ShroomNET

Neuronale Netze spezialisiert auf die Bestimmung von Pilzarten

Maturarbeit von Jan Obermeier
G15E | NKSA | 2018

Betreuende Lehrperson: Dr. Nicolas Ruh

Abstrakt

Diese Maturarbeit beschäftigt sich mit der Entwicklung und Optimierung eines Neuronalen Netzes für die Bestimmung von Pilzarten. Ziel der Arbeit ist es, die Umsetzbarkeit eines solchen Neuronalen Netzes zu prüfen. Zur Vereinfachung beschränkt sich die Erkennung vorerst auf die 20 häufigsten Pilzarten der Nordwestschweiz.

Im Theorieteil der Dokumentation soll ein grundlegendes Verständnis für *Neuronale Netze* und *Deep Learning* vermittelt werden, um die darauf folgende Dokumentation der Umsetzung nachvollziehbar gestalten zu können.

Bei der Umsetzung des Algorithmus werden verschiedene Techniken und Vorgehensweisen für den Aufbau und das Training des Neuronalen Netzes in Betrachtung gezogen, um eine möglichst akkurate Bestimmung der Pilzart ermöglichen zu können. Dabei werden verschiedene Netzwerkarchitekturen wie auch Datenaufbereitungsmethoden gegeneinander abgewägt und auf deren spezifische Vor- und Nachteile untersucht.

In einer Zusammenfassung soll schliesslich über die Anwendbarkeit, Zuverlässigkeit und Erweiterbarkeit des Algorithmus diskutiert werden.

Vorwort

Für *künstliche neuronale Netze* hatte ich schon immer eine gewisse Faszination. Ein im Grunde genommen statischer Code modelliert organisches Verhalten: Er kann lernen und sich dadurch verbessern. Und wie wir auch nicht genau wissen, wie unser Gehirn funktioniert, so wissen wir es auch nicht von den *neuronalen Netzen* — sie *sind* gewissermassen ein "elektronisches Hirn". Weiss man aber dieses "elektronische Hirn" zu belehren und einzusetzen, so eröffnen sich grenzenlose Möglichkeiten.

Codezeilen haben mich schon lange begleitet; die ersten Schritte machte ich mit meinem Vater: Variablen definieren, Werte daraus berechnen, erste Schleifen. Mit der Mittelschule und dem *infcom*-Kurs kamen Objekte, die Steuerung von grafischen Elementen und Game-Loops hinzu. Nebenbei hielt ich mich mit der aktuellen Technik auf dem Laufenden, welche aber zu viel mehr imstande war: Bilder und Sprache erkennen, später lernten sie auch, den Menschen in den komplexesten Brettspielen[1] und Computerspielen[2] zu schlagen. Weiter erschienen Videoaufnahmen von absurden Reden bekannter Staatsoberhäuptern, bei denen das Auge nicht mehr von echt oder gefälscht unterscheiden konnte [3]. Diese und weitere Möglichkeiten, aber auch die Genialität dieser Algorithmen haben mein Interesse an den *künstlichen neuronalen Netzen* geweckt, jedoch fehlten mir damals die Programmiergrundlagen, um ein solches Projekt umzusetzen. Mit der ersten grösseren Arbeit "EvoSim"[4] habe ich meine bisherigen Programmierkenntnisse festigen und erweitern können. Zudem habe ich mich in das Themengebiet der sogenannten *genetischen Algorithmen* begeben; eine rein durch Zufall und Selektion vorangetriebene Methode des *maschinellen Lernens*. Mit der Maturarbeit will ich somit einen Schritt weiter gehen und den Einstieg in die komplexere Welt der *künstlichen neuronalen Netze*, den "elektronischen Hirnen", machen.

Die Anwendung von *künstlichen neuronalen Netzen* kann sehr vielfältig sein; enorm stark vertreten sind sie im Bereich der Bilderkennung. Da viele bekannte Problemstellungen wie Gesichts- und Handschrifterkennung schon zu genüge behandelt worden sind, habe ich mich auf den Impuls von Dr. Nicolas Ruh auf ein für mich eher exotischeres Anwendungsgebiet eingelassen: die Erkennung von Pilzarten. Es ist bekannt, dass die zuverlässige Bestimmung von Pilzen langjährige Expertise voraussetzt, da sich Arten zum Teil nur anhand von wenigen Details auseinanderhalten lassen. Liesse sich dieser Prozess zu einem gewissen Teil von *neuronalen Netzen* übernehmen, so könnte man diese "Expertise" für jedermann mit einem Mobiltelefon in der Tasche zugänglich und nutzbar machen.

Daraus entstand der konkrete Plan, einen auf *künstlichen neuronalen Netzen* ba-

sierenden Algorithmus zu entwickeln, welcher für die Pilzartenbestimmung ausgelegt ist — daher auch der Name *ShroomNET*. Der Erkennungsalgorithmus soll der zentrale Punkt dieser Arbeit sein, wobei der Fokus auf die Entwicklung und Optimierung gelegt wird.

Während den Recherchen für diese Arbeit fanden wir ein Team von Studenten der Universität Helsinki, welches für ihr Projekt namens "Deep Shrooms"[5] ein ähnliches Konzept hatten. Jedoch war das Projekt leider erfolglos, weswegen ich in dieser Arbeit die selbe Hürde umso mehr zu nehmen versuchen will.

Inhaltsverzeichnis

1 Einführung

Wer behaupten will, er habe noch nie von *Deep Learning* gehört, muss gewissermassen unter einem Stein gelebt haben. Durch *Deep Learning* haben die *künstlichen Intelligenzen*[1]in den letzten Jahren ein Comeback erlebt und haben es auch schon einige Male in die Schlagzeilen geschafft:

> "Googles AlphaGo KI [1] schlägt
> Go-Weltmeister Lee Sedol 4-1"[7]

The Guardian, 2016

Das chinesische Brettspiel *Go* basiert auf ganz simplen Regeln: Schwarze bzw. weisse Spielsteine werden abwechslungsweise auf ein 19x19 Felder grosses Spielfeld gesetzt. Das Ziel des Spiels ist es, den Gegner einzukreisen und zu erobern [8]. Trotz den simplen Regeln gibt es fast unzählige Möglichkeiten: Insgesamt lassen sich die Steine in rund 10^{170} verschiedenen Arten anordnen. Ein kurzer Magnitudenvergleich: Schach hat etwa 10^{43} verschiedene Anordnungsmöglichkeiten [9], das beobachtbare Universum enthält schätzungsweise 10^{80} Atome [10].
Diese schiere Anzahl an Möglichkeiten machten *Go* bei KI-Programmierern sehr hoch angesehen, denn im Gegensatz zu Schach können nicht alle relevanten Züge in absehbarer Zeit vorausberechnet werden[2]. Trainierte Go-Spieler können das auch nicht, dafür verlassen sie sich auf ihre Erfahrung und auf ihre Intuition, etwas, was Computern nur schwierig beizubringen ist. Und doch passierte es im März 2016: Der Go-Weltmeister Lee Sedol wurde geschlagen — von der *KI AlphaGo* [1].

AlphaGo unterscheidet sich grundlegend von konventionellen Spiele-KIs, weswegen es auch in *Go* gegen Lee Sedol antreten konnte. *AlphaGo* basiert auf dem Algorithmus namens *Deep Learning* (kurz *DL*). Diese Algorithmen sind in der Lage, aus Beispielen Muster zu erkennen, zu lernen und sich selber dadurch zu verbessern. Durch die Analyse von zahlreichen von Menschen gespielten Partien lernte *AlphaGo* die Grundlagen und Grundstrategien des Spiels. Daraufhin liess man den Algorithmus einige Tausend Male gegen sich selber spielen, wodurch sich *AlphaGo* sukzessive verbesserte und fortgeschrittenere Strategien entwickelte, sodass der Algorithmus schliesslich in der Lage war, gegen die besten Go-Spieler der Welt anzutreten.

[1]Künstliche Intelligenz: Modellierung/Nachahmung von "intelligentem" Verhalten durch Rechner, z.B. Schachcomputer, Übersetzungs-Tools oder Text-/Spracherkennung

[2]vgl. *Deep Blue*: Erster Schachcomputer (entwickelt von IBM), welcher 1997 den damaligen Weltmeister G. Kasparov geschlagen hat. *Deep Blue* verwendete eine *Brute-Force-Methode* (Ausprobieren aller Möglichkeiten), weswegen ein Entwickler sogar die "Intelligenz" hinter der KI *Deep Blue* bestritt[11].

Deep Learning (*DL*) gehört dem Überbegriff "*künstliche neuronale Netze*" (kurz *KNN*s) an. *KNN*s sind im Grunde genommen der Versuch der Informatik, den Aufbau und die Funktionsweise eines biologischen Gehirns zu modellieren und nachzuahmen, wobei *DL* eine Weiterentwicklung dieses Grundprinzipes ist. Wie ein biologisches Gehirn muss auch ein *KNN* lernen, man muss es *trainieren*. Mit dieser Lernfähigkeit werden die Anwendungsmöglichkeiten enorm vielseitig und komplex: In Spiele-KIs, Spam-Filtern und der Bildbearbeitung, aber auch in der Gesichtserkennung und der Krebsdiagnose finden sie heutzutage Verwendung.

1.1 Zielsetzung

Diese Arbeit behandelt die Entwicklung eines auf *KNN*s basierenden Algorithmus, welcher auf die Bestimmung von Pilzarten optimiert ist. Das Ziel der Arbeit ist es, durch Abwägung verschiedener Techniken und Vorgehensweisen einen möglichst zuverlässigen Bestimmungsalgorithmus für Pilzarten zu entwickeln. Als Grundlage für die Bestimmung dienen Fotografien von den zu bestimmenden Pilzen, sekundär können ggf. weitere Eingenschaften des Pilzes angegeben werden, welche in die Bestimmung einfliessen.

Da es sich im Rahmen dieser Arbeit hauptsächlich um die Evaluierung der Umsetzbarkeit geht, wird die Anzahl der bestimmbaren Pilzarten auf die 20 häufigsten Sorten der Nordwestschweiz beschränkt. Des Weiteren werden aus den selben Gründen auf Anwendbarkeit und Benutzerfreundlichkeit des Algorithmus verzichtet.

1.1.1 Datenbeschaffung & Datenaufbereitung

Ein wichtiger Teil für *KNN*s ist die Beschaffung von vielen Trainingsdaten. Aus dem Kapitel *Datenbeschaffung & Datenaufbereitung* geht hervor, wie die Trainingsdaten beschaffen worden sind und bearbeitet worden sind, um sich als Eingangsdaten für ein *KNN* zu eignen. Die ergriffenen Massnahmen werden auf Auswirkungen auf den Trainingsprozess sowie der Genauigkeit der Pilzbestimmung untersucht.

1.1.2 Entwicklung & Optimierung

Bei der Umsetzung sollen verschiedene Methoden sowie Vorgehensweisen für den Aufbau eines *KNN*s genauer untersucht werden, um eine möglichst akkurate Bestimmung der Pilzart zu ermöglichen. Es werden verschiedene Netzwerkarchitekturen, aber auch andere Parameter wie Netzwerkgrösse und verschiedene Trainingsdaten auf Stärken und Schwächen untersucht, um eine möglichst optimale Kombination für die Erkennung von Pilzarten evaluieren zu können. Im Kapitel *Vorgehen* wird nachvollziehbar auf die Verfahren und Massnahmen eingegangen, welche zu einer wesentlichen Verbesserung des Ergebnisses beigetragen haben.

1.1.3 Theorie

Für das erleichterte Verständnis der Dokumentation soll das Kapitel *Grundlagen KNNs & DNNs* einen Einblick in die Entwicklung der verwendeten Algorithmen geben sowie die grundlegende Funktionsweise derer beschreiben. Für das Verständnis des Theorieteils wird ein solides technisches und mathematisches Grundwissen vorausgesetzt.

2 Grundlagen *KNNs* & *DNN*s

In diesem Kapitel werden die theoretischen Grundlagen für die in dieser Arbeit verwendeten Algorithmen behandelt. In einer Einleitung werden die Begrifflichkeiten erläutert sowie die Motivation wie auch Schwierigkeiten für die Entwicklung von *maschinellen Lernalgorithmen* (*ML*) geschildert. Der Hauptteil behandelt den technischen Aspekt von *KNN*s und *DNN*s, worin das Funktionsprinzip, verschiedene Architekturen als auch Umsetzungstechniken vorgestellt werden.

Es ist anzumerken, dass es sich bei *maschinellem Lernen* und *künstlichen neuronalen Netzen* um ein sehr komplexes Themengebiet handelt. Für das Verständnis des Theorieteils wird somit ein solides Grundwissen für technische als auch mathematische Konzepte vorausgesetzt, um die relevanten Informationen in einem angemessenen Umfang behandeln zu können.

2.1 *KI*, *KNN*, *DL*, usw.

In den Medien wie auch im Internet hört und liest man die Ausdrücke und Abkürzungen immer wieder: *KI*s, *Deep Learning* und *neuronale Netze*, *AI*s und *DNN*s. Man könnte meinen, sie seien ähnlich, jedoch bedeuten sie nicht alle dasselbe. Folgender Überblick soll helfen, die verwirrenden Bezeichnungen und Fachausdrücke korrekt zu interpretieren. Der Inhalt greift dabei etwas vor, um späteres Nachschlagen zu ermöglichen.

KI: Künstliche Intelligenz *(engl. artificial intelligince, AI)*:
Überbegriff in der Informatik für automatisierte Prozesse, welche "intelligentes" Verhalten modellieren/nachahmen. Mangels einer klaren Definition von "Intelligenz" ist der Begriff sehr weit anwendbar[12], d.h. einen Roboter auf zwei Beinen gehen zu lassen gehört genau so zur *KI* wie die Erkennung von Gesichtern.

ML: Maschinelles Lernen *(engl. machine learning)*:
Teilbereich der *künstlichen Intelligenzen*, welcher sich mit künstlichen Lernprozessen befasst. Konventionellen Programmen gibt man einen Satz an Regeln mit, um z.B. Muster erkennen zu können. Beim *maschinellen Lernen* ist es hingegen das Ziel, den Algorithmus durch Training die Regeln und Gesetzmässigkeiten selber erkennen und bestimmen zu lassen. Für den Trainingsprozess werden meistens bezeichnete Trainingsdaten verwendet[13].

KNN/NN: Künstliches neuronales Netz *(engl. artificial neural network, ANN)*:
Programmtechnik aus dem Gebiet des *maschinellen Lernens*, welche inspiriert biologischen neuronalen Netzen (Gehirnen) Neuronen und deren

Interaktionen zu modellieren versucht. Die *Neuronen* sind in Schichten angeordnet (sog. *Layers*) und untereinander verbunden. *KNN*s sind in der Lage aus Trainingsdaten zu lernen[14]. Im Kontext der Informatik wird zwischen *künstlichen neuronalen Netzen* und *neuronalen Netzen* (*NN*s) nicht differenziert.

SNN: Shallow Neural Network:
Eher selten gebrauchter Begriff, *KNN* impliziert z.T. ein *SNN*[3]. "Shallow" weist dabei auf die geringe Anzahl der *Layers* hin (i.d.R. 1, höchstens 2 *Hidden Layers*). Wegen den wenigen *Layers* können *SNN*s nur einfachere Probleme lösen[14].

DL/DNN: Deep Learning/Deep Neural Network: Im Gegensatz zu *SNN*s bezeichnet man mit *Deep Learning* Netzarchitekturen, welche 2 oder mehr *Hidden Layers* besitzen. Mit speziellen *Layers* können dem *KNN* andere Eigenschaften oder Charakteristiken gegeben werden, so z.B. verbesserte Verallgemeinerung oder ein "Kurzzeitgedächtnis". *Deep Neural Network* weist spezifisch auf die Umsetzung eines *DL*-Algorithmus mit *NN*s hin, jedoch werden die beiden Begriffe meist synonym verwendet[15].

CNN: Convolutional Neural Network:
Spezielle Art eines *Deep Neural Networks*, welche sich v.a. in der Bilderkennung bewährt. Mit verschiedenen *Layers* werden die relevanten Informationen gefiltert und auf immer kleinere *Layers* konzentriert. Diese Massnahme ermöglicht sehr komplexe und leistungsfähige Netze in absehbarer Zeit zu trainieren.

[3]Um Missverständnisse zu meiden, werden in dieser Arbeit die Begriffe *KNN* und *NN* als **Überbegriff** von *SNN*s und *DNN*s verwendet.

2.2 Einleitung

Abbildung 1: Finger

Sie haben wahrscheinlich ohne zu zögern das Bild gesehen und sofort erkannt, dass darauf eine Hand mit 3 ausgestreckten Fingen gezeigt wird. Es scheint auf den ersten Blick ein simpler Prozess zu sein. So simpel, dass sogar Kleinkinder dazu fähig sind. Mit dieser Vorstellung waren ersten *KI*-Forscher der 1960er der Meinung, sie könnten Rechnern die visuellen Interpretation von Bildern wie auch andere menschliche Fähigkeiten beibringen — in der Zeitspanne von nur zwanzig Jahren. Wie es sich herausstellte, haben sie die Hürde massiv unterschätzt. Erst heute mit der steigenden Rechenleistung und neuen Entwicklungen beginnen die Computer langsam die Versprechen der 60er zu erfüllen. In anderen Worten: Ihr visueller Kortex hat für die Erkennung dieser Finger soeben das geleistet, wofür Rechner Jahrzehnte an Forschung und Entwicklung benötigt haben.

> "Maschinen werden in 20 Jahren zu jeder Arbeit fähig sein, die ein Mensch verrichten kann." [17]
>
> *Herbert A. Simon*
> *KI-Pionier, 1965*

Um die Schwierigkeiten der Bilderkennung zu verstehen, muss das Problem aus der sich von Rechnern betrachtet werden. Was wir als ein Bild von einer Hand erkennen sind für einen Computer Zahlen. Jeder Bildpunkt (*Pixel*) besteht aus drei Intensitätswerten zwischen 0 und 255 für die Grundfarben rot, grün und blau; im Raster angeordnet ergeben sie das Bild. Die Frage besteht in diesem Beispiel darin, ob eine mathematische Funktion[4] für diese Pixelwerte gefunden werden kann, welche etwas über die Anzahl der gezeigten Finger aussagen kann. In Worten formuliert sähe die Funktion wie folgt aus: "Ein ausgestreckter Finger ist leicht rötlich, länglich

[4]Laut Alan Turing kann jede mathematische Funktion als Computerprogramm umgesetzt werden[18], weswegen das Finden dieser Funktion das eigentliche Ziel ist.

und von der Hand gespreizt. Diese gilt es zu zählen". Will man nun aber einen Algorithmus designen, welche diese Eigenschaften erkennen kann, so verliert man sich schnell in unzähligen Regeln und komplexen Ausnahmen. Es sei denn, man lässt den Algorithmus anhand von Beispieldaten die Muster und Regeln selber finden:

2.3 Maschinelles Lernen

Maschinelle Lernalgorithmen sind auf den künstlichen Wissenserwerb spezialisiert. Dabei ist es wichtig, nicht durch "Auswendiglernen", sondern durch Erkennen und Verallgemeinern von Gesetzmässigkeiten Daten zu verarbeiten. Es sind viele verschiedene Umsetzungsansätze für lernfähige Algorithmen bekannt; so z.B. *genetische Algorithmen*, *Entscheidungsbäume* und *künstliche neuronale Netze*.

2.3.1 Training

Um "Wissen zu erwerben" werden maschinelle Lernalgorithmen *trainiert*. Abstrakt kann man das Ziel vom Training folgendermassen formulieren: Eine unbekannte Funktion $f(x)$ kann fehlerlos jedem x das korrekte y zuordnen, seinen x und y Zahlenwerte oder Bilder (x) und die dazugehörigen Namen (y). Für diese unbekannte Funktion $f(x)$ gilt es nun im Training, mithilfe von wenigen bekannten Datenpunkten (x, y) (den *Trainingsdaten*) eine möglichst genaue Modellierung zu finden.

In der Praxis sieht der *Trainingsprozess* generell folgendermassen aus: Ein Datenpunkt x aus den *Trainingsdaten* wird in den zu trainierenden Algorithmus $F(x)$ eingespeist, worauf das erhaltene Ergebnis mit dem erwünschten Ergebnis y verglichen wird. Durch Anpassung der *Parameter* des Algorithmus wird versucht, das Resultat zu verbessern. Dieser Prozess wird mit allen Datenpunkten der Trainingsdaten wiederholt, wodurch der Algorithmus eine möglichst genaue Repräsentation für $f(x)$ finden soll: $F(x) \approx f(x)$.

2.3.2 Verzerrung-Varianz-Dilemma

Das Verzerrung-Varianz-Dilemma[6] ist eine generelle Schwierigkeit von Lernalgorithmen. Im Grunde genommen will man, dass der Algorithmus mit den *Trainingsdaten* als auch mit komplett neuen Daten akkurate Ergebnisse liefern kann. Dazu muss man beim Designen eines maschinellen Lernalgorithmus die richtige Balance zwischen Genauigkeit (*Varianz*) und Toleranz (*Verzerrung*) finden. Ist das Gleichgewicht jedoch nicht bewerkstelligt, so äussert sich das mit den Symptomen der *Überanpassung* oder *Unteranpassung*.

Überanpassung Erhält man ein verdächtig akkurates Ergebnis mit den Trainingsdaten, läuft man Gefahr der *Überanpassung*: Der Algorithmus erstellt zu detaillierte

Regeln für die Trainingsdaten spezifisch und verallgemeinert schlecht (er "lernt auswendig"). *Überanpassung* führt dazu, dass der Algorithmus zwar ausserordentliche Leistungen innerhalb der Trainingsdaten erzielt, jedoch ungelernte Daten nur mit einer sehr grossen Fehlerquote verarbeiten kann.

Um eine Überanpassung des Algorithmus zu vermeiden, verwendet man bei maschinellen Lernalgorithmen zwei sehr menschliche Eigenschaften:

- Nicht perfekt sein
- Annahme, dass manche Details ignoriert werden können

Diese Eigenschaften erlauben es dem Algorithmus durch Weglassen von Details das Rauschen in den Trainingsdaten zu glätten. Es werden weniger Regeln aufgestellt, wodurch die Funktion weniger komplex wird. Die Reduktion der Komplexität führt aus rein probabilistischen Gründen dazu, dass der Algorithmus toleranter wird und somit verallgemeinert wird[19].

Unteranpassung Wenn man nun aber zu viele Annahmen und Fehler zulässt, unterliegt man der Gefahr der *Unteranpassung*, bei der die Leistung des Algorithmus über alle gelernten sowie ungelernten Daten konsistent schlecht ist: Der Algorithmus verallgemeinert zwar sehr gut, aber macht zu viele/falsche Annahmen und ignoriert für die Bestimmung relevante Details. Unteranpassung tritt dann auf, wenn der Algorithmus zu simpel ist oder unpassend für die Problemstellung designet worden ist.

Validierung Um die Balance zwischen Genauigkeit und Toleranz zu gewährleisten, werden während dem Training regelmässig Validierungsdaten eingespeist, auf welche der Algorithmus **nicht** trainiert ist. Mit den durch die Validierungsdaten gelieferten Ergebnissen kann auf die Leistung des Algorithmus ausserhalb der Trainingsumgebung geschlossen werden. Sobald die Leistung mit den Validierungsdaten sinkt während die der Trainingsdaten weiterhin steigt, kann man davon ausgehen, dass der Algorithmus ein Optimum erreicht hat und dass weiteres Trainieren zu einer Überanpassung führen wird. Wenn der trainierte Algorithmus in seinem Optimum trotzdem keine zufriedenstellenden Ergebnisse liefert, ist von einer *Unteranpassung* auszugehen. Durch Ändern der Trainingsparameter[5]oder durch Umgestaltung des Algorithmus kann der *Unteranpassung* wie auch der *Überanpassung* entgegengewirkt werden.

[5]Nicht zu verwechseln mit den *Parametern* des Algorithmus! Trainingsparameter beeinflussen den Trainingsprozess, d.h. Dauer des Trainings, Anzahl Trainings-/Validierungsdaten, etc. Die *Parameter* des Algorithmus werden trainiert und berechnen aus den Eingangsdaten Ausgangsdaten. Um die Verwechslung dieser beiden relevanten Begriffe zu vermeiden, wird für die Trainingsparameter wie auch für die Architektur des Algorithmus der Begriff **Meta-Parameter** verwendet.

2.4 Künstliche neuronale Netze

Obwohl die *KNN*s erst in den letzten Jahren ihren grossen Durchbruch erlebt hatten, lässt sich die ursprüngliche Idee von *künstlichen neuronalen Netzen* bis in die 40er Jahre zurückdatieren. In den 50er Jahren wurden die ersten Versuche mit Rechnern der damaligen Zeit durchgeführt, wobei mit dem Vorgänger des *Neurons*, dem *Perceptron*, nur müssiger Fortschritt erzielt worden ist. Erst mit der Entwicklung der sehr effizienten *Gradient Descent-Trainingsmethode* und der Einführung der heute gängigen *Neuronen* kam die Forschung an *KNN*s in den 70er Jahren wieder in Fahrt. Der richtige Durchbruch der *KNN*s kam aber erst in den letzten beiden Jahrzehnten mit der rapiden Steigerung der Prozessorleistung, aber auch dem Aufkommen des Internets und der schieren Menge an gespeicherten und zugänglichen Daten. Das ermöglichte mit neuen Entwicklungen im Bereich des *Deep Learnings* höchst komplexe Systeme. So waren solche *DNN*s in der Lage, Aufgaben zu meistern, welche für Algorithmen lange als sehr schwierig oder unmöglich erachtet worden sind. Mit neusten Erfolgen in der Bilderkennung, aber auch der Entscheidungsfindung (siehe Kapitel 1 *AlphaGo*) und vielen praktischen Anwendungen hat *Deep Learning* öffentliche Aufmerksamkeit erhalten[14].

Die technischen Informationen zu *KNN*s und *DNN*s in den folgenden Abschnitten sowie im Kapitel 2.5 basieren auf dem öffentlich verfügbaren Lernmittel *Neural Networks and Deep Learning* von Michael Nielsen[20].

2.4.1 Die Natur als Vorbild

Um die in Kapitel 2.3 erwähnten Lernverhalten simulieren zu können, bedienen sich die *KNN*s dem Vorbild der Natur: das Gehirn. Der Aufbau des Gehirns lässt sich in wenigen Worten folgendermassen formulieren: Das Gehirn besteht aus einer Vielzahl an Neuronen, welche durch Verbindungen miteinander vernetzt sind. Diese Neuronen sind in der Lage, Signale aus den eingehenden Verbindungen mittels Ionenkanäle zu gewichten und mithilfe der in der Zelle herrschenden Ladung miteinander zu verrechnen. Sobald die Ladung einen gewissen Schwellenwert übersteigt, feuert das Neuron und gibt das Signal an die folgenden Neuronen weiter, welche den selben Prozess wiederholen. Der Lernprozess wird durch Belohnung von nützlichen und häufig verwendeten Verbindungen bewerkstelligt, wobei diese Verbindungen verstärkt und stärker gewichtet werden und ungebrauchte oder ungünstige Verbindungen verkümmern[21].

Auf diesen Grundlagen basiert die Idee der *KNN*s, weswegen sie auch noch eine weitere wichtige Eingenschaft gemeinsam haben: Als Aussenstehender ist einem genaue das genaue Funktionsprinzip unbekannt. So sind wir wie auch *KNN*s dazu fähig, handschriftliche Ziffern voneinander zu unterscheiden, wie es aber im Detail

bewerkstelligt wird, ist und bleibt unklar[6]. Für die algorithmische Umsetzung wurden jedoch einige Anpassungen vorgenommen, daher sind heutige *KNN*s nur noch im entferntesten Sinne mit biologischen neuronalen Netzen verwandt.

2.4.2 Neuronen & Verbindungen

Das *Neuron* ist der Grundbaustein von *KNN*s. Wie das biologische Vorbild enthält es mehrere Signaleingänge sowie einen Signalausgang, alle Eingangssignale werden mit einem Faktor Gewichtet und zudem hat jedes *Neuron* seinen Schwellenwert. Die Gewichtungen wie auch der Schwellenwert sind die zu trainierenden *Parameter* des *KNN*s.

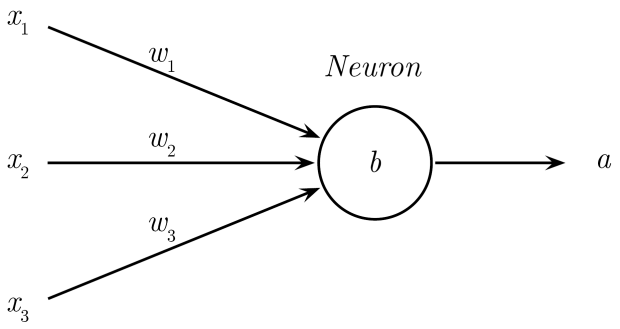

Abbildung 2: Schema von einem künstlichen Neuron

Den Eingangssignalen werden die Variablen $x_1, x_2, ..., x_n$ gegeben, wobei sie jeden Wert zwischen 1 und 0 annehmen können. Zur Verallgemeinerung der Formeln wird n für die Anzahl der Eingangssignale verwendet. Jeder Verbindung wird ein Gewichtungs-*Parameter* $w_1, w_2, ..., w_n$[7]gegeben, dessen Zahlenwert nicht begrenzt ist und daher auch negativ sein kann. Die Eingangssignale werden mit dem jeweiligen Gewichtungs-*Parameter* multipliziert $(x_i \cdot w_i)$, wodurch man das gewichtete Signal vom i-ten Eingang erhält. Im nächsten Schritt gilt es, alle gewichteten Signale zu summieren, welches in folgender Formel bewerkstelligt wird:

$$\sum_{i=1}^{n} x_i \cdot w_i \tag{1}$$

Nun soll der Schwellenwert-*Parameter*[7] als $b \equiv -$Schwellenwert implementiert werden. b kann somit ganz simpel mit dem Ergebnis der Formel 1 addiert werden, wodurch man $\sum_{i=1}^{n} x_i \cdot w_i + b$ erhält. An dieser Stelle soll zur Übersicht dieser gesamte

[6]Ironischerweise war eine Motivation für die Entwicklung von *KNN*s, uns ein besseres Verständnis für die Funktionsweise von intelligentem Verhalten zu geben. Jedoch stellte es sich heraus, dass weder *KNN*s noch das biologische Vorbild im Detail verstanden werden können.

[7]In den folgenden Kapiteln werden die Gewichtungen $w_1, w_2, ..., w_n$ und der Schwellenwert b eines Neurons allgemein als **Parameter** bezeichnet. Nicht zu verwechseln mit den **Meta-Parametern**(siehe Fussnote 5)

Term gleich zur Variable z zusammengefasst werden:

$$\sum_{i=1}^{n} x_i \cdot w_i + b \equiv z \tag{2}$$

Wenn z kleiner als 0 ist, bedeutet dies eine Unterschreitung des Schwellenwertes, ist z grösser als 0, eine Überschreitung. Um für das Ergebnis a vom *Neuron* wieder einen Wert im Bereich $0 \leqslant a \leqslant 1$ zu erhalten, wird die stufenlose *Sigmoid*-Funktion[8] zur Standardisierung von z angewandt. Diese Funktion lässt die Werte für a im Unendlichen gegen 1 resp. 0 gehen, jedoch ermöglicht sie im Bereich um $z = 0$ eine Feinjustierung des Ausgangswertes. Zudem eignet sich die *Sigmoid*-Funktion auch, weil sie sich differenzieren lässt.

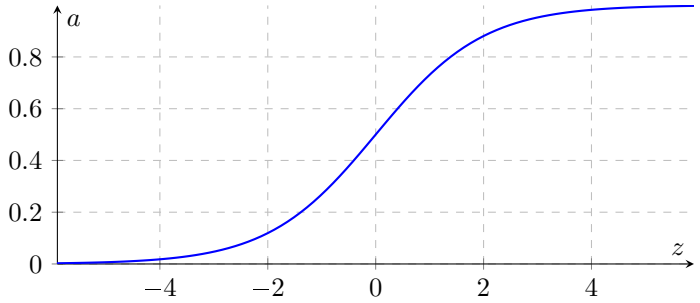

Abbildung 3: Sigmoid-Funktion $\sigma(z) = a$

Mit der *Sigmoid*-Funktion als ideale Kurve zur Standardisierung von z gilt es nur noch, alle Elemente miteinander zu verknüpfen:

$$a = \sigma\left(\sum_{i=1}^{n} x_i \cdot w_i + b\right) \qquad \text{oder} \qquad a = \sigma\left(z\right) \tag{3}$$

wobei $x_1, ..., x_n$ den Eingangswerten und a dem Ausgangswert a entspricht. Die Variablen $w_1, ..., w_n$ und b sind *Parameter* und verändern das Verhalten des Neurons.

Dies ist die Formel eines jeden *Neurons* und bildet (z.T. mit geringfügigen Modifikationen, siehe Kapitel 2.7.1) den Grundbaustein aller *KNN*s.

Anzumerken ist, dass die Funktion 3 aufgrund der Differenzierbarkeit der *Sigmoid*-Funktion und der Linearität der restlichen Operationen ebenfalls differenzierbar ist. Diese Eingenschaft wird später essenziell, um einen effizienten Trainingsalgorithmus für die Justierung der *Parameter* zu entwickeln.

[8] Definition der *Sigmoid*-Funktion: $\sigma(x) = \frac{1}{1+e^{-x}}$

2.4.3 Layers

Da ein einzelnes *Neuron* alleine nicht viel leisten kann, werden sie miteinander vernetzt — daher auch der Name *künstliches neuronales Netz*. Die einfachste Art der Vernetzung schlägt folgende Struktur vor:

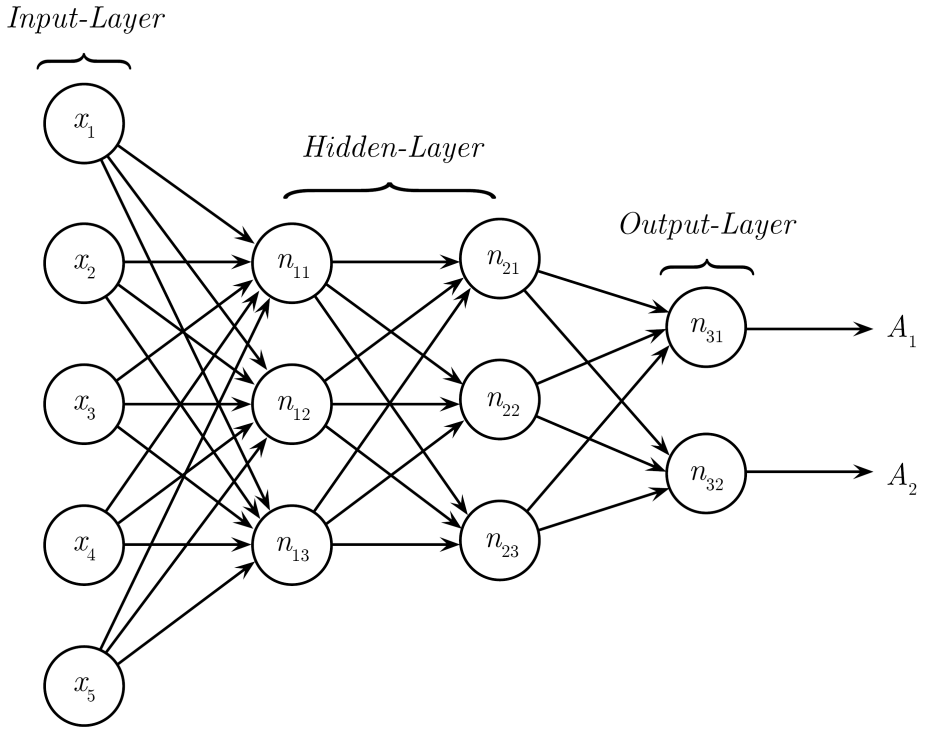

Abbildung 4: Schema von einem kleinen *KNN*

Die *Neuronen* im Netz werden in Schichten strukturiert, in sog. *Layers*[9]. Die Neuronen eines *Layers* werden mit allen Ausgangssignalen der *Neuronen* im vorherigen *Layer* verbunden (ein sog. *Fully Connected Layer*), wobei jede Verbindung separat gewichtet wird. Anzumerken ist, dass die vielen Ausgänge eines *Neurons* alle dem selben Ausgangswert a (siehe Kapitel 2.4.2) entsprechen.

Generell gibt es drei Typen von *Layern*:

- Eingangs-*Layers*, sog. *Input-Layers*
- Zwischen-*Layers*, sog. *Hidden-Layers*
- Ausgangs-*Layers*, sog. *Output-Layers*

Jedes *KNN* hat genau einen *Input-Layer* sowie einen *Output-Layer*. Dazwischen befinden sich ein oder mehrere *Hidden-Layers*.

Die Eingangsdaten $x_1, ..., x_n$ (in Abbildung 4 $n = 5$) werden in die *Neuronen* des

[9]Da die Literatur der *KNN*s praktisch ausschließlich in der englischen Sprache niedergeschrieben wird, wird auf weitere deutsche Übersetzungen der Fachausdrücke verzichtet.

Input-Layers geschrieben[10]. Dies sind z.B. Testwerte von einem Versuch oder Pixel eines Bildes (wobei n der Pixelanzahl entsprechen muss).

Darauf werden die Werte durch die *Neuronen* der *Hidden-Layers* verarbeitet. Zwar bestehen die Rechenoperationen nur aus denen im letzten Kapitel 2.4.2 beschriebenen Funktion 3, jedoch können durch richtige Justierung der *Parameter* w und b im Trainingsprozess (siehe Kapitel 2.4.5) auch sehr komplexe Funktionen modelliert werden.

Die Neuronen im *Output-Layer* geben das berechnete Resultat $A_1, ..., A_m$ (in Abbildung 4 $m = 2$) aus. Wird das *KNN* zur Kategorisierung von Daten, z.B. Bildern, verwendet, wird jedem A_i eine Kategorie zugeordnet, z.B. $A_1 \equiv$ Hund und $A_2 \equiv$ Katze. Die erhaltenen Werte für $A_1, ..., A_m$ entsprechen dabei der Wahrscheinlichkeit, dass die Eingangsdaten zur jeweiligen Kategorie gehören.

Die Grösse der *Input-* und *Output-Layers* müssen den Eingangsdaten respektive der Anzahl Kategorien angepasst werden. Die Grösse und Anzahl der *Hidden-Layers* ist hingegen frei wählbar und gehört zu den *Meta-Parametern*[11] des *KNN*s. Kleinere Probleme lassen sich noch mit einem einzelnen *Hidden-Layer* bewerkstelligen, wobei von einem *Shallow Neural Network* die Rede ist. Bei komplexeren Problemen empfiehlt sich jedoch die Verwendung eines *DNN*s, in welchem mehrere (oft spezialisierte) *Hidden-Layers* in Serie geschaltet werden.

Mathematisch gesehen werden bei der Vernetzung der *Neuronen* die Formeln ineinander verschachtelt, wodurch sich das ganze *KNN* als mathematische Funktion schreiben lässt. Die gesamte Formel ist zwar sehr komplex und unübersichtlich zu notieren, jedoch werden die ursprünglichen Eigenschaften der Funktion 3 beibehalten, d.h. es bleibt eine differenzierbare mathematische Funktion. Das *KNN* soll als Funktion zusammengefasst geschrieben werden als:

$$N(x_1, ..., x_n) = A_1, ..., A_m \tag{4}$$

wobei $N(...)$ das *KNN* als Funktion darstellt, n die Anzahl der Eingangswerte $x_1, ..., x_n$ (sowie der *Input-Neuronen*) ist und m die Anzahl der Ausgangswerte $A_1, ..., A_n$ (sowie der *Output-Neuronen*) ist. Selbstverständlich sind die w- und b-*Parameter* nach wie vor Teil des *KNN*s, einfachheitshalber werden sie in der allgemeinen Darstellung weggelassen, das sie in der Regel konstant sind.

[10] Die *Neuronen* des *Input-Layers* sind daher an sich keine *Neuronen*, da sie nur den Eingangswert annehmen und direkt an die folgenden *Neuronen* weitergeben ($x = a$).

[11] Parameter, die die Architektur des *KNN*s (Grösse, Tiefe, etc.) aber auch den Trainingsprozess (Dauer, Daten, etc.) beeinflussen

2.4.4 Leistung von KNNs messen

Das *künstliche neuronale Netz* ist erstellt, doch bevor es mit trainiert werden kann, wird eine Methode zur Evaluation der Leistung des *KNN*s benötigt. Eine Art dies zu bewerkstelligen, ist die *Quadratic-Cost*-Funktion:

$$C(A_1, ..., A_m, y_1, ..., y_m) = \sum_{i=1}^{m} (A_i - y_i)^2 = Err \tag{5}$$

wobei m die Anzahl der *Output-Neuronen* ist. $A_1, ..., A_m$ sind die durch das *KNN* erhaltene Ergebnisse und $y_1, ..., y_m$ die korrekten Ergebnisse der Trainingsdaten. Die neue Variable Err steht für den erhaltenen Fehler.

Sind die Differenzen zwischen A_i und y_i gross, so erhält man einen grossen Wert für Err und vice versa. Das heisst, wenn $Err \approx 0$ ist, hat das *KNN* das Ergebnis $y_1, ..., y_m$ mithilfe der Eingangsdaten sehr genau vorausbestimmen können. In anderen Worten: Beim Training gilt es diesen Wert Err zu minimieren um die Leistung zu maximieren.

Wie in den vorherigen Kapiteln 2.4.2 und 2.4.3 betont wurde, kann das gesamte *KNN* auch als Funktion dargestellt werden. Dadurch lässt sich der Fehler Err auch als Funktion der Eingangsdaten $x_1, ..., x_n$ (sowie aller Gewichtungen w und Schwellenwerten b) eines *KNNs* $N(...)$ und dem erwarteten Ergebnis $y_1, ..., y_m$ ausdrücken:

$$Err = C(N(x_1, ..., x_n), y_1, ..., y_m) \tag{6}$$

2.4.5 Training von KNNs

Bis jetzt wurden die *Parameter* b und w der *Neuronen* als nicht-veränderliche Werte angenommen. Im Trainingsprozess geht es nun um die Justierung dieser Werte, um den Fehlerwert Err des Netzes zu minimieren. Alle *Parameter* werden am Anfang des Trainingsprozesses zufällig initialisiert, somit wird der Fehler Err entsprechend gross sein.

Eine Möglichkeit, den Fehler zu minimieren besteht darin, einen einzelnen *Parameter* p aus dem Netz ein wenig zu ändern um dann dessen Auswirkung auf Err zu erkennen zu können. Wird Err grösser, versucht man die andere Richtung, wird Err kleiner, ist man dem Ziel einen Schritt näher und wiederholt dasselbe mit allen anderen Parameter. Zwar klingt diese Methode verständlich und machbar, jedoch ist sie sehr ineffizient, da viel Ausprobieren involviert ist. Wie also kann man die Auswirkung einer Parameteränderung auf den Fehler Err am effizientesten berechnen?

Um sich zuerst ein intuitives Bild von dieser Problemstellung machen zu können, muss man es sich als Parameterlandschaft vorstellen: In drei Dimensionen entsprächen die Ortskoordinaten zwei *Parametern*, die Höhe stünde für den zu den

Parametern zugehörige Fehlerwert *Err*, die "Landschaft" mit ihren Bergen und Tälern wird durch die Trainingsdaten bestimmt (entspricht der Funktion $f(x)$ aus Kapitel 2.3.1). Da *KNN*s weithin mehr als zwei *Parameter* besitzen, enthält deren Parameterlandschaft so viele "Ortsdimensionen" wie das *KNN Parameter* hat. Das Ziel ist es in dieser Landschaft, ein tiefes Tal zu finden — mit einer Schwierigkeit: Man ist blind ($f(x)$ ist unbekannt).

In diesem Gedankenmodell sähe die vorher erwähnte Brute-Force-Strategie zum Finden des Minimums folgendermassen aus: Man macht einen Stritt in eine beliebige Richtung. Ist man höher als vorher, geht man zwei Schritte in die entgegengesetzte Richtung und wiederholt den Prozess für die anderen Dimension(en)/*Parameter*, bis man in einem Tal angekommen ist (eine Bewegung in jede Richtung bewirkt eine Verschlechterung des Ergebnisses).

Gradient Descent Eine elegantere Lösung lässt sich mithilfe der Analysis finden, weswegen bei der Herleitung der Formeln stets auf die Differenzierbarkeit geachtet wurde. Durch *partielles Ableiten* der Funktion 6 nach allen *Parametern* lässt sich die Steigung in der Parameterlandschaft im momentanen Punkt approximieren. Der erhaltene Richtungsvektor soll dabei als $\vec{v}$ bezeichnet werden, bei dem jede Dimension einem *Parameter* entspricht. Auf das Gedankenmodell angewendet zeigt dieser Vektor $\vec{v}$ in die Richtung, in der man am meisten Höhe verliert, obwohl die umliegende Landschaft nicht bekannt ist. Da man dem Gradienten entlang absteigt, nennt man dieses Optimierungsverfahren *Gradient Descent* (zu Deutsch *Gradientenverfahren*). Das Ausprobieren fällt weg, wodurch sich das Training massiv beschleunigt.

Bei der *Gradient Descent*-Trainingsmethode lassen sich einige solcher *Meta-Parameter* einstellen. Diese beeinflussen die Lerngeschwindigkeit, Genauigkeit und die Dauer des Trainings.

- **Lernrate** η bestimmt die Grösse der "Schritte" in der Parameterlandschaft ($\vec{v} \cdot \eta$). Bei zu kleinen Werten wird das Training nur langsam vorangehen; bei zu grossen Stritten hingegen riskiert man das Überspringen eines Minimums und z.T. sogar eine Verschlechterung der Leistung.

- **Mini-Batch** Der Richtungsvektor $\vec{v}$ wird in der Regel über mehrere Trainingsexemplare gemittelt, wofür alle Trainingsdaten zufällig in sog. *Mini-Batches* unterteilt werden. Der Trainingsalgorithmus berechnet für alle Exemplare eines *Mini-Batches* die Steigung und mittelt diese, um die Richtung $\vec{v}$ zu erhalten. Erst dann werden die *Parameter* angepasst (ein "Schritt"). Dieser Prozess wird mit dem nächsten *Mini-Batch* für die neue Position in der Parameterlandschaft wiederholt.

Je grösser die Mini-Batches sind, umso genauer repräsentiert $\vec{v}$ die Parameterlandschaft aller Trainingsdaten, jedoch müssen für einen "Schritt" mehr Berechnun-

gen angestellt werden, welche das Training verlangsamen.

• **Lernepochen** Sind alle *Mini-Batches* schon einmal für die Berechnung des Gradienten verwendet worden, ist eine Lernepoche vorüber. Die Mini-Batches werden neu gemischt und können wieder in den Trainingsalgorithmus eingespeist werden.

Je mehr Epochen das Training durcharbeitet, umso genauer können die *Parametern* eingestellt werden, jedoch läuft man bei zu vielen Epochen Gefahr der *Überanpassung*, welches man mithilfe von Validierungsdaten zu verhindern versucht (siehe Kapitel 2.3.2: Validierung).

2.5 Deep Neural Networks

Theoretisch liesse sich jedes Problem mit nur einem *Hidden-Layer* lösen, jedoch ist es bewiesen, dass ein *Shallow Neural Network* schon für die simpelste Logik-Funktionen z.B. um die Gleichheit mehrerer Signale zu prüfen, exponentiell mehr *Hidden-Neuronen* benötigt[23]. Weshalb sollte man also nicht die Anzahl der *Hidden-Layers* erhöhen? Stichwort: *Deep Neural Networks*. Die Idee dieser *KNN*s ist es, durch weitere, spezialisierte *Hidden-Layers* deren Komplexität und Leitung zu steigern.

Mit dem Wissen der vorherigen Kapitel liesse sich ganz einfach ein sehr tiefes *KNN*s für die Bilderkennung bauen. Man braucht nur eine Menge *Hidden-Layers* und schaltet diese in Serie, indem man alle *Neuronen* miteinander verbindet[12]:

Per Definition handelt es sich dabei um ein *DNN*, jedoch wird man feststellen, dass sich die Leistung trotz viel langsameren Training nur marginal verbessert. Der Grund dafür liegt in der Mathematik des *Gradient Descent*-Trainingsverfahrens selber: Während dem Training sind Gradienten für *Parameter* der ersten Layers entweder verschwindend klein oder extrem gross, welches das Training destabilisieren. Man nennt dieses Phänomen *Unstable Gradients*. Um das Problem zu umgehen, werden die *Hidden-Layers* in *DNN*s vereinfacht und spezialisiert, um die Anzahl der Verbindungen und somit die der *Parameter* zu verringern.

In der Praxis gibt es einige verbreitete *DNN*-Typen: *RNN*s (*Recurrent Neural Networks*), *LSTM RNNs* (*Long short-Term Memory RNN*), *Deep Belief Networks*, etc., jedoch ist das *Convolutional Neural Network* (*CNN*) der geeignetste Typ für Bilderkennungsprobleme. Da es in der Arbeit Pilze anhand von Bildern zu kategorisieren gilt, wird im nächsten Kapitel der Fokus nur auf *CNN*s gelegt.

[12] Ab nun empfiehlt es sich, sich die *Layers* als **2D-Ebenen** aus *Neuronen* vorzustellen, da das Kapitel 2.6 auf die Anwendung für die Verarbeitung von 2D-Bildern eingeht. An den Formeln ändert sich jedoch nichts; die *Layers* werden nur anders dargestellt.

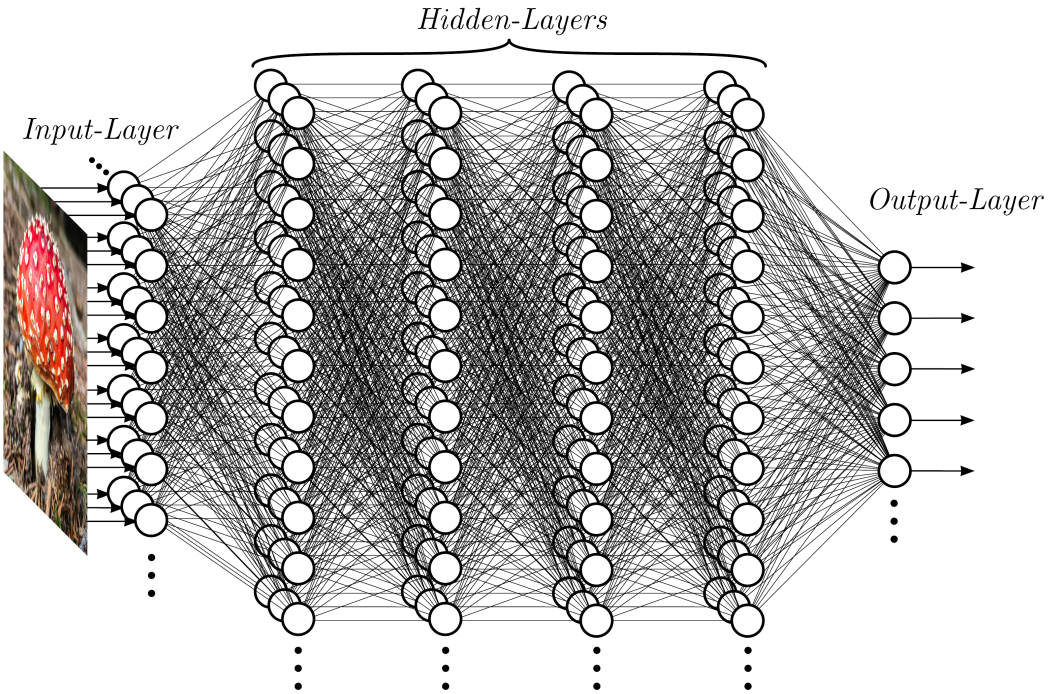

Abbildung 5: Schema von einem *Fully Connected Deep Neural Network*

2.6 Convolutional Neural Networks

Will man ein *DNN* für die Erkennung von Bildern verwenden, so ist das in der Abbildung 5 gezeigte Netz aus mehreren Gründen ungeeignet: Einerseits leidet es u.a. wegen den vielen Verbindungen weiterhin an *Unstable Gradients* und ist daher schwierig zu trainieren, andererseits werden durch *Fully Connected Layers* die räumlichen Zusammenhänge der Eingangs-Pixel komplett ignoriert. *CNN*s verhindern dies mit sog. *Convolution-* und *Pooling-Layers*, wobei mit sog. *Feature-Maps* Muster und Regelmässigkeiten erkannt werden können.

2.6.1 Convolution Layer

Das *Local Receptive Field*, zu Deutsch "lokales rezeptives Feld", engt die Eingangssignale eines *Neurons* auf einen lokalisierten Bereich des vorhergehenden Layers ein. Hat man z.B. den *Input-Layer* und einen darauf folgenden *Convolution-Layer*, auf die eine *Local-Receptive-Field*-Grösse von 5×5 *Neuronen* angewandt wird, so ist ein *Neuron* des *Convolution-Layers* nur mit einem 5×5 *Neuronen* grossen Bereich des *Input-Layers* verbunden. Für das erste *Neuron* des *Convolution-Layers* wird es folgendermassen aussehen:

Für das nächste *Neuron* des *Convolution-Layers* verschiebt sich die gesamte Maske einen Schritt in die jeweilige Richtung. Wie die Grösse des *Local-Receptive-Fields* ist auch die Schrittgrösse ein *Meta-Parameter* und kann angepasst werden, in diesem Beispiel wird eine Schrittgrösse von einem *Neuron*(/Pixel) verwendet:

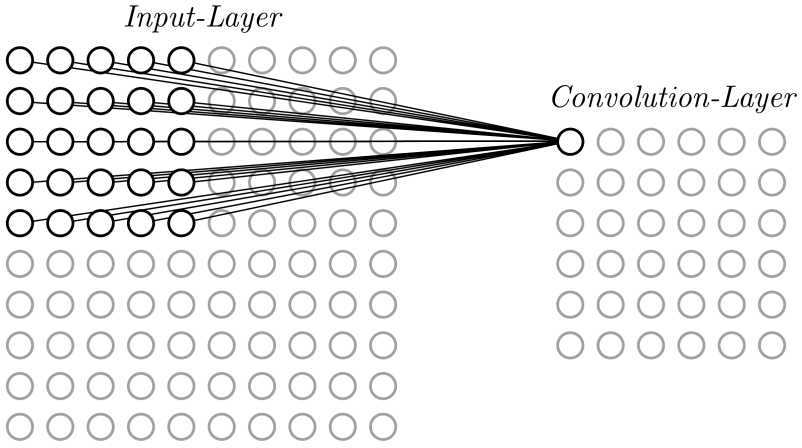

Abbildung 6a: Beispiel für das *Local-Receptive-Field* vom 1. *Neurons* des *Convolution-Layers*

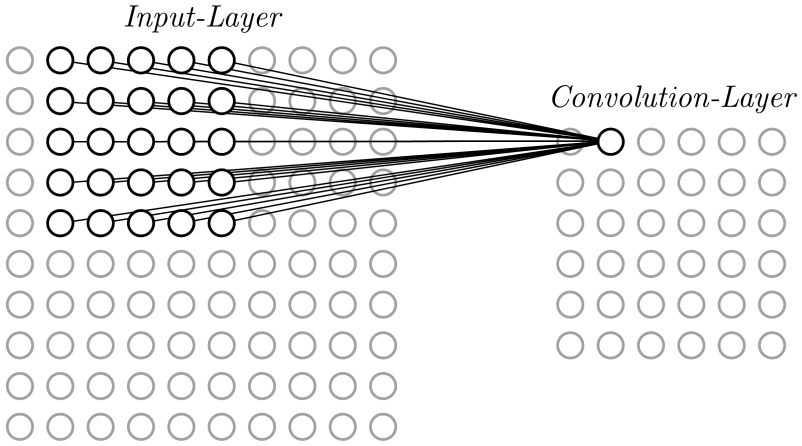

Abbildung 6b: Beispiel für das *Local-Receptive-Field* vom 2. *Neurons* des *Convolution-Layers*

2.6.2 Shared Weigths & Feature Maps

Ein weiterer verwendeter Trick ist, dass alle Neuronen des *Convolution-Layers* dieselben *Parameter* verwenden. Dabei ist die Rede von sog. *Shared weigths*, d.h. das $Neuron_{ij}$ im *Local-Receptive-Field* wird für jedes *Convolution-Neuron* mit w_{ij} gewichtet, wobei i und j die Indexes für die horizontale resp. vertikale Position im *Local-Receptive-Field* bezeichnet. Im 5×5-Beispiel von oben gibt es somit nur 25 verschiedene Gewichtungen für die Verbindungen, der Schwellenwert b ist überall der selbe.

Intuitiv kann man sich den *Convolution Layer* als Schablone für ein durch die *Parameter* bestimmte Muster vorstellen, welche Schrittweise über das Eingangsbild geschoben wird. Entspricht das Bild dem Muster, wird ein hoher Wert zurückgegeben und vice versa. Je nach *Parameter* kann man z.B. Kanten oder Ecken erkennen. Man hat somit eine sog. *Feature Map* erstellt, welches das Vorkommen eines bestimmtes Musters im Bild kartiert. Da ein Bild nicht nur aus einem einzigen Muster besteht, werden in einem *Convolution-Layer* sogleich mehrere *Feature-Maps* erstellt, welche

sich auf verschiedene Muster spezialisieren können.

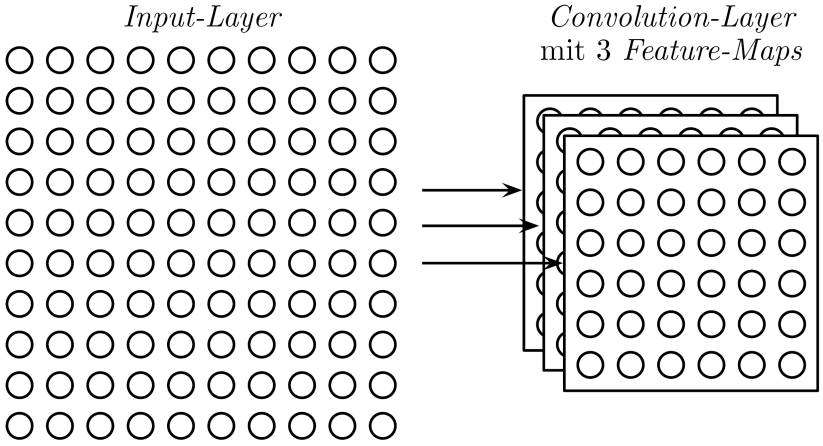

Abbildung 7: Veranschaulichung von *Feature-Maps*

2.6.3 Pooling

Um die vom *Convolution-Layer* erhaltenen *Feature-Maps* weiter zu konzentrieren, folgt in der Regel ein sog. *Pooling*-Layer. Dabei wird wiederum aus einem Bereich des vorherigen *Layers* nur der höchste Wert (*max-Pooling*) oder der durchschnittliche Wert (average-Pooling) an den nächsten *Layer* weitergegeben.

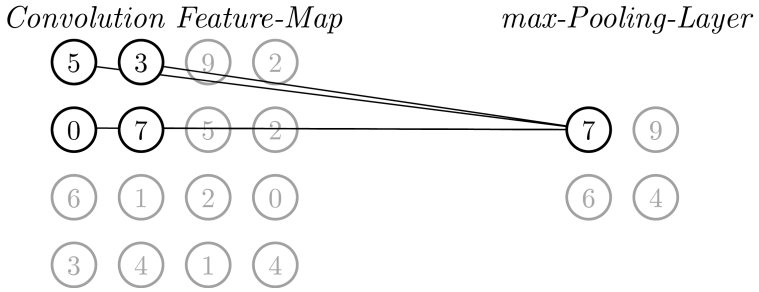

Abbildung 8: Funktionsweise eines *max-Pooling-Layers* mit Filtergrösse 2×2 und Schrittgrösse 2

2.6.4 Aufbau eines CNNs

Beim Aufbau eines *CNN*s werden einem viele Freiheiten gelassen, jedoch hat sich folgende Grundstruktur für viele Bildklassifizierungsprobleme bewährt: Nach dem *Input-Layer* werden ein *Convolution-Layer* und ein *Pooling Layer* hintereinandergeschaltet, z.T. werden noch weitere *Layers* (z.B. *ReLU-Layer*, siehe Kapitel 2.7.1) hinzugefügt. Zusammen bilden diese eine Einheit, welche einige Male wiederholt wird. Die Grösse der Layers nimmt mit der Tiefe ab, wodurch sich der charakteristische trichterförmige Aufbau von *CNN*s bildet. Auf die letzte Einheit folgen wenige *fully connected Layers*, wodurch die erkannten Muster miteinander kombiniert wer-

den. Schliesslich werden mithilfe eines sog. *Softmax-Layers* die Ausgangssignale auf Werte zwischen 0 und 1 normalisiert, um sich für Klassifizierung zu eignen.

2.7 Zusätzliche Meta-Parameter und Modifikation

Die in den vorherigen Kapiteln beschriebenen *KNN*s und *DNN*s als auch deren Training sind keineswegs starr und lassen sich in vielen Aspekten modifizieren und justieren. Auch diese *Meta-Parameter* sind für spätere Kapitel relevant, weswegen sie in diesem Abschnitt kurz beschrieben werden.

2.7.1 ReLU-Layer

Die *Sigmoid*-Kurve aus Kapitel 2.4.2 ist nicht die einzige Funktion, die zur Standardisierung des gewichteten Ausgangssignals z eines Neurons (siehe Formel 2) verwendet wird. Neben der *Sigmoid*-Funktion haben sich auch die sog. *Rectified-Linear*-Funktion bewährt. Ein *Neuron*, welches diese Funktion verwendet, nennt man *Rectified-Linear-Unit* (kurz *ReLU*); ein *ReLU*-Layer ist somit eine Schicht zusammengesetzt aus diesen *ReLU-Neuronen*.

Im Gegensatz zur *Sigmoid*-Kurve liefert diese Funktion einen ganz anderen Ansatz: Bei $z > 0$ verhält sie sich linear und bei $z \leqslant 0$ gibt sie einen konstanten Wert von 0 zurück, d.h. der Ausgangswert ist nach oben nicht begrenzt und immer positiv.

$$\sigma(z) = \begin{cases} z, & \text{wenn } z > 0 \\ 0, & \text{wenn } z \leqslant 0 \end{cases} \tag{7}$$

Als Graph sieht die in der Formel 7 beschriebene *Rectified-Linear*-Funktion verglichen mit der *Sigmoid*-Funktion folgendermassen aus:

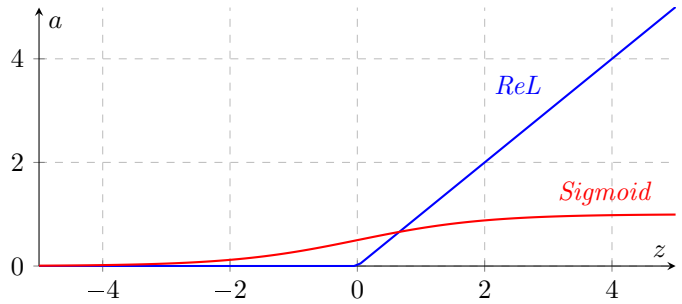

Abbildung 9: *Rectified-Linear*-Funktion $\sigma(z) = max(z, 0)$ verglichen mit der *Sigmoid*-Funktion

Vorteile: Aufgrund der Linearität kann auch für sehr grosse positive z-Werte eine signifikante Steigung für die Justierung der *Parameter* berechnet werden, wo hingegen die *Sigmoid*-Funktion in diesem Bereich nur eine verschwindend kleine Steigung

liefert. Für negative Werte ist die Steigung hingegen konstant 0, wodurch die *Parameter* jenes Neurons nicht mehr durch das Training justiert werden können und gewissermassen "einfrieren". Diese Massnahme kann in machen Anwendungsszenarien die Leistung des *KNN*s verbessern.

2.7.2 Dropout-Layer

Da die vielen Verbindung von *fully connected Layers* anfällig sind für Überanpassung, versucht man dies mit sog. *Dropout-Layers* zu minimieren. Diese *Layers* werden ähnlich wie *fully connected Layers* in einem *KNN* eingesetzt, jedoch unterscheiden sie sich wesentlich im Trainingsprozess:

Ein gewisser Prozentsatz der *Neuronen* eines *Dropout-Layers* wird für das Training temporär deaktiviert, d.h. sie werden bei der Berechnung des Gradienten nicht berücksichtigt noch justiert. Diese Methode macht das *KNN* robuster, reduziert Überanpassung und beschleunigt dazu noch das Training.

2.7.3 Data Augmentation

Ein wichtiger Aspekt für die Entwicklung eines zuverlässigen maschinellen Lernalgorithmus ist der Umfang der Trainingsdaten. Wenn zu wenige vorhanden sind, kann der Algorithmus keine allgemeine Funktion finden und riskiert das Überanpassen auf die wenigen Exemplare.

Mithilfe von *Data augmentation* lässt sich die Anzahl der Trainingsexemplare jedoch künstlich vergrössern. Dazu werden die Trainingsbilder um einige Pixel verschoben, gespiegelt, ein wenig rotiert oder verzerrt. Man generiert dadurch viele weitere Trainingsexemplare, welche wiederum das Risiko einer Überanpassung senken.

2.7.4 Transfer Learning

2.7.5 Ensemble

Literatur

[1] AlphaGo. https://deepmind.com/research/alphago/, 2017. Abgerufen am 30.07.2018.

[2] Openai five. https://blog.openai.com/openai-five/, 2018. Abgerufen am 09.08.2018.

[3] anonym. Deepfake. https://www.deepfakes.club/, 2017. Abgerufen am 09.08.2018.

[4] Jan Obermeier Nils Schlatter. Evosim, 2018.

[5] Jonas Harjunpää Teemu Koivisto, Tuomo Nieminen. Deep shrooms. https://tuomonieminen.github.io/deep-shrooms/, 2017. Abgerufen am 25.07.2018.

[6] Stuart Geman René Doursat, Elie Bienenstock. Neural networks and the dias/variance dilemma, 1992. Massachusetts Institute of Technology.

[7] Steven Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals- 4-1-victory-over-grandmaster-lee-sedol, 2016. Abgerufen am 09.08.2018.

[8] British Go Association. Play go. https://www.britgo.org/files/pubs/playgo.pdf, 2014. Abgerufen am 30.07.2018.

[9] Claude E. Shannon. Programming a computer for playing chess. Kapitel General Considerations, 1949. Bell Telephone Laboratories, Inc., Murray Hill, N.J.

[10] Wikipedia. Oberservable universe. https://en.wikipedia.org/wiki/Observable_universe. Abgerufen am 30.07.2018.

[11] Wikipedia. Deep blue. https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer), 2017. Abgerufen am 09.08.2018.

[12] Wikipedia. Artificial intelligence. https://en.wikipedia.org/wiki/Artificial_intelligence. Abgerufen am 10.08.2018.

[13] Wikipedia. Machine learning. https://en.wikipedia.org/wiki/Machine_learning. Abgerufen am 10.08.2018.

[14] Wikipedia. Artificial neural network. https://en.wikipedia.org/wiki/Artificial_neural_network. Abgerufen am 10.08.2018.

[15] Wikipedia. Deep learning. https://en.wikipedia.org/wiki/Deep_learning. Abgerufen am 10.08.2018.

[16] Wikipedia. Convolutional neural network. https://en.wikipedia.org/wiki/Convolutional_neural_network. Abgerufen am 10.08.2018.

[17] H. A. Simon. *The Shape of Automation for Men and Management*. 1965. New York: Harper & Row, S. 96.

[18] Alan Turing. On computable numbers, with an application to the entscheindungsproblem, 1936. Proceedings of the London Mathematical Society.

[19] Welch Labs. Learning to see : Part 8. https://www.youtube.com/user/Taylorns34/, 2017. Abgerufen am 11.08.2018.

[20] Michael A. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/. Abgerufen am 05.06.2018.

[21] Roman Claus et al. *Natura, Biologie für Gymnasien*. 1991. Klett, ISBN 3-12-042200-2, S. 252 ff.

[22] Christopher J.C. Burges Yann LeCun, Corinna Cortes. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Abgerufen am 12.08.2018.

[23] Johan Hastad. On the correlation of parity and small-depth circuits. https://eccc.weizmann.ac.il//report/2012/137/, 2012.

Abbildungsverzeichnis