BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

FB VI: Informatik und Medien

Skizzenerkennung mit tiefen neuronalen Netzen

Autor:
Hagen Tarner
Sophienstraße 4
10317 Berlin

Matrikelnummer 833592

Betreuer:
Prof. Dr.-Ing. Kristian Hildebrand

Gutachter:
Prof. Dr.-Ing. Hartmut Schirmacher

Zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

im Studiengang Medieninformatik.

31.03.2017

Abstract

Zur visuellen Kommunikation behelfen sich Menschen seit prähistorischen Zeiten einfacher Strichzeichnungen. Diese sind universal und von jeglicher Sprache losgelöst verständlich. Wurden diese früher auf Höhlenwände und später auf Papier gezeichnet, verwendet der moderne Mensch ein Gerät mit Touchscreen.

Um die gezeichnete Information maschinenlesbar und damit weiterverarbeitbar, zu machen ist eine Software zur Erkennung von Skizzen nötig. Die Autoren Eitz, Hays und Alexa haben 2012 einen Datensatz veröffentlicht, der es ermöglicht eine Maschine so zu trainieren, dass diese in späteren Arbeiten Erkennungsraten liefert, die die eines Menschen übertreffen.

Die vorliegende Arbeit nimmt Ansätze aus dem aktuellen Forschungsgebiet des *deep learnings*, um die Erkennung von ebendiesen Skizzen zu erhöhen. Die höchste so erzielte Erkennungsrate von 80,22% übertrifft dabei die gemessene Leistung eines Menschen (73,10%) deutlich.

Im Rahmen dieser Arbeit sind drei verschiedene neuronale Netze zur Skizzenerkennung und eine Anwendung um mit diesen zu interagieren entstanden.

Eidesstattliche Erklärung

Ich versichere, dass ich meine Abschlussarbeit selbstständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt habe.

Unterschrift: _____

Datum: _____

Inhaltsverzeichnis

Kapitel 1

Einleitung

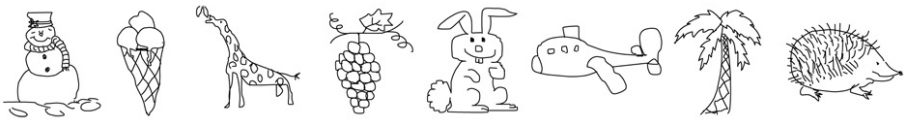


Abbildung 1.1: Einige der im Datensatz enthaltenen Skizzen [13].

Zur visuellen Kommunikation behelfen sich Menschen seit prähistorischen Zeiten einfacher Strichzeichnungen. Diese sind universal und von jeglicher Sprache losgelöst verständlich. Wurden diese früher auf Höhlenwände und später auf Papier gezeichnet, verwendet der moderne Mensch ein Gerät mit Touchscreen.

Die Autoren Eitz, Hays und Alexa haben 2012 einen Datensatz veröffentlicht [13], welcher 20.000 Skizzen in 250 verschiedenen Kategorien umfasst. Die Abbildung 1.1 zeigt beispielhaft einige der enthaltenen Skizzen. Eine Skizze ist schwarz/weiß und wird freihändig auf ein geeignetes Display gezeichnet. Im Gegensatz zu Fotos bieten Skizzen sehr wenig Details und bestehen aus genau zwei Farben: weißer Hintergrund und schwarze Strichfarbe.

Um die gezeichnete Information maschinenlesbar und damit weiterverarbeitbar, zu machen ist eine Software zur Erkennung von Skizzen nötig. Gerade für das Erkennen und computergestützte Verarbeiten von visuellen Informationen bietet das Forschungsgebiet des Maschinellen Lernens passende Ansätze. Genau genommen ist maschinelles Lernen ein Untergebiet des Forschungsfeldes der künstlichen Intelligenz. Kernziel ist es nach der Beendigung einer Lernphase *erlerntes* Wissen innerhalb einer Domäne zu verallgemeinern.

Maschinelles Lernen findet aktuell nur in Expertensystemen Anwendung. Also Systemen, die für einen speziellen Fall erschaffen und trainiert werden und deren Wissen sich nicht auf andere Domänen übertragen lässt. Es ist zum aktuellen Zeitpunkt also nicht möglich mit Techniken des maschinellen Lernens so etwas wie eine universelle Intelligenz zu schaffen, die jegliche Klasse von Problemen lösen könnte.

Dennoch ist es beeindruckend, welche Leistungen aktuelle Machine Learning Systeme zu leisten im Stande sind. Wurde der Schachweltmeister Garri Kasparow bereits 1997 von IBM's Deep Blue geschlagen, ist dies 2016 einer Maschine in Go gelungen (Googleś AlphaGo gegen Lee Sedol). Maschinen können heutzutage Autos fahren und sprechen [2]. Hierbei gilt allerdings auch wieder die Einschränkung aus dem letzten Paragrafen: diese Leistungen sind Ergebnisse vieler einzelner Systeme und nicht die eines einzelnen, *intelligenten* Systems.

Ziel der vorliegenden Arbeit ist es mittels aktueller Techniken des deep learnings ein System zu entwickeln, das bei der Erkennung von Skizzen eine Erkennungsrate aufweist, die höher als die eines Menschen ist. Dafür werden entsprechende Netzarchitekturen evaluiert und anschließend mit dem Datensatz der Autoren Eitz, Hays und Alexa trainiert.

1.1 Mögliche Anwendungen für die Skizzenerkennung

Die Skizze als Eingabemedium ist vielseitig einsetzbar. Immer dann, wenn geschriebene oder gesprochene Sprache keine Mittel zur Kommunikation sind, können Sketche als Vereinfachung dienen. Als Eingabe für ein Suchfeld in einem großen Datenbestand wäre dies denkbar. Möchte man beispielsweise das Angebot eines Schuhhändlers durchsuchen, weiß aber nicht, wie die gesuchten Schuhe heißen kann man den Umriss zeichnen, wenige Details hinzufügen und findet den gesuchten Artikel. Ein weiterer Einsatzzweck wäre die Übersetzung. Fehlt einem beispielsweise das englische Wort für *Ananas*, zeichnet man einfach ebendiese in die Anwendung, wählt *Englisch* als Zielsprache aus und erhält *pineapple*.

Die vorgestellten Netze sind allesamt auf die Erkennung von Skizzen optimiert und können auf andere Domänen (z.B. das Erkennen von Straßenschildern in Fotos) nicht angewendet werden. Ein denkbarer Ansatz dies zu ändern wäre nicht die vorgestellten Netze kompatibler zu machen, sondern die Eingabedaten zu transformieren. Wenn man es beispielsweise schafft die zu erkennenden Merkmale eines Fotos dahingehend zu extrahieren, dass sie den hier gezeigten Eingabedaten ähneln (Reduktion der Farbinformationen, betonen der Kanten) ist eine hohe Erkennungsrate mit den hier gezeigten Architekturen denkbar.

1.2 Bisherige Arbeiten zur Skizzenerkennung

Die maschinelle Auswertung händisch gezeichneter Skizzen von einfachen Gegenständen oder Alltagsobjekten ist aktives Forschungsgebiet. Beispielsweise werden diese als Eingaben zur Erzeugung von Foto-Collagen [14] oder zur

Erzeugung von Eingaben in Suchfeldern für Bildsuchen (*sketch-based image retrieval*) [15, 16] genutzt. Auch das Erstellen von dreidimensionalen Volumen anhand von Skizzen [44] ist ein Beispiel, wie diese genutzt und erforscht werden.

Diese Arbeiten nehmen jeweils eigens entworfene, sehr kleine Datensätze zur Grundlage.

Erhebung des Datensatzes und Erkennung mit linearen Klassifizierern

Eitz, Hays und Alexa [13] veröffentlichen 2012 den dieser Arbeit zu Grunde liegenden Datensatz (im Folgenden auch *TU Datensatz* genannt). Wie die Kategorien und Skizzen entstanden sind, ist Teil von Kapitel 3.2. Dies stellt die erste Veröffentlichung eines umfassend erhobenen Datensatzes von Skizzen in einer wissenschaftlichen Veröffentlichung dar.
Die Autoren nutzen zur Erkennung der Skizzen in ihrer Arbeit eine Support Vector Machine (SVM). Eine SVM ist ein linearer Klassifikator aus dem Bereich des überwachten Lernens. Damit die SVM die Skizzen klassifizieren kann, ist eine formale Beschreibung jedes einzelnen nötig. Für diesen Zweck entwerfen die Autoren einen Beschreibungsvektor und trainieren damit den Klassifikator.

Schneider und Tuytelaars [45] nutzen ebenfalls eigens erstellte Featuredeskriptoren und verbessern die Erkennungsrate auf insgesamt 68,9%. Damit erreichen sie fast die Klassifikationsleistung eines Menschen. Diese wird von Eitz, Hays und Alexa für den vorliegenden Datensatz mit 73,10% beziffert (3.2).

Die Autoren schlagen außerdem eine Revision des Datensatzes vor, da die ursprüngliche Version zu ähnliche Klassen und einige unverständliche Zeichnungen besäße (siehe hierzu auch Kapitel 4.2.1, welches auf die Probleme mit dem Datensatz genauer eingeht). Der reduzierte Datensatz umfasst lediglich noch 160 Klassen mit mindestens 56 Zeichnungen pro Klasse.

Schneider und Tuytelaars sind die ersten, die die Reihenfolge der Striche innerhalb einer Skizze eingehen. Sie identifizieren den ersten Strich jedes Sketches als längsten und für die Erkennung wichtigsten. Diese Erkenntnis wird in der vorliegenden Arbeit in der Generierung der Vollständigkeitsstufen eines Sketches von großer Bedeutung sein.

Erkennung mit Convolutional Neural Network (CNN) und Recurrent Neural Network (RNN)

Lineare Klassifikatoren liefern eine niedrige Erkennungsrate für den TU Datensatz. Eine weitaus höhere Erkennungsrate bieten hier aktuell CNNs. Gängige CNNs sind für die Erkennung von Objekten in Fotos optimiert. Da sich die

vorliegenden Skizzen jedoch stark von Fotos unterscheiden liefern diese Netzwerke keine zufriedenstellenden Resultate. Ein speziell für diesen Verwendungszweck erstelltes Netzwerk ist also notwendig.

Sarvadevabhatla und Babu [43] setzen als erste CNNs zur Erkennung von Skizzen ein. Ihr Ansatz beschränkt sich darauf vortrainierte Netze (AlexNet und LeNet) zu verwenden. Da diese jedoch für den Einsatz mit Fotos konzipiert wurden bleibt die Erkennungsrate niedrig. Die Neuerung in dieser Arbeit liegt darin, dass die Autoren diese nicht mit dem Ziel der Erkennung von Skizzen, sondern mit der Suche in einem Datenbestand verfasst haben. Dazu trainieren sie die Netze erst und nehmen deren Output nicht zur Klassifikation, sondern zur Erzeugung von 4096 dimensionalen Featurevektoren, die danach weiterverarbeitet werden.

Yang und Hospedales [56] konstruieren ihr Netzwerk *Sketch-a-Net* speziell für die Skizzenerkennung. Um die Erkennungsrate noch weiter zu verbessern werden außerdem zwei verschiedene Arten der Aufbereitung vor dem eigentlichen Training des Netzwerks verwendet. Die Autoren deformieren und entfernen einzelne Striche aus den Zeichnungen. So erreichen sie im Endeffekt eine Erkennungsrate von 77,06%, die die eines Menschen um etwa 4% übertrifft.

Seddati, Dupont und Mahmoudi [47] entwickeln 2015 DeepSketch. Das Netzwerk erreicht mit 77,69% die bis dato höchste Erkennungsrate auf dem TU Datensatz. Die Autoren greifen die Arbeit von Sarvadevabhatla und Babu auf und verbessern auch das *sketch-based image retrieval*. Hierzu extrahieren die Autoren die Features des letzten Convolutional Layers und benutzen diese als Eingabedaten für eine Ähnlichkeitssuche.

Seddati, Dupont und Mahmoudi [46] veröffentlichen 2016 eine Erweiterung von DeepSketch mit der es möglich ist Skizzen während der Entstehung bereits zu Erkennen. Hierzu untersuchen sie die Möglichkeit Sketche in Stufen verschiedener Vollständigkeit zu zerlegen, um so zeitlich basierte Sequenzinformationen zu generieren. Diese dienen danach als Eingabedaten für ein Long short-term memory (LSTM) Netz.

Kapitel 2

Grundlagen

Häufig werden die Begriffe *Maschinelles Lernen* und *künstliche Intelligenz* in der Nichtfachliteratur synonym benutzt, wenn es darum geht neue Forschungsergebnisse zu beschreiben. Die Vorstellung von einer Maschine mit menschenähnlichem Verstand ist ein aus Literatur und Film bekanntes Motiv und häufig aufgegriffenes Thema.

Klassifikation und Regression

Maschinelles Lernen ist im Grunde das Erkennen von Mustern in einem Datenbestand, der für einen einzelnen Menschen auf Grund seiner Größe nicht mehr verarbeitbar ist. In Zeiten, in denen Webseiten Petabytes an Daten zur Verfügung haben[1], sind effektive Methoden um Rückschlüsse und Vorhersagen aus diesen ziehen zu können notwendig. Maschinelles Lernen hat hierbei zwei Haupteinsatzzwecke: Klassifikation und Regression.

Als Klassifikation wird die Einteilung von Objekten anhand bestimmter übereinstimmender Merkmale bezeichnet. Jedes Datum ist dabei einer Klasse (auch *label*) zuzuordnen. Ziel dabei ist es Daten, die nicht dem originalen Datenbestand angehörig sind, in die existierenden Klassen einzusortieren. Beispielsweise die Entscheidung, ob eine eingetroffene E-Mail *spam* oder *ham* ist, ist ein typisches Klassifizierungsproblem. Hierbei wäre die E-Mail das Datum und *ham* und *spam* die Klassen oder *labels*.

Soll aus einem bestehenden Datenbestand und einer Menge dazugehöriger Klassen ein neues Datum prognostiziert werden spricht man von Regression. Ist beispielsweise die mittlere Miete für eine Wohnung über einen Zeitraum bekannt, kann versucht werden den Mietpreis für Zeiträume in der Zukunft vorherzusagen.

[1] Facebook gibt an in 2014 etwa 4 PB [55] an Daten pro Tag zu generien.

Überwachtes und unüberwachtes Lernen

Grundsätzlich unterscheidet man Machine Learning Ansätze in zwei Kategorien: überwachtes (engl.: *supervised*) und unüberwachtes Lernen (engl.: *unsupervised Learning*). Beim überwachten Lernen gibt es einen *Lehrer*, der dem Algorithmus während einer Trainingsphase eine Kombination aus Datum und erwünschter Ausgabe zeigt. Der Algorithmus probiert daraufhin die Assoziation zwischen Ein- und Ausgabe zu formalisieren und im Anschluss auf unbekannte Daten anwenden zu können. Die in dieser Arbeit vorgestellten künstlichen neuronalen Netze, CNNs und RNNs sind allesamt Verfahren des überwachten Lernens. Auch die von Eitz, Hays und Alexa benutzte SVM ist ein Beispiel für supervised Learning.

Das unüberwachte Lernen unterscheidet sich hierbei, da es keinen Lehrer gibt, der erwartete Ausgaben zu jeder Eingabe präsentiert. Methoden des unüberwachten Lernens finden also die Assoziationen von Ein- und Ausgabe in einem Datenbestand selbstständig. Typische Anwendungsfälle sind die Segmentierung (engl.: *clustering*) von Daten oder die Komprimierung zur Dimensionsreduktion.

2.1 Zur Historie des Maschinellen Lernens

Die Anfänge

Die Anfänge des maschinellen Lernens lassen sich bis in die vierziger Jahre des letzten Jahrhunderts nachverfolgen. McCulloch und Pitts erschaffen 1943 das erste Modell eines künstlichen Neurons: die McCulloch-Pitts-Zelle [35]. Die Autoren zeigen, dass sich damit logische und arithmetische Funktionen berechnen lassen. Mit dieser Arbeit wird das Forschungsfeld der künstlichen neuronalen Netze begründet.

Ende der 1940er Jahre formuliert Hebb seine Hebbsche Lernregel [23], welche das Zustandekommen des Lernens in künstlichen neuronalen Netzwerken beschreibt. Diese Regel wird heutzutage als klassische Methodik des unüberwachten Lernens verstanden.

Die ersten Erfolge

Von Hebb beeinflusst veröffentlicht Rosenblatt 1958 die Arbeit „The perceptron: A probabilistic model for information storage and organization in the brain.“ [39] und erschafft damit die Idee von Neuronen in einer Maschine. Maschine ist hier durchaus wörtlich zu nehmen, da das *Mark I Perceptron* eine etwa schrankgroße Aparatur war, das mit 20x20 Cadmium Fotozellen

ausgestattet ein 400 Pixel großes Bild verarbeiten konnte. Die Gewichte des Mark I ließen sich nicht etwa trainieren, sondern wurden händisch über eine Reihe von Potentiometern eingestellt.

In ihrem 1969 veröffentlichten Buch *Perceptrons.* geben Minsky und Papert [36] eine genaue mathematische Analyse von Rosenblatts Perceptron. Im Laufe ihrer Arbeit kommen die Forscher zu dem Schluss, dass es fundamentale Probleme gäbe, die von den zu der Zeit aktuellen Perceptrons nicht gelöst werden können. Allen voran das Problem, dass das Perceptron nur linear separierbare Daten verarbeiten kann. Dieses Fazit leitet den ersten *AI Winter* ein.

In der Geschichte der Forschung zu künstlicher Intelligenz kommt es immer wieder dazu, dass auf Grund von Ergebnissen komplette Forschungsetats gestrichen werden. Von einem Durchbruch in der Forschung gesteigerte Interessen bei Finanziers, wie Regierungen oder Militärs, konnten nicht erfüllt werden. Während dieser Zeit geht die Zahl der Veröffentlichungen stark zurück und es wird kaum Fortschritt gemacht. Daher der Name *AI Winter* (aus dem Englischen: *AI* als Abkürzung für *Artificial Intelligence* = künstliche Intelligenz). Die beiden größten AI Winter fanden Ende der 1970er und Ende der 1980er Jahre statt.

Der Wiederaufbau nach dem ersten AI Winter

In die Zeit nach dem ersten AI Winter fallen die ersten Veröffentlichungen mehrlagiger Perzeptrons. Als Beispiel sei hier das *Neocognitron* von Fukushima [17] genannt. Als Netzwerk zur Erkennung handschriftlicher Eingaben gedacht legt es den Grundstein für CNNs.

Durch die Arbeiten von Hopfield 1985 und Rumelhart, Hinton und Williams 1986 werden Minsky und Papert widerlegt. Die Forscher zeigen mit der generalisierten Anwendung der Fehlerrückführung (engl.: *Backpropagation*), dass es möglich ist mehrschichtige Perzeptrons zu trainieren und so nicht-linear separierbare Probleme zu lösen. Die Idee der Backpropagation geht auf Werbos [54] zurück.

Lineare Klassifikation statt *deep learning*

Während der 1990er Jahre reicht die zur Verfügung stehende Rechenleistung aktueller Hardware nicht aus, um tiefe neuronale Netze effektiv zu trainieren. Dies soll sich erst Anfang der 2000er Jahre mit der Möglichkeit die Graphics processing unit (GPU) für das Training zu verwenden, ändern. Da lineare Klassifikatoren wesentlich weniger Speicher und Berechnungszeit benötigen,

sind sie zu dieser Zeit sehr populär. So veröffentlichen Cortes und Vapnik 1995 die SVM.

Währenddessen identifiziert Hochreiter 1991 *vanishing gradient* als zentrales Problem aktueller künstlicher neuronaler Netze [25]. Er beschreibt dabei den Zusammenhang zwischen den damals üblichen Aktivierungsfunktionen eines Neurons, der Tiefe des Netzwerks und der schlechten Trainingsquote der ersten Layer eines Netzwerks. 1997 lösen Hochreiter und Schmidhuber das *vanishing gradient problem* für RNNs mit der Vorstellung der LSTMs [26]. Für den Kontext der CNN ist das Problem weiterhin ungelöst, lässt sich allerdings durch die enorm gestiegenen Hardwareressourcen, die zur Berechnung bereitstehen, vernachlässigen.

Als Grundlage moderner CNNs kann die Veröffentlichung „Gradient-based learning applied to document recognition“ von LeCun u. a. [33] angesehen werden. Die Autoren kreieren darin ein System (LeNet-5 genannt; wohl ein Kofferwort aus dem Nachnamen des Autors **Le**Cun, *neural **Net*** und der Anzahl an Convolutional Layern im Netz) zur Erkennung handschriftlich erfasster Dokumente. Die Leistung dieser Arbeit liegt u. a. darin, dass diese beweist, dass CNNs fähig sind komplexe Eingaben korrekt zu verarbeiten. Bis dato erreichten nur Erkennungsmechanismen mit zuvor händisch geschaffenen Merkmalsbeschreibungen (engl.: *feature descriptors*) zufriedenstellende Resultate. Dieser Schritt fällt bei CNNs jedoch weg, da diese die Merkmale der Eingabedaten selbstständig lernen und somit wesentlich effizienter sind.

Grafikkarten ermöglichen das Training sehr tiefer neuronaler Netze

Durch das Aufkommen der Videospielindustrie Ende der 1990er Jahre kommen die ersten dedizierten Grafikkarten für Heimcomputer auf den Markt. Die für 3D-Grafik benötigten Berechnungen (hauptsächlich Vektor- und Matrizenrechnungen) sorgen für eine Spezialiserung der verbauten Chips. Da sich die Berechnungen denen des Trainings von künstlichen neuronalen Netzen stark ähneln, ist die Verwendung der GPU seitdem Stand der Forschung.

Die erste Veröffentlichung von tiefen neuronalen Netzen, die auf Grafikkarten trainiert wurden, stammt aus dem Jahr 2005 [50]. 2006 veröffentlicht die NVIDIA Corporation mit CUDA eine API für einfaches GPGPU [38]. Damit ist ein effizientes und effektives Training auch sehr tiefer Netzwerke möglich. So erreichen Cireşan, Meier und Schmidhuber 2012 mit einem CNN, das auf einer GPU trainiert wurde, als erste eine menschenähnliche Performance [5] auf dem von LeCun u. a. veröffentlichten MNIST Datensatz. Vorher hatten sie bereits mit einer ähnlichen Technik eine übermenschliche Performance bei der Erkennung von Straßenschildern gezeigt [6].

Der exponentiell gestiegen Leistung der zur Verfügung stehenden Hardware ist es zu verdanken, dass maschinelles Lernen sich wieder in einer Hochphase befindet. Das gesteigerte Interesse sorgt auch für neue, bahnbrechende Forschungsergebnisse. So veröffentlichen Hinton u. a. 2012 die Technik des *Dropouts* (siehe dazu auch Kapitel 4.1.2) und ermöglichen damit starke Generalisierung der trainierten Netzwerke [24]. Mit dieser Technik veröffentlichen sie im selben Jahr die wegweisende Arbeit „Imagenet classification with deep convolutional neural networks“ [30]. Diese Arbeit erreicht ein bis dato unerreichtes Level an Genauigkeit bei der Klassifizierung des ImageNet Datensatzes der Stanford Universität. Die Besonderheit des Datensatzes liegt dabei in dessen Größe: über 15 Millionen hochauflösende Bilder in 22.000 Kategorien [8]. Diese Arbeit verdeutlicht so, dass auch große Datensätze auf GPUs effizient zu trainieren sind.

Machine Learning als Thema in der breiten Öffentlichkeit

In 2016 ist Machine Learning als Thema in der breiten Öffentlichkeit angekommen: nahezu jede große IT-Firma entwickelt ein eigenes Framework oder unterstützt bereits veröffentlichte Open Source Software [1]. Das öffentliche Interesse an Themen des maschinellen Lernens wächst stetig. Themen wie selbstfahrende Kraftfahrzeuge, Personal Assistant Software (wie Siri oder Google Assistant) oder digitale Assistenten mit dedizierter Hardware (wie Google Home oder Amazon Echo) betreffen längst nicht mehr nur eine Forschergemeinde, sondern finden Einzug in die Agenden von Politikern, Datenschützern und Endanwendern.

Ob auf diese Phase der gesteigerten Aufmerksamkeit und die daraus resultierenden Erwartungen an die Leistungen, die maschinelles Lernen zu vollbringen vermag wieder ein AI Winter folgen wird bleibt eine offene Frage.

2.2 Lineare Klassifikation

Zur Klassifikation einfacher Datensätze eignen sich lineare Klassifikatoren (engl.: *linear classifiers*). Für die Klassifikation des TU Datensatzes benutzen Eitz, Hays und Alexa eine SVM als linear classifier [13]. Diese bilden die Eigenschaften der Eingabedaten als Vektoren (engl.: *feature descriptors*) in einem Hyperraum ab. Jeder Datenpunkt entspricht einem Punkt in diesem Raum. Der Klassifikator errechnet dann eine Hyperebene, die den von den Vektoren aufgespannten Raum unterteilt und ermöglicht so eine Klassifikation neuer Datenpunkte (siehe Abbildung 2.1), abhängig von ihrer Lage zu der Hyperebene. Voraussetzung hierfür ist, dass diese linear trennbar sind. Erst durch die Einführung des *Kernel-Tricks* [3] sind SVMs in der Lage auch nicht linear

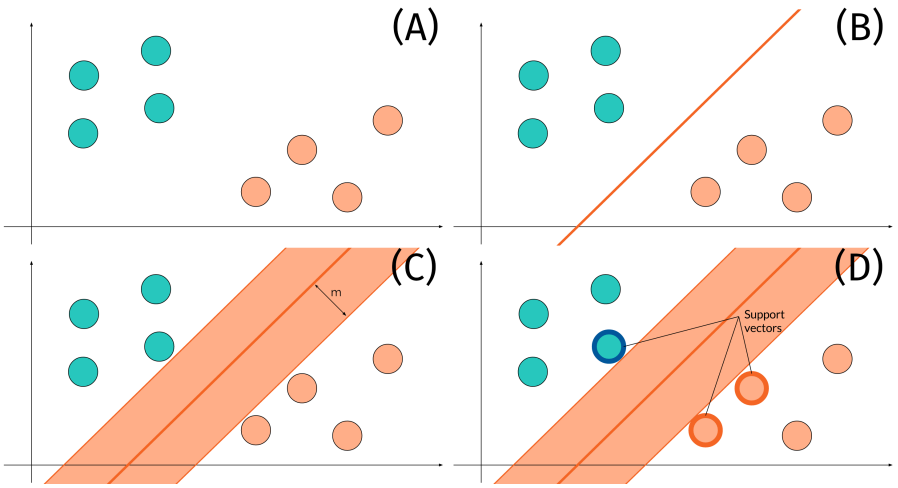


Abbildung 2.1: Grafische Abbildung der Funktionsweise einer SVM: Datensatz wird eingegeben (A), SVM berechnet Kandidaten für die Hyperebene (B), Kandidat muss möglichst großen Abstand m zu allen Datenpunkten besitzen (C) und nutzt zur Berechnung (D) der optimalen Hyperebene lediglich die nächsten Datenpunkte als Stützvektoren (engl.: *support vectors*).

trennbare Datensätze zu verarbeiten. Dieser überführt die Daten solange in höherdimensionale Räume, bis sich eine lineare Trennbarkeit einstellt.

Die ursprüngliche SVM ist ein binärer Klassifikator, der die Eingabedaten in genau zwei Klassen aufteilen kann. Ist eine Klassifizierung in mehr als zwei Klassen nötig spricht man von einem *multiclass problem*. Damit die SVM dieses Problem lösen kann, wird das multiclass problem in viele einzelne binäre Klassifizierungsprobleme unterteilt [10].

Für die erfolgreiche Funktion einer SVM ist es notwendig, dass die Eingabedaten formal beschrieben sind. Dazu werden die charakteristischen Merkmale des Inputs zunächst gesammelt und dann für jedes Eingabedatum angegeben, welches dieser Merkmale auf das Datum zutrifft. Das Merkmal wird dabei auch *feature* und die Menge aller Features *feature descriptor* genannt.

Für die SVM in der Vorlage wurde beispielsweise ein 64-dimensionaler feature descriptor händisch erstellt. So lässt sich jeder Sketch im TU Datensatz durch eine Wertekombination in diesem Vektor mit 64 Dimensionen abbilden. Hier liegt aber auch einer der Nachteile der SVM. Das Erstellen des descriptors ist aufwendig und erfordert viel Wissen. Abhilfe schaffen hier die neuronalen Netze, die nicht auf solche Beschreibungsvektoren angewiesen sind, sondern die Merkmale der zu untersuchenden Daten selbstständig erlernen.

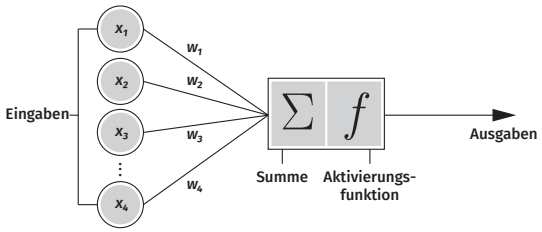


Abbildung 2.2: Schematische Darstellung eines einzelnen künstlichen Neurons mit den Eingabewerten, ihren Gewichten und der Aktivierungsfunktion.

2.3 Ein einzelnes Neuron

Um den praktischen Teil der Arbeit zu verstehen, bedarf es der Erklärung einiger theoretischer Grundlagen. Zunächst wird auf den Formalismus und die Beschreibung eines einzelnen Neurons eingegangen, um dann in Kapitel 2.4 die Wechselwirkung mehrerer Neuronen zu definieren.

Neuronen in künstlichen neuronalen Netzen orientieren sich grob an Neuronen, die in den Gehirnen und Nervensystem komplexer Wirbeltiere zu finden sind. Diese Analogie geht soweit, dass Forscher probieren, explizit Erkenntnisse aus der Biologie als Grundlage für Fortschritte im maschinellen Lernen zu verwenden (dazu: Kapitel 2.5).

Das Neuron stellt die kleinste Einheit in einem künstlichen neuronalen Netz dar. Es besitzt in seiner einfachsten Konfiguration eine Reihe von Eingängen, eine Aktivierungsfunktion und eine Reihe von Ausgängen. Die Anzahl der Ein- und Ausgänge hängt von der Konfiguration des Netzes ab. Die Aktivierungsfunktion des Neurons ist ein Hyperparameter.

Formal lässt sich ein einzelnes Neuron so darstellen:

$$a_i = f(\sum_i w_i x_i + b)$$

Hierbei sei w_i das Gewicht an Eingang i, x_i der Input an Eingang i, b der Bias, f die Aktivierungsfunktion und a_i der Aktivierungswert.

Abbildung 2.2 zeigt eine vereinfachte Darstellung eines Neurons. Das Neuron multipliziert zunächst erst pro Eingang den Input mit dem eingangsspezifischen Gewicht, summiert danach die Ergebnisse und dann den Bias. Das Ergebnis dieser Rechnung ist die Eingabe in die Aktivierungsfunktion und

der Aktivierungswert die Ausgabe des Neurons. So ist es möglich für eine beliebige Anzahl an Eingaben genau einen Aktivierungswert zu berechnen. Über die Gewichte an jedem Eingang lässt sich das Ergebnis parametrieren.

Der Wahl der Aktivierungsfunktion kommt hierbei eine wichtige Rolle zu, denn diese bestimmt, wie sich die Eingaben durch das Netzwerk bewegen, da sie darüber bestimmen ob ein Neuron aktiviert wird oder auf eine Eingabe nicht *reagiert*. So hat sie großen Einfluss auf die Ausgabe des Netzwerks.

Lineare Aktivierung

Im einfachsten Falle wird eine lineare Aktivierungsfunktion verwendet.

$$\sigma(x) = x$$

Bei dieser Funktion ist die Ausgabe gleich der Eingabe.

Die Menge aller Aktivierungsfunktionen eines Netzwerks kann als Gesamtfunktion betrachtet werden. Jede Eingabe in das Netzwerk ist also eine Eingabe in diese Gesamtfunktion. Wenn jetzt allerdings nur lineare Aktivierungsfunktionen gewählt werden bleibt auch die Gesamtfunktion linear, da die Komposition mehrerer linearer Funktionen lediglich eine weitere lineare Funktion produziert. Gerade die Fähigkeit eines künstlichen neuronalen Netzes diese Linearität hinter sich zu lassen und damit auch komplexere Probleme zu lösen unterscheidet es von den *linear classifiers*, wie der SVM. Daher ist eine Alternative zu der linearen Aktivierung notwendig.

ReLU

Die Rectified Linear Unit (ReLU) Aktivierungsfunktion ist nicht linear und erzeugt nur Werte ≥ 0. Ihr Vorteil liegt in der sehr kurzen Berechnungszeit.

$$\sigma(x) = max(0, x)$$

Laut LeCun, Bengio und Hinton [31] ist ReLU die populärste Aktivierungsfunktion momentan.

Sigmoid

Die Sigmoidfunktion erzeugt nur positive Werte, ist aber im Vergleich zu ReLU rechenintensiver. Die Sigmoidfunktion ist eine logistische Funktion mit einem S-förmigen Graphen (vgl. Abbildung 2.3). Sie ist im Wesentlichen eine skalierte

und verschobene Tangens-hyperbolicus-Funktion und weist entsprechende Symmetrien auf (siehe Abbildung 2.3).

$$sig(x) = \frac{x}{1 + e^{-x}}$$

Anwendung als Aktivierungsfunktion in künstlichen neuronalen Netzen findet sie hauptsächlich auf Grund der Tatsache, dass sie Werte innerhalb definierter Grenzen liefert und wegen ihrer einfachen Differenzierbarkeit (siehe *Gradientenverfahren* in Kapitel 2.4.2).

$$sig'(x) = sig(x)(1 - sig(x))$$

Tangens Hyperbolicus

Speziell in RNNs wird häufig der *tangens hyperbolicus* (kurz: *tanh*) verwendet. Ähnlich der Sigmoidfunktion besitzt er einen S-förmigen Graphen (siehe Abbildung 2.3). tanh erzeugt nur Aktivierungen zwischen -1 und 1.

$$\sigma(x) = \tanh(x)$$

Softmax

Die Softmax-Funktion erzeugt aus einem K-dimensionalen Vektor x mit beliebigen reellen Werten einen K-dimensionalen Vektor $\sigma(x)$ mit positiven Werten < 1, deren Summe genau 1 ergibt. Diese Eigenschaft sorgt dafür, dass die Ausgabe von Softmax als Aktivierungsfunktion in einem Output Layer als Vektor mit Klassenwahrscheinlichkeiten verstanden werden kann.

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \text{ für } j = 1, \dots, K$$

2.4 Verschaltete Neuronen bilden ein künstliches neuronales Netz

Ein einzelnes Neuron ist noch nicht in der Lage komplexe Probleme zu lösen. Erst durch die Verschaltung mehrerer Neuronen entwickelt sich ein künstliches neuronales Netz, das dazu in der Lage ist. Hierbei werden die Ausgänge eines Neurons mit den gewichteten Eingängen eines nachfolgenden Neurons verknüpft. Neuronen der gleichen Tiefe (also derselben benötigten Anzahl an

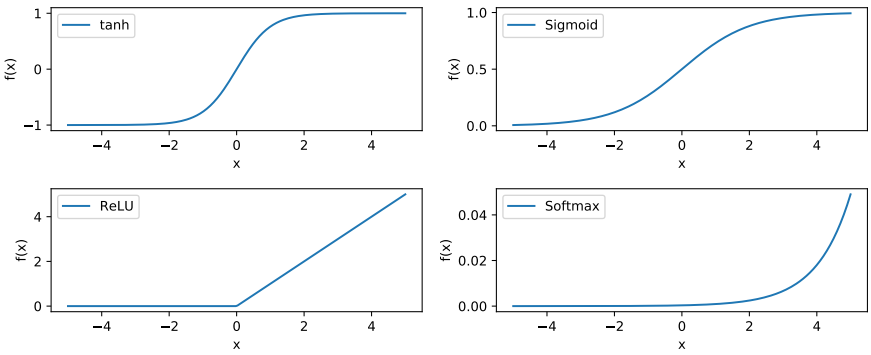


Abbildung 2.3: Plots der behandelten Aktivierungsfunktionen: Tangens Hyperbolicus, Sigmoid, ReLU und Softmax.

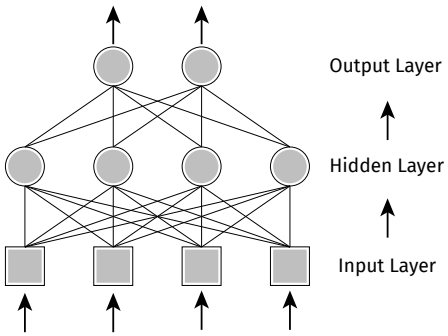


Abbildung 2.4: Ein einfaches künstliches neuronales Netz mit einem Input, einem Hidden und einem Output Layer.

Schritten bis zum nächsten Input-Neuron) werden in Schichten (engl.: *layers*) zusammengefasst.

Das kleinstmögliche künstliche neuronale Netz besteht aus drei Schichten:

1. genau ein Input Layer zur Eingabe der Daten in das Netzwerk
2. Hidden Layer
3. genau ein Output Layer

Jede dieser Schichten besteht aus einzelnen Neuronen. Die mittlere Schicht wird hierbei *hidden* (dt.: *versteckt*) genannt, da ihre Ein- und Ausgaben von außerhalb des Netzwerks nicht zugänglich sind. Abbildung 2.4 zeigt eine schematische Darstellung eines solchen Netzes. Deutlich zu erkennen: die Eingaben bewegen sich nur in einer Richtung (vom Input zum Output Layer) durch das Netzwerk und jedes Neuron einer Schicht ist mit jedem Neuron der Folgenden verknüpft.

Die Anzahl der Neuronen im Input Layer hängt von den Eingabedaten ab. Die der Neuronen im Output Layer von der erwarteten Ausgabe. Bei Klassifizierungsproblemen beispielsweise findet man häufig ein Neuron pro Klasse im

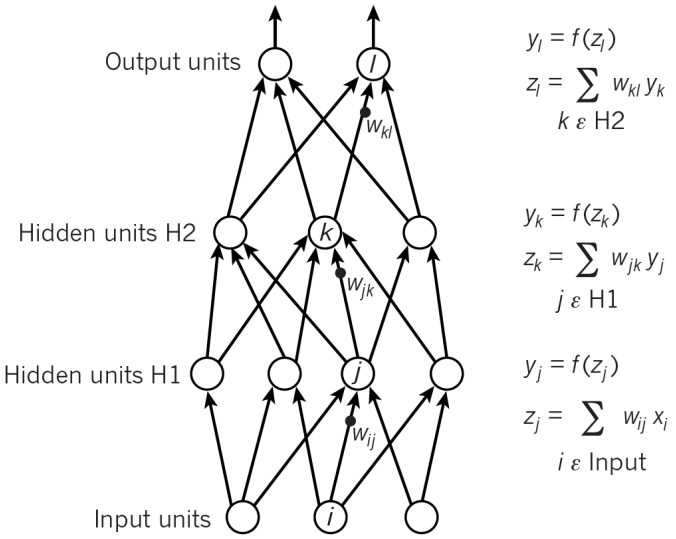


Abbildung 2.5: Vereinfachte Darstellung der Outputberechnung eines feedforward Netzwerks [31].

Output Layer.
Die Anzahl der Hidden Layer und deren jeweilige Anzahl an Neuronen ist frei wählbar und bestimmt maßgeblich die Komplexität und Mächtigkeit eines künstlichen neuronalen Netzes. Während die ersten Netzwerke in der Mitte des letzten Jahrhunderts selten mehr als einen Hidden Layer besaßen, sind heutzutage Netzwerke mit viel größeren Tiefen möglich[2].

2.4.1 Feedforward — Berechnung einer Ausgabe zu einer Eingabe

Wie aber ist es genau möglich selbst mit verschachtelten Neuronen komplexe Probleme zu lösen, wenn jedes Neuron effektiv nur eine nicht-lineare Funktion auf das Ergebnis einer Summe anwendet? Die Antwort auf diese Frage besteht aus zwei Teilen: Feedforward und Gradientenabstiegsverfahren.

Eingaben in ein künstliches neuronales Netz bewegen sich in einer bestimmten Richtung (vom Input Layer hin zum Output Layer) durch selbiges. Eine Eingabe wird also während eines Durchlaufs niemals von einem Neuron zweimal berechnet[3]. Diese Eigenschaft nennt man *feedforward*. Ein beispielhafter feedforward Durchlauf ist in Abbildung 2.5 dargestellt.

Die Eingabe bewegt sich hier vom Input Layer durch die beiden Hidden Layer H_1 und H_2 hin zum Output Layer. Jedes Neuron (i, j, k und l) nimmt dabei

[2] Das *VGG19* Netzwerk beispielsweise besteht aus 19 und *GoogLeNet* sogar aus 22 Layern. Beide Netzwerke tauchen in Kapitel 3.4 als mögliche Kandidaten für diese Arbeit erneut auf.
[3] Eine Ausnahme bilden hier die zyklischen *recurrent* Layer eines RNN. Diese werden in Kapitel 2.6 behandelt.

die Ausgabe (y) des Vorgängers, multipliziert mit dem eingangsspezifischen Gewicht (w ist das Gewicht und z die Summer aller Gewichte) und gibt anschließend den von der Aktivierungsfunktion (f) berechneten Wert an das nachfolgende Neuron weiter.

Das Ergebnis eines feedforward Durchlaufs ist die Ausgabe des Netzwerks. Da es sich bei feedforward neuronalen Netzen um eine Methodik des überwachten Lernens handelt, sind am Ende eines solchen Laufes die Eingabe, die erwartete Ausgabe und die tatsächliche Ausgabe vorhanden. Um zunächst die Leistung des Netzwerks beurteilen zu können wird der Unterschied zwischen den beiden letzteren ermittelt. Dieser wird als Fehler (engl.: *loss*) bezeichnet. Die Methode um den Fehler zu bestimmen wird in der Literatur *loss* oder *cost function* genannt.

Kreuzentropie

Als Beispiel für eine solche loss function soll die Kreuzentropie dienen. Diese wird häufig bei Klassifikationsproblemen mit mehr als zwei Ausgabeklassen eingesetzt.

Wie in Kapitel 2.3 bereits beschrieben, ist eine der Eigenschaften von Softmax als Aktivierungsfunktion im Output Layer, dass die Ausgabe als Vektor mit Klassenwahrscheinlichkeiten gedeutet werden kann. Für eine Eingabe x erhalten wir also zusätzlich zu dem bereits vorhandenen, korrekten Wahrscheinlichkeitsvektor p den vom Netzwerk berechneten Wahrscheinlichkeitsvektor q.

Für diskrete Wahrscheinlichkeiten ist die Kreuzentropie definiert als:

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

Das Ergebnis ist ein Wert, der die Distanz zwischen den beiden Wahrscheinlichkeiten wiedergibt. Im übertragenen Sinne also ein Maß dafür wie *gut* die Klassifizierung der Eingabe durch das Netzwerk ist.

Damit ist es nun möglich den Fehler pro Eingabe auszurechnen. In der Praxis hat sich dabei durchgesetzt, dass man den Fehler des Netzwerks nicht nach jedem Durchlauf berechnet, sondern erst eine gewisse Anzahl an Eingaben abwartet. Diese Anzahl heißt *mini batches* oder einfach nur *batches* und stellt eine wichtige Stellgröße im erfolgreichen Trainieren eines Netzwerks dar (vgl.: Kapitel 3.5.2).

Damit das künstliche neuronale Netz aber trainiert und bessere Ausgaben liefert, muss der Fehler jetzt rückgeführt und das Netzwerk vor dem nächsten feedforward Durchlauf optimiert werden.

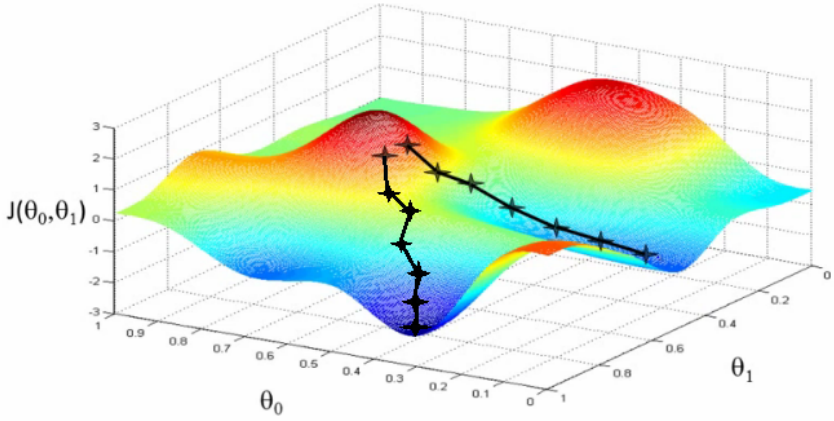


Abbildung 2.6: Grafische Darstellung eines vereinfachen Durchlaufs des Gradientenverfahrens. Hierbei sind θ_0 und θ_1 Parameter eines Neurons und $J(\theta_0, \theta_1)$ der dazugehörige Fehler.
Quelle: [37]

2.4.2 Fehlerbestimmung und -rückführung

Die einfachste Möglichkeit eines künstlichen neuronalen Netzes auf Eingaben und deren erzeugten Fehler zu reagieren ist es die Gewichte an jedem Eingang eines Neurons zu verändern. Da die Gesamtanzahl der Gewichte meist sehr hoch ist, stellt sich das Identifizieren des zu optimierenden Gewichts als Herausforderung dar. Woher weiß das Netzwerk welches Gewicht wie zu verändern ist, um den Gesamtfehler positiv zu beeinflussen?

Häufig wird für das Ermitteln der optimierten Gewichte das Gradientenverfahren (auch *Gradientenabstiegsverfahren*, *Verfahren des steilsten Abstiegs* oder engl.: *gradient descent*) verwendet. Dieses Verfahren bietet eine Möglichkeit allgemeine Optimierungsprobleme zu lösen. Im Falle des Fehlers eines künstlichen neuronalen Netzes handelt es sich um ein Minimierungsproblem. Das Gradientenverfahren versucht dabei in Richtung des negativen Gradienten in kleinen Schritten voran zu schreiten und ein Minimum der cost function zu finden (siehe Abbildung 2.6).

Wie in Kapitel 2.3 bereits beschrieben, sind die Aktivierungsfunktionen eines Neurons nicht linear. Für das Gradientenverfahren bedeutet das also, dass es nicht möglich ist dank des gefundenen Fehlers instantan alle zu aktualisierenden Werte für die einzelnen Gewichte zu berechnen. An dieser Stelle kommt deshalb die Fehlerrückführung (engl.: *backpropagation*) zum Einsatz. Damit ist es möglich beim Output Layer des Netzes anzufangen und für jedes Neuron einzeln den Beitrag zum Gesamtfehler zu berechnen. Dies geschieht durch partielle Ableitung der Fehlerfunktion im Bezug auf die Gewichte für

jedes Neuron einzeln. Inwiefern die Gewichtung jedes einzelnen Eingangs dabei geändert werden sollte, ist Ergebnis des Gradientenverfahrens.

Die partielle Ableitung der Fehlerfunktion ergibt sich aus der Verwendung der Kettenregel:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j} \frac{\partial o_j}{\partial s_j} \frac{\partial s_j}{\partial w_{ij}}$$

Hierbei sind E die zuvor vorgestellte Fehlerfunktion, a_j der Aktivierungswert des Neurons j und s_j die Summer aller gewichteten Eingänge plus Bias an Neuron j.

Je nach Lage des Neurons im Netz ergibt sich daraus:

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = \alpha \delta_j a_j$$

mit

$$\delta_j = \begin{cases} \varphi'(s_j)(t_j - a_j) & \text{falls } j \text{ Ausgabeneuron ist,} \\ \varphi'(s_j) \sum_k \delta_k w_{jk} & \text{falls } j \text{ verdecktes Neuron ist.} \end{cases}$$

Hierbei sind Δw_{ij} die Änderung des Gewichts zwischen Neuron i und Neuron j, α die Lernrate, φ die Aktivierungsfunktion des Neurons, t die erwartete Ausgabe für die erfolgte Eingabe und k der Index der nachfolgenden Neuronen von j.

Nachdem nun die vorzunehmende Änderung (Δw_{ij}) berechnet wurde, können die Gewichte entsprechend aktualisiert werden:

$$w_{ij}^{\text{neu}} = w_{ij}^{\text{alt}} + \Delta w_{ij}$$

Ein typischerweise auftretender Fehler ist dabei, dass das Gradientenabstiegsverfahren zwar ein lokales, aber nicht das globale Minimum der zu optimierenden loss function findet. Um hier entgegenzuwirken, führen aktuelle Erweiterungen des Verfahrens den Parameter *Momentum* ein. Das Momentum ist dabei die Konstante γ (meist: $\gamma = 0.9$), die mit dem Wert des letzten Parameterupdates multipliziert wird und anschließend zu der Änderung des Gewichts addiert wird:

$$\Delta w_{ij_t} = \gamma \Delta w_{ij_{t-1}} \alpha \delta_j a_j$$

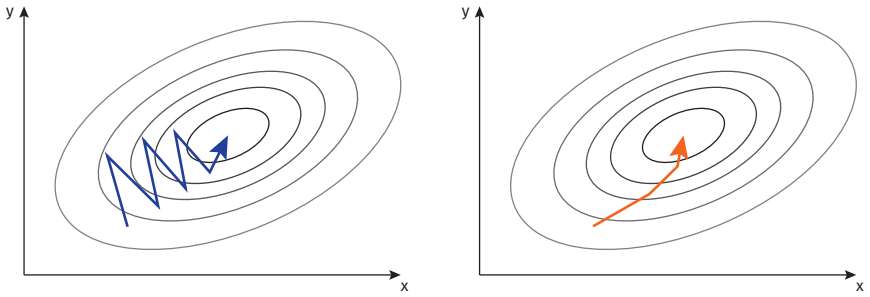


ABBILDUNG 2.7: Gradientenverfahren ohne Momentum (links) und mit Momentum (rechts).

Hierbei sei t der aktuelle Zeitpunkt der Berechnung und $t-1$ der des zuvor durchgeführten Durchlaufs.

So werden konsekutive Updates des Gradienten in eine Richtung beschleunigt und Updates in andere Richtung gebremst. Das Ergebnis sind eine schnellere Konvergenz und reduzierte Oszillation der approximierten Funktion [40, Kapitel 4.1].

Abbildung 2.7 zeigt einen beispielhaften Versuch des Gradientenverfahrens das Minimum einer zweidimensionalen Fläche (in diesem Falle liegt dies in der Mitte) zu finden. Deutlich erkennbar: die Anzahl der benötigten Schritte beider Varianten unterscheiden sich deutlich.

Adadelta

Adadelta ist eine Weiterentwicklung von Adagrad, was wiederum auf Stochastic Gradient Descent (SGD) beruht. In ihrem Kern sind diese Optimierer alles Erweiterungen des Gradientenverfahrens. Während das normale Gradientenverfahren (auch *batch gradient descent* genannt) den Gradienten jeweils über den gesamten Datensatz berechnet, benötigt SGD lediglich ein Sample zur Gradientenberechnung. Dadurch ist SGD wesentlich schneller und ressourcenschonender, als batch gradient descent.

Adagrad (als Abkürzung von *adaptive gradient algorithm*) [11] passt die Lernrate von SGD an die zu berechnenden Parameter an. Jeder Parameter erhält so eine eigene Lernrate. Dies ermöglicht große Sprünge für häufig auftretende und kleine Sprünge für selten auftretende Parameter in dem Versuch ein Minimum des Fehlers zu finden [40, Kapitel 4.3].

Adadelta, als Weiterentwicklung von Adagrad, geht dabei noch einen Schritt weiter und eliminiert die Lernrate aus der Formel zur Berechnung der Gewichtsaktualisierungen[40, Kapitel 4.4] bei gleichbleibenden Ergebnissen und Berechnungszeiten. Dies sorgt dafür, dass die Lernrate als zu evaluierender Hyperparameter wegfällt.

2.5 Convolutional Neural Networks

Bei den bisher vorgestellten Netzen ist jedes Neuron mit jedem Neuron des Vorgänger- und des Nachfolgelayers verbunden. Diese Eigenschaft eines Layers bezeichnet man als *fully connected*. Da jedes Neuron für jeden Eingang auch ein Gewicht speichert, wächst mit der Anzahl der Eingänge auch die Anzahl der Parameter. Dies kann schnell zu unverarbeitbar großen Datenmengen führen.

Ein Beispiel: als Eingabedaten für ein künstliches neuronales Netz sollen monochrome Bilder der Größe 64x64 pixel dienen. Ein Neuron der ersten Schicht bräuchte also $64 \times 64 = 4096$ Gewichte. Des Weiteren wären ebenso $64 \times 64 = 4096$ Neuronen nötig. Demzufolge bestünde der erste Hidden Layer aus bereits $4096 \times 4096 = 16.777.216$ (trainierbaren) Parametern. Werden weitere Hidden Layer hinzugefügt, die Eingabebilder größer oder ändert sich der Farbraum der Bilder, wächst diese Zahl schnell an.

Eine Lösung dieses Problems (auch bekannt als *Fluch der Dimensionalität*, engl.: *curse of dimensionality*) stellen die CNNs dar. Dieses Kapitel stellt zunächst die neuen Layertypen *Convolutional* und *Pooling* vor und geht anschließend auf aktuelle Layerarchitekturen moderner CNNs ein.

Ursrpünglich erdacht um visuelle Eingabedaten zu verarbeiten, werden CNNs schon seit einigen Jahren beispielsweise auch für das Natural language processing (NLP) verwendet[27]. Hierbei werden textuelle Informationen als eindimensionale Vektoren dem Netzwerk gezeigt und anschließend klassifiziert. Anwendung findet dies beispielsweise in der Verarbeitung von Suchanfragen [48][57].

2.5.1 Convolutional Layers

Vereinfacht gesagt lernen Convolutional Layers Muster in Eingabedaten. Was für Muster dies sind, hängt von den Eingabedaten und der Position des Layers innerheralb des Netzes ab. Für Bilder beispielsweise lernen die ersten Convolutional Layer einfache Objekte wie Kanten und Ecken, während Layer in der Mitte des Netzwerkes diese Erkennungen nutzen, neu zusammenfügen und in dem Ergebnis komplexe Objekte erkennen können (vgl. Abbildung 2.10).

Convolutional Layers unterscheiden sich in drei grundsätzlichen Eigenschaften von den bisher gezeigten (multilayer) perceptrons:

1. lokale Konnektivität (engl.: *local connectivity*, als Gegensatz zu *fully connected*)
2. geteilte Gewichte (engl.: *shared weights*)

3. räumliche Anordnung der Neuronen (engl.: *spatial arrangement*)

Mit dem Begriff *local connectivity* ist gemeint, dass jedes Neuron nicht mehr mit den gesamten Vorgängerneuronen, sondern nur noch mit einem Teil davon verknüpft ist. Der verknüpfte Bereich wird auch rezeptives Feld (engl.: *receptive field*) genannt (vgl.: Abbildung 2.8). Zudem sind die Gewichte für alle Neuronen eines Convolutional Layers identisch (geteilte Gewichte, englisch: shared weights). Dies führt zum einen dazu, dass beispielsweise jedes Neuron im ersten Convolutional Layer codiert, zu welcher Intensität eine Kante in einem rezeptiven Feld des Eingabebildes vorliegt und zum anderen dazu, dass die Anzahl der trainierbaren Parameter erheblich reduziert wird.

Der Zusatz *convolutional* (von engl. *convolution*, zu deutsch: Faltung) rührt daher, dass die Aktivität jedes Neurons über eine diskrete Faltung berechnet wird. Dabei wird schrittweise eine Faltungsmatrix (im Folgenden auch *Filter*) geringer Größe (üblich sind im ersten Convolutional Layer Größen bis 15x15 Pixel) über die Eingabe bewegt. Der Eingabewert berechnet sich dann als Skalarprodukt aus dem Filterkernel mit dem aktuellen Bildausschnitt.

Die räumliche Anordnung der Neuronen spielt eine Rolle bei der Dimensionalität der Ausgabe eines Convolutional Layers. Bestimmt wird diese über die folgenden drei Faktoren:

1. *depth* gibt an, wieviele Neuronen eines Layers den gleichen Input verarbeiten.
2. *stride* gibt die Schrittweite des Filters an
3. *zero-padding* bestimmt ob der Input des Neurons mit 0-Werten umrandet werden soll. Dies ist hilfreich um die Ausgabe des Neurons auf eine bestimmte Größe zu fixieren.

Abbildung 2.8 verdeutlicht diese Eigenschaften grafisch. Das Eingabevolumen (blau) wird dabei mit Zero Padding = 1 umgeben (weißer Rahmen). Der 3×3 große Filter (grauer *Schatten*) gleitet mit stride = 2 über die Eingabe (deutlich an der Verschiebung des Mittelpunkts des Filters zwischen den drei Abbildungen zu sehen). Das Ausgabevolumen (grün) hat eine Größe von 3×3. In diesem Beispiel sind Ein- und Ausgabe des Convolutional Layers zweidimensional. In der Praxis sind diese jedoch häufig dreidimensional. Dazu wird das Ausgabevolumen in sogenannten *depth slices* geschichtet. Die Filter eines depth slices teilen sich dabei die Gewichte.

Um die tatsächliche Ausgabe eines Convolutional Layers berechnen zu können dient folgende Formel:

$$(W - F + 2P)/S + 1$$

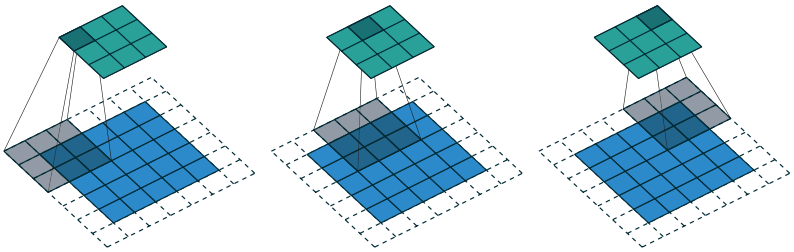


Abbildung 2.8: Der 3×3 Filter des Convolutional Layers (grau) gleitet mit stride = 2 über das Eingabevolumen (blau). Dieses ist mit Zero Padding = 1 umrandet (weiß). Diese Operation erzeugt das Ausgabevolumen (grün) [12].

Hierbei seien W die Größe der Eingabedaten, F die Größe der Filter, S die Schrittweite der Filter und P die Anzahl der zur Umrahmung hinzugefügten 0. Die Tiefe des Layers wird im Folgenden mit K angegeben.

Als Beispiel sei hier das künstliche neuronale Netz *DeepSketch* der Autoren Seddati, Dupont und Mahmoudi herangezogen. Der erste Convolutional Layer von DeepSketch empfängt monochrome Bilder der Größe 224×224 pixel ($W = 224$), besitzt eine Filtergöße von 7×7 ($F = 7$), eine Schrittweite von 2 ($S = 2$), wendet kein zero-padding an ($P = 0$) und hat eine Tiefe von $K = 64$. Daraus ergibt sich:

$$\begin{aligned} \text{out}_1 &= (W - F + 2P)/S + 1 \\ &= (224 - 7 + 2 \times 0)/2 + 1 \\ &= 109,5 \end{aligned}$$

Das Ergebnis wird abgerundet und ergibt mit K eine finale Ausgabegröße von $(109, 109, 64)$.

Nun wird auch die Wichtigkeit der geteilten Gewichte erneut deutlich. Für das berechnete Ausgabevolumen wären ohne Gewichtteilung $109 * 109 * 64 = 760.384$ Neuronen mit jeweils $7 \times 7 \times 1$ Gewichten (die 1 gibt hier die Anzahl der Farbkanäle in den Bildern wieder) plus Bias, also $760.384 * (49+1) = 38.019.200$ Parameter zu trainieren. Da sich nun aber Filter eines *depth slices* die Gewichte teilen, sind in diesem Layer lediglich $64 \times 7 \times 7 \times 1 + 64 = 3200$ Parameter (inklusive der 64 einzelnen Biase) vorhanden.

2.5.2 Pooling

Pooling Layer wenden eine Form des nicht-linearen Downsamplings auf die Eingabedaten an. Die am häufigsten verwende Art ist dabei das *max pooling*.

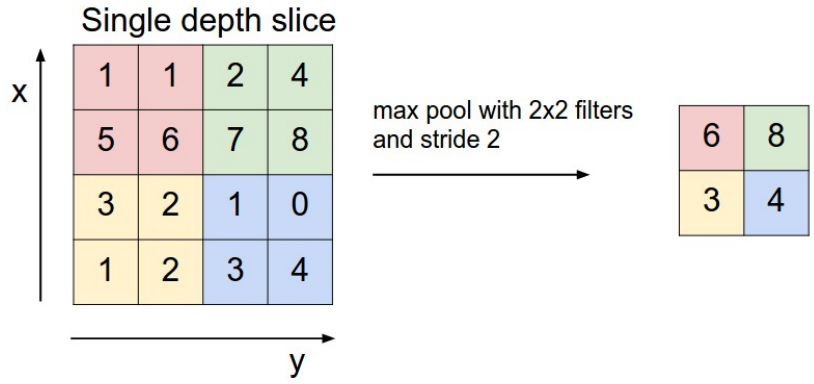


ABBILDUNG 2.9: Max Pooling reduziert das Volumen des Output und verstärkt räumliche Beziehungen aktivierter Filter [28]

Max Pooling unterteilt den Input in gleich große, sich nicht überlappende Rechtecke und gibt für jedes Rechteck den höchsten Wert aus (siehe Abbildung 2.9). Dies fördert vor Allem die Translationsinvarianz gelernter Filter. Das Max Pooling betont also nicht die exakte Position der Aktivierung, sondern die räumliche Beziehung zu anderen Aktivierungen.

Ein weiterer Vorteil des Max Poolings ist, dass dessen Anwendung die Größe des Output Volumens verringert und dadurch Rechenzeit gespart werden kann. Auch als Maßnahme gegen Overfitting spielt Pooling eine große Rolle (siehe dazu Kapitel 4.1.2).

2.5.3 CNN-Architekturen

Für die Verwendung von Convolutional und Pooling Layern hat sich im Laufe der Zeit ein Muster gebildet. So werden die Layer nicht etwa einzeln, sondern in Blöcken in den Architekturen von CNNs verbaut. Karpathy formalisiert [28] dieses Muster als:

$$\text{INPUT} \rightarrow ((\text{CONV} \rightarrow \text{RELU}) \times N \rightarrow \text{POOL?}) \times M \rightarrow (\text{FC} \rightarrow \text{RELU}) \times K \rightarrow \text{FC}$$

Hierbei steht *CONV* für einen Convolutional, *POOL* für einen Pooling, *FC* für einen Fully Connected und *RELU* für einen Aktivierungslayer mit ReLU als Aktivierungsfunktion. Die Variablen N, M und K geben dabei die Multiplizitäten der Blöcke an und das Fragezeichen symbolisiert einen optionalen Teil. Für die Variablen hat Karpathy die Werte $0 \leq N \leq 3$, $M \geq 0$ und $0 \leq K < 3$ identifiziert.

Für das DeepSketch Netzwerk ergibt sich daraus folgende Notation:

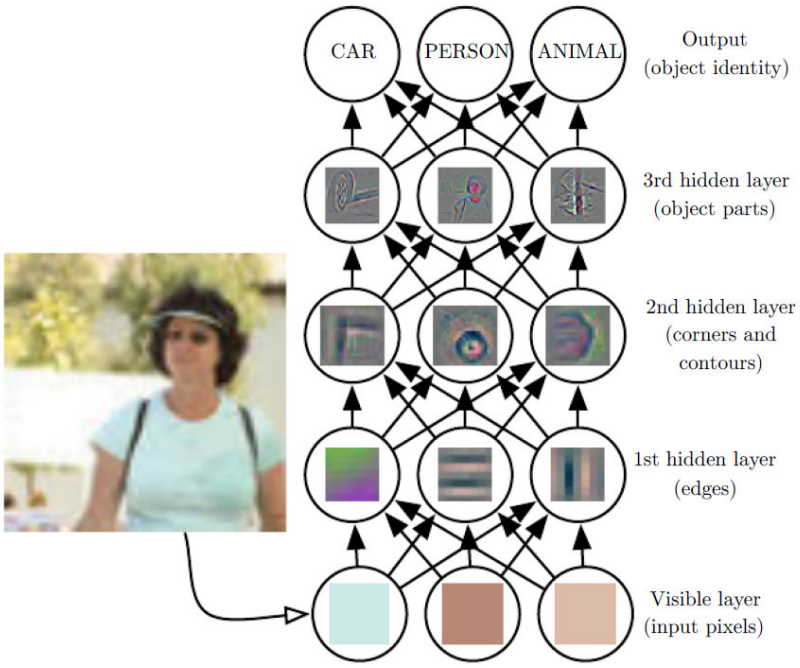


Abbildung 2.10: Darstellung der erlernten Features in den verschiedenen Schichten eines CNN. Quelle: [20]

$$\text{INPUT} \rightarrow (\text{CONV} \rightarrow \text{RELU} \rightarrow \text{POOL}) \times 2 \rightarrow (\text{CONV} \rightarrow \text{RELU}) \times 2 \rightarrow \text{POOL} \rightarrow (\text{CONV} \rightarrow \text{RELU}) \rightarrow \text{DROPOUT} \rightarrow \text{CONV}$$

Erst durch diese Blöcke wird eine zentrale Eigenschaft der CNNs ermöglicht: das Erlernen komplexer Zusammenhänge. Während die ersten Layerblöcke für das Erlernen einfacher Merkmale (engl.: *features*) zuständig sind, sind die nachgeschalteten Blöcke fähig diese Informationen zu immer komplexeren Eigenschaften zusammenzusetzen. Dies wird in Abbildung 2.10 verdeutlicht. Der erste Layer lernt hierbei einfache Eigenschaften, wie Farbverläufe oder Kanten, im Input zu erkennen. Diese werden in einem weiteren Layer zu annähernd erkennbaren Gebilden (wie etwa der Umriss eines Gesichts) zusammengesetzt. Der dritte Hidden Layer kann diese Gebilde dann zu beispielsweise einem Gesicht zusammensetzen und liefert damit den Input für den Klassifizierungslayer.

2.6 Recurrent Neural Networks

Während CNNs den Input falten und in einzelne Merkmale unterteilen, die zusammengesetzt dann den korrekten Output liefern, verfolgen die rekurrenten (=auf sich selber Bezug nehmend) Layer eines RNN einen anderen Ansatz.

Das Alleinstellungsmerkmal eines solchen Layers ist ein feedback loop, in welchem der Ouput eines Neurons an den Input eines vorangehenden Neurons verknüpft wird. Dies stellt einen Gegensatz zu den bisher vorgestellten feedforward-Netzen da, bei denen sich der Input in nur einer Richtung durch das Netz bewegt.

Speziell für die Analyse von zusammenhängenden Daten (auch *Sequenz* genannt) hat sich diese Vorgehensweise als erfolgreich herausgestellt, da somit Zeitinformation erlernt werden können. Beispiel für die Verwendung sind hier die Analyse von Handschrift[22], gesprochener Sprache[21] oder Übersetzungen[51].

Hochreiter und Schmidhuber erweitern das bisher bekannte Konzept der RNN um die Möglichkeit auch zeitlich weit voneinander getrennte Zusammenhänge zu lernen und nennen diesen Layertypen LSTM [26]. Die Idee dahinter ist die Recurrent Layer um *Input*, *Ouput* und *Forget Gate* zu erweitern[18]. Damit ist es dem Layer möglich Eingaben zu speichern und auch wieder zu löschen. Dazu bereitet das Input Gate den Input so auf, dass er in dem *Gedächtnis*[4] (engl.: *cell state*) des Layers gespeichert wird. Bevor das Input Gate nun die neuen Eingabewerte im cell state persistiert, entscheidet das forget gate darüber ob der cell state überschrieben oder aktualisiert wird. Beim Überschreiben wird also das *Gedächtnis* der Zelle gelöscht. Anschließend bereitet das Output Gate den cell state so auf, dass die nachgeschalteten Neuronen die Ausgabe verwenden können.

[4] In der Praxis ist dies meist eine Menge von Vektoren, die eine Reihe von Aktivierungen enthält.

Kapitel 3

Die Erkennung einfacher Skizzen mit neuronalen Netzen

3.1 Verwendete Hard- und Software

Das Trainieren tiefer neuronaler Netze besteht aus sehr vielen simplen Einzelberechnungen (Aktualisieren des Gradienten, Backpropagation). Da diese unabhängig voneinander geschehen können, empfiehlt es sich diese zu parallelisieren. Für diesen Einsatzzweck besonders geeignet sind moderne Grafikkarten, deren GPUs auf ebensolche Berechnungen spezialisiert sind.

Für die vorliegende Arbeit wurde von der Beuth Hochschule ein System mit einer NVIDIA Titan X Grafikkarte bereitgestellt. Diese beruht auf der Pascal Architektur und verfügt über 16 GB Arbeitsspeicher. In dem System sind weiterhin eine Intel Xeon 8 Kern Central processing unit (CPU) mit 3.40 GHz und 16 GB Arbeitsspeicher verbaut. Als Betriebssystem kommt Ubuntu 16.04 zum Einsatz. Obwohl der vorliegende Datensatz mit nur 20.000 Bildern insgesamt eher klein ausfällt, reicht die verwendete Hardware nicht aus, um die Bilder in voller Größe auf einmal in den Arbeitsspeicher zu laden. Welche exakten Maße für welches Netz verwendet wurden, ist Gegenstand von Kapitel 4.2.

Für die Erstellung der Netzwerke wird das Framework Keras von François Chollet [4] verwendet. Keras ist ein in Python geschriebenes Framework, das auf TensorFlow als Backend zurückgreift. TensorFlow ist ein u.a. von Google entwickeltes Framework für maschinelles Lernen [34]. TensorFlow selber ist in C++ geschrieben, bietet aber die Möglichkeit via Python gesteuert zu werden. Ebendiese Aufgabe übernimmt Keras. TensorFlow benutzt weiterhin die Bibliotheken `cudnn`, `cuda`, `cublas`, `cufft` und `curand` um möglichst effizient

die Leistung der Grafikkarte abzurufen. Diese werden vom Hersteller der Grafikkarte angeboten[1].

3.2 Der Datensatz

Der für die vorliegende Arbeit benötigte Datensatz entstammt der Arbeit „How Do Humans Sketch Objects?“ der Autoren Eitz, Hays und Alexa aus dem Jahre 2012. Dieser besteht aus insgesamt 20.000 einzelnen Dateien, die wiederum in 250 Klassen sortiert wurden. Jede Datei ist monochrom (schwarzer Strich auf weißem Grund) und liegt in verschiedenen Dateiformaten vor. Für diese Arbeit wurden die Portable Network Graphics (PNG) und Scalable Vector Graphics (SVG) Versionen verwendet.

Für die Erfassung des Datensatzes haben die Autoren auf Amazon Mechanical Turk (AMT) zurückgegriffen. AMT ist eine von Amazon betriebene Plattform, auf der Forscher und Firmen einfache Aufgaben bereitstellen können, die dann von Menschen gelöst werden. Dieses wird häufig verwendet um Datensätze von Menschen klassifizieren zu lassen und das Ergebnis anschließend zum Training von Maschinen zu benutzen. Eitz, Hays und Alexa definieren als Task beispielsweise “Skizzieren Sie einen Apfel” und erhalten als Ergebnis ein von einem Menschen gemalten Apfel. Hierbei existiert also zuerst das Label und das Datum wird generiert.

Die 250 zu erfassenden Klassen entstammen dem *LabelMe* Datensatz, dem *Princeton Shape Benchmark* und dem *Caltech 256 dataset* [13, Kapitel 3.1]. Bei der Erstellung der Klassen haben die Autoren besonderen Wert darauf gelegt, dass jede Klasse *erschöpfend*, *wiedererkennbar* und *genau* ist. Mit erschöpfend ist gemeint, dass die Gesamtheit der Klassen möglichst umfassend versucht das alltägliche Leben abzubilden, ohne Themengebiete auszulassen. Eine Klasse ist wiedererkennbar, wenn ihre äußere Form ohne jede Zugabe von Kontext ausreicht um erkannt zu werden. Als *genau* wird eine Klasse bezeichnet, wenn die Bandbreite ihrer möglichen visuellen Repräsentationen gering ist. Um Beispielsweise die Vogelfamilie *Laridae* zu beschreiben wäre *Möwe* der richtige Klassenname, während *Tier* zu ungenau wäre. Mit diesen Bedingungen haben Eitz et al. insgesamt 250 Klassen aus den obigen Quellen identifiziert.

Nach der Erfassung des Datensatzes haben die Autoren in einem zweiten AMT Task den Datensatz von Menschen klassifizieren lassen. Jedem Probanden wurde ein zufällig ausgewählter Sketch gezeigt und es galt die korrekte Klasse aus der Menge der 250 zu wählen. Das Ergebnis ist, dass Menschen mit einer Genauigkeit von 73,1% die korrekte Klasse auswählen. Dieser Wert dient als

[1] Siehe: `https://developer.nvidia.com/cudnn`

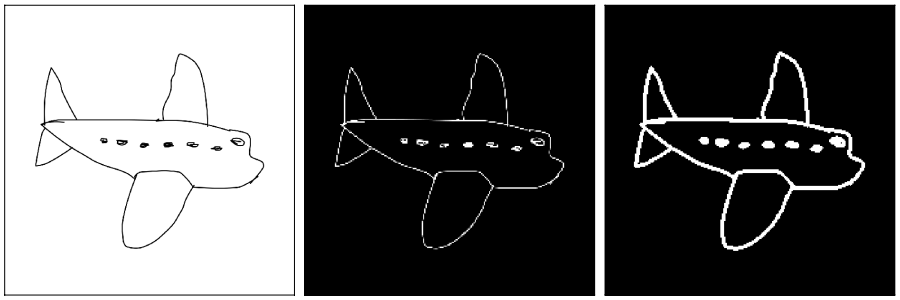


ABBILDUNG 3.1: Die drei Schritte der Vorverarbeitung: verkleinern (links), invertieren (Mitte) und Dilatation (rechts).

Messlatte für jedes Programm, da es diesen zu erreichen, oder im besten Falle übertreffen, gilt.

3.3 Aufbereitung des Datensatzes

Essentiell für das erfolgreiche Trainieren eines künstlichen neuronalen Netzwerks ist die Aufbereitung bzw. Vorverarbeitung (engl.: *preprocessing*) des Datensatzes. Dieses Kapitel beschreibt mit welchen Ansätzen der vorliegende Datensatz bearbeitet wurde.

3.3.1 Bilddimensionen und Farbinformationen

Bevor jeder Sketch in den Input Layer des Netzwerks gegeben wird, durchläuft er drei grundlegende Preprocessingschritte:

1. Verkleinern des Sketches auf die vom Netzwerk erwartete Eingangsgröße. Diese unterscheidet sich je nach verwendetem Netz und ist entweder 224×224 Pixel oder 168×168 Pixel (siehe Kapitel 4.2)
2. Reduktion auf genau zwei Farben und anschließende Inversion
3. Dilatation mit einem (2, 2) Kernel, damit nach der Verkleinerung in Schritt 1 keine Details verloren gehen

Der Großteil der Vorverarbeitung wird mit der Python Imaging Library (PIL) realisiert. Das Öffnen und direkte Verkleinern des Bildes, sowie die Invertierung dessen Farben sind mit dieser Bibliothek einfache Funktionsaufrufe (siehe Listing 3.1, Zeile 1f und Zeile 4). Für den dritten Preprocessingschritt wird die Funktion `binary_dilation` aus dem SciPy Paket[2] verwendet (siehe Listing 3.1, Zeile 6).

Die Ergebnisse dieser drei Schritte sind in Abbildung 3.1 zu sehen.

[2] `http://www.scipy.org/`, zuletzt abgerufen am 08.03.2017

```
img = Image.open(filepath)
img = img.resize(input_size)
img = img.convert('L')  # nach s/w konvertieren
img = PIL.ImageOps.invert(img)
img = np.asarray(img, dtype='float32')
img = ndimage.binary_dilation(img).astype(img.dtype)
```

Quellcode 3.1: Die Vorverarbeitung eines einzelnen Sketches in Python.

Binäre Dilatation

Die binäre Dilatation ist eine der Basis-Operationen der digitalen Bildverarbeitung. Hierbei wird ein strukturierendes Element auf den Inhalt eines binären (also einem Bild mit exakt zwei Farbtönen) Bildes angewandt. Dazu wird ebendieses Element auf jeden Bildpunkt der Eingabe angewendet und weitet diesen entsprechend aus (*dilatiert* diesen).

Im vorliegenden Falle wird ein 2×2 Pixel großes strukturierendes Element auf den bereits verkleinerten Sketch angewendet. Dadurch wird jeder weiße Pixel im Ursprungsbild auf zwei Pixel Breite und zwei Pixel Höhe erweitert. Dies hat zu Folge, dass einzelne Details, die bei der Verkleinerung verloren gegangen sind, wieder sichtbar werden. Dem mittleren Sketch in Abbildung 3.1 ist beispielsweise deutlich zu entnehmen, dass der Strich, der die Decke des Flugzeugs darstellt zum Heck hin nach der Verkleinerung nicht mehr vollständig ist. Dieses fehlende Stück könnte für eine Fehlklassifizierung des Netzwerks sorgen. Nach der Dilatation (gleiche Abbildung, rechte Grafik) ist diese Linie jedoch wieder vollständig geschlossen.

3.3.2 Randomisierte Transformationen

Des Weiteren wird der Datensatz durch eine Reihe randomisierter Transformationen künstlich vergrößert. Hierbei kann ein Bild:

- Horizontal gespiegelt werden
- um bis zu 5 Grad rotiert werden
- Änderung der Position relativ zum Zentrum in Abhängigkeit von Höhe und Breite von bis zu 10%
- um bis zu 5% heran- bzw herausgezoomt werden

Hierfür wird der von Keras bereitgestellte `ImageDataGenerator` verwendet (siehe Listing 3.2).

```
datagen = ImageDataGenerator(
        horizontal_flip=True,
        rotation_range=5,
        width_shift_range=0.1,
        height_shift_range=0.1,
        zoom_range=0.05
)
```

QUELLCODE 3.2: Die Parameter des `ImageDataGenerator` zur Erzeugung randomisierter Transformationen.

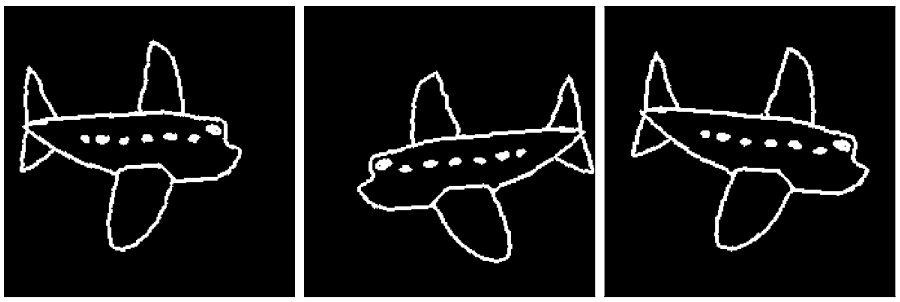


ABBILDUNG 3.2: Beispiele für die zufällige Vorverarbeitung des Datensatzes durch den Keras `ImageDataGenerator`.

Ein Beispiel für die Anwendung der randomisierten Transformationen ist Abbildung 3.2 zu entnehmen.

Mit dem Generator ist es jetzt also möglich eine beliebige Epochenlänge anzugeben. Reicht der Umfang des Datensatzes dabei nicht aus, generiert Keras automatisch nach den oben definierten Regeln neue Skizzen und füllt die Epoche damit auf.

3.3.3 Standardisierung des Datensatzes

In der Statistik ist es üblich Stichproben vor der Berechnung zu normalisieren. Häufig wird dafür der Mittelwert des gesamten Stichprobenumfangs subtrahiert und anschließend durch die Standardabweichung dividiert (beispielhafte Implementierung siehe Listing 3.3).

```
def standardize_inputs(x):
        x -= np.mean(x)
        x /= (np.std(x) + 1e-7)
        return x
```

QUELLCODE 3.3: Standardisierung eines Datensatzes in Python unter Zuhilfenahme von *numpy*.

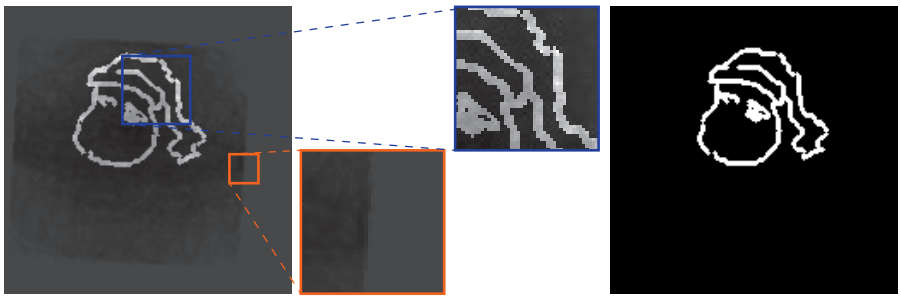
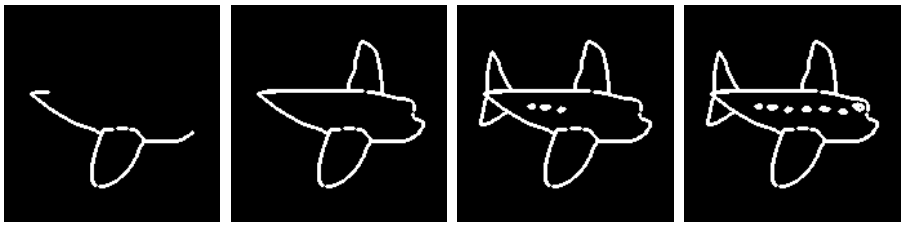


Abbildung 3.3: Die normalisierte Version eines vorverarbeiteten Sketches (links) und die nicht-normalisierte Version (rechts). In den Vergrößerungen deutlich zu erkennen: Kontrastverlust (oben) und deutliche Bildung unnatürlicher Kanten durch die Verarbeitung mit dem *ImageDataGenerator* (unten).



(A) (B) (C) (D)

Abbildung 3.4: Aufteilung eines Sketches in eine Sequenz aus vier Vollständigkeitsstufen: (A) 25%, (B) 50%, (C) 75% und (D) 100%.

Im Falle des vorliegenden Datensatzes führt die Normalisierung dazu, dass der Sketch insgesamt heller wird und so der Kontrast von Strich zu Hintergrund geringer wird. Das Ergebnis ist Abbildung 3.3 zu entnehmen.

3.3.4 Entfernen einzelner Striche

Eine weitere Möglichkeit den bestehenden Datensatz zu vergrößern, ohne Skizzen hinzuzufügen ist die Manipulation der vorhandenen Daten. Eitz, Hays und Alexa stellen den Datensatz in mehreren Dateiformaten bereit, u.a. auch als SVG[3]. Dieses Textformat beruht auf Extensible Markup Language (XML) und beinhaltet pro Sketch eine Reihe einzelner, der Reihenfolge nach sortierter, `Path`-Elemente. Es ist möglich die SVG-Dateien einzuladen und Striche anteilig zu entfernen. So sind pro Sketch vier neue Sketches mit jeweils unterschiedlichen Vollständigkeitsstufen des Originals entstanden (siehe Abbildung 3.4).

[3] `http://cybertron.cg.tu-berlin.de/eitz/projects/classifysketch/sketches_svg.zip`, zuletzt abgerufen am 13.03.2017

Zur automatisierten Generierung der Vollständigkeitsstufen werden für jeden Sketch zunächst die 25, 50, 75 und 100 Prozent Perzentile berechnet und anschließend als einzelne, bereits vorverarbeitete (siehe Kapitel 3.3.1) PNG-Dateien gespeichert. Hierbei ist aufgefallen, dass es einen beträchtlichen Teil an Sketches mit weniger oder genau vier Strichen gibt. In diesen Fällen wurde das 100% Perzentil so häufig wiederholt, bis insgesamt vier neue Sketches pro Originalsketch entstanden sind. Insgesamt ist der Datenbestand so von 20.000 auf 80.000 Sketches gewachsen.

Neben der Vervielfachung des Datensatzes hat die Trennung eines Sketches in einzelne Striche noch einen weiteren Vorteil. So ist es möglich die Reihenfolge der Striche innerhalb eines Sketches als zeitliche Abfolge zu verstehen. Durch das Aufteilen des Sketches in vier Vollständigkeitsstufen erhalten wir also drei Zwischenstände und einen Endstand für den Prozess der Sketcherstellung. Diese zeitlichen Informationen lassen sich benutzen um Sequenzen als Input für ein RNN (siehe Kapitel 2.6) zu erstellen. Für das Ergebnis des Versuchs CNN und LSTM zu kombinieren siehe Kapitel 4.2.3.

3.4 Das verwendete Netzwerk

Bei der Verwendung künstlicher neuronaler Netze ist es üblich auf die Architektur bestehender Netzwerke aufzubauen. Oft ist es sogar möglich bereits vortrainierte Netze zu verwenden. Für die Klassifikation des vorliegenden Datensatzes sollen zunächst also in der Bildklassifikation gängige Netzwerke auf Tauglichkeit untersucht werden, um einen gültigen Startpunkt für weitere Evaluationen zu finden.

Weit verbreitete Netzwerke zur Klassifikation von Fotos sind u.a. *VGG16* [49], *VGG19* [49] und *GoogLeNet* [52]. Ebenso in die erste Auswahl aufgenommen wurde das CIFAR10 Netz[4]. All diese Netze sind jedoch für den Einsatz auf Datensätzen, die aus Fotos bestehen, spezialisiert. Fotos enthalten in der Regel deutlich detailliertere Objekte, mehrere Objekte pro Foto und mehr Farben als die Skizzen aus dem TU Benchmark. Ob diese Netze für die Klassifikation von Sketches verwendet werden können, soll im Folgenden gezeigt werden.

Speziell für den Einsatz mit dem vorliegenden Datensatz konzipiert wurden die Netze *Sketch-a-Net* [58] und *DeepSketch* [47]. Beide werden ebenso in die Evaluation aufgenommen.

Des Weiteren wird das MNIST CNN[5] aus den Beispielen des Keras Frameworks aufgenommen. Trotz seiner geringen Komplexität (es besteht im größten

[4] https://github.com/fchollet/keras/blob/master/examples/cifar10_cnn.py, zuletzt abgerufen am 08.03.2017

[5] https://github.com/fchollet/keras/blob/master/examples/mnist_cnn.py, zuletzt abgerufen am 08.03.2017

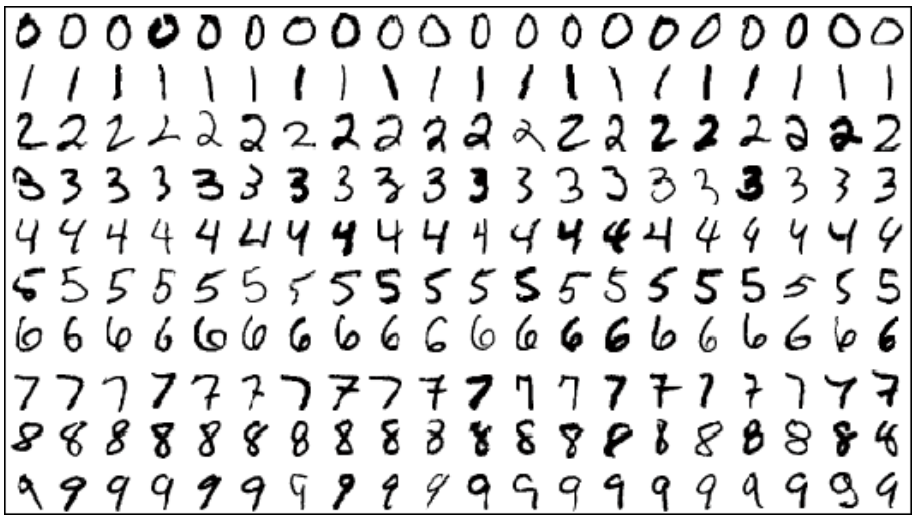


Abbildung 3.5: Beispiele aus dem MNIST Datensatz[32], die eine Nähe zum TU Datensatz verdeutlichen: monochrome Bilder, geringe Detailtiefe und lediglich ein Objekt pro Bild.

Teil aus lediglich zwei convolutional und zwei fully connected layers) ist das MNIST CNN auf Grund der Tatsache, dass Merkmale des MNIST Datensatzes mit dem des vorliegenden übereinstimmen (siehe Abbildung 3.5) ein möglicher Kandidat.

Für die Evaluation werden die von Eitz, Hays und Alexa bereitgestellten PNG Dateien[6] entpackt und einzeln den jeweiligen Netzwerken, ohne jegliche Art der Vorverarbeitung, für 30 Epochen gezeigt. Mit Epoche ist hierbei der Zeitraum gemeint, den das Netzwerk benötigt um den ganzen Datensatz einmal komplett zu lernen. Die Größe der Epoche, also die Anzahl der gezeigten Sketches, ist variabel (siehe 3.3.2) und wurde hier auf 20.000 gesetzt. Dies entspricht dem unveränderten Datensatz.
Bei den Netzwerken handelt es sich um Implementationen der Netzwerkarchitekturen in Keras. Keines der Netzwerke ist vortrainiert. Die Ergebnisse sind Tabelle 3.1 zu entnehmen.

Zur Bewertung der Klassifikation des Netzwerks wird der Datensatz vorab in zwei Teile geteilt: das Trainings- und das Testset. Das Trainingsset ist dabei doppelt so groß, wie das Testset. Diese Dreiteilung (zwei Teile Training und ein Teil Test) wird auch *3fold* genannt. Während des Trainings werden dem Netzwerk dabei nur Samples aus dem Trainingsset gezeigt und anschließend anhand der Klassifikation des bis dahin unbekannten Testsets die Genauigkeit ermittelt.

Die Top1-Genauigkeit gibt dabei an, in wie vielen Fällen die vom Netzwerk als

[6] `http://cybertron.cg.tu-berlin.de/eitz/projects/classifysketch/sketches_png.zip`, zuletzt abgerufen am 08.03.2017]

Tabelle 3.1: Die Ergebnisse der durchgeführten Evaluation möglicher Netzwerkkandidaten.

Netzwerk	Maximale Top1-Genauigkeit
DeepSketch	57,07%
Sketch-a-Net	50,35%
VGG16	45,44%
CIFAR10	31,41%
MNIST	14,21%
VGG19	✗
GoogLeNet	✗

am wahrscheinlichsten bestimmte Klasse mit der tatsächlichen Klasse übereinstimmt. Eine weitere häufig auftretende Metrik ist die Top5-Genauigkeit, bei der angegeben wird, wie hoch die Wahrscheinlichkeit ist, dass die korrekte Klasse unter den 5 vom Netzwerk ausgegeben ist.

Das VGG19-Netzwerk und GoogLeNet sind für die bereitgestellte Hardware zu aufwändig und konnten während der Evaluation keine Ergebnisse produzieren. VGG16, die kleinere Variante von VGG19[7], erzielte mit 45,44% Top1-Genauigkeit den dritten Platz. CIFAR10, als weiteres für Fotos optimiertes Netzwerk, erreicht mit 31,41% Genauigkeit den vierten Platz. Weit abgeschlagen auf Platz fünf, mit nur 14,21% und damit weniger als der Hälfte der Genauigkeit im Vergleich zu Platz vier ist das MNIST Netzwerk. Die ersten beiden Plätze belegen die bereits für Skizzenerkennung optimierten Netzwerke DeepSketch und Sketch-a-Net mit 57,07% und 50,35% Genauigkeit.

Die erste Evaluation bestätigt die erwarteten Ergebnisse. Die bereits für die Skizzenerkennung optimierten Netzwerke schneiden signifikant (Unterschied DeepSketch zu VGG16: 11,63%) besser ab. Die für Fotos optimierten Netzwerke VGG16 und CIFAR10 erbringen im unoptimierten Zustand keine brauchbare Leistung und das Netzwerk zur Erkennung handschriftlicher Ziffern erkennt nicht einmal jede fünfte gezeigte Skizze.

Die schlechte Performance des MNIST Netzwerks ist auf dessen geringe Komplexität zurückzuführen (Stichwort: *underfitting*. Siehe Kapitel 4.1.2). Trotz der eingangs erwähnten Ähnlichkeiten beider Datensätze ist es dem Netzwerk so nicht möglich eine zufriedenstellende Top1-Genauigkeit zu erreichen.

[7] Die Zahl im Namen der beiden Netze gibt die Anzahl der concolutional layers wieder: VGG16 besitzt 16 und VGG19 insgesamt 19 solcher layers.

Tabelle 3.2: Die Architektur von DeepSketch, Quelle: [47]

#	Typ	Filtergröße	-anzahl	Stride	Pad
1	Convolutional	7x7	64	2	0
2	ReLU	–	–	–	–
3	Maxpool	3x3	–	2	0
4	Convolutional	5x5	128	2	2
5	ReLU	–	–	–	–
6	Maxpool	3x3	–	2	–
7	Convolutional	3x3	256	1	1
8	ReLU	–	–	–	–
9	Convolutional	3x3	512	1	0
10	ReLU	–	–	–	–
11	Maxpool	3x3	–	2	–
12	Convolutional	5x5	4096	1	0
13	ReLU	–	–	–	–
14	Dropout	–	–	–	–
15	Convolutional	1x1	250	1	0

3.4.1 Architektur von *DeepSketch*

Wie im vorherigen Kapitel deutlich wurde, bietet DeepSketch im unoptimierten Zustand die beste Erkennungsrate und soll damit als Grundlage für diese Arbeit dienen. Die Autoren Seddati, Dupont und Mahmoudi beschreiben die Architektur von DeepSketch wie in Tabelle 3.2 abgebildet.

Auffällig bei der Architektur sind die geringe Filtergröße in Layer 1, der häufige Einsatz von Zero-Padding und die hohe Anzahl an Filtern und deren Größe in Layer 12. Die Filtergröße in Layer 1 begründen die Autoren damit, dass Sketche im Gegensatz zu Fotos häufiger klare Kanten besitzen, die so einfacher zu detektieren seien. Der Kantenerkennung im ersten Layer kommt weiterhin zu Gute, dass die Skizzen ja bereits nur aus zwei Farben bestehen, der Kontrast also maximal ist.

Die Verwendung von Zero-Padding soll der Translationsinvarianz dienen. So werden Merkmale, die sich außerhalb der Mitte befinden, auch nach mehrmaligem Falten und Downsampling durch die Convolutional und Max Pooling Layer noch erfolgreich erkannt.

Die erhöhte Anzahl an Filtern und deren Größe (5×5) in Layer 12 sorgt dafür, dass das Netzwerk in der Lage ist die erkannten Merkmale gut in eine räumliche Beziehung zueinander zu setzen. Anders als bei Fotos, wo Informationen

wie Texturen oder Farbe weiteren Kontext liefern, fehlt dieser bei den Skizzen. Daher ist die räumliche Bezugnahme von hoher Bedeutung.

3.5 Hyperparameteroptimierung

Neben den eigentlichen Parametern der Neuronen, die während des Trainings erlernt werden, gibt es noch eine ganze Reihe anderer Parameter, die das endgültige Modell beeinflussen. Diese werden Hyperparameter genannt. Zum Training eines künstlichen neuronalen Netzes gehört die Suche nach den optimalen Werten dieser Hyperparameter ebenso dazu, wie das eigentliche Finden der optimalen Werte für die Parameter.

Da die Architektur des Netzes aus [47] übernommen wurde, wird in dieser Arbeit darauf verzichtet, eventuelle Layer Blöcke als Hyperparameter zu betrachten und deren beste Konstellationen zu suchen.

Im Folgenden wurden die Optimierungsfunktion der Fehlerrückfühung, die *batch size* und die Anzahl der Filter im letzten Convolutional Layer als zu untersuchende Hyperparameter identifiziert.

3.5.1 Optimierungsfunktion der Fehlerrückführung

Die Wahl der Optimierungsfunktion der Fehlerrückführung hat Auswirkungen auf die maximale Erkennungsrate des Modells und die Dauer einer Epoche. Sie ist zum Beispiel dafür zuständig, dass das Finden der optimalen Parameterkombination für die Gewichte der einzelnen Neuronen funktioniert und die Minimierung der Fehlerfunktion nicht in einem lokalen Minimum versiegt. Da sich die möglichen Optimierungsfunktionen in ihrem erzielten Ergebnis je nach Einsatzgebiet unterscheiden, gilt es die für die vorliegende Arbeit beste zu finden.

In Keras sind SGD, RMSprop[53], Adagrad[11], Adadelta[59], Adam[29], Adamax[29] und Nadam[9] implementiert.

3.5.2 Batchgröße und Anzahl der Filter

Ein Hyperparameter der nicht direkt mit der Architektur des CNN und dem daraus entstehenden Modell verknüpft ist, sondern mit dessen Trainingsvorgang ist die *batch size*. Diese Stellgröße definiert nach wie vielen Samples die Fehlerrückführung stattfinden soll und damit die Gewichte des Netzwerks aktualisiert werden. Die batch size hat dabei Einfluss auf die Dauer des Trainings. Hierbei korreliert die Größe der batch size nicht mit der Dauer. In Versuchen

wurde die längste Dauer für eine batch size von 32 gemessen (58 min und 24 s für 50 Epochen), während das gleiche Parameterset bei einer batch size von 128 nur 48 min und 1 s für 50 Epochen benötigte.
Die Frage, obb die batch size Einfluss auf die Erkennungsrate hat, ist Gegenstand von Kapitel 4.1.3.

Die von Seddati, Dupont und Mahmoudi beschrieben Architektur von DeepSketch sieht in Layer 12 4096 Filter vor. Die Autoren argumentieren, dass diese hohe Anzahl nötig sei, um möglichst viele der erkannten Features in räumliche Beziehung zueinander zu setzen. Zur Überprüfung dieser Hypothese wird die Filteranzahl in Layer 12 als Hyperparameter ausgewählt und im Folgenden getestet.

Kapitel 4

Ergebnisse

Die in Kapitel 3.3 beschriebenen Preprocessingmethoden wurden auf die von Eitz, Hays und Alexa bereitgestellten PNG Dateien angewandt und das Ergebnis gespeichert. Das anschließende Training der Netze ist Gegenstand dieses Kapitels.

4.1 Training des Netzwerks

Mit der Möglichkeit einzelne Striche eines Sketches zu entfernen (siehe 3.3.4) ergeben sich drei verschiedene Ansätze für das Training von DeepSketch:

1. als Eingabedaten die 20.000 vorverarbeiteten Skizzen aus dem Datensatz verwenden
2. einzelne Striche entfernen und die so entstandenen 80.000 Dateien in das gleiche Netzwerk geben
3. aus den Sketches Sequenzen anfertigen und diese einer LSTM als Eingabedaten zuführen

Das Ergebnis dieser drei Ansätze ist in Abbildung 4.1 zu sehen. Die Abbildung zeigt die Top1-Genauigkeit der Kandidaten. Zum direkten Vergleich sind die Erkennungsraten eines Menschen (Quelle: [13]), die der von [13] vorgestellten SVM und die von DeepSketch und Sketch-a-Net angegebenen hinzugefügt. Da sich letztere um weniger als ein Prozentpunkt unterscheiden wurden diese aus Gründen der Übersicht zusammengefasst.

4.1.1 DeepSketch als Netzwerkarchitektur

Als erstes ist eine Implementation der DeepSketch Architektur in Keras entstanden (siehe Tabelle 4.1). Diese orientiert sich stark am Vorbild, weist aber einige Unterschiede auf. Die Layer 3, 6 und 13 sind nicht im Originalentwurf

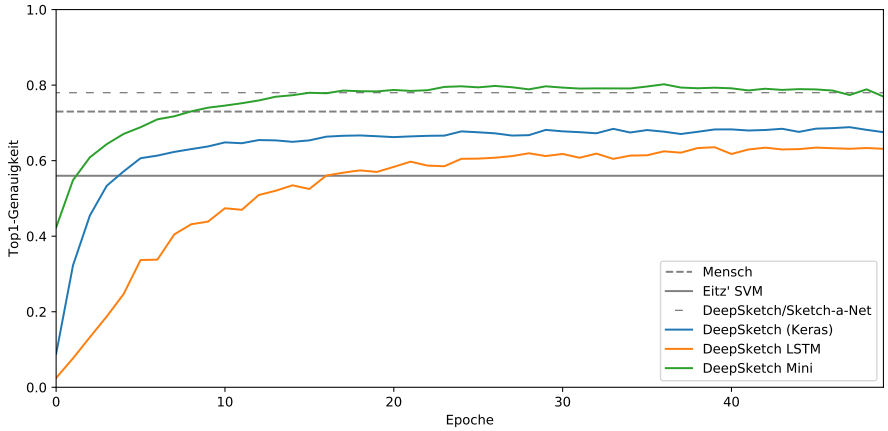


Abbildung 4.1: Ergebnis der Evaluation: DeepSketch (Keras) erreicht maximal 68,87%, DeepSketchMini 80,22% und DeepSketchLSTM 63,56% Top1-Genauigkeit.

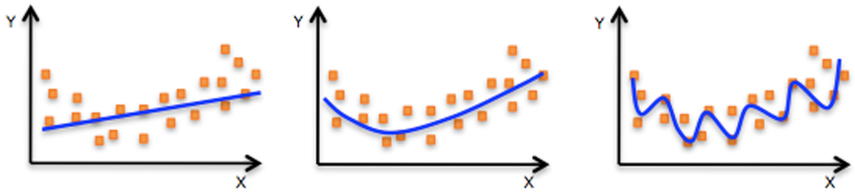


Abbildung 4.2: Gefahren beim Training von Klassifizieren: underfitting (links) und overfitting (rechts). Ein korrektes Training gleicht der Grafik in der Mitte. [19]

vorhanden, mussten aber für den Einsatz mit Keras ergänzt werden. Die Funktion der Zero-Padding Layer ist selbsterklärend, während der Flatten Layer eine Dimensionsreduktion seines Eingabevolumens vornimmt. Dies ist nötig um den Output des Convolutional Layers an Stelle 12 in den Dense Layer an Position 14 zu übergeben. *Dense* ist dabei der Keras-eigene Name für einen Fully Connected Layer. Layer 14 taucht in der Arbeit von Seddati, Dupont und Mahmoudi nicht auf. Dort wird die Ausgabe des Convolutional Layers verwendet, da die Autoren nicht an einer Klassifizierung, sondern nur an den vom Netz gewonnenen Features interessiert sind. Da die vorliegende Arbeit aber eine Klassifizierung des Datensatzes anstrebt, wurde der letzte Layer hinzugefügt und mit einer Softmax Aktivierung versehen.

4.1.2 Over- und underfitting vermeiden

Bevor das Training eines Klassifikators begonnen werden kann, müssen noch die Begriffe *under-* und *overfitting* (dt.: *Unter-* und *Überanpassung*) eingeführt werden. Beides sind Phänomene, die, wenn sie auftreten, das Ergebnis stark beeinträchtigen.

Tabelle 4.1: Die Architektur der DeepSketch Implementierung in Keras

	Typ	Filtergröße	-anzahl	Stride	Aktivierung
1	Convolutional	7x7	64	2	ReLU
2	Maxpool	3x3	–	2	–
3	Zero-Padding	2x2	–	–	–
4	Convolutional	5x5	128	2	ReLU
5	Maxpool	3x3	–	2	–
6	Zero-Padding	2x2	–	–	–
7	Convolutional	3x3	256	1	ReLU
8	Convolutional	3x3	512	1	ReLU
9	Maxpool	3x3	–	2	–
10	Convolutional	5x5	4096	1	ReLU
11	Dropout	–	–	–	–
12	Convolutional	1x1	250	1	ReLU
13	Flatten	–	–	–	–
14	Dense	–	250	–	Softmax

Ein Modell wird als underfitted (siehe Abbildung 4.2 links) bezeichnet, wenn der vorliegende Datensatz nicht genau genug klassifiziert werden kann. Die errechneten Klassen bilden dabei nicht die korrekten Klassen ab. Dies kann an einem zu komplexen Datenbestand oder einem zu simplen Modell liegen. Eine Erhöhung der Modellkomplexität (etwa durch das Hinzufügen weiterer Hidden Layer) kann hier entgegenwirken.

Overfitting bezeichnet dabei das genaue Gegenteil: das Modell passt sich dem Datenbestand zu gut an und ist nicht mehr fähig auf neue, bisher unbekannte Eingaben zu reagieren. Das Modell lernt den Datensatz sozusagen *auswendig* und erreicht während der Evaluation so Erkennungsraten von bis zu 100%. Gründe hierfür können ein zu simpler Datensatz oder ein zu langes Training sein. Maßnahmen gegen das Overfitting sind Gegenstand aktueller Forschung. Der eingangs erwähnte Dropout Layer (siehe Kapitel 2.1) stellt ein Beispiel hierfür dar. Der Dropout Layer sorgt dafür, dass nachstehende Neuron randomisiert beim Training ausgelassen werden. So sorgt er dafür, dass die noch aktiven Neuronen sich weiter generalisieren müssen, um einen korrekten Output zu liefern. Einer zu großen Spezialisierung der Neuronen wird so zuvorgekommen. Dropout findet in den hier verwendeten Netzen Anwendung (siehe Layer 11, Tabelle 4.1).

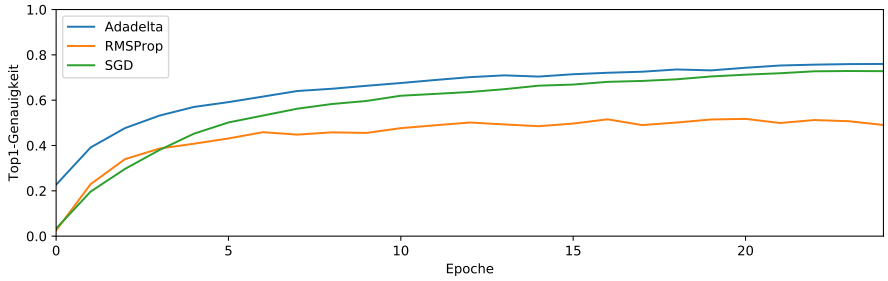


Abbildung 4.3: Adadelta erzeugt unter den getesteten Optimierungsfunktionen das beste Ergebnis.

4.1.3 Wahl der Hyperparameter

Optimerungsfunktion der Fehlerrückführung

In der Auswertung der verschiedenen Optimierungsmethoden für die Fehlerrückführung während des Trainings (siehe Abbildung 4.3) wird deutlich, dass diese sich hauptsächlich in Geschwindigkeit und maximal erreichter Genauigkeit unterscheiden.

Für den abgebildeten Versuch wurden alle in Keras implementierten Optimierer[1], die in ihren Standardeinstellungen ein Ergebnis lieferten, getestet. Dies sind: Adadelta [59], RMSProp, und SGD.

Da Adadelta am schnellsten konvergiert, wird dieser Optimierer für alle Experimente in dieser Arbeit verwendet.

Batch Size und Filteranzahl

In Kapitel 3.5.2 sind die Hyperparameter batch size und Filteranzahl als mögliche Kandidaten zur Verbessrung der Erkennungsleistung des Netzes identifiziert worden. Um die beste Wertekombination zu testen wurde für jeden Parameter ein Wertebereich definiert und pro Wertebereich vier Kandidaten gewählt.

Für die batch size sind dies 32, 48, 64 und 128. Die Obergrenze für diesen Bereich leitet sich aus der Literatur ab. Sketch-a-Net bespielsweise benutzt eine batch size von 135. Dieser Wert wurde auf das nächste Vielfache von zwei abgerundet (128), zwei Mal halbiert (64 und 32) und um genaue Kenntnis über eine kleinere batch size (und damit das häufige Anwenden der Fehlerrückführung) zu erhalten, wurde das letzte Interval nochmals halbiert (48 als Mitte zwichen 32 und 64).

[1] Siehe: `https://keras.io/optimizers/`, zuletzt abgerufen am 13.03.2017.

Tabelle 4.2: Evaluationsmatrix der Hyperparameteroptimierung. batch size (senkrecht) / Filteranzahl (horizontal). Die Werte zeigen die maximal erreichte Top1-Genauigkeit in Prozent.

	512	**1024**	**2048**	**4096**
32	66.09	66.60	66.25	66.68
48	66.24	66.69	66.26	66.31
64	66.50	66.35	66.34	66.82
128	65.00	66.19	66.24	65.65

Bei der Anzahl der Filter in Layer 10 wurde die Obergrenze durch die zur Verfügung stehende Hardware gegeben. Mehr als 4096 Filter waren nicht im Arbeitsspeicher haltbar. Die anderen Parameterausprägungen ergeben sich wieder durch weiteres Halbieren der Obergrenze (2048, 1024 und 512).

Für jede mögliche Parameterkombination wurde ein Netzwerk für 50 Epochen lang auf dem gesamten Datensatz trainiert. Das Ergebnis ist Tabelle 4.2 zu entnehmen.

Die Parameterkombination mit der höchsten erreichten Top1-Genauigkeit ist batch size = 64 und Filteranzahl = 4096 und wird im Laufe dieser Arbeit für das Training jedes Netzwerks verwendet.

Auffällig ist, dass diese Werte alle sehr nah beieinander liegen. Die minimal erzielte Genauigkeit liegt hier bei 65%. Der Mittelwert der Ergebnisse liegt bei 66,26% und die Standardabweichung beträgt gerade einmal 0,42. Der Einfluss der Hyperparameter batch size und Filteranzahl kann also als gering für das verwendete Netz und dessen Architektur eingestuft werden.

Mit diesem Versuch kann also die These der Autoren Seddati, Dupont und Mahmoudi belegt werden. Eine hohe Filterzahl gegen Ende der Netzwerkarchitektur fördert eine erfolgreiche Klassifizierung.

4.1.4 Normalisierung der Eingabedaten

Da es sich bei den vorliegenden Daten um *binäre* Werte handelt (jeder Pixel in einem Sketch hat nach dem Preprocessing entweder den Wert 0 oder 1), ist eine Standardisierung nicht nur nicht nötig, sondern steht einer erfolgreichen Verarbeitung sogar im Wege (siehe Abbildung 4.4).

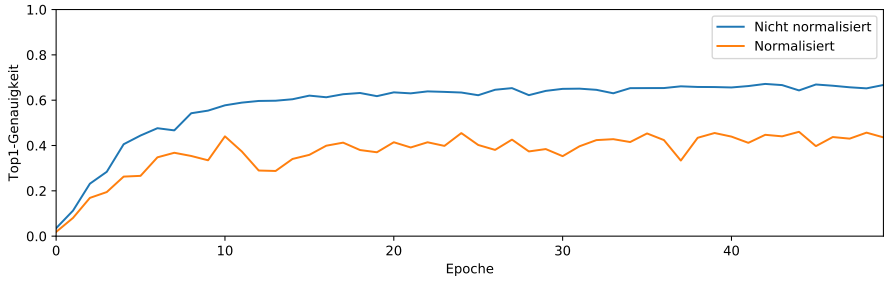


Abbildung 4.4: Vergleich der Top1-Genauigkeit von DeepSketch mit und ohne Normalisierung der Eingangsdaten.

Ein weiteres Problem stellt die Verarbeitung der normalisierten Daten mit dem Keras `ImageDataGenerator` dar (siehe hierzu Kapitel 3.3.2). Da die Transformationen erst nach der Normalisierung angewendet werden, wird der den Sketch umgebende Bereich mit einer konstanten Farbe aufgefüllt. In diesem Falle berechnet Keras diese Farbe aus dem Mittelwert des Samples. Dies ergibt ein dunkles Grau. Dadurch bilden sich unnatürliche Kanten in dem neu entstandenen Bild, die vom Netzwerk als Merkmal erkannt werden und so die Erkennungsrate mindern. Diese Phänomene sind Abbildung 3.3 zu entnehmen.

Für die folgenden Versuche wurde daher auf eine Normalisierung des Datensatzes verzichtet.

4.2 Ergebnisse des Trainings

Die drei vorgestellten Ansäze wurden alle mit einer batch size von 64, 4096 Filtern in Layer 10 und Adadelta als Optimierungsfunktion trainiert. Sofern nicht anders angegeben wurde für 50 Epochen mit einem dreigeteilten Datensatz (Trainingsdatensatz doppelt so groß wie Testdatensatz) trainiert.

4.2.1 CNN mit unveränderten Sketches als Eingabedaten

Der erste Ansatz (im Folgenden *DeepSketch (Keras)* genannt) war es die Originalbilder vorzuverarbeiten und dem Netz zum Training zu zeigen. Hierfür mussten die Sketches auf 224×224 Pixel (von ursprünglich 1111×1111 Pixel) verkleinert werden. Die Anzahl der Sketches wurde durch die Verwendung randomisierter Transformationen (siehe Kapitel 3.3.2) vervierfacht, sodass die Epochenlänge für dieses Training auf 52.800 für den Trainingsdatensatz und 27.200 für den Testdatensatz angestiegen ist. Das Ergebnis sind 68,87% Top1-Genauigkeit in der Klassifizierung des Testdatensatzes. Damit liegt DeepSketch (Keras) knapp vier Prozent unter der Leistung eines Menschen.

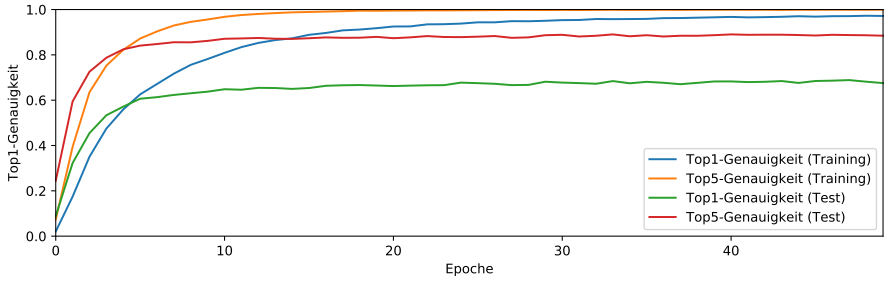


ABBILDUNG 4.5: DeepSketch (Keras) Erkennungsraten für einen Durchlauf mit 50 Epochen.

Bei der Betrachtung der Ergebnisse fällt auf, dass die Top5-Genauigkeit für das Testset einen Wert von genau 1.0, also eine Erkennungsrate von 100% erreicht. Dies kann ein Indikator für ein Overfitting des Modells sein. Maßnahmen wären hier das Netzwerk zu vereinfachen oder den Datensatz komplexer zu machen. Letzteres wurde in Versuch 2 mit der Erkennung partieller Sketches probiert (siehe Kapitel 4.2.2).

Die von Eitz, Hays und Alexa vorgestellte Arbeit enthält zur Klassifizierung eine SVM. Diese funktioniert jedoch erst nach Eingabe eines feature descriptors zur Beschreibung der Merkmale eines Sketches. Während dieser Beschreibungsformalismus bei den Autoren händisch angefertig wurde, lernen CNNs die zur Beschreibung der Eingabedaten notwendigen Merkmale eigenständig. Zum Verständnis dieser Arbeitsweise ist es möglich die erlernten Gewichte des ersten Convolutional Layers zu visualisieren (siehe Abbildung 4.6). Die Visualisierung zeigt auf der linken Seite eine Matrix, die die 64 Gewichte stark vergrößert abbildet. Zur Aktivierung wurden der Rahmen eines Karos (Mitte) und ein gefülltes Quadrat (rechts) verwendet. Die jeweils nur 7×7 Pixel großen Filter bestimmen auf welche Eingabewerte das jeweilige Neuron reagiert. Die orange markierten Beispiele zeigen einen vertikalen Verlauf. Die Aufhellung im rechten Bild an der linken Kante des Quadrats zeigt, dass dieser Filter durch die Eingabe aktiviert wurde. Gleiches ist bei den blau markierten Filtern und deren Aktivierungen durch das Karo zu erkennen.

Um zu verstehen, warum das Netzwerk in knapp einem Drittel der Fälle eine falsche Klasse zuweist wurden zunächst die Klassen mit den meisten falschen Vorhersagen gefunden. Hierzu wurden alle Sketches dem trainierten Netzwerk gezeigt und jene gezählt, bei denen die korrekte Klasse nicht unter den fünf vom Netzwerk als wahrscheinlichst (Top5-Genauigkeit) ausgewählten zu finden war.

Bei genauerer Durchsicht der Klassen mit den drei höchsten Fehlerwahrscheinlichkeiten (*knife, lobster* und *bulldozer*) wird auch klar, warum das Netzwerk diese nicht korrekt klassifiziert. Die Zeichnungen sind nicht repräsentativ für eine einzelne Klasse. Sie sind entweder schlecht gezeichnet oder

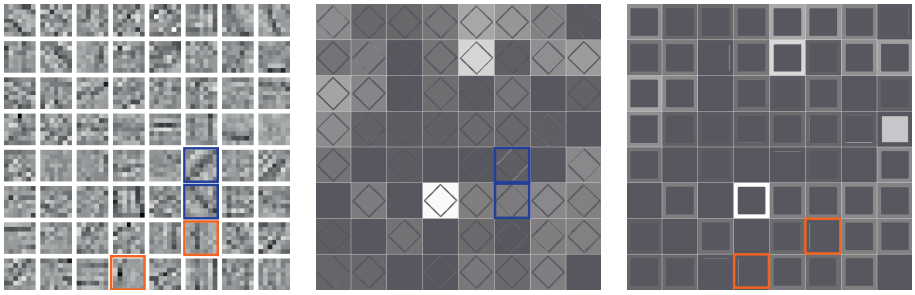


Abbildung 4.6: Visualisierung der erlernten Filter des ersten Convolutional Layers (links) und eine Darstellung der Aktivierungen mit einer Raute (Mitte) und einem Quadrat (rechts). Die hervorgehobenen Aktivierungen zeigen, welcher Filter auf welche Teile des Bildes reagiert (helle Stellen).

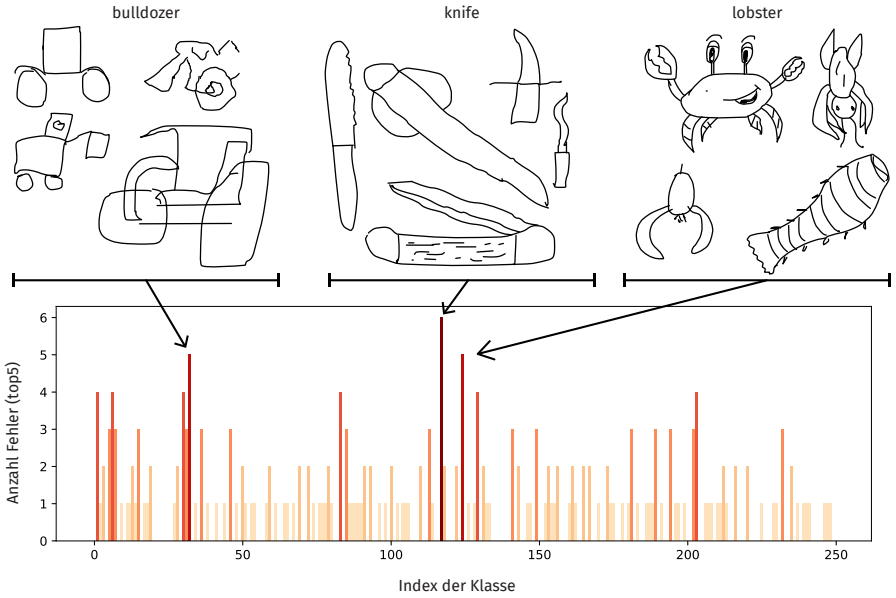


Abbildung 4.7: Beispiele für die Klassen mit der höchsten Anzahl an falsch erkannten Skizzen.

aber die Klasse ist so grob gewählt, dass die Person, die den Sketch angefertigt hat, sich nicht sicher gewesen sein muss, was von ihr verlangt wurde. Die Klasse *knife* beispielsweise zeigt deutlich, dass zum einen Brotmesser (oben links), Klappmesser (unten Mitte) oder Dolche (rechts) gezeichnet wurden. Dies kann auf Ungenauigkeiten in der Sprache zurückgeführt werden. Hier hätte eine genauere Spezifizierung der Klasse geholfen.

Die Klasse *bulldozer* weist hier Beispiele für unzureichende Fertigkeiten der Probanden die Klasse zu visualisieren auf. Hier hätte eine vereinfachte Eingabe, zum Beispiel die Verwendung eines Grafiktabletts, statt einer Computermaus möglicherweise geholfen, einfacher zu erkennende Skizzen anzufertigen.

Die Klasse *lobster* lässt vermuten, dass Probanden schlichtweg nicht wussten, wie ein Hummer aussieht. So wäre die anthropomorphisierte Krabbe oben links und die Skizze unten rechts beispielsweise zu erklären.

Die Beispiele verdeutlichen außerdem, warum auch Menschen keine Erkennungsrate von 100% erreichen. Den Problemen des Datensatzes widmet sich unter Anderem die Arbeit von Schneider und Tuytelaars [45]. Die Autoren zeigen in ihrer Arbeit, dass 40% der von Menschen gemachten Fehlern auf unzureichend ausgesuchte Kategorien zurückzuführen sind. Der Datensatz enthält beispielsweise die Klassen *Chair* und *Armchair*. Die Maschine kann hier unterscheiden, der Mensch kann dies erst nach langer Eingewöhnungsphase. Er muss erst alle Klassen des Datensatzes verinnerlichen um das korrekte Label anzuwenden.

Die Autoren haben 2014 eine neue Version des Datensatzes veröffentlicht. Zur Erstellung haben sie zunächst alle von Menschen falsch klassifizierten Sketche aus dem TU Datensatz entfernt und danach nur Klassen mit mehr als 56 (=70%) verbliebenen Sketchen beibehalten. Übrig blieben so 160 Klassen mit jeweils mindestens 56 Skizzen.

4.2.2 CNN zur Erkennung partieller Sketches

Für den zweiten Ansatz (im Folgenden *DeepSketchMini* genannt) wurden, wie in Kapitel 3.3.4 beschrieben, aus jedem Sketch vier Vollständigkeitsstufen generiert, sodass bei diesem Training ebenfalls Epochenlängen von 52.800 bzw. 27.200 Samples entstehen. Da hierbei aber tatsächlich viermal so viele Daten wie bis dahin vorliegen, wird auch viermal so viel Arbeitsspeicher benötigt. Da so viel Arbeitsspeicher aber nicht zur Verfügung stand, musste die Größe der Datengrundlage verringert werden. Hierzu wurden die Sketches noch einmal verkleinert und liegen nun in einer Größe von 168×168 Pixel vor.

Tabelle 4.3: Die Architektur von DeepSketch Mini zur Erkennung partieller Sketches.

	Typ	Filtergröße	-anzahl	Stride	Aktivierung
1	Convolutional	7x7	64	2	ReLU
2	Maxpool	3x3	–	2	–
3	Zero-Padding	2x2	–	–	–
4	Convolutional	5x5	128	2	ReLU
5	Maxpool	3x3	–	2	–
6	Zero-Padding	2x2	–	–	–
7	Convolutional	3x3	256	1	ReLU
8	Convolutional	3x3	512	1	ReLU
9	Maxpool	3x3	–	2	–
10	Convolutional	**3x3**	4096	1	ReLU
11	Dropout	–	–	–	–
12	Convolutional	1x1	250	1	ReLU
13	Flatten	–	–	–	–
14	Dense	–	250	–	Softmax

Damit das Netzwerk die verkleinerten Sketches entgegennehmen konnte, musste die Architektur verändert werden (siehe Tabelle 4.3). Dazu ist die Filtergröße des Convolutional Layers an Stelle 10 von 5×5 auf 3×3 verringert worden.

Das so trainierte Netzwerk erreicht mit 80,22% Top1-Genauigkeit die höchste Erkennungsrate der hier gezeigten Ansätze. Diese übertrifft die der von Eitz, Hays und Alexa für einen Menschen angegeben (73,10%) um sieben Prozentpunkte.

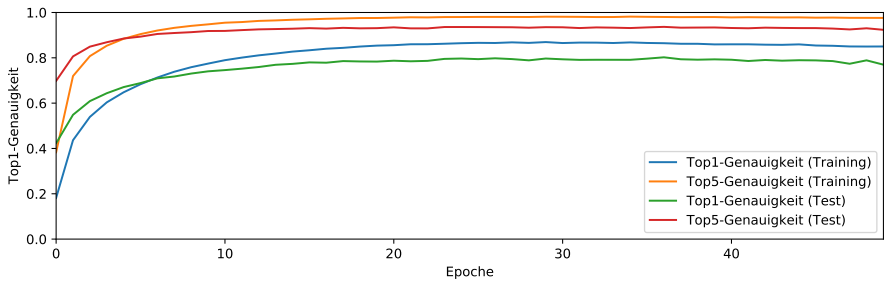


Abbildung 4.8: DeepSketchMini Erkennungsraten für einen Durchlauf mit 50 Epochen.

TABELLE 4.4: Die Architektur von DeepSketch LSTM zur Erkennung von zeitlichen Informationen in Sketches. TD steht hier als Abkürzung für den Keras Layertypen `TimeDistributed`.

	Typ	Filtergröße	-anzahl	Stride	Aktivierung	TD
1	Convolutional	7x7	64	2	ReLU	✔
2	Maxpool	3x3	–	2	–	✔
3	Zero-Padding	2x2	–	–	–	✔
4	Convolutional	5x5	128	2	ReLU	✔
5	Maxpool	3x3	–	2	–	✔
6	Zero-Padding	2x2	–	–	–	✔
7	Convolutional	3x3	256	1	ReLU	✔
8	Convolutional	3x3	512	1	ReLU	✔
9	Maxpool	3x3	–	2	–	✔
10	Convolutional	3x3	4096	1	ReLU	✔
11	Dropout	–	–	–	–	✔
12	Convolutional	1x1	250	1	ReLU	✔
13	Flatten	–	–	–	–	✔
14	LSTM	–	512	–	tanh	✗
15	Dense	–	250	–	Softmax	✗

4.2.3 Kombination aus CNN und LSTM zur Erkennung von Sketchsequenzen

Für den dritten Ansatz (im Folgenden *DeepSketchLSTM* genannt) wurde versucht die Merkmalsbestimmung eines CNN mit der Fähigkeit eines RNN zeitliche Zusammenhänge zu erkennen zu paaren. In der Fachliteratur ist dieser Ansatz als Convolutional, Long Short-Term Memory, Fully Connected Deep Neural Network (CLDNN) bekannt [42].

Um aus den eigentlichen Sketchen Zeitinformationen zu generieren, wurde wieder auf das Löschen einzelner Striche zurückgegriffen. Dies wurde jedoch dahingehend erweitert, dass sichergestellt wurde, dass jede Vollständigkeitsstufe einen klar definierten Sequenzindex bekam und so einem konkreten *Zeitpunkt*[2] zugeordnet werden konnte. So sind insgesamt 20.000 Sequenzen (pro Bild im Originaldatensatz eine Sequenz) mit jeweils vier Bildern entstanden. Für das Training standen so 13.200 Sequenzen (=52800 Einzelbilder) und für den Test weitere 6800 Sequenzen (=27200 Einzelbilder) zur Verfügung.

[2] Mit Zeitpunkt ist hier kein konkretes Datum, sondern der Zeitpunkt der Erstellung relativ zu dem der anderen Vollständigkeitsstufen gemeint. Beispiel: die 25% Vollständigkeitsstufe wird mit Index 1 versehen und ist damit zeitlich klar vor der 75% Vollständigkeitsstufe (Index 3) einzuordnen.

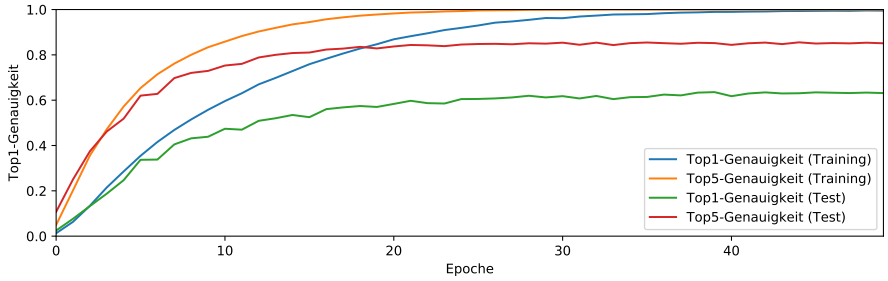


Abbildung 4.9: DeepSketchLSTM Erkennungsraten für einen Durchlauf mit 50 Epochen.

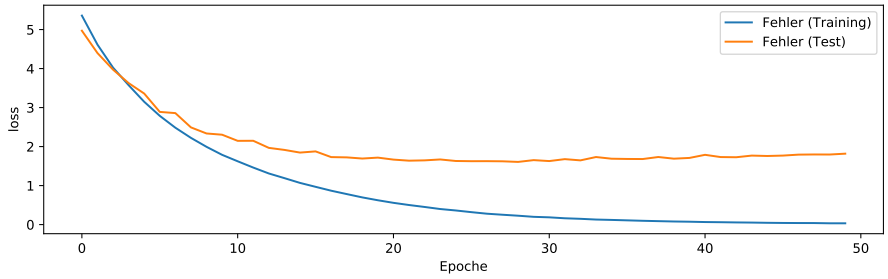


Abbildung 4.10: DeepSketchLSTM loss für einen Durchlauf mit 50 Epochen. Optimierungsfunktion ist Adadelta.

Damit das Netzwerk die neue Datenstruktur verarbeiten kann musste dessen Eingabevolumen um eine Dimension erweitert werden. Diese neue Dimension enthält den Sequenzindex jedes Sketches. Für diese Extrainformation stellt Keras den Layer Typen `TimeDistributed` zur Verfügung. Wie in Tabelle 4.4 zu erkennen ist wurden alle Layer bis hin zum LSTM Layer in einen Time Distributed Layer eingebettet. Der LSTM Layer selber arbeitet mit 512 internen Neuronen und wird über eine Tangens Hyperbolicus Funktion aktiviert. Der anschließende Dense Layer deckt sich mit denen in den anderen Netzen und muss nicht in einen Time Distributed Layer eingebettet werden, da der vorangehende LSTM Layer die Sequenzinformationen nicht im Output beibehält. Die Klassifizierung findet weiterhin im letzten Layer mit einer Softmax Funktion statt.

DeepSketchLSTM erreicht mit 63,56% Top1-Genauigkeit den schlechtesten Wert im Test und liegt damit zwischen der SVM (56%) der Autoren Eitz, Hays und Alexa und dem Ergebnis von DeepSketch (Keras).

Ein möglicher Grund für die Leistung des Netzwerks wird in Abbildung 4.10 deutlich. Die Abbildung zeigt den loss für Training- und Testset über den gesamten Trainingsverlauf. Hierbei fällt auf, dass der loss Wert für das Testset im Verlauf aufhört zu sinken und gegen den Wert 2 zu konvergieren scheint.

Eine Vermutung ist, dass dies mit der verwendeten Optimierungsfunktion

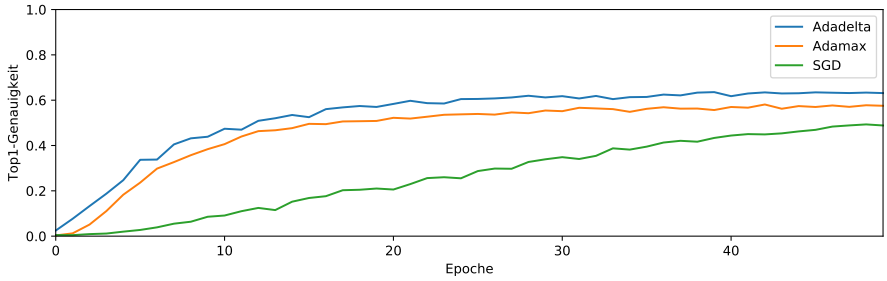


Abbildung 4.11: Adadelta ist weiterhin die beste Optimierungsfunktion für DeepSketchLSTM.

Adadelta zusammenhängt. In einem weiteren Versuch wurden erneut alle Optimierungsfunktionen mit DeepSketchLSTM getestet. Im Gegensatz zu DeepSketch (Keras) erzeugt RMSProp kein Ergebnis, dafür jedoch Adamax. Für 50 Epochen wurden so SGD, Adamax und Adadelta erneut getestet. Das Ergebnis ist Abbildung 4.11 zu entnehmen.

Adadelta bleibt weiterhin die Optimierungsfunktion der Wahl. Während des gesamten Trainingsverlaufes erzielt diese rund sechs Prozentpunkte mehr, als Adamax bei der Top1-Genauigkeit im Testset. Eine Begründung für die Performance des gesamten Netzwerks ist somit nicht gegeben.

Kapitel 5

Eine interaktive Anwendung zum Testen der Erkennungsrate

Ebenfalls im Rahmen dieser Arbeit ist die Anwendung *Draw me something!* entstanden. Diese soll auf spielerische Weise ein vereinfachtes Interface zu den Netzwerken bereitstellen. So soll es möglich sein, über ein angeschlossenes Eingabegerät oder den Touchscreen einen Sketch zu zeichnen und dem Netzwerk als Input zu liefern.

Die Antwort des Netzwerks soll anschließend sichtbar gemacht werden. Es sollen die fünf Klassen mit den höchsten Wahrscheinlichkeiten einer korrekten Vorhersage ausgegeben werden. Die wahrscheinlichste Klasse soll dabei gesondert dargestellt werden.

5.1 Realisierung als Browseranwendung

Um die Hürde für den Benutzer so gering wie möglich zu gestalten, wird die Anwendung als Webapp realisiert. Jegliche für den Gebraucht benötigte Software ist also ein moderner Browser. Dieser ist auf nahezu jedem aktuellen Betriebssystem (egal ob Desktop- oder Mobilgerät) bereits vorinstalliert.

Dem User wird eine Leinwand mit der Aufforderung *Draw me something!* präsentiert (siehe Abbildung 5.1). Nach jeder Eingabe eines Striches soll das Netzwerk befragt werden und eine Vorhersage treffen. Diese wird dann wiederum dem Benutzer angezeigt.

Draw me something! ist als Client-Server-Anwendung realisiert. Ein in ECMAScript 2015 (ES2015) geschriebenes Frontend schickt nach jedem Strich die gesamte Skizze an ein Python Backend. Das Backend übernimmt dann die Vorverarbeitung der Eingabedaten und die Übergabe an das Netzwerk. Die Antwort des

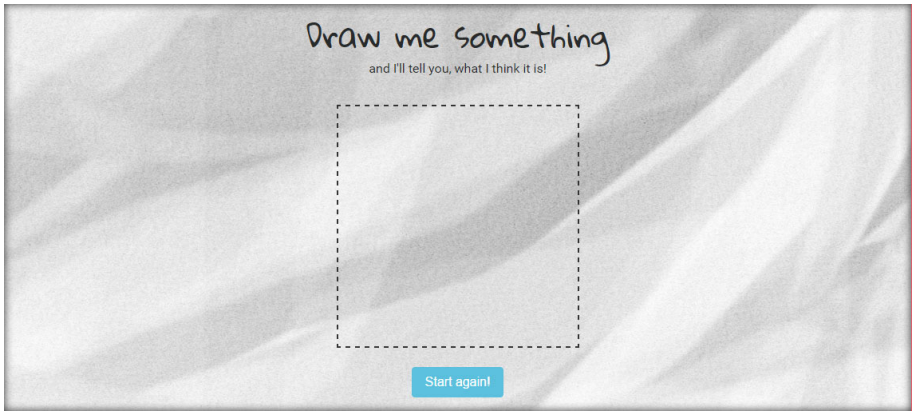


Abbildung 5.1: Der Startzustand der entstandenen Webapp *Draw me Something!*

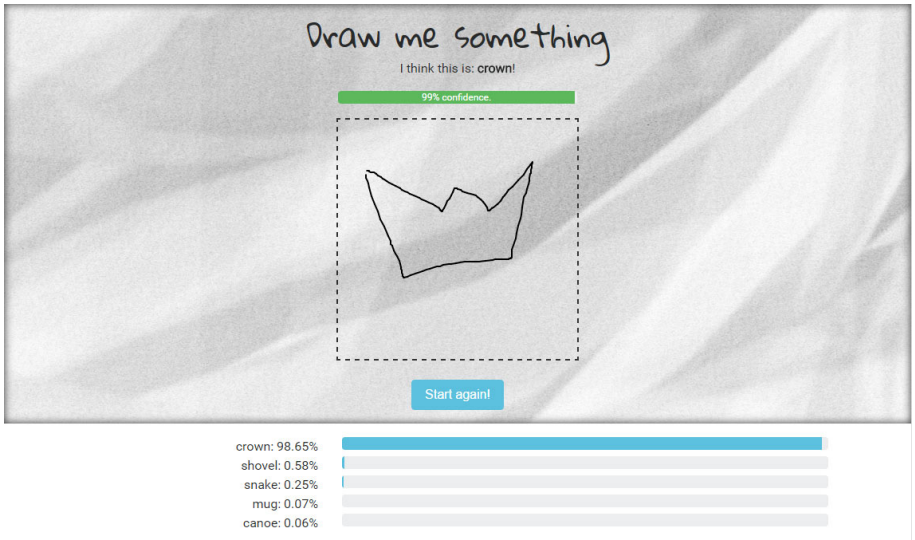


Abbildung 5.2: Die Anwendung stellt die Klasse mit der höchsten Top1-Genauigkeit oberhalb der canvas dar. Die Top5-Klassen werden unterhalb in einer Liste dargestellt.

Backends enthält die Vorhersagen des Netzwerkes, welche dann im Frontend wiederum angezeigt werden.

Die Übermittlung der Skizze erfolgt als String via HTTP Post. Hierfür wird die *canvas* API des HyperText Markup Language Version 5 (HTML5) Standards verwendet. Diese bietet die Möglichkeit die Inhalte des Elements, als base64 kodierten String auszugeben. Für die Kommunikation mit dem Backend und die Interaktion mit der canvas wird *Angular.js*[1] in Version 1.6 verwendet.

Das Backend beruht auf dem Python Framework *flask*[2] in Version 0.12. Es nimmt die Zeichnung als base64 String entgegen und erzeugt zunächst wieder ein Bild aus den Daten. Diese werden, wie in Kapitel 3.3 beschrieben,

[1] Webseite des Angular.js Projekts: `https://angularjs.org`, zuletzt abgerufen am 08.03.2017
[2] Webseite des flask Projekts: `http://flask.pocoo.org/`, zuletzt abgerufen am 08.03.2017

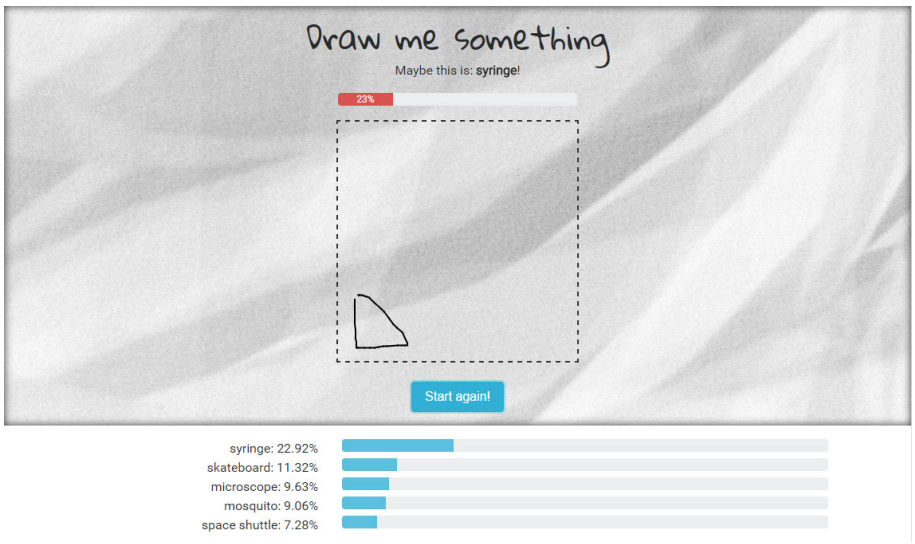


ABBILDUNG 5.3: Die Zuversicht des Netzwerks in die erkannte Klasse wird oberhalb der canvas als Prozentbalken dargestellt und je nach Wert farblich ($\leq$ 25%: rot, $\leq$ 50%: gelb, $\leq$ 75%: blau, $\geq$ 75%: grün) hinterlegt.

vorverarbeitet und anschließend an das Netzwerk übergeben. Die Antwort des Netzwerks wird an das Frontend übermittelt und dort angezeigt (siehe Abbildungen 5.2 und 5.3).

5.2 *Ensemble Fusion* und *democratic vote*

Da das DeepSketch Mini Netzwerk die höchste Erkennungsrate liefert, wird dieses auch in der Webapp verwendet. Um die Erkennung der bisher unbekannten Eingabedaten zu erhöhen, wird auf die Technik der *ensemble fusion* zurückgegriffen. Hierbei wird nicht nur ein einzelnen Netzwerk, sondern mehrere verschieden trainierte Kopien gleichzeitig befragt. Hierfür sind fünf Netzwerke trainiert worden. Da alle Gewichte des Netzwerks mit vor dem Training Zufallsdaten initialisiert werden und das Aufteilen des Datensatzes in Trainings- und Testset ebenfalls randomisiert geschieht, sind die fünf Kandidaten nicht identisch.

Für eine Vorhersage wird nun der Input einzeln in jedes Netz gegeben und dort verarbeitet. Der Output des Netzes wird dann mit denen der anderen vier Kandidaten zusammen gespeichert und an das Frontend übergeben. Das Frontend sucht nun die Vorhersage mit dem häufigsten Auftreten in der Menge an vorhersagen und zeigt diese dem Benutzer an. Da jedes Netz sozusagen eine *Stimme* in der Wahl der korrekten Klasse hat wird diese Vorgehensweise auch *democratic vote* genannt.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Die Wissenschaftler Eitz, Hays und Alexa haben 2012 einen Datensatz bestehend aus 20.000 handschriftlich erfassten Skizzen von Alltagsgegenständen veröffentlicht. Ziel der vorliegenden Arbeit war es nun aktuelle Techniken aus dem Bereich des Deep Learnings anzuwenden um diesen Datensatz zu verarbeiten und so eine Software zu schreiben, die in der Lage ist, bisher unbekannte Skizzen korrekt zu klassifizieren. Ermöglicht wurde dies durch die Verwendung einer CNN Architektur, die auf dem bereits veröffentlichen künstlichen neuronalen Netz DeepSketch beruht.

Die Arbeit zeigt, welche Maßnahmen zur Vorverarbeitung der Eingabedaten nötig sind, welche Hyperparameter des Netzes wie ausgesucht wurden und beschreibt dessen Training.

Zur eigentlichen Erkennung wurden drei Ansätze verfolgt: die Erkennung der Skizzen ohne, dass einzelne Striche entfernt wurden, die Erkennung verkleinerter Skizzen mit entfernten Strichen und die Erkennung von partiellen Skizzen mittels Sequenzinformationen.

Das Ziel der vorliegenden Arbeit war es mittels aktueller Techniken des deep learnings ein System zu entwickeln, das bei der Erkennung von Skizzen eine Erkennungsrate aufweist, die höher als die eines Menschen ist. Das beste Ergebnis liefert dabei die Kombination von *reinem* CNN mit partiellen Sketchen. Die Erkennungsrate der entstandenen Software liegt einem Ergebnis von 80,22% Top1-Genauigkeit damit über der eines Menschen (73,10%).

6.2 Ausblick

Während des Trainings der Netzwerkkandidaten ist immer wieder aufgefallen, dass die zur Verfügung stehende Hardware nicht ausreicht. Deswegen musste zum Beispiel der Testdatensatz mit den partiellen Sketches auf eine Größe von 168×168 Pixel verkleinert werden. Ob die Erkennungsrate bei größeren Sketchdimensionen weiter ansteigt, muss mit einem System mit mehr Arbeitsspeicher überprüft werden.

Sowohl [47] als auch [58] leisten mit ihren vorgeschlagenen Architekturen wertvolle Grundarbeit. Eine Möglichkeit der Optimierung könnte es sein, simultan mehrere Netzwerke zu befragen. Beispielsweise könnte man pro Vollständigkeitsstufe ein einzelnes CNN trainieren und deren Vorhersage gewichtet auswerten. Auch sollte die Möglichkeit Zeitinformationen, als Sequenzdaten codiert, für eine Vorhersage durch ein LSTM zu nutzen, weiter erforscht werden. Die hier präsentierte Erkennungsrate von 63,56% lässt sich sicherlich durch geschicktes Training und Aufbereiten des Datensatzes weiter optimieren. Ebenso verspricht eine Sequenzlänge von mehr als vier Elementen bessere Ergebnisse. Hierbei muss allerdings darauf geachtet werden, dass viele Sketches des Datensatzes über weniger als vier Striche insgesamt verfügen. Eventuell ist es möglich vorhandene Striche zu teilen und so *künstliche* Vollständigkeitsstufen zu generieren.

Die Erfassung von Sequenzinformationen in der präsentierten Webapp stellt auch eine weitere Optimierungsmöglichkeit dar. So wäre es beispielsweise möglich in einer Session die bisherigen Eingaben des Nutzers zu speichern und vereint an den Server zur Auswertung zu schicken, statt wie bisher nach jedem Strich ein einzelnes Bild.

Eine weitere Möglichkeit die entstandene Webanwendung zu optimieren, wäre es den *democratic vote* des Netzensembles zu gewichten. Dafür könnte man während des Trainings die maximal erzielte Genauigkeit speichern und diesen Wert als Gewicht an die Vorhersage des Netzes binden. So haben erfolgreicher trainierte Netze ein höheres *Mitspracherecht* bei der Wahl des richtigen Kandidaten für die Vorhersage.

Abkürzungsverzeichnis

AMT Amazon Mechanical Turk

API Application programming Interface

CLDNN Convolutional, Long Short-Term Memory, Fully Connected Deep Neural Network

CNN Convolutional Neural Network

CPU Central processing unit

ES2015 ECMAScript 2015

GPGPU General-Purpose computing on GPU

GPU Graphics processing unit

HTML5 HyperText Markup Language Version 5

LSTM Long short-term memory

NLP Natural language processing

PIL Python Imaging Library

PNG Portable Network Graphics

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

SGD Stochastic Gradient Descent

SVG Scalable Vector Graphics

SVM Support Vector Machine

XML Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] Xavier Amatriain. *These Were The Best Machine Learning Breakthroughs Of 2016*. 30. Dez. 2016. URL: https://www.forbes.com/sites/quora/2016/12/30/these-were-the-best-machine-learning-breakthroughs-of-2016 (besucht am 20. 03. 2017) (siehe S. 9).

[2] Sercan O. Arik u. a. „Deep Voice: Real-time Neural Text-to-Speech“. In: (25. Feb. 2017). arXiv: 1702.07825v2 [cs.CL] (siehe S. 2).

[3] Bernhard E. Boser, Isabelle M. Guyon und Vladimir N. Vapnik. „A training algorithm for optimal margin classifiers“. In: *Proceedings of the fifth annual workshop on Computational learning theory - COLT '92*. Association for Computing Machinery (ACM), 1992. DOI: 10.1145/130385.130401 (siehe S. 9).

[4] François Chollet. *Keras: Deep Learning library for Theano and TensorFlow*. Version 2.0.2. URL: https://keras.io/ (siehe S. 27).

[5] Dan Cireşan, Ueli Meier und Jürgen Schmidhuber. „Multi-column Deep Neural Networks for Image Classification“. In: *CVPR 2012, p. 3642-3649* (13. Feb. 2012). arXiv: 1202.2745v1 [cs.CV] (siehe S. 8).

[6] Dan Cireşan u. a. „A committee of neural networks for traffic sign classification“. In: *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE. 2011, S. 1918–1921 (siehe S. 8).

[7] Corinna Cortes und Vladimir Vapnik. „Support-vector networks“. In: *Machine Learning* 20.3 (Sep. 1995), S. 273–297. DOI: 10.1007/bf00994018 (siehe S. 8).

[8] Jia Deng u. a. „Imagenet: A large-scale hierarchical image database“. In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE. 2009, S. 248–255 (siehe S. 9).

[9] Timothy Dozat. *Incorporating Nesterov momentum into Adam*. Techn. Ber. Stanford University, Tech. Rep., 2015, 2015 (siehe S. 37).

[10] Kai-Bo Duan und S. Sathiya Keerthi. „Which Is the Best Multiclass SVM Method? An Empirical Study“. In: *Multiple Classifier Systems*. Springer Nature, 2005, S. 278–285. DOI: 10.1007/11494683_28 (siehe S. 10).

[11] John Duchi, Elad Hazan und Yoram Singer. „Adaptive subgradient methods for online learning and stochastic optimization“. In: *Journal of Machine Learning Research* 12.Jul (2011), S. 2121–2159 (siehe S. 19, 37).

[12] Vincent Dumoulin und Francesco Visin. „A guide to convolution arithmetic for deep learning“. In: (23. März 2016). arXiv: 1603.07285v1 [stat.ML] (siehe S. 22).

[13] Mathias Eitz, James Hays und Marc Alexa. „How Do Humans Sketch Objects?“ In: *ACM Trans. Graph. (Proc. SIGGRAPH)* 31.4 (2012), 44:1–44:10 (siehe S. iii, 1–3, 6, 9, 28, 32, 34, 39, 45, 48, 50, 57).

[14] Mathias Eitz u. a. „Photosketcher: interactive sketch-based image synthesis“. In: *IEEE Computer Graphics and Applications* 31.6 (2011), S. 56–66 (siehe S. 2).

[15] Mathias Eitz u. a. „Sketch-based image retrieval: Benchmark and bag-of-features descriptors“. In: *IEEE transactions on visualization and computer graphics* 17.11 (2011), S. 1624–1636 (siehe S. 3).

[16] Mathias Eitz u. a. „Sketch-based shape retrieval.“ In: *ACM Trans. Graph.* 31.4 (2012), S. 31–1 (siehe S. 3).

[17] Kunihiko Fukushima. „Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position“. In: *Biological Cybernetics* 36.4 (Apr. 1980), S. 193–202. DOI: 10.1007/bf00344251 (siehe S. 7).

[18] Felix A Gers, Jürgen Schmidhuber und Fred Cummins. „Learning to forget: Continual prediction with LSTM“. In: *Neural computation* 12.10 (2000), S. 2451–2471 (siehe S. 25).

[19] Amar Gondaliya. *REGULARIZATION IMPLEMENTATION IN R : BIAS AND VARIANCE DIAGNOSIS*. 22. Mai 2014. URL: http://pingax.com/regularization-implementation-r/ (besucht am 26. 03. 2017) (siehe S. 40).

[20] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (siehe S. 24).

[21] Alex Graves und Jürgen Schmidhuber. „Framewise phoneme classification with bidirectional LSTM and other neural network architectures“. In: *Neural Networks* 18.5 (2005), S. 602–610 (siehe S. 25).

[22] Alex Graves und Jürgen Schmidhuber. „Offline handwriting recognition with multidimensional recurrent neural networks“. In: *Advances in neural information processing systems*. 2009, S. 545–552 (siehe S. 25).

[23] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005 (siehe S. 6).

[24] Geoffrey Hinton u. a. „Improving neural networks by preventing co-adaptation of feature detectors“. In: *arXiv preprint arXiv:1207.0580* (2012) (siehe S. 9).

[25] Sepp Hochreiter. „Untersuchungen zu dynamischen neuronalen Netzen“. Diss. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991 (siehe S. 8).

[26] Sepp Hochreiter und Jürgen Schmidhuber. „Long Short-Term Memory“. In: *Neural Computation* 9.8 (Nov. 1997), S. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735 (siehe S. 8, 25).

[27] Nal Kalchbrenner, Edward Grefenstette und Phil Blunsom. „A Convolutional Neural Network for Modelling Sentences“. In: (8. Apr. 2014). arXiv: 1404.2188v1 [cs.CL] (siehe S. 20).

[28] Andrej Karpathy. *CS231n Convolutional Neural Networks for Visual Recognition*. 2016. URL: https://cs231n.github.io/ (besucht am 23. 03. 2017) (siehe S. 23).

[29] Diederik P. Kingma und Jimmy Ba. „Adam: A Method for Stochastic Optimization“. In: (22. Dez. 2014). arXiv: 1412.6980v9 [cs.LG] (siehe S. 37).

[30] Alex Krizhevsky, Ilya Sutskever und Geoffrey Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Advances in neural information processing systems*. 2012, S. 1097–1105 (siehe S. 9).

[31] Yann LeCun, Yoshua Bengio und Geoffrey Hinton. „Deep learning“. In: *Nature* 521.7553 (Mai 2015), S. 436–444. DOI: 10.1038/nature14539 (siehe S. 12, 15).

[32] Yann LeCun, Corinna Cortes und Christopher J.C. Burges. *The MNIST database of handwritten digits*. 1998. URL: http://yann.lecun.com/exdb/mnist/ (siehe S. 34).

[33] Yann LeCun u. a. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (1998), S. 2278–2324 (siehe S. 8).

[34] Martín Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/ (siehe S. 27).

[35] Warren S. McCulloch und Walter Pitts. „A logical calculus of the ideas immanent in nervous activity“. In: *The Bulletin of Mathematical Biophysics* 5.4 (Dez. 1943), S. 115–133. DOI: 10.1007/bf02478259 (siehe S. 6).

[36] Marvin Minsky und Seymour Papert. *Perceptrons*. MIT press, 1969 (siehe S. 7).

[37] Andrew Ng und John Duchi. *Lecture 2 | Machine Learning (Stanford)*. 22. Juli 2008. URL: https://www.youtube.com/watch?v=5u4G23_OohI?t=27m20s (besucht am 23. 03. 2017) (siehe S. 17).

[38] NVIDIA Corporation. *HISTORY OF GPU COMPUTING*. URL: https://www.nvidia.com/object/cuda_home_new.html (besucht am 18. 03. 2017) (siehe S. 8).

[39] Frank Rosenblatt. „The perceptron: A probabilistic model for information storage and organization in the brain.“ In: *Psychological review* 65.6 (1958), S. 386 (siehe S. 6).

[40] Sebastian Ruder. „An overview of gradient descent optimization algorithms“. In: (15. Sep. 2016). arXiv: 1609.04747v1 [cs.LG] (siehe S. 19).

[41] David E. Rumelhart, Geoffrey Hinton und Ronald J. Williams. „Learning representations by back-propagating errors“. In: *Nature* 323.6088 (Okt. 1986), S. 533–536. DOI: 10.1038/323533a0 (siehe S. 7).

[42] Tara N Sainath u. a. „Convolutional, long short-term memory, fully connected deep neural networks“. In: *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE. 2015, S. 4580–4584 (siehe S. 49).

[43] Ravi Kiran Sarvadevabhatla und R. Venkatesh Babu. „Freehand Sketch Recognition Using Deep Features“. In: (1. Feb. 2015). arXiv: 1502.00254v2 [cs.CV] (siehe S. 4).

[44] Ryan Schmidt u. a. „Shapeshop: Sketch-based solid modeling with blobtrees“. In: *ACM SIGGRAPH 2007 courses*. ACM. 2007, S. 43 (siehe S. 3).

[45] Rosália G. Schneider und Tinne Tuytelaars. „Sketch classification and classification-driven analysis using Fisher vectors“. In: *ACM Transactions on Graphics* 33.6 (Nov. 2014), S. 1–9. DOI: 10.1145/2661229.2661231 (siehe S. 3, 47).

[46] O. Seddati, S. Dupont und S. Mahmoudi. „DeepSketch 2: Deep convolutional neural networks for partial sketch recognition“. In: *2016 14th International Workshop on Content-Based Multimedia Indexing (CBMI)*. Juni 2016, S. 1–6. DOI: 10.1109/CBMI.2016.7500261 (siehe S. 4).

[47] O. Seddati, S. Dupont und S. Mahmoudi. „DeepSketch: Deep convolutional neural networks for sketch recognition and similarity search“. In: *2015 13th International Workshop on Content-Based Multimedia Indexing (CBMI)*. Juni 2015, S. 1–6. DOI: 10.1109/CBMI.2015.7153606 (siehe S. 4, 22, 33, 36–38, 40, 43, 58).

[48] Yelong Shen u. a. „Learning semantic representations using convolutional neural networks for web search“. In: *Proceedings of the 23rd International Conference on World Wide Web*. ACM. 2014, S. 373–374 (siehe S. 20).

[49] K. Simonyan und A. Zisserman. „Very Deep Convolutional Networks for Large-Scale Image Recognition“. In: *CoRR* abs/1409.1556 (2014) (siehe S. 33).

[50] D. Steinkraus, I. Buck und PY. Simard. „Using GPUs for machine learning algorithms“. In: *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*. IEEE. 2005, S. 1115–1120 (siehe S. 8).

[51] Ilya Sutskever, Oriol Vinyals und Quoc V Le. „Sequence to sequence learning with neural networks“. In: *Advances in neural information processing systems*. 2014, S. 3104–3112 (siehe S. 25).

[52] Christian Szegedy u. a. „Going Deeper with Convolutions“. In: (17. Sep. 2014). arXiv: 1409.4842v1 [cs.CV] (siehe S. 33).

[53] Tijmen Tieleman und Geoffrey Hinton. „Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude“. In: *COURSERA: Neural networks for machine learning* 4.2 (2012) (siehe S. 37).

[54] Paul Werbos. „Beyond regression: New tools for prediction and analysis in the behavioral sciences“. In: (1974) (siehe S. 7).

[55] Janet Wiener und Nathan Bronson. *Facebook's Top Open Data Problems*. 22. Okt. 2014. URL: https://research.fb.com/facebook-s-top-open-data-problems/ (besucht am 24. 03. 2017) (siehe S. 5).

[56] Yongxin Yang und Timothy M. Hospedales. „Deep Neural Networks for Sketch Recognition". In: *CoRR* abs/1501.07873 (2015). URL: http://arxiv.org/abs/1501.07873 (siehe S. 4).

[57] Wen-tau Yih, Xiaodong He und Christopher Meek. „Semantic Parsing for Single-Relation Question Answering." In: *ACL (2)*. Citeseer. 2014, S. 643–648 (siehe S. 20).

[58] Qian Yu u. a. „Sketch-a-Net that Beats Humans". In: (30. Jan. 2015). arXiv: 1501.07873v3 [cs.CV] (siehe S. 33, 58).

[59] Matthew D. Zeiler. „ADADELTA: An Adaptive Learning Rate Method". In: (22. Dez. 2012). arXiv: 1212.5701v1 [cs.LG] (siehe S. 37, 42).