und mehr zu erwartenden Bildern in den Trainings- und Testdatensätzen würde dies ein enormer Zeitaufwand mit sich bringen. Da über alle Daten gesehen nur relativ wenige solche Interpretationsprobleme zu erwarten sind, wird sich dieser Aufwand nicht lohnen und für diese Arbeit nicht betrieben.

Ein weiteres Problem ist das Fehlen der Intuitivität eines Computers. Ein Beispiel dazu ist in Abbildung 3.1c zu finden. Ein Mensch erkennt hier sofort intuitiv, dass zwei Finger gezeigt werden, auch wenn diese leicht übereinander liegen. Für einen Computer ist diese Situation deutlich schwieriger zu erkennen. Ein möglicher Ansatz ist eine Kantenerkennung. Diese funktioniert nur mit scharfen Bildern unter idealer Beleuchtung gut. Eine andere Möglichkeit ist die Ermittlung der relativen Größe der gezeigten Finger. Hier setzen perspektivische Verzerrungen allerdings schnell Grenzen. Wie beim Interpretationsproblem wird auch dieses Problem relativ selten auftreten, weswegen auch dafür kein extra Aufwand betrieben wird.

(a) (b) (c)

Abbildung 3.2.: Interpretationsprobleme bei der Erkennung 2

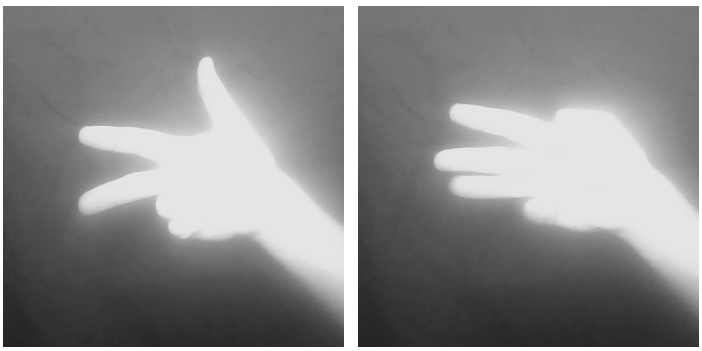
Auch die verwendete Hardware und deren Performance kann ein Problem darstellen. Je nach Aufnahme- und Verarbeitungsgeschwindigkeit der Kamera sowie Bewegungsgeschwindigkeit der Hand kann es zu Unschärfe und Verwischungen kommen. Zwei Beispiele dazu sind in Abbildung 3.2a und 3.2b zu sehen. In 3.2a wird der kleine Finger weitestgehend horizontal zur Kamera bewegt. Dadurch lässt sich dieser trotz Unschärfe noch als Finger erkennen. Der Daumen in 3.2b wird vertikal zur Kamera bewegt. Die Unschärfe ist an dieser Stelle so groß, dass sich dieser nicht mehr deutlich als Finger identifizieren lässt. Beide Fälle sind lediglich eine Momentaufnahme, weswegen diese Probleme nur bei Einzelbildern auftreten. In einem Bildstrom werden diese Probleme sich von allein eliminieren, sofern sich die Hand oder Kamera nicht ständig sehr stark bewegt.

Die Verdeckung ist auch ein Problem. Hierbei sind noch zwei Arten zu unterscheiden: Die perspektivische Verdeckung, sowie das Verdecken durch andere Objekte. Bei der perspektivischen Verdeckung werden einige Finger durch die relative Position und Drehung der Hand verdeckt. Ein Beispiel hierzu ist in Abbildung 3.2c zu sehen. Da alle Finger einer Hand unterschiedlich groß sind, ließe sich hierfür mithilfe von mathematischen Modellen zumindest eine Teillösung realisieren. Diese würde allerdings viel Aufwand bedeuten, ohne dass eine vollständige Lösung sichergestellt werden kann.
Die andere Art von Verdeckung ist die durch Objekte. Dies können sich bewegende Objekte sein, wie z. B. eine andere Person oder sich bewegende Maschinenteile. Das Problem bei dieser Art der Verdeckung löst sich selbstständig durch die Bewegung. Bei statischen Objekten, wie z. B. eine Wand oder eine Säule, müsste sich die Hand aus dem verdeckten Bereich bewegen. Dies ist allerdings nicht immer möglich. Eine mögliche Lösung, welche auch bei der perspektivische Verdeckung funktioniert, ist auf Hardwareebene zu finden: Die Verwendung von

mehreren Kameras. Dazu werden die Bilddaten aller Kameras ausgewertet. Wenn eine Kamera keine Hand oder eine Verdeckung erkennt, wird die Auswertung dieser Kamera ignoriert. Für den Fall, dass mehrere Kameras unterschiedliche Ergebnisse liefern, kann man einen Algorithmus entwickeln, welche die plausibelste Möglichkeit ermittelt. Im schlimmsten Fall wird das Ergebnis verworfen und keine Geste erkannt.

Des Weiteren könnten Bewegungen im Hintergrund ein Problem darstellen. Objekte könnten z. B. eine Form haben, die die Computer als Hand identifiziert. Oder durch eine ungünstige Farbkonstellation könnte die Hand teilweise oder sogar vollständig mit dem Hintergrund verschmelzen. Eine Lösung könnte auch daraus bestehen mehrere Kameras zu verwenden. Ob und wie gut das klappt müsste in einer praktischen Versuchsanlage getestet werden, welche für diese Arbeit nicht zur Verfügung steht. Deswegen wird in dieser Arbeit nur festgestellt, dass Bewegungen im Hintergrund problematisch sein können.

Weitere, nicht zu unterschätzende Probleme entstehen aus Beleuchtung und Kontrast. Bei vielen starken Kontrasten im Bild kann es vorkommen, dass die Hand nicht mehr, oder nur ganz schwierig, zu erkennen ist. Ebenso kann die Hand bei geringen Kontrasten mit dem Hintergrund nahezu verschmelzen, wodurch die Erkennung erheblich erschwert wird und im schlimmsten Fall sogar unmöglich ist. Sofern das gesamte Bild kontrastarm ist, lässt sich mittels Histogrammausgleich in der Vorverarbeitung der Kontrast erhöhen. Bei einem punktuell geringen Kontrast wird auch diese Methode an seine Grenzen stoßen. Außerdem benötigt dies mehr Leistung, weshalb eine Implementation nur bei ausreichend leistungsfähiger Hardware Sinn macht.



(a) (b)

Abbildung 3.3.: Interpretationsprobleme bei der Erkennung 3

Bei einer ungünstigen Beleuchtung kann es z. B. zu einer Überbeleuchtung der Hand kommen, wodurch diese von der Kamera unscharf wahrgenommen wird. Beispiele hierzu sind in 3.3 zu finden. Eine zu starke Überbeleuchtung kann dazu führen, dass die Hand nicht mehr als solche erkannt wird, sondern nur noch als heller Fleck im Bild auftaucht. Weitere Probleme kann es in Kombination mit einer schlecht gewählten Kameraposition geben. Da Kameralinsen oftmals gekrümmt sind, können sogenannte Halo-artige Effekte[3] auftreten, welche die Hand verdecken können.

Trotz aller möglichen Probleme bei der Erkennung macht es Sinn eine solche zu testen und zu bewerten, wie es Zweck dieser Bachelorarbeit ist. Viele der Probleme sind auf ein einzelnes Bild begrenzt. In einem kontinuierlichen Bildstrom tauchen diese nicht auf, bzw. fallen dort kaum bis gar nicht ins Gewicht. Die Interpretationsprobleme z. B. sind in den meisten Fällen darauf zurück zu führen, dass die Finger gerade während der Aufnahme eines Bildes bewegt wurden, um eine andere Geste oder eine andere Anzahl Finger zu zeigen. Die Probleme der Verdeckung und der fehlenden Intuition bei Computern lassen sich meistens durch kleine Bewegungen der Hand beheben. Ebenso ist die Unschärfe meistens nur eine Momentaufnahme.
Die größten tatsächlichen Probleme dürften die der Beleuchtung und des Kontrastes sein. Diese lassen sich in den meisten Fällen schon durch eine vorherige Abstimmung von Lichtquellen und Kameraposition minimieren, oder gar eliminieren.

[3] Halo Lichteffekte, `http://www.meteoros.de/themen/halos/`

4. Analyse der Netzkonfigurationen

In diesem Kapitel werden die einzelnen Experimente mit unterschiedlichen Eingabedaten und Netzkonfigurationen analysiert und anschließend verglichen. Bei den Eingabedaten wird mit einfachen Daten angefangen. Mit jedem Experiment soll die Komplexität, also die Erkennungsschwierigkeit, steigen. So ist beispielsweise eine deutliche sichtbare Hand vor einem Hintergrund mit starkem Kontrast einfacher zu erkennen, als eine Hand in einem verschwommenen Bild mit einem geringen Kontrast zum Hintergrund. Außerdem soll so ermittelt werden, ob ab einer bestimmten Komplexität die Erkennungsrate deutlich sinkt oder möglicherweise von der genauen Eingabe abhängig ist. So wäre vorstellbar, dass von zwei Handkonfigurationen mit vergleichbarer Komplexität eine gut erkannt wird, während die andere große Probleme bereitet.

Zu den verschiedenen einstellbaren Parametern, die Metaparameter, gehören die Größen der Faltungsmasken, die Subsamplingraten, die Dimension des letzen Faltungslayers und die Lernraten. Diese werden anfangs aufgrund fehlender Erfahrungswerte nach dem Zufallsprinzip ausprobiert. Im Verlauf werden dann zuerst solche Metaparameter verwendet, welche sich in vorherigen Schritten als gut erwiesen haben. Da zu unterschiedlichen Eingabedaten unterschiedliche Netzkonfigurationen passend sein können, wird weiterhin auch mit Alternativen experimentiert. Ziel ist es dabei, einen Parametersatz zu finden, welche über alle Experimente hinweg durchschnittlich die besten Ergebnisse liefert.

4.1. Datensammlung

Die für diese Experimente benötigten Bilddaten werden selbst aufgenommen. Dazu dient ein Netbook mit integrierte Webcam, auf dem Ubuntu läuft. Die installierte Lush-Version stammt direkt aus den offiziellen Paketquellen. Aufgenommen werden die Bilder im Graustufenmodus mit einer Auflösung von 640*480 Pixel, welche die native Auflösung der Webcam entspricht. Um möglichst viele Daten auf einmal zu sammeln, wird ein Bildstrom mit zehn Bildern pro Sekunde aufgenommen. Die genaue Anzahl Bilder wird aus Flexibilitätsgründen als Parameter an die Aufnahmefunktion übergeben. Da weniger Testdaten als Trainingsdaten benötigt

werden, werden für erstere tendenziell kürzere Bildströme verwendet. Die Verarbeitung, das Training und das Testen wird auf einem Desktop durchgeführt, auf dem ebenfalls Ubuntu läuft. Der Grund dafür ist dessen mit 3.4GHz getakteter Intel Core i5 Prozessor, der um ein vielfaches schneller ist als das Netbook.

Bei der Verarbeitung werden zuerst die Ränder der Bilder abgeschnitten, damit diese quadratisch werden. Das hat den Grund, dass Faltungsnetze aufgrund der Faltungsmasken besser damit zurecht kommen. Außerdem wird bei der Aufnahme der Bilder darauf geachtet, dass die Hand möglichst in der Bildmitte bleibt. Daher sind diese Ränder überflüssig und würden die Performance negativ beeinflussen.

Nach dem Zuschneiden werden die Bilder auf 120*120 Pixel skaliert. Diese Größe hat sich als die minimale Größe herausgestellt, damit die Finger im Bild noch gut erkannt werden können. Eine Auflösung von 60*60 Pixeln wäre auch möglich, allerdings muss die Hand dann ganz nah an der Kamera sein um die Finger noch deutlich genug erkennen zu können. Da dies eine hohe Konzentration erfordern und möglicherweise zu einem hohen Ausschuss bei den Daten führen würde, wird die höhere Trainingsdauer mit den größeren Bildern in Kauf genommen. Anschließend folgt die Klassifikation. Wie auch für die vorherigen Aufgaben wurde hierfür eine eigene Funktion geschrieben, mit der dieses ganz einfach und relativ schnell zu erledigen ist. Zur Kalibrierung der automatischen Erkennung werden im Test- und Trainingsdatensatz für jedes Bild die Anzahl gezeigten Finger per Tastatur angegeben. Nach dem letzten Bild eines Datensatzes werden die Klassifizierungen automatisch im richtigen Format gespeichert, damit diese Daten direkt vom Faltungsnetz zum trainieren und testen verwendet werden können. Am Ende der Verarbeitung werden alle Trainings- und Testdaten jeweils zu einem Datensatz zusammengefasst, damit diese dem Netz übergeben werden können.

4.2. Versuchsaufbau

Als Grundlage für alle Versuche dient die LeNet5 Demo, welche im Installationsverzeichnis von Lush unter ”packages/gblearn2/demos/lenet5.lsh” zu finden ist. Die Datei ”dsource-mnist.lsh”, welche im gleichen Verzeichnis zu finden ist, dient als Grundlage für die Datensätze.
Für diese Arbeit wird die Demo entsprechend angepasst. Es wird spezifiziert, dass sechs Klassen einzuteilen sind, die die Fingerzahl von null bis fünf repräsentieren. Als nächstes werden die target-Werte gesetzt. Diese wird beim Mean-Squared-Error-Training verwendet, wo ein

positiver Wert die Zugehörigkeit zu einer Klasse und ein negativer Wert die Zugehörigkeit zu einer anderen Klasse entspricht. Danach wird ein Vektor mit den trainierbaren Parametern gesetzt. Zur Erstellung des Netzes wird dann das Modul ”überwachtes Lernen” verwendet, dass das eigentliche Netz mit den verschiedenen Layern, eine Funktion zur Berechnung der euklidischen Distanz und eine Klassifikatorfunktion beinhaltet. Hier werden auch die Größen der Faltungsmasken, die Subsamplingraten, sowie die Outputgröße des letzten C-Layers festgelegt. Anschließend wird ein überwachter Trainer erstellt, welche nach dem gradientenbasierten Abstiegsverfahren lernt. Danach werden die Messobjekte angelegt, welche die Fehlerraten ermitteln. Darauf folgend werden die Kantengewichte initialisiert und die Trainings- und Testdatensätze aufgebaut. Diese bestehen jeweils aus einem 3D Array, welches die Bilder beinhaltet, sowie ein Vektor mit den dazugehörigen Klassifikationen. Danach werden die diagonale Hesse-Matrix und die einzelnen Lernparameter berechnet. Abschließend wird das Netz trainiert und die Trainings- und Testdaten getestet. Dies wird standardmäßig fünfmal durchlaufen, während dabei die Laufzeit gemessen wird.

Im Laufe der Experimente wurden an dieser Datei einige Änderungen vorgenommen. So wurden z. B. alle häufig geänderten Parameter am Anfang der Datei als globale Parameter definiert, damit man diese schnell ändern kann. Außerdem wurde die Möglichkeit implementiert ein trainiertes Netz zu speichern und zu laden. Dies hat den Vorteil, dass man z. B. ein Netz später weiter trainieren kann. Des Weiteren wurde eine Funktion erstellt, mit der einzelne Bilder getestet werden können. Dies ist hilfreich um zu erkennen, welche Gesten das Netz erkennt und wo es Probleme gibt.

4.3. Experiment: hoher Kontrast

Das erste Experiment soll zeigen, dass die Handgestenerkennung mittels Faltungsnetzen überhaupt möglich ist. Daher wird vom einfachsten Szenario ausgegangen: ein hoher Kontrast zwischen Hand und Hintergrund, keine Verdeckung von Teilen der Hand und diese immer deutlich im Bild. Erwartet wird hier eine eine geringe Fehlerrate bei Trainings- und Testdaten.

4.3.1. Datensatz 1800 Bilder

Den Anfang macht ein Trainingsdatensatz von 1800 Bildern. Da sich diese stark ähneln, wird angenommen, dass diese Menge zumindest für eine grobe Tendenz reicht. Zum testen wird ein

Satz mit 300 Bildern verwendet. Einige Beispiele aus diesem Datensatz sind in Abbildung 4.1 zu finden.

Abbildung 4.1.: Beispiele aus dem Datensatz mit 1800 Daten

Aufgrund der fehlenden Erfahrungen mit den Netzparametern werden für den ersten Durchlauf die Standardparameter von LeNet5 verwendet. Für die Datensätze entsprechen das einem Bias von null und einem Koeffizienten von 0.01. Die Faltungsmasken in den Layern C0 und C1 haben eine Auflösung von 5*5 Pixeln, die Subsampling-Layer S0 und S1 bestitzen Abtastraten von 2*2. Der F-Layer entspricht einem 120 dimensionalen Vektor. Die Größe der Faltungsmaske in Layer C2 wird automatisch berechnet. Diese muss die gleiche Auflösung wie die Ausgabegröße von S1 haben, da der nachfolgende F-Layer ein Vektor mit 120 Dimensionen ist. Die Berechnung von C2 erfolgt nach folgenden Formeln:

$$ki2 = (((image - height - (ki0 - 1)/si0) - (ki1 - 1))/si1)$$
$$kj2 = (((image - width - (kj0 - 1)/sj0) - (kj1 - 1))/sj1)$$

kiX und kjX entsprechen dabei der Höhe und Breite der einzelnen Faltungsmasken, siX und sjX die Höhe und Breite der Subsampling-Layer. Bei einer Bildauflösung von 120*120 Pixeln entspricht das bei den Standardeinstellungen eine Faltungsmaskengröße von 27*27 in C2.
Layer C0 bietet sechs Feature-Maps, welche alle mit dem Input verbunden sind. 16 Feature-Maps sind es in Layer C1, welche 60 Verbindungen zu S0 hat. C2 ist mit S1 voll verknüpft, was 1920 Verbindungen entspricht. Zusammen ergibt das für dieses Netz 1 402 242 trainierbare Parameter.
Zur Berechnung der Lernraten der einzelnen Parameter und der diagonalen Hesse-Matrix werden 100 Lernschritte und ein Schutzfaktor von 0.02 verwendet. Dieser Schutzfaktor sorgt dafür, dass alle Parameterwerte in einem sinnvollen Rahmen bleiben. Die globale Lernrate des Trainers liegt bei 0.0001.

Durchlauf	Trainingsdaten	Testdaten
1	89.44%	82.67%
2	89.44%	82.67%
3	89.44%	82.67%
4	69.33%	76.33%
5	61.61%	78.00%

Tabelle 4.1.: Fehlerraten im ersten Versuch

Parameter	trainmin	trainmax	testmin	testmax	Dauer
9 4 9 4 120	87.83%	89.44%	82.67%	90.00%	254s
9 4 9 4 480	89.44%	89.44%	82.67%	85.33%	267s
11 5 9 2 240	89.44%	89.44%	81.00%	87.67%	321s
5 2 5 2 120	80.17%	89.44%	82.33%	87.67%	302s

Tabelle 4.2.: Fehlerraten in weiteren Versuchen mit dem ersten Datensatz

Die Standardvorgehensweise beinhaltet fünf Durchläufe mit Training, testen der Trainingsdaten und testen der Testdaten. Das Ergebnis ist in Tabelle 4.1 zu sehen.
In den ersten drei Durchläufen hat sich an den Fehlerraten nichts geändert. Fast 90% bei den Trainingsdaten sind weitaus schlechter als erwartet. Beim vierten und fünften Durchlauf verbessern sich diese Werte etwas. Über 60% Fehlerrate bei den Trainingsdaten und fast 80% bei den Testdaten ist allerdings immer noch weitaus schlechter als anfangs vermutet. Die Dauer für diesen Versuch beträgt 303s.

Weitere Testläufe mit unterschiedlichen Metaparametern folgen. Diese sind in Tabelle 4.2 zu sehen. In der ersten Spalte sind die verwendeten Metaparameter eingetragen. Diese entsprechen den C0, S0, C1, S1 und F-Layern. In den weiteren Spalten sind jeweils die minimalen und maximalen Fehlerraten des Testlaufs mit jeweils fünf Durchgängen eingetragen.
In allen Versuchen sind die Fehlerraten sehr hoch. Eine deutliche Tendenz ist nicht zu erkennen, außer dass sie alle noch schlechter abschneiden als der erste Versuch. Eine mögliche Erklärung hierfür ist, dass durch zu große Faltungsmasken und Subsampling-Werten für die nachfolgenden Layer nur noch so wenig Bildinformationen übrig sind, dass diese nicht mehr als zufällig raten können.

Bei den Zeiten ist dagegen eine deutliche Tendenz sichtbar. Je größer die Faltungsmasken und S-Layer, umso schneller ist das Netz. Die Erklärung hierfür liegt in den Anzahl Parametern. Je größer die ersten Layer, umso kleiner sind die Faltungsmasken in C2. Da an dieser Stelle

Durchlauf	Trainingsdaten	Testdaten
1	39.86%	79.33%
2	18.50%	75.50%
3	11.39%	76.50%
4	6.94%	74.17%
5	3.61%	74.83%

Tabelle 4.3.: Fehlerraten mit künstlich verdoppeltem Datensatz

generell die meisten Faltungsmasken vorhanden sind, macht sich eine Verkleinerung deutlicher bemerkbar als eine Vergrößerung der Masken im ersten oder zweiten Layer.

Die dritte Zeile der Tabelle umfasst die Werte von zwei Versuchen. Für den zweiten Versuch wurden der Bias des Datensatzes von null auf 0.02 und der Koeffizient von 0.01 auf 0.1 gesetzt. Bis auf eine Verbesserung von 0.67% bei der minimalen Testerrorrate und eine um 0.12s längere Dauer hat sich nichts verändert. Aufgrund der geringen Auswirkungen wird mit diesen Werten nicht weiter experimentiert. Für die letzte Zeile wurden 50 Schritte für die Berechnung der einzelnen Parameterlernraten verwendet. Die Fehlerraten deuten hier an, dass durch die geringere Datenmenge bei der Berechnung der initialen Lernraten diese schlechter gewählt werden. Auch mit diesem Wert wird weiterhin nicht mehr experimentiert. Zwar könnte eine Erhöhung etwas bessere Ergebnisse liefern, da diese ohnehin weit hinter den Erwartungen zurückgeblieben sind, wird primär mit anderen Parametern experimentiert.

Da an dieser Stelle die Vermutung aufgekommen ist, dass zuwenige Daten vorliegen, werden diese künstlich verdoppelt. Dazu wird der aktuelle Datensatz kopiert, permutiert und zusammengefügt. Somit ergibt sich ein Datensatz aus 3600 Trainingsbilder und 600 Testbilder. Da sich im ersten Versuch die Standardeinstellungen als bestes Ergebnis herausgestellt haben, werden diese hier auch wieder verwendet. Die Ergebnisse dieses Versuchs sind in Tabelle 4.3 zu finden. Die Dauer dieses Versuchs beträgt mit 605s erwartungsgemäß etwa doppelt so lang als mit 1800 Daten.
Die Vermutung bezüglich zu wenige Trainings- und Testdaten hat sich mit den geringeren Fehlerraten bestätigt. Schon gleich im ersten Durchgang wird eine Fehlerrate von unter 40% bei den Trainingsdaten erreicht, welche damit deutlich besser sind als bei allen vorherigen Versuchen nach fünf Durchläufen. Bei den Testdaten hat sich eine Verbesserung von etwa 10% ergeben.
Da dies noch deutlich hinter den Erwartungen liegt, werden auch für diese Daten wieder meh-

Durchlauf	v1 Trainingsdaten	v1 Testdaten	v2 Trainingsdaten	v2 Testdaten
1	22.44%	77.17%	20.61%	76.17%
2	13.06%	75.17%	12.22%	77.50%
3	8.56%	75.17%	8.72%	77.17%
4	5.11%	75.83%	6.00%	76.83%
5	3.56%	76.17%	4.22%	77.83%

Tabelle 4.4.: Fehlerraten in weiteren Versuchen mit dem ersten Datensatz

Durchlauf	Trainingsdaten	Testdaten
1	56.83%	81.83%
2	35.33%	70.00%
3	26.00%	67.50%
4	18.75%	68.00%
5	12.64%	69.33%

Tabelle 4.5.: Fehlerraten in weiteren Versuchen mit künstlich verdoppelten Daten

rere Konfigurationen getestet. Die erste Alternative verwendet 240 anstatt 120 Dimensionen für den Layer C2. Die anderen Metaparameter bleiben gleich. Dieser Versuch wird zweimal durchgeführt. Beide Versuche benötigen 771s. Tabelle 4.4 zeigt die Ergebnisse dieses Versuchs. Dieser Vergleich zeigt, dass das Lernen ein dynamischer Prozess ist. In den ersten drei Durchläufen werden im zweiten Versuch die besseren Ergebnisse bei den Trainingsdaten erreicht, während in den letzten beiden Durchläufen das Netz im ersten Versuch besser ist.
Des weiteren ist die Gegenüberstellung zum Netz mit 120 Dimensionen in C2 interessant. Während dort am Anfang deutlich schlechtere Ergebnisse erzielt wurden, haben diese sich nach den fünf Durchläufen auf ein gleiches Niveau eingependelt. Dafür ist die erste Variante bedingt durch die deutlich geringere Anzahl trainierbare Parameter um über 20% schneller.
An dieser Stelle deutet sich schon an, dass die Standardeinstellungen von LeNet5 bereits sehr gut gewählt sind. Es wird zwar weiterhin noch ein wenig experimentiert, allerdings werden die Werte nicht mehr so weit abweichen wie noch in den ersten Versuchen.

Eine weitere Alternative benutzt 9*9 Faltungsmasken in C0 und C1, sowie eine Abtastrate von 4*4 in den S-Layern. C2 ist wieder ein 120 dimensionaler Layer. Diese Ergebnisse sind in Tabelle 4.5 zu finden.
Hier ergibt sich das gleiche Bild wie schon bei den ersten Versuchen. Diese Konfiguration liefert schlechtere Ergebnisse als die Standardkonfiguration. Allerdings werden auch bei diesem Versuch die Trainingsdaten mit jedem Durchlauf besser erkannt. Anders verhalten sich

die Testdaten. Zuerst sinken die Fehlerraten deutlich, um sich anschließend wieder leicht zu verschlechtern. Dies deutet auf ein überlernen der Trainingsdaten hin. Als wahrscheinlichster Grund wird ein zu kleiner Datensatz angenommen.

4.3.2. Datensatz 3600 Bilder

Da im vorherigen Abschnitt festgestellt wurde, dass ein Datensatz von 1800 Daten bei weitem nicht ausreicht werden diesmal 3600 Bilder verwendet. Dazu werden weitere 1800 Bilder für die Trainingsdaten aufgenommen. Die Testdaten werden um 900 Bilder erweitert, was sich auf 1200 Testbilder insgesamt addiert. Wie schon bei den letzten Versuchen werden die Daten künstlich mittels einer permutierten Kopie verdoppelt, woraus sich 7200 Trainings- und 2400 Testdaten ergeben. Einige Beispiele aus diesem Datensatz sind in Abbildung 4.2 zu finden. Bei einem Teil der Aufnahmen wurde vergessen die Beleuchtung im Raum anzupassen. Dadurch ist die Hand auf einen Teil der Bilder etwas überbeleuchtet. Da sich die Hand noch deutlich vom Hintergrund abhebt und die Anzahl Finger noch deutlich zu erkennen sind, wurden diese Daten trotzdem verwendet.

Abbildung 4.2.: Beispiele aus dem Datensatz mit 3600 Daten

Als erste Konfiguration wird ein Netz verwendet, welches wieder 5*5 Faltungsmasken in C0 und C1 und 2*2 Subsampling verwendet. Für C2 werden 240 Dimensionen benutzt. Die fünf Durchläufe brauchen bei diesem Versuch 1588s, etwas mehr als doppelt so lange im Vergleich zum vorherigen Datensatz. Hierfür zeigt Tabelle 4.6 die Resultate.
Wie schon bei den letzten Versuchen wird die Erkennungsrate im Trainingsdatensatz mit jedem Durchlauf besser. Problematisch sind hier abermals die Testdaten, dessen Erkennung immer schlechter wird. Auch zwei weitere Versuche mit den gleichen Einstellungen zeigen vergleichbare Ergebnisse. An dieser Stelle verstärkt sich die Vermutung, dass ein künstliches

Durchlauf	Trainingsdaten	Testdaten
1	14.82%	67.75%
2	8.88%	69.00%
3	5.90%	71.75%
4	4.46%	72.92%
5	3.12%	73.58%

Tabelle 4.6.: Fehlerraten mit erweitertem Datensatz

Durchlauf	Trainingsdaten	Testdaten
1	71.72%	70.33%
2	71.72%	70.33%
3	71.72%	72.92%
4	70.42%	71.92%
5	66.83%	75.17%

Tabelle 4.7.: Fehlerraten ohne künstliche Verdopplung

Verdoppeln der Daten zwar Arbeit spart, allerdings keine brauchbaren Ergebnisse liefert. Ein weiterer Test mit einer 7*7 Faltungsmatrix in C0 und 3*3 Subsampling in S1 bestätigen die bisherigen Ergebnisse. Zwar ist diese Konfiguration etwa 33% schneller, dafür ist die Fehlerrate nach den fünf Durchläufen etwa doppelt so hoch.

Aufgrund der bisherigen Erfahrungen und Vermutungen wird ein Test mit dem aktuellen Datensatz von 3600 Bildern durchgeführt. Es wird wieder die bisher schon häufig verwendete Konfiguration mit 5*5 Faltungsmasken und 2*2 Subsampling, sowie 240 Dimensionen in C2 verwendet. Als Testdatensatz wird der bisherige weiter verwendet. Da mit diesen Daten nicht trainiert wird, sollte diese keine Auswirkungen auf die Erkennungsrate haben. Die Laufzeit beträgt in diesem Fall 839s. Tabelle 4.7 zeigt die Ergebnisse.
Diesmal ist die Fehlerrate erwartungsgemäß deutlich höher als bei den künstlich verdoppelten Datensätzen. Eine leicht positive Tendenz ist bei den Trainingsdaten zu erkennen, während die Testdaten keine klare Richtung zeigen. Aufgrund dieser Tendenz wurden zwei weitere Versuche mit jeweils zehn Durchlaufen gemacht, wobei sich Netzkonfigurationen nicht verändert haben. Ergebnisse hier waren 52.50% und 58.58% respektive 40.17% und 38.78% nach fünf bzw. zehn Durchläufen. Bei den Testdaten schwankten diese zwischen 68.71% und 79.83%, ohne das sich dort eine Tendenz abgezeichnet hat.

Parameter	trainmin	trainmax	testmin	testmax	Dauer
5 2 5 3 240	62.42%	71.17%	70.58%	73.96%	649s
5 2 5 6 240	65.67%	69.50%	65.92%	73.62%	533s
5 2 5 6 120	64.06%	70.61%	64.08%	73.33%	508s
5 2 5 3 120	53.83%	71.17%	69.71%	77.25%	569s

Tabelle 4.8.: Fehlerraten in weiteren Versuchen mit dem zweiten Datensatz

In einem weiteren Versuch wurde die globale Lernrate von 0.0001 auf 0.001 erhöht. In allen fünf Durchläufen wurde hier eine Fehlerrate von 71.72% bei den Trainings- und 70.33% bei den Testdaten erreicht. Auch wenn diese Ergebnisse auf den ersten Blick etwas merkwürdig wirken können, gibt es dennoch eine Erklärung dafür. Zwar ist 0.001 nur eine sehr kleine Zahl, allerdings um den Faktor zehn höher als der Standardwert. Bei einem kleinen Datensatz führt dies dazu, dass das Netz scheinbar nichts neues mehr lernt. Grund dafür könnte die in Kapitel 2.2 beschriebene Oszillation sein.

Um noch weitere, möglicherweise geeignete Konfigurationen zu finden, werden noch einige Versuche gestartet. Da sich große Änderungen bisher als schlecht herausgestellt haben, liegt der Fokus an dieser Stelle auf Änderungen in den Layern S1 und C2. Die Resultate sind in Tabelle 4.8 zu finden.
Wie schon erwartet gibt es bei der Dauer keine neuen Erkenntnissen. Je weniger trainierbare Parameter im Netz vorhanden sind, umso schneller sind die Durchläufe beendet. Einige Auffälligkeiten sind bei den Fehlerraten zu finden. Während ein Subsampling von 3*3 in S1 zu besseren Ergebnissen bei den Trainingsdaten führt, zeigt 6*6 bessere Resultate bei den Testdaten. Die Unterschiede bei der Anzahl Feature-Maps in C2 sind zwar gering, dennoch zeigt sich 120 als die bessere Wahl, da etwas geringer Fehlerraten erreicht werden und diese Konfiguration weniger Zeit zum lernen benötigt. Eine kleine Überraschung bietet der letzte Versuch. Hier werden nahezu die gleichen Werte wie mit der Standardkonfiguration erreicht, allerdings wird dafür deutlich weniger Zeit benötigt. Sogar ein weiterer Durchlauf wäre noch schneller und könnte dabei bessere Ergebnisse liefern.

Die Versuche mit diesem Datensatz zeigen sehr deutlich die Grenzen eines Computers auf. Während wir Menschen nur wenige Bilder brauchen um Objekte klassifizieren zu können, braucht eine Maschine tausende Bilder und hat sogar dann noch Schwierigkeiten gleiche Objekte zu erkennen. Die Hauptursache hierfür ist die Komplexität unseres Gehirns. Während dieses aus Abermilliarden flexiblen Neuronen besteht, werden diese Strukturen in Form vom

oftmals nur wenigen Millionen künstlichen Neuronen im Computer nachgebildet. Trotz dieser Einschränkungen haben die in den letzten Versuchen verwendete Konfigurationen sich nicht als vollkommen unbrauchbar herausgestellt. Daher werden diese auch bei weiteren Versuchen als Alternativen zur Auswahl stehen.

4.3.3. Datensatz 6000 Bilder

Auch ein Datensatz von 3600 Bildern hat sich als nicht ausreichend herausgestellt. Zwar lässt sich hier schon eine positive Tendenz bei den Tests mit Trainingsdaten erkennen, allerdings noch längst nicht so gut wie erwartet. Die Testdaten deuten sogar ein Überlernen des Netzes an. Das heißt, dass das Netz zwar die Trainingsdaten gut erkennt, aber bei neuen Daten keine guten Ergebnisse erzielt.

Abbildung 4.3.: Beispiele aus dem Datensatz mit 6000 Daten

Für die nächsten Versuche werden die Datensätze auf 6000 fürs Training und 2400 fürs Testen erweitert. Es wird keine künstlichen Verdopplungen mehr geben, da sich diese als unbrauchbar herausgestellt hat. Außerdem werden alle Daten jetzt nur noch permutiert verwendet. Einige Beispiele aus den neu hinzugekommenen Daten sind in Abbildung 4.3 zu finden. Da ein Faltungsnetz positions- und skalierungsinvariant ist, wurde die Hand bei diesen Aufnahmen mehr im 3D Raum bewegt.

Angefangen wird mit der 5*5 Faltungsmasken in C0 und C1, 2*2 und 3*3 Subsampling in S0 und S1, sowie 120 Dimensionen in C2. Für die fünf Durchläufe werden 903s, also über 15 Minuten benötigt. Die Ergebnisse sind in Tabelle 4.9 zusammengefasst.

Durchlauf	Trainingsdaten	Testdaten
1	33.07%	67.96%
2	15.08%	65.00%
3	4.32%	61.04%
4	1.42%	61.17%
5	0.68%	61.38%

Tabelle 4.9.: Fehlerraten bei 6000 Trainings- und 2400 Testdaten

Durchlauf	v1 train	v1 test	v2 train	v2 test
1	41.58%	66.75%	40.28%	69.12%
2	21.75%	61.83%	21.33%	68.46%
3	6.95%	56.96%	8.53%	64.29%
8	0.25%	53.38%	0.13%	58.08%
9	0.13%	52.29%	0.07%	57.96%
10	0.10%	52.38%	0.05%	58.29%

Tabelle 4.10.: Fehlerraten in weiteren Experimenten

Diesmal werden nach den fünf Durchläufen schon Fehlerraten unter einem Prozent bei den Trainingsdaten erreicht, deutlich besser als alle bisherigen Ergebnisse. Bei den Testdaten liegen diese mit knapp über 61% zwar auch deutlich unter den bisherigen Resultaten, allerdings sind sie immer noch sehr hoch. Nach dem dritten Durchlauf steigen die Fehlerraten bei den Testdaten wieder leicht an. Um festzustellen, ob dieses Zufall ist oder hier tatsächlich der beste Wert erreicht wurde, werden weitere Tests durchgeführt.

Zuerst werden die gleichen Einstellungen verwendet, allerdings mit zehn Durchlaufen. Bei einem weiteren Versuch wird für S1 ein Subsampling von 2*2 verwendet. Alle anderen Metaparameter bleiben gleich. Erwartungsgemäß verhalten sich die Laufzeiten mit 1803s beim ersten und 2121s beim zweiten Test. Exemplarisch werden jeweils nur die ersten und letzten drei Durchläufen in Tabelle 4.10 dargestellt.
Die Fehlerraten bei den Trainingsdaten sind, wie im vorherigen Test, sehr gering. Die Testdaten werden jetzt bei einem Subsampling von 3*3 fast zur Hälfte erkannt. Die 2*2 Variante schneidet hier sowohl bei den Ergebnissen als auch bei der Laufzeit schlechter ab. Daher werden für folgende Experimente primär die Einstellungen aus dem ersten Versuch verwendet.

Aufgrund einiger Änderungen im Code wurde in einem Versuch vergessen die Netzgewichte zu initialisieren. Die Auswirkungen zeigen die Ergebnisse ganz deutlich. In allen drei Durchläufen

	0	1	2	3	4	5	6	7			0	1	2	3	4	5	6	7
0	x		x		x	x				12		x	x		x	x		
1	x		x		x		x			13		x	x		x		x	
2	x		x		x			x		14		x	x		x			x
3	x		x			x	x			15		x	x			x	x	
4	x		x			x		x		16		x	x			x		x
5	x		x				x	x		17		x	x				x	x
6		x		x	x	x				18	x			x	x	x		
7		x		x	x		x			19	x			x	x		x	
8		x		x	x			x		20	x			x	x			x
9		x		x		x	x			21	x			x		x	x	
10		x		x		x		x		22	x			x		x		x
11		x		x			x	x		23	x			x			x	x

Tabelle 4.11.: Verbindungen zwischen S0 und C1

wurde eine Fehlerrate von 67.75% auf den Trainings- und 74.00% auf den Testdaten erreicht. Grund dafür sind die standardmäßige Initialisierung der Matrizen mit null (als Wert), sowie die Multiplikationsoperationen damit.
In einem weiteren Versuch wurde die Lernrate der trainierbaren Parameter von 0.02 auf 0.05 gestellt. Dies resultiert in einer geringeren Fehlerrate auf den Trainingsdaten im ersten Durchlauf. Dafür fallen die Ergebnisse nach drei Durchläufen schlechter aus. Ebenso sind die Fehlerraten bei den Testdaten höher als bei einer Lernrate von 0.02.

Die bisherigen Ergebnisse lassen die Vermutung aufkommen, dass die Konfiguration der Feature-Maps für die Erkennung der Anzahl gezeigten Finger nicht ausreicht. Anstatt bisher sechs Feature-Maps in C0 werden diese auf acht erhöht. In C1 werden diese von 16 auf 24 erhöht. C0 ist mit dem Input wieder voll verknüpft, während Tabelle 4.11 die 96 Verknüpfungen zwischen S0 und C1 zeigt, welche die gleiche Belegung wie im Artikel von Huang und LeCun (2006) entspricht. Durch das erhöhen der Anzahl Feature-Maps wird erhofft, dass die Fehlerrate bei den Testdaten sinkt. Mit den bisherigen Einstellungen wurden sogar im besten Fall weniger als die Hälfte der Testfälle korrekt erkannt. Für eine Anwendung in der Industrie ist das deutlich zu wenig.
Mit dieser neuen Konfiguration der Feature-Maps wird wieder der gleiche Datensatz aus 6000 Trainings- und 2400 Testdaten verwendet. Diesmal wird mit unterschiedlichen Einstellungen in C1 und S1 experimentiert. Verwendet werden Kombinationen aus 5*5 und 7*7 Faltungsmasken mit 3*3 oder 6*6 respektive 2*2 oder 4*4 Subsampling. Dabei werden zuerst fünf Durchläufe gemacht und anschließend das bereits trainierte Netz geladen und mit weiteren fünf Durchläu-

Parameter	trainmin	trainmax	testmin	testmax	Dauer
5 2 5 3 120	0.10%	54.90%	51.12%	68.38%	1342s+1344s
5 2 5 6 120	0.65%	57.10%	47.33%	66.38%	1195s+1199s
5 2 7 2 120	0.05%	38.00%	56.71%	69.29%	2108s+2110s
5 2 7 4 120	0.12%	48.58%	47.00%	64.92%	1807s+1816s

Tabelle 4.12.: Fehlerraten mit neuer Feature-Map Konfiguration

Durchlauf	train	test
1	43.82%	62.67%
5	4.14%	47.58%
6	2.58%	44.58%
10	0.69%	48.25%

Tabelle 4.13.: Fehlerraten in weiteren Experimenten

fen weiter trainiert. Die Ergebnisse sind in Tabelle 4.12 zu sehen.
Bei diesen Versuchen zeigt sich eine interessante Tendenz ab. Eine größere Faltungsmaske in C1 führt zwar zu einer deutlich geringeren Fehlerrate auf den Trainingsdaten nach dem ersten Durchlauf, nach den insgesamt zehn Durchlaufen ist dieser Unterschied äußerst gering geworden. Bei den Testdaten sind die Unterschiede nur sehr gering. Ein vergrößern der Subsamplingrate hingegen resultiert in deutlich bessere Ergebnisse auf den Testdaten. Auch lernt das Netz dadurch schneller. Die Trainingsdauer ist zwar für spätere Anwendungen irrelevant, der Zusammenhang mit den besseren Ergebnissen ist zum experimentieren allerdings sehr angenehm.

4.3.4. Weitere Datensätze

Auch wenn inzwischen über die Hälfte der Testdaten korrekt erkannt wird, sind diese Ergebnisse dennoch in keinerlei Weise zufriedenstellend. Deswegen wird für einen weiteren Versuch die Hälfte der Testdaten zum Trainingsdatensatz hinzugefügt. Das ergibt 7200 Daten zum trainieren und 1200 zum testen. Da sich die Konfiguration mit 5*5 Faltungsmasken und 2*2 und 6*6 Subsampling bisher als am Schnellsten herausgestellt hat und dabei auch vergleichsweise gute Ergebnisse geliefert hat, wird diese für diesen Versuch verwendet. Die Resultate der Durchläufe eins, fünf, sechs und zehn sind in Tabelle 4.13 aufgeführt. Da das Lernen in zwei Stufen durchgeführt wurde, entsprechen diese jeweils den ersten und letzten Durchgang der beiden Versuche.
Die Erkennungsrate der Testdaten hat sich leicht auf 44.58% verbessert. Ein neuer Bestwert, wel-

cher aber immer noch nicht ausreichend gut ist. Unter Annahme einer linearen Verbesserung wären noch über 15 000 weitere Trainingsdaten notwendig um eine Fehlerrate unter 10% zu erreichen. Da diese Annahme unrealistisch optimistisch ist, stellt sich die Frage, ob irgendetwas übersehen wurde. Daher wurden alle Daten nochmal kontrolliert. Dabei ist aufgefallen, dass relativ viele interpretationsfähige Bilder dabei sind. Diese könnten dazu führen, dass falsche Features gelernt werden, was wiederum in höhere Fehlerraten resultieren könnte. Um das Problem zu umgehen bieten sich zwei Möglichkeiten an: Zum Einen könnte man alle Bilder neu klassifizieren. Dazu müsste eine weitere Klasse eingeführt werden, in der alle interpretationsfähige Bilder einsortiert werden. Diese Möglichkeit hat das Problem, dass das Netz versucht, Features zu lernen. Eine Aussage wie ”Alles, was sich nicht eindeutig zuordnen lässt, fällt in diese Klasse” ist dadurch nicht möglich. Daher ist anzunehmen, dass dieser Lösungsansatz eher zu noch schlechteren Ergebnissen führen würde.

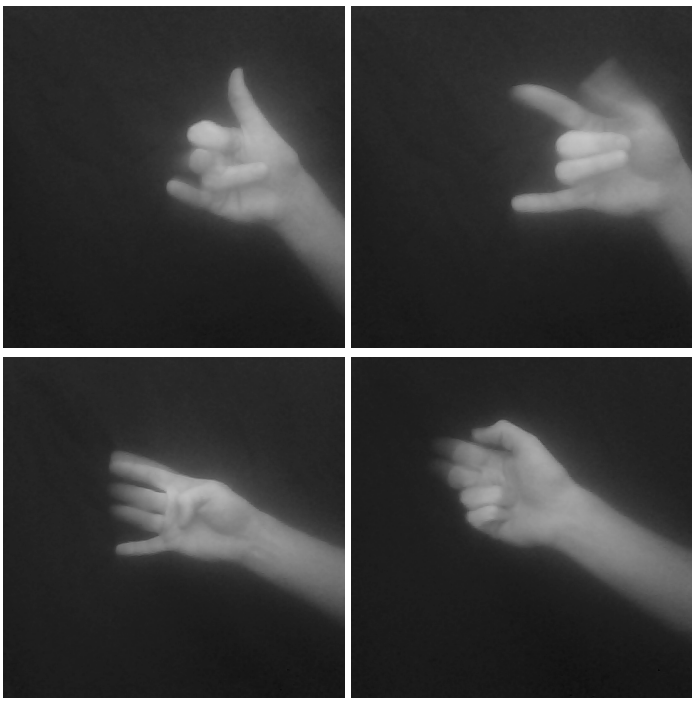


Abbildung 4.4.: Beispiele einiger gelöschter Bilder

Die Alternative ist das Aussortieren der Bilder. Das Resultat ist ein sauberer Datensatz, in dem für Menschen keine mehrdeutigen Interpretationen mehr möglich sind. Da diese Möglichkeit die besseren Ergebnisse verspricht, werden die Datensätze gesäubert. Aufgrund des doch relativ hohen Aufwands alle 8400 Daten zu sichten wurde nach Abarbeitung der ersten

Hälfte der Trainingsdaten ein Testlauf durchgeführt. Zu diesem Zeitpunkt beinhaltete der Trainingsdatensatz 6654 Bilder. Das heißt, dass von den ersten 3600 Bildern 546 aussortiert wurden, was über 15% Ausschuss entspricht. Obwohl insgesamt weniger Bilder vorhanden sind und noch nicht alles kontrolliert wurde, wurde in diesem Zwischenversuch mit einer Fehlerrate von 43.17% ein neuer Bestwert erzielt.

Daraus lässt sich deutlich feststellen, dass eine saubere Klassifikation sehr wichtig ist. Verschwommene Bilder sollten ebenso wenig verwendet werden wie Bilder, welche mehrere Interpretationsmöglichkeiten bieten. Einige solcher Beispiele, die aus den Datensätzen gelöscht wurden, sind in Abbildung 4.4 zu finden. in einem fortlaufenden Bildstrom ist die Klassifikation unter Berücksichtigung der vorherigen Bilder möglich. Nach der Permutation wird auch für Menschen jedes Bild als ein eigenständiges Bild gesehen, weswegen dann keine saubere Interpretation mehr möglich ist. Des Weiteren lässt sich aus dieser Erkenntnis schlussfolgern, dass die bereits im vorherigen Kapitel angesprochene Verdeckung für den in dieser Arbeit beschriebenen Ansatz vermutlich ein unlösbares Problem darstellt. Daher finden in dieser Arbeit keine Experimente diesbezüglich statt.

Es hat sich herausgestellt, dass die zweite Hälfte des Datensatzes weniger problematische Klassifikationen enthielt. Nachdem alle Bilder kontrolliert und aussortiert wurden, sind 6530 Trainings- und 1023 Testdaten übrig geblieben. Das bedeutet einen Ausschuss von 9.3% respektive 15%. Diesmal wurde im vierten Durchlauf eine Fehlerrate von 38.32% erreicht. Das entspricht eine Verbesserung von etwa 6% durch das Säubern der Datensätze. Trotz Verbesserung bleibt diese deutlich hinter den Erwartungen zurück. Daher wird der Trainingsdatensatz abermals erweitert, jetzt auf 7970 Daten. Einige Beispiele sind in Abbildung 4.5 zu finden.

Abbildung 4.5.: weitere neue Bilder

Durchlauf	v1 train	v1 test	v2 train	v2 test
1	60.36%	67.06%	54.10%	62.85%
8	0.24%	35.48%	1.38%	39.49%
10	0.09%	36.07%	0.55%	40.18%

Tabelle 4.14.: Fehlerraten in den letzten Experimenten

Diesmal wird die Hand etwas gedreht. Da sich dadurch die relative Position der Finger zueinander nicht ändert, wird erhofft, dass das Netz robuster wird. Zwei Versuche mit jeweils zwei Durchgängen werden damit durchgeführt. Im ersten Versuch wird die bisherige beste Konfiguration verwendet. Da eine Erhöhung des Subsamplings in S1 bisher zu besseren Ergebnissen geführt hat, wird auch S0 erhöht. Die Faltungsmasken bleiben 5*5, S0 und S1 haben jetzt eine Subsamplingrate von 4*4 respektive 5*5. Die relevanten Ergebnisse dieser Versuche sind in Tabelle 4.14 zu finden. Im ersten Versuch wurde mit 35.48% ein neuer Bestwert erreicht. Es werden also nahezu zweidrittel der Gesten korrekt erkannt. Der zweite Versuch hat keine besseren Ergebnisse erzielt.

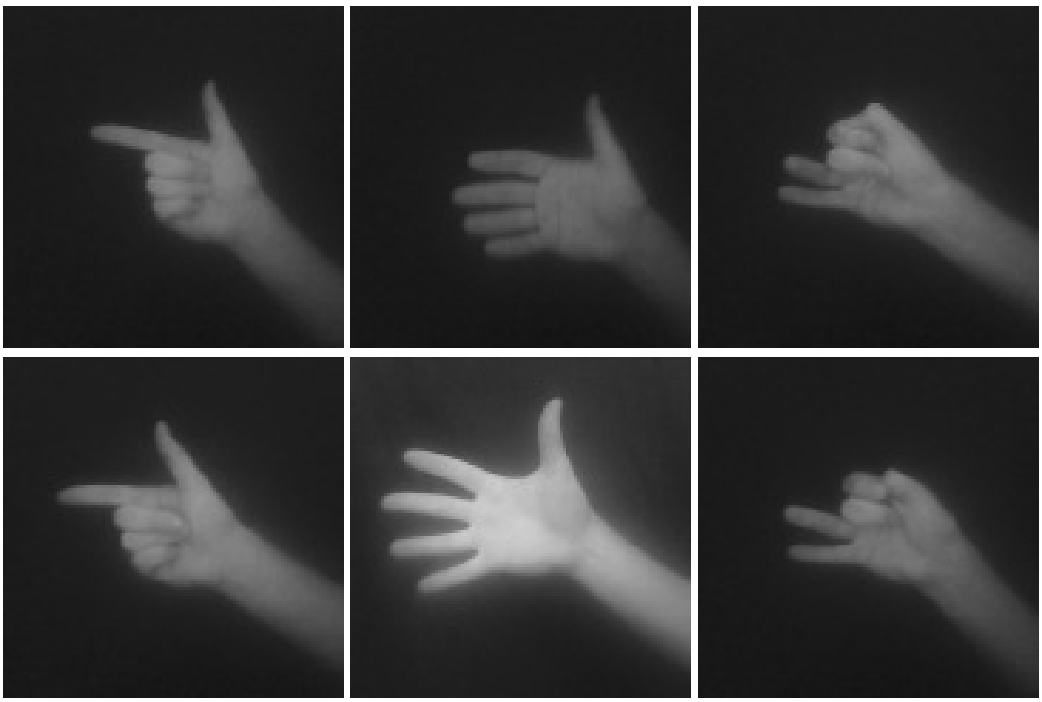


Abbildung 4.6.: oben: korrekt erkannt, unten: falsch erkannt

Um festzustellen, wo genau es Probleme mit der Erkennung gibt, werden einige Bilder des Testdatensatzes manuell kontrolliert. Einige Beispiele dazu sind in Abbildung 4.6 zu sehen. Die oberen drei Bilder wurden korrekt erkannt, während die unteren Bilder falsch erkannt wurden. Dies zeigt deutlich, dass nicht die erwünschten Features gelernt wurden. Diese Ergebnisse erwecken den Eindruck, dass das Netz bestimmte Kontraste gelernt hat, anstatt Kanten der Finger. Während sich die Beispiele links für einen Menschen kaum unterscheiden, sind die Unterschiede mittig und rechts tendenziell besser zu erkennen.

4.4. Experiment 2, Einschränkung der Gesten

Nachdem die bisherigen Ergebnisse nicht für eine praxistaugliche Anwendung sprechen, kam erneut die Frage auf, wo das Problem noch liegen könnte. Dabei haben sich die Anzahl mögliche Gesten als sehr umfangreich herausgestellt. Es gibt jeweils fünf Möglichkeiten einen oder vier Finger zu zeigen, jeweils zehn mit zwei oder drei Finger und jeweils eine Möglichkeit um fünf und null Finger zu zeigen. Das ergibt 32 Arten, welche auf sechs Klassen verteilt sind. Dazu kommen noch die relative Positionen zueinander. Beispiele zu allen Möglichkeiten finden sich im Anhang.

Abbildung 4.7.: Beschränkung auf drei Gesten

Um die Erkennung deutlich zu vereinfachen wurde die Anzahl Gesten auf drei reduziert. Diese sind in Abbildung 4.7 zu sehen. Damit hat das Netz deutlich weniger zu lernen und sollte entsprechend deutlich bessere Resultate liefern. Der Grund für genau diese Gesten liegt in ihre unterschiedlichen Erkennungsmerkmale. Der Daumen zeigt als einziger Finger seitlich von der Hand weg und ist, im Gegensatz zu den anderen Fingern, nur von der Seite sichtbar. Durch leichte Krümmungen lässt sich das Netz so robuster gegenüber die exakte Form trainieren. Zeigefinger und Mittelfinger lassen sich sehr intuitiv zusammen zeigen. Außerdem lassen sich

Parameter	trainmax	testmin	testmax	Dauer
5 2 5 6 120	0.97%	2.86%	10.76%	458s
5 2 5 3 120	28.85%	2.52%	57.31%	516s
5 2 5 2 120	4.39%	3.03%	24.37%	604s
5 2 7 4 120	0.04%	2.35%	4.03%	693s
5 2 7 4 160	0.13%	2.52%	3.19%	704s
5 2 5 6 160	50.82%	3.03%	65.71%	463s

Tabelle 4.15.: Fehlerraten bei drei Gesten

diese deutlich vom Daumen unterscheiden. Aufgrund des Alleinstellungsmerkmal des Daumens sollte dieser nicht für weitere Gesten verwendet werden. Da sich auch die Kombination aus Zeigefinger, Mittelfinger, Ringfinger und kleiner Finger intuitiv zeigen lässt wird diese als dritte Geste verwendet. Ein weiterer Grund für diese Auswahl liegt in der Klassifikation. Diese erfolgt zahlenbasiert, zusammenhängend und fängt bei null an. Somit ergeben sich null, eins und zwei als Klassen, welche als Exponent in 2^n die Anzahl gezeigten Finger entsprechen.

Aufgrund des verringerten Gestensatzes besteht die Möglichkeit, dass sich eine andere Netzkonfiguration besser eignet. Deswegen werden mehrere Möglichkeiten ausprobiert. Für dieses Experiment werden 2371 Trainings- und 595 Testdaten verwendet. Wie in den letzten Versuchen wird darauf geachtet, dass diese keine sehr unscharfen Bilder beinhalten und nur eine eindeutige Klassifikation zulassen. Die Ergebnisse wurden in Tabelle 4.15 aufgearbeitet.

Diesmal erreichen die Fehlerraten in allen Konfigurationen sehr gute Werte. Spätestens nach dem fünften Durchlauf wurden im Trainingsdatensatz alle Gesten korrekt erkannt. Von den Testdaten wurden nur maximal etwa drei Prozent nicht richtig erkannt. Bei den 595 Bildern entspricht das zwischen 14 und 18 falsch erkannte Gesten. Die Unterschiede der Fehlerraten zwischen den einzelnen Konfigurationen sind so gering, dass sich nicht eine bestimmte als die Beste herausgestellt hat. Zwar deutet sich in Teilen ein ähnliches Bild an wie schon im ersten Experiment, um dies zu belegen oder widersprechen müssten allerdings signifikant mehr Trainings- und Testdaten verwendet werden. Diese Resultate sprechen für eine praxistaugliche Anwendung, sofern nur wenige Befehle bzw. Gesten unterschieden werden müssen.

Die maximalen Fehlerraten in Tabelle 4.15 wurden jeweils im ersten Durchlauf erzielt. Interessant sind hier die großen Unterschiede bei der Erkennung, welche sich nicht nur mit den kleinen Änderungen an den Netzparametern erklären lässt. Vielmehr dürfte hier das Zusammenspiel zwischen diesen und die zufällige Initialisierung der Kantengewichte eine

Durchgang	1	2	3	4	5
Testfehlerraten	65.62%	60.94%	58.51%	56.08%	54.51%

Tabelle 4.16.: Fehlerraten der Testdaten mit der linken Hand

Rolle spielen. Sofern die Lernvorgänge ausreichend oft wiederholt werden, pendeln sich die Ergebnisse unabhängig der initialen Gewichte in einem Zielbereich ein.

4.5. Experiment 3, Spiegelung

In einer weiteren Untersuchung werden die gleichen Trainingsdaten wie aus dem vorherigen Experiment verwendet. Die Testdaten unterscheiden sich allerdings in einer Kleinigkeit: Anstatt nur Bilder der rechten Hand zu verwenden wird ein Datensatz mit Bilder der linken Hand verwendet. Die linke Hand eines Menschen ist zwar nur eine Spiegelung der Rechten und umgekehrt, das Ergebnis ist allerdings offen. Auch wenn wir Menschen keine Probleme bei dieser Unterscheidung haben, könnte dies für einen Computer eine große Hürde darstellen.

Abbildung 4.8.: Gesten der linken Hand

Für dieses Experiment steht ein Testdatensatz mit 576 Bildern zur Verfügung. Beispiele sind in Abbildung 4.8 zu sehen. Als Netzkonfiguration dienen 5*5 respektive 7*7 Faltungsmasken in C0 und C1, sowie 2*2 respektive 4*4 Subsampling in S0 und S1. Die Ausgabe von C2 ist ein 160 dimensionaler Vektor. Die Fehlerraten auf den Testdaten sind in Tabelle 4.16 dargestellt.

Nach dem ersten Durchlauf werden knapp zweidrittel falsch erkannt. Da es nur drei Gesten gibt, gleicht das zufälligem Raten. Mit jedem Durchlauf steigt zwar die Erkennungsrate, allerdings werden auch nach fünf Durchläufen noch weniger als die Hälfte der Gesten korrekt erkannt.

Da im dritten Durchlauf schon alle Trainingsdaten korrekt erkannt wurden, sind keine großen Verbesserungen zu erwarten.

In einem kleinen Zwischenexperiment werden die Testdaten gespiegelt. Dadurch sehen diese aus wie mit der rechten Hand aufgenommen. Zu erwarten sind somit ähnliche Ergebnisse wie im vorherigen Experiment. Diese Erwartungen wurden allerdings schon im ersten Durchlauf übertroffen. Erreicht wurden Fehlerraten von 2.78% respektive 1.22% auf den Trainings- und Testdaten. In allen weiteren Durchläufen betrug die Fehlerrate null Prozent, das heißt, dass alle Gesten korrekt erkannt wurden.

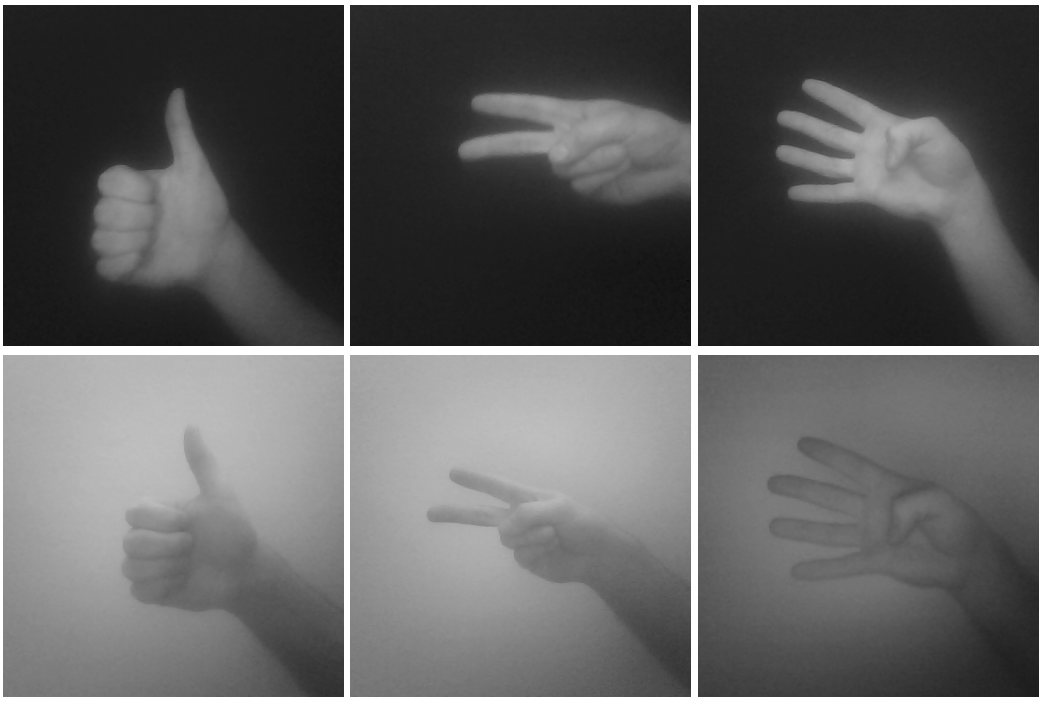


Abbildung 4.9.: Kontrastvergleich alt - neu

4.6. Experiment 4, Kontraste

Alle bisherigen Experimente wurden unter den gleichen Bedingungen durchgeführt, nämlich eine helle Hand vor einem dunklen Hintergrund. Mit diesen Daten lässt sich allerdings schlecht feststellen, ob tatsächlich die gewünschten Features, also die Fingerpositionen und -formen,

Durchgang	1	2	3	4	5
Testfehlerraten	67.39%	75.13%	88.24%	89.24%	88.91%

Tabelle 4.17.: Fehlerraten der Testdaten mit geringem Kontrast

Durchgang	1	2	3	4	5
Testfehlerraten	66.39%	67.39%	26.89%	15.63%	14.79%
Testfehlerraten	67.39%	53.95%	15.13%	9.58%	9.58%
Testfehlerraten	67.73%	67.73%	67.73%	9.41%	3.03%
Testfehlerraten	3.53%	2.35%	2.35%	2.35%	2.35%

Tabelle 4.18.: Fehlerraten der Testdaten mit geringem und hohem Kontrast

gelernt wurden oder irgendwelche anderen Eigenschaften wie mögliches Bildrauschen. Daher wird in einem weiteren Experiment Bilder mit einem deutlich geringeren Kontrast als Testdatensatz verwendet. Dazu diese vor einem anderen Hintergrund aufgenommen, in diesem Fall eine weiße Wand. Wie groß dieser Unterschied ist, zeigt der direkte Vergleich in Abbildung 4.9.

Auch wenn die neuen Bilder für einen Menschen relativ leicht zu erkennen sind, gibt es kaum eine bessere Möglichkeit, die gelernten Features zu überprüfen. Da nach Anwendung eines Kantendetektionsfilters die Finger nach wie vor zu erkennen sind, ist das Ergebnis offen, auch wenn eine deutlich schlechtere Erkennung als bisher vermutet wird. Tabelle 4.17 zeigt die Resultate.

Der erste Durchlauf spricht für ein Erraten der Gesten, während die weiteren deutlich schlechtere Ergebnisse liefern. Dies deutet eine kontrastabhängige Ermittlung der Features an. Beim Versuch, dem entgegen zu wirken wurde der Trainingsdatensatz auf 3559 Bilder erweitert, wobei die Neuen alle mit dem geringen Kontrast aufgenommen wurden. Mit einer Verteilung von zwei zu eins sind dafür deutlich weniger Beispiele vorhanden. Daher wurden vier Versuche durchgeführt, zweimal den neuen Testdatensatz mit dem niedrigen und zweimal den alten mit dem hohen Kontrast. Die Ergebnisse sind in Tabelle 4.18 aufgeführt.

Diese Versuche zeigen einmal mehr, wie unterschiedlich die Ergebnisse ausfallen können, wenn nur die Kantengewichte mit anderen Werten initialisiert werden. Auch spiegeln diese Ergebnisse die Erwartungen wider. Die Testdaten mit dem hohen Kontrast wurden deutlich besser erkannt als jene mit dem geringen Kontrast. Für letzteres wurde das trainierte Netz noch einmal neu geladen und weiter trainiert. Die Fehlerrate in den Testdaten hat sich bis auf

12.10% verbessert, obwohl die Trainingsdaten bereits alle korrekt erkannt wurden.

Dass sich das Ergebnis überhaupt noch verbessern kann, liegt in der Struktur des Netzes. Jede Klasse passt mit einer bestimmten Wahrscheinlichkeit zu einem Eingabebild. Die Klasse mit der höchsten Wahrscheinlichkeit wird als Ergebnis betrachtet. Bei drei Klassen reicht also schon eine Verteilung von 34/33/33 aus, um ein korrektes Ergebnis zu erzielen. Mit jedem weiteren Durchgang werden die Kantengewichte so verändert, dass ein Bild möglichst zu 100% korrekt klassifiziert wird.

4.7. Performance

Mit der standardmäßigen Implementierung erfasst ”LeNet5” nur die Gesamtdauer von Training und Testen. Damit lassen sich zwar Aussagen über die Lernperformance treffen, allerdings nicht über die Klassifikationsgeschwindigkeit. Diese lässt sich ermitteln, indem ein Testvorgang manuell gestartet wird und dabei die Laufzeit ermittelt wird. Aus der Größe des Datensatzes und die Dauer lässt sich dann errechnen, wie lange für ein einziges Bild bzw. Geste gebraucht wird.

Auf einem Core i5 mit 3.4GHz benötigt die Klassifikation eines Bildes etwa 8.9ms. Das entspricht einem Durchsatz von etwa 112 Frames pro Sekunde. Diese Geschwindigkeit sollte für eine praktische Anwendung reichen, auch wenn etwas Vorverarbeitung notwendig sein wird, um ausreichend kleine Bilder als Eingabe für das Netz zu bekommen. Da eine der Ziele eine gute Performance auf weniger starke Hardware ist, wird zum Vergleich die Laufzeit auf einem etwas älteren Pentium Dualcore mit 1.3GHz gemessen. Dort wird eine Dauer von 38.7ms pro Bild ermittelt. Dies bedeutet einen Durchsatz von etwa 25 Frames pro Sekunde. Zwar würde das theoretisch für eine praktische Anwendung noch ausreichen, allerdings kommt da noch eine Vorverarbeitung zum Verkleinern des Inputs hinzu. Hier ist also eine schnelle, effiziente Vorverarbeitung wünschenswert um eine tatsächlich brauchbare Performance zu erreichen.

5. Zusammenfassung

Zur Erkennung von einfachen Handgesten wurden in dieser Arbeit Faltungsnetze verwendet. Diese stellen eine besondere Form von neuronalen Netzen dar, welche sich gut zur Erkennung von Mustern in Bildern eignen. Zuerst wurden keine Einschränkungen der Gesten vorgenommen. So mussten 32 Möglichkeiten der Anzahl gezeigter Finger in sechs Klassen untergebracht werden. Bei den vergleichsweise kleinen Datensätzen hat sich das als nicht praktikabel herausgestellt. Da mit immer größeren Datensätzen immer bessere Ergebnisse erzielt wurden, wird angenommen, dass mit ausreichend großen Datensätzen auch eine Unterscheidung aller 32 Möglichkeiten mit guten Erkennungsraten erreicht werden kann.

Auch hat sich herausgestellt, dass eine saubere Klassifikation sehr wichtig ist. Sobald verschwommene Bilder verwendet werden, steigt die Fehlerrate. Gleiches gilt für den Fall, dass mehrere Interpretationen einer gezeigten Geste möglich sind.

Nachdem die Anzahl Gesten auf drei reduziert wurden, wurden sehr gute Resultate erzielt. Das hat die Grenzen eines Computers deutlich aufgezeigt. Während wir Menschen schon mit wenigen Beispielen gut klassifizieren können, brauchen Maschinen wesentlich mehr. Auch die weiteren Experimente zeigen das deutlich. Eine einfache Spiegelung stellt schon eine große Hürde dar, welche explizit angelernt werden muss. Gleiches gilt auch für unterschiedliche Kontraste.

Ein weiteres Resultat dieser Arbeit ist die Erkenntnis, dass eine Gleichverteilung der Trainingsdaten wichtig ist. Das hat das Experiment mit den unterschiedlichen Kontrasten deutlich gemacht. Dort waren weniger Beispiele mit geringem Kontrast vorhanden. Analog dazu waren die Fehlerraten bei dem Testdatensatz mit solchen Beispielen deutlich höher.

6. Ausblick

6.1. Weitere Experimente

Diese Arbeit bietet noch einige Möglichkeiten für weitere Experimente, beispielsweise mit weiteren Kontrasten. Durch die Verwendung unterschiedlicher Kontraste sind robustere Netze zu erwarten. Auch andere Hintergründe, welche nicht nur aus einer Farbe bestehen, eignen sich dafür. Als Beispiele können tatsächliche Arbeitsplätze dienen, wo ein solches System zur Erkennung von Handgesten in Frage kommen würden. Eine weitere Idee wäre das Verwenden von Aufnahmen, wo Personen unterschiedliche Handschuhe tragen. Auch dies würde deutlich realistischere Bedingungen bedeuten und vermutlich zu deutlich robusteren Netzen führen.

Für solche weiteren Experimenten sind wesentlich größere Datensätze erforderlich. Um diese zu sammeln wäre es von Vorteil, wenn sich mehrere Personen daran beteiligen. Zum Einen vereinfacht und beschleunigt das das Datensammeln, zum Anderen bewegen sich Menschen individuell, was am Ende zu besseren Ergebnissen führen könnte. Gleichzeitig ist dabei eine Erweiterung des Gestensatzes denkbar. Anstatt drei Gesten z. B. fünf, oder noch mehr.

Ein anderer Ansatz könnte die Aufteilung der Klassifikation auf mehrere Netze sein. Dazu müssen zuerst mehrere Gruppen ermittelt werden, welche übergeordnete Eigenschaften von bestimmten Gesten zugeordnet bekommen. Diese werden von einem Netz klassifiziert. Anschließend werden diese an Subnetze weitergeleitet, welche die Klassifikation der einzelnen Gruppen auf unterster Ebene durchführt. Da sich bisher die Klassifikation von einigen wenigen Klassen als deutlich einfacher erwiesen hat, könnte dies dazu beitragen, dass sich auch viele, komplexere Gesten besser unterscheiden lassen.

6.2. Kombination mit anderen Ansätzen

Eine Kombination mit weiteren Ansätzen aus bisherigen Arbeiten klingt ebenso vielversprechend. Denkbar wäre die Verwendung einer Vorverarbeitung wie z. B. in Nagi u. a. (2011). Dazu

müsste dann ein System oder eine Methode gefunden werden, mit der sich die Hand im Bild lokalisieren lässt, ohne nach bestimmten Farben zu suchen. Hierzu könnten sich Verfahren zu Bewegungsdetektion in einer Abfolge von Bildern eignen. Sobald die Position der Hand erkannt wurde, kann in der Vorverarbeitung der Bildausschnitt verkleinert werden, was zu einer besseren Performance des Faltungsnetzes führen sollte.

Eine weitere Idee ist die Kombination mit einer Herangehensweise wie z. B. eine dreidimensionale Erkennung der Hand. Das könnte die Erkennung der Handposition im Bild deutlich vereinfachen. Auch dort könnte dann wieder mit einer Vorverarbeitung experimentiert werden.

A. Anhang

A.1. Lush Codebeispiele

Um einen kleinen Einblick in Lush zu bekommen folgen einige Code-Beispiele mit Erklärung. Wie in Lisp besteht auch in Lush alles aus Listen. Diese werden mittels Klammern erstellt. Dabei versucht Lush diese Listen standardmäßig zu evaluieren. Der erste Eintrag einer Liste ist ein Operator oder eine Funktion. Die weiteren Einträge sind die Parameter für diesen Operator oder diese Funktion.

```
1 (setq var1 2)
2 (setq var2 "Lush code")
3 (setq var3 '(1 2 3))
4 (+ 2 3)
5 (* 2 3 4 5)
6 (setq var4 (- 20 3))
```

Listing A.1: Variablen und Operatoren

Um in Lush Variabeln zu setzen gibt es die Funktion ”setq”. Als Parameter werden Variablenname und Wert übergeben. Dabei kann es sich um Zahlen, Strings, Listen, Matrizen, Objekte oder Pointer handeln. In Zeile eins wird die Variable ”var1” mit der Zahl zwei belegt. Die nächsten beiden Zeilen zeigen eine Belegung mit einem String bzw. einer Liste. Das ' vor der Liste sorgt dafür, dass diese Liste nicht evaluiert wird. Zeilen vier und fünf zeigen einfache mathematischen Operationen. Dabei kann man mehr als zwei Werte übergeben. Die gängige Schreibweise für Zeile fünf lautet $2 * 3 * 4 * 5$. In der letzten Zeile wird eine Liste als Wert übergeben. Da hier allerdings kein ' vorangestellt ist, wird diese Liste evaluiert und das letzte Ergebnis dieser Evaluierung in der Variable var4 gespeichert.

```
(setq testmatrix (matrix 4 4))
(setq testmatrix [[1 2 3 4][3 4 5 6][9 8 7 6][7 6 5 4]])
(testmatrix 0 0)
(testmatrix 0 0 5)
(select testmatrix 1 2)
```

Listing A.2: Matrizen

Dieses Beispiel zeigt einige Matrizenoperationen. In der ersten Zeile wird eine 4*4 Matrix angelegt. Initialisiert wird sie mit null (als Zahl, Typ double). Alternativ kann man eine Matrix gleich mit Werten belegen, wie Zeile zwei zeigt. Auslesen und beschreiben eines bestimmten Wertes wird wie in Zeilen drei und vier gemacht. Die letzte Zeile zeigt eine der vielen praktischen Matrixoperationen. Aus ”testmatrix” wird die Dimension mit dem Index eins weggelassen. Das Ergebnis ist ein Vektor. Welches Teil aus der weggelassenen Dimension ausgewählt wird bestimmt der letzte Parameter. In diesem Beispiel ist das die dritte Spalte, also der Vektor (3 5 7 5).

```
(setq z (new v4l2device "/dev/video0" 640 480 10 4))
(when (not window) (new-window 0 0 660 540 "window title"))
(==> z get-frame-grey frame-grey)
(rgb-draw-matrix 10 10 frame-grey)
```

Listing A.3: Objekte

Dieses Beispiel zeigt die Verwendung einer Webcam. Dazu wird ein neues ”v4l2device” Objekt angelegt. Als Parameter werden die Kamera, die Auflösung, die Framerate und die Anzahl Framebuffer übergeben. Die zweite Zeile erstellt ein neues Fenster, sofern noch keins vorhanden ist. Eine Methode auf einem Objekt aufrufen erfolgt wie in Zeile drei dargestellt. ”==>” bedeutet sovie wie ”Sende Objekt eine Nachricht”. Es folgen das Objekt, die Methode und die Parameter. In diesem Fall wird dem ”v4l2device” Objekt angewiesen ein Graustufenbild aufzunehmen und in der Variable ”frame-grey”, eine Matrix, zu speichern. Die letzte Zeile zeichnet das gerade aufgenommene Bild in dem zuvor erstellten Fenster.

A.2. Gesten

Eine Auflistung aller 32 möglichen Gesten von null bis fünf Finger. Da man einen Finger entweder zeigen oder nicht zeigen kann, entspricht das einem binären System, also 2^n Möglichkeiten, wobei n für die Anzahl Finger steht.

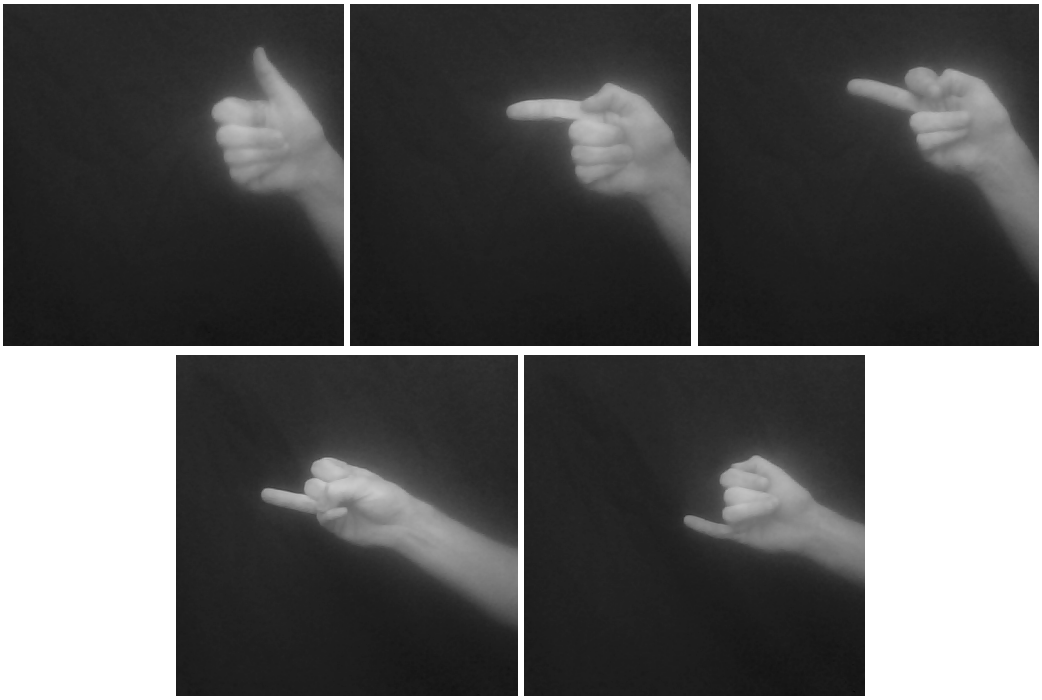


Abbildung A.1.: Möglichkeiten mit einem Finger

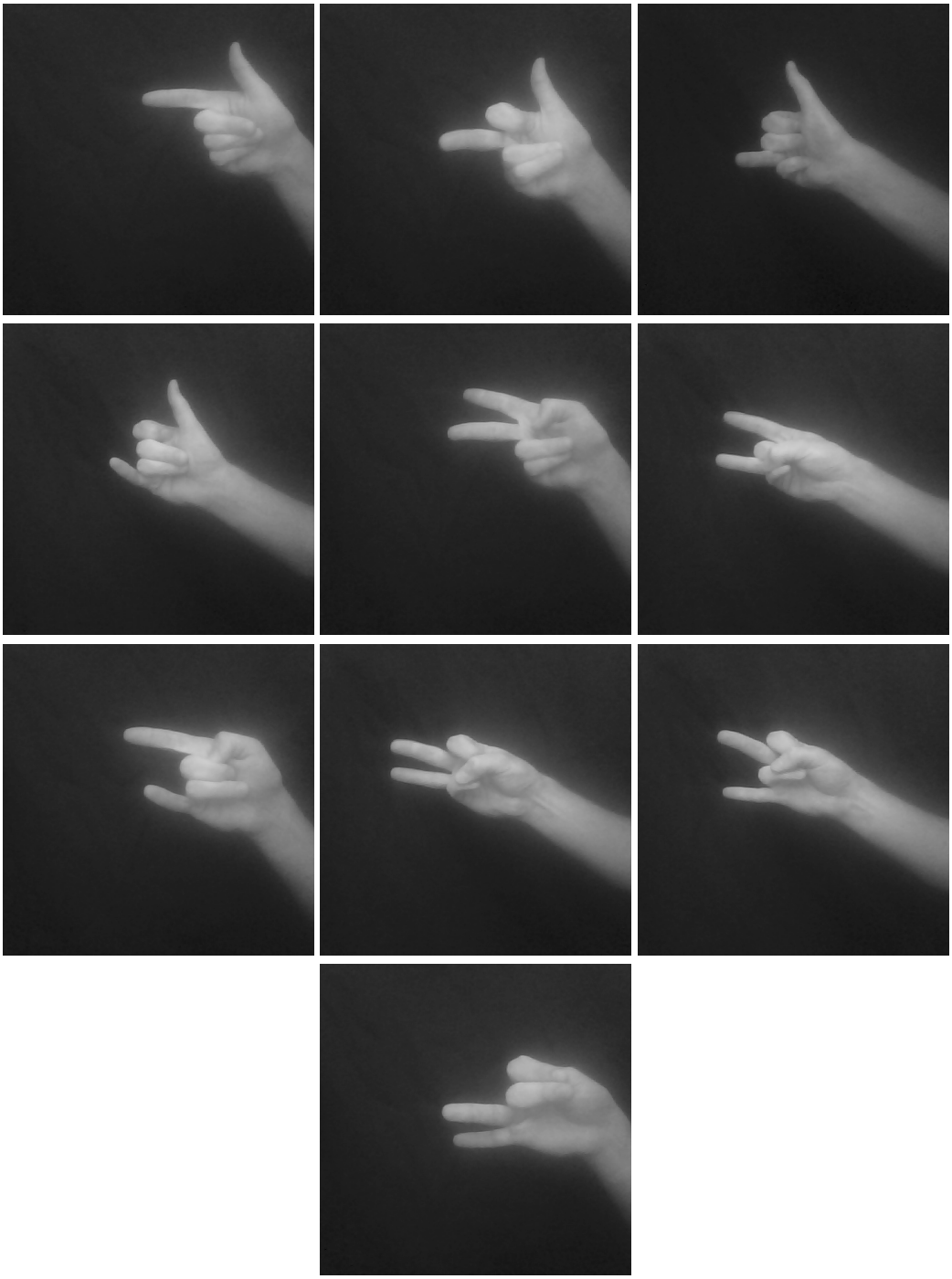


Abbildung A.2.: Möglichkeiten mit zwei Finger

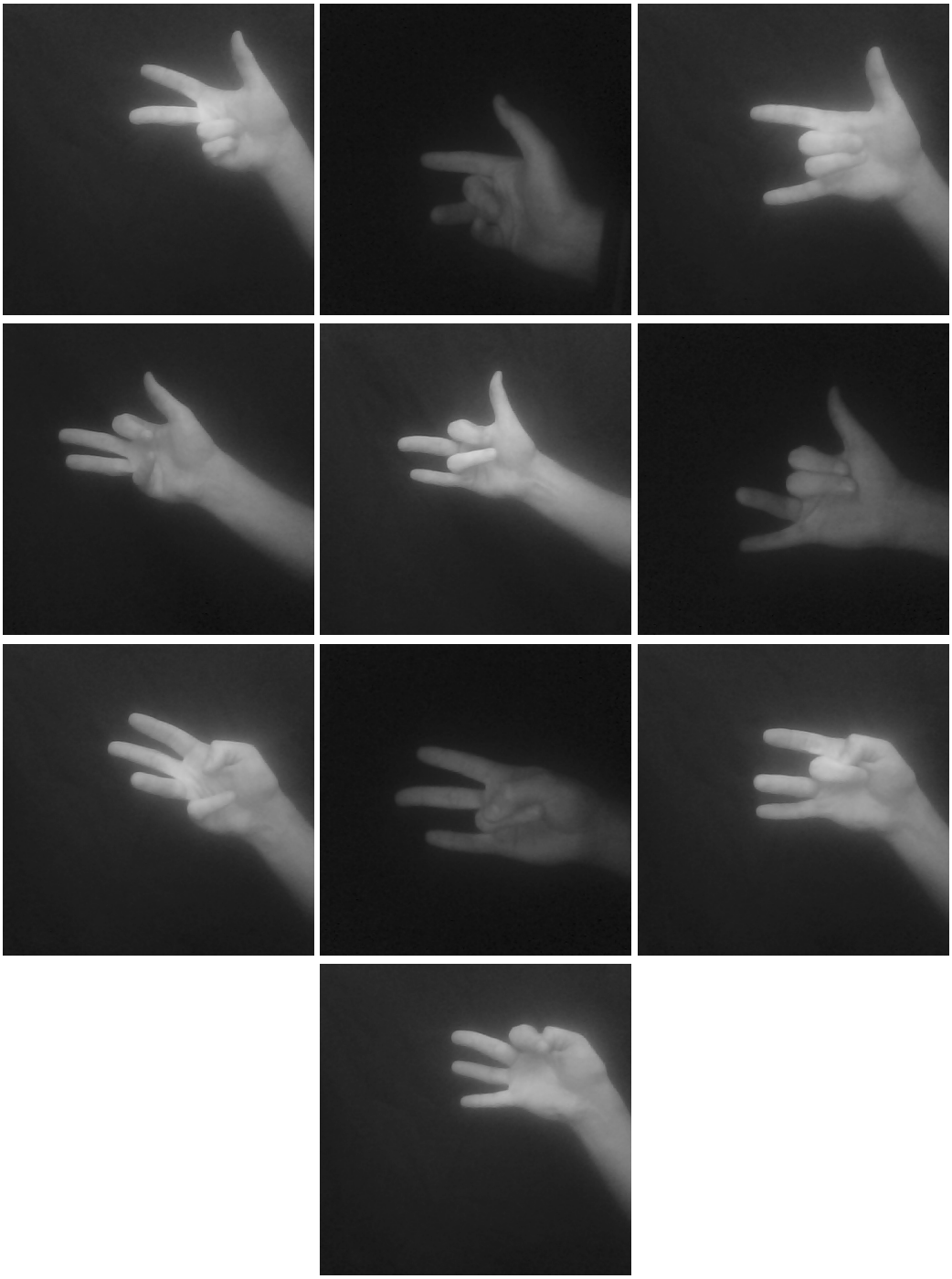


Abbildung A.3.: Möglichkeiten mit drei Finger

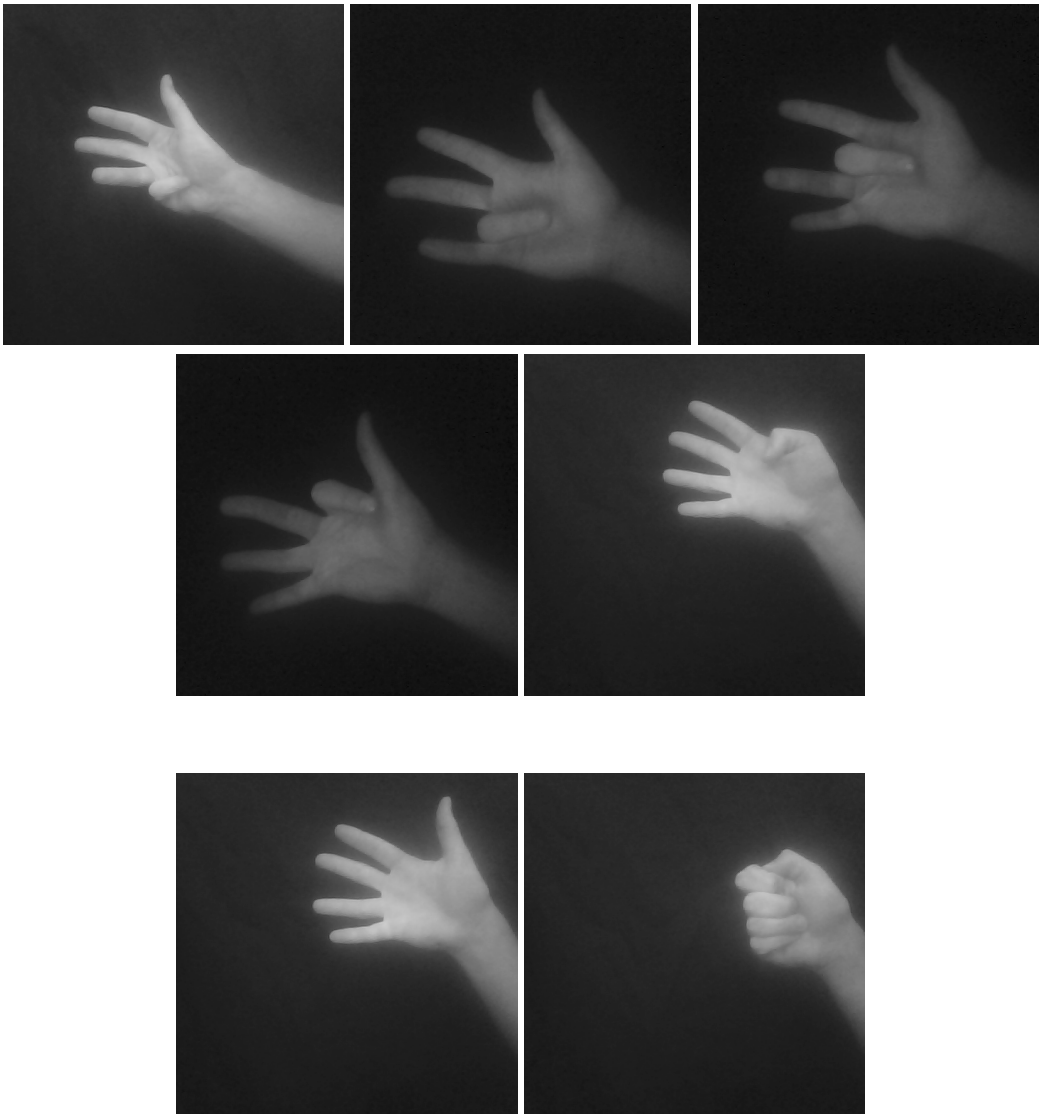


Abbildung A.4.: Möglichkeiten mit vier, fünf und null Finger

Literaturverzeichnis

[einfuehrungnn] *Einführung in Neuronale Netze.* – URL https://web.archive.org/web/20010515082546/http://wwwmath.uni-muenster.de/SoftComputing/lehre/material/wwwnnscript/startseite.html. – (Version vom 15. Mai 2001 im Internet Archive)

[faltungsmatrix] *GIMP Dokumentation, Faltungsmatrix.* – URL http://docs.gimp.org/de/plug-in-convmatrix.html. – Zugriff: 24.09.2014

[lushhp] *Lush Homepage.* – URL http://lush.sourceforge.net/. – Zugriff: 16.09.2014

[Bottou und LeCun] Bottou, Leon ; LeCun, Yann: *Lush Dokumentation.* – URL http://lush.sourceforge.net/doc.html. – Zugriff: 26.09.2014

[Huang und LeCun 2006] Huang, Fu-Jie ; LeCun, Yann: Large-Scale Learning with SVM and Convolutional Nets for Generic Object Categorization. In: *Proc. Computer Vision and Pattern Recognition Conference (CVPR'06)*, IEEE Press, 2006

[Kim u. a. 2008] Kim, Ho-Joon ; Lee, Joseph S. ; Park, Jin-Hui: Dynamic hand gesture recognition using a CNN model with 3D receptive fields. In: *2008 IEEE International Conference Neural Networks and Signal Processing*, 2008

[Konda u. a. 2012] Konda, Kishore ; Königs, Achim ; Schulz, Hannes ; Schulz, Dirk: *Real Time Interaction with Mobile Robots using Hand Gestures.* 2012

[Kriesel 2007] Kriesel, David: *Ein kleiner Überblick über Neuronale Netze.* URL http://www.dkriesel.com, 2007

[Krizhevsky u. a. 2012] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: *ImageNet Classification with Deep Convolutional Neural Networks.* 2012

[LeCun und Bengio 1995] LeCun, Y. ; Bengio, Y.: Convolutional Networks for Images, Speech, and Time-Series. In: Arbib, M. A. (Hrsg.): *The Handbook of Brain Theory and Neural Networks*, MIT Press, 1995

[LeCun u. a. 1998] LeCun, Y. ; Bottou, L. ; Bengio, Y. ; Haffner, P.: Gradient-Based Learning Applied to Document Recognition. In: *Proceedings of the IEEE* 86 (1998), November, Nr. 11, S. 2278–2324

[LeCun] LeCun, Yann: *LeNet-5, convolutional neural networks*. – URL `http://yann.lecun.com/exdb/lenet/index.html`. – Zugriff: 23.09.2014

[LeCun u. a. 2004] LeCun, Yann ; Huang, Fu-Jie ; Bottou, Leon: Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting. In: *Proceedings of CVPR'04*, IEEE Press, 2004

[LeCun u. a. 2010] LeCun, Yann ; Kavukvuoglu, Koray ; Farabet, Clément: Convolutional Networks and Applications in Vision. In: *Proc. International Symposium on Circuits and Systems (ISCAS'10)*, IEEE, 2010

[MacKay 2003] MacKay, David: *Information Theory, Inference and Learning Algorithms*. S. 467–554, URL `http://www.inference.phy.cam.ac.uk/mackay/itprnn/book.html`, 2003

[Meisel 2012] Meisel, Andreas: *Robot Vision Vorlesungsscript*. 2012

[Mirowski u. a. 2008] Mirowski, Piotr ; LeCun, Yann ; Madhavan, Deepak ; Kuzniecky, Ruben: Comparing SVM and Convolutional Networks for Epileptic Seizure Prediction from Intracranial EEG. In: *Proc. Machine Learning and Signal Processing (MLSP'08)*, IEEE, 2008

[Nagi u. a. 2012] Nagi, Jawad ; Caro, Gianni A. D. ; Giusti, Alessandro ; Nagi, Farrukh ; Gambardella, Luca M.: Convolutional Neural Support Vector Machines: Hybrid Visual Pattern Classifiers for Multi-robot Systems. In: *2012 IEEE International Conference on Machine Learning and Applications*, 2012

[Nagi u. a. 2011] Nagi, Jawad ; Ducatelle, Frederick ; Caro, Gianni A. D. ; Ciresan, Dan ; Meier, Ueli ; Giusti, Alessandro ; Nagi, Farrukh ; Schmidhuber, Jürgen ; Gambardella, Luca M.: Max-Pooling Convolutional Neural Networks for Vision-based Hand Gesture Recognition. In: *2011 IEEE International Conference on Signal and Image Processing Applications (ICSIPA2011)*, 2011

[Platt und Nowlan 1995] Platt, John ; Nowlan, Steve: A Convolutional Neural Network Hand Tracker, Neural Information Processing Systems Foundation, January 1995. – URL `http://research.microsoft.com/apps/pubs/default.aspx?id=68392`

[Rey und Beck] Rey, Günter D. ; Beck, Fabian: *Neuronale Netze, Eine Einführung*. – URL `http://neuronalesnetz.de/`. – Zugriff: 16.09.2014

[Sermanet u. a. 2012] Sermanet, Pierre ; Chintala, Soumith ; LeCun, Yann: Convolutional Neural Networks Applied to House Numbers Digit Classification. In: *International Conference on Pattern Recognition (ICPR 2012)*, 2012

[Simard u. a. 1999] Simard, P. ; Bottou, L. ; Haffner, P. ; LeCun, Y.: Boxlets: a fast convolution algorithm for neural networks and signal processing. In: *Advances in Neural Information Processing Systems (NIPS 1998)* Bd. 11, MIT Press, 1999

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

Hamburg, 01. Oktober 2014 Marten Boessenkool