Walter Stein

unter Mitwirkung von Lena Maxine Lenkeit

Künstliche Intelligenz

Eine Einführung für den Schulunterricht mit Programmbeispielen

Walter Stein

unter Mitarbeit von Lena Maxine Lenkeit

Künstliche Intelligenz

Eine Einführung für den Schulunterricht mit Programmbeispielen

Vorwort

Künstliche Intelligenz – Ein Hype, der zurzeit alle Medien erfasst. In China schaut man in Sachen künstlicher Intelligenz positiv in die Zukunft. In Deutschland haben viele Angst vor dieser Zukunft. Werden Maschinen mit ihren hochentwickelten Algorithmen die Herrschaft übernehmen?

Die Künstliche Intelligenz (KI) hat zwar ab 2010 mittels Deep Learning gewaltige Fortschritte erzielt, doch manch ein Zeitgenosse würde sie noch als künstliche Dummheit bezeichnen, da sie nur ganz spezielle Probleme lösen kann. Obwohl wir Menschen auch nur beschränkt intelligent sind, denken wir nur an Umweltzerstörung und Kriege, so sind wir heutzutage der künstlichen Intelligenz, was allgemeines Problemlösungsverhalten angeht, noch deutlich überlegen. Wir haben das Steuer noch in der Hand. Von dem Punkt der technologischen Singularität, also dem Zeitpunkt, ab dem sich die Maschinen mittels KI so schnell verbessern, dass wir abgehängt werden, sind wir zurzeit noch weit entfernt.

Trotzdem wird die heute schon zur Verfügung stehende Künstliche Intelligenz unsere zukünftige Welt umwälzend verändern. Sie kann zu großen gesellschaftlichen Problemen führen, wie dem Verlust von Arbeitsplätzen, autonomen Waffensystemen oder der totalen Überwachung. Sie kann aber auch unser Leben bereichern, indem sie uns von stumpfsinniger körperlicher und geistiger Arbeit befreit, ein bedingungsloses Grundeinkommen ermöglicht, Sprachbarrieren abbaut sowie rasante Fortschritte in der Medizin und in anderen Wissenschaftsdisziplinen hervorbringt. Umso wichtiger ist es, dass möglichst viele in der menschlichen Gemeinschaft diese Entwicklung kenntnisreich und positiv mitgestalten. Schon in der Schule müssen Kenntnisse hierzu in unterschiedlichen Fächern vermittelt werden, denn gerade junge Menschen müssen sich dieser Entwicklung stellen, damit sie nicht Berufe ergreifen, für die es keine Zukunft mehr gibt. Schülerinnen und Schüler sollten aber nicht nur im SoWi-Unterricht über künstliche Intelligenz diskutieren, sondern sie sollten auch lernen, wie man einfache KI-Programme erstellt. Nur so bekommen sie ein fundiertes Verständnis, was künstliche Intelligenz zurzeit leisten kann und was nicht. Wichtig ist auch, dass man bei Schülerinnen und Schülern eine Offenheit für Neuerungen sowie eine kritische Bereitschaft zur Mitgestaltung erwirkt. Denn was die digitale Zukunft angeht, da gibt es gerade in Deutschland sehr viele Bedenkenträger. Wir brauchen aber mehr Menschen mit Mut und mit Fähigkeiten, um diese digitale Zukunft positiv zu gestalten. Wenn es in einem Land mehr Bedenkenträger als Gestalter gibt, dann ist dies in der Tat - sehr bedenklich!

Ob ich zu diesem Thema ein kleines Buch für Schülerinnen und Schüler sowie für Lehrerinnen und Lehrer schreiben soll, habe ich mir lange überlegt. Stehen hierzu doch unzählige Beiträge im Internet. Man kann aber schnell den Überblick verlieren und muss bei der Suche nach guten Beiträgen auch viele Kröten schlucken. Dies wollte ich den oben genannten Personen ersparen. Das vorliegende Buch ist aber kein normales, textlastiges Buch. Es führt den Leser anhand von zahlreichen Links, in der Mehrzahl YouTube-Videos, durch das Thema Künstliche Intelligenz. Diese Links und eigene Texte werden junge Leser in die Lage versetzen, nicht nur konkrete Programme zur Künstlichen Intelligenz zu schreiben, sondern auch die Auswirkungen der KI auf ihr zukünftiges Leben abschätzen und hoffentlich auch mitgestalten zu können.

Danken möchte ich an dieser Stelle meiner ehemaligen Schülerin Lena Maxine Lenkeit. Sie hat mir den entscheidenden Schubs gegeben, dieses kleine Buch zu verfassen und sie hat die Programme hierzu beigetragen.

Walter Stein — Bad Münstereifel im Dezember 2019

Inhaltsverzeichnis

1 Einführung in das Thema Künstliche Intelligenz 1

2 Deep Learning 5

3 Python und Jupyter-Notebook 11

4 Google Colaboratory, TensorFlow und Keras 13

5 Praktische Übungen 17

5.1 Einfache Zahlenfolgen 17

5.2 Logische Operatoren 19

5.3 Handgeschriebene Ziffern erkennen 23

5.4 Tiere erkennen 29

5.5 Kleidungsstücke erkennen 35

5.6 Weitere Datensammlungen 37

6 Unterrichtsmaterialien und Schülerwettbewerbe 39

7 Ausblick 41

8 Anmerkung 43

9 Index 45

1 Einführung in das Thema Künstliche Intelligenz

Wie ich schon im Vorwort erwähnt habe, enthält dieses Buch viele Links. Insbesondere Links zu YouTube-Videos. Dies geschieht nicht aus Bequemlichkeit, sondern weil die Studie "Jugend / YouTube / Kulturelle Bildung. Horizont 2019" des Rats für Kulturelle Bildung zu dem folgenden Schluss kommt. Zitat: „ Audiovisuelles Lernen in Form von Webvideos ist für Jugendliche zwischen 12 und 19 Jahren von großer Bedeutung.“ Diese aktuelle Studie findet man unter dem Link https://www.rat-kulturelle-bildung.de/publikationen/studien

Als Einstieg in das Thema Künstliche Intelligenz empfehle ich fünf deutschsprachige YouTube-Videos in der aufgeführten Reihenfolge. Also, entspannt zurücklehnen und Videos gucken:)

Video 01: Ein Vortrag von Prof. Dr. Christian Bauckhage. Dauer: 49 Minuten

https://www.youtube.com/watch?v=4yrRKIgRWXk

Video 02: Ist KI überbewertet oder unterschätzt? – Gespräch mit Prof. Wolfgang Wahlster, Direktor des Deutschen Forschungszentrums für Künstliche Intelligenz von 1997 bis Januar 2019. Dauer: 37 Minuten

https://www.youtube.com/watch?v=FDh66yNiWws

Video 03: Aljoscha Burchardt: Künstliche Intelligenz – Wird ein Traum wahr? Dauer: 8 Minuten

https://www.youtube.com/watch?v=495uojRW_ow

Video 04: arte: Mehrere Videos zu Chancen und Risiken der Künstlichen Intelligenz

https://www.arte.tv/de/videos/RC-018563/kuenstliche-intelligenz-chancen-und-risiken/

Video 05: Ranga Yogeshwar: Der große Umbruch Dauer: 88 Minuten

https://www1.wdr.de/mediathek/video/sendungen/video-der-grosse-umbruch--wie-kuenstliche-intelligenz-unser-leben-veraendert-100.html

Zur weiteren Entspannung empfehle ich das kurze, sehr amüsante, aber nicht weniger lehrreiche Video über Bots im Internet. Es trägt den Titel: **„Wie Maschinen lernen“.**

https://www.youtube.com/watch?v=R9OHn5ZF4Uo

Bei diesem YouTube-Video wird der englische Text sehr schnell vorgetragen. Man kann sich die entsprechenden Untertitel aber dank KI in Deutsch anzeigen lassen. Dazu klickt man, wie in der folgenden Abbildung 1.1 dargestellt, auf 1, dann auf 2 und schließlich auf 3. Die Icons von Abbildung 1.1 befinden sich rechts unten im Videofenster von YouTube.

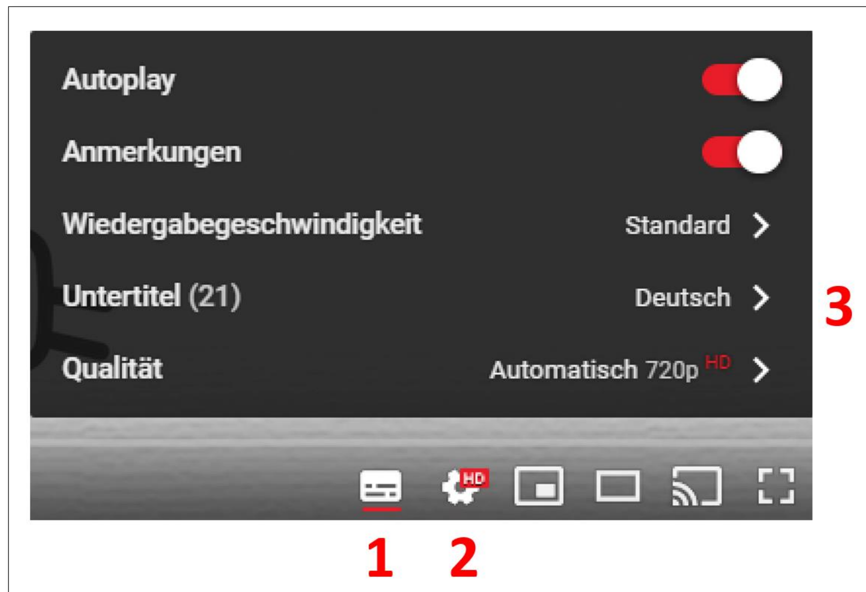

Abbildung 1.1: So fügt man deutsche Untertitel bei YouTube-Videos ein

Ebenfalls zur Entspannung kann man sich Musik anhören, die mittels KI erzeugt wurde. Hier sind zwei Links zu AIVA (Artificial Intelligence Virtual Artist). AIVA ist eine KI, die in Zusammenarbeit mit Menschen Musik komponiert.

https://www.youtube.com/playlist?list=PLv7BOfa4CxsHAMHQj0ScPXSbgBlLglRPo

https://www.youtube.com/watch?v=H6Z2n7BhMPY

Dass Künstler in zunehmendem Maße KIs zum Komponieren ihrer Musikstücke benutzen, zeigt auch die YouTuberin Taryn Southern. Hier ist der Link zu ihrem Song *Break Free*:

https://www.youtube.com/watch?v=XUs6CznN8pw

In einem Interview beschreibt Taryn Southern, wie sie ihre Musik mittels KI erstellt.

https://www.youtube.com/watch?v=hVq5ZcE2d_Q

Auch muss man heutzutage kein großer Künstler sein, um ein schönes Bild zu malen. Aus einfachen Kritzeleien erzeugt das Deep-Learning-Modell von NVIDIA ansprechende und realistische Bilder. Als Einstieg schaut man sich das Video des ersten Links an und danach testet man seine künstlerischen Fähigkeiten anhand des zweiten Links.

https://www.youtube.com/watch?v=p5U4NgVGAwg&t=40s

http://nvidia-research-mingyuliu.com/gaugan/

Ob die Künstliche Intelligenz sich zum Vorteil oder Nachteil für alle Menschen entwickeln wird, dies entscheiden zurzeit immer noch Menschen. Die Künstliche Intelligenz in Form von Deep Learning ist uns Menschen zwar in speziellen Disziplinen überlegen, doch ihre Fähigkeiten sind auf diese speziellen Disziplinen begrenzt. Das menschliche Gehirn hat diese Beschränkung aber nicht, obwohl es sehr viel kleiner und sehr viel energieeffizienter ist als heutige Großrechenanlagen.

Trotzdem wird die Künstliche Intelligenz in Form von Deep Learning die Welt grundlegend verändern.

Hoffen wir, dass die Menschen, die in der Politik oder Wirtschaft an entscheidenden Positionen sitzen, Deep Learning nicht dazu benutzen, um ihre Macht und ihren Reichtum auf Kosten der Allgemeinheit weiter zu vermehren. Hoffen wir, dass Entscheidungsträger nicht aus Unwissenheit und Dummheit Entwicklungen in die Wege leiten, die sich zum Nachteil für viele Menschen entwickeln. Wichtig ist, dass man die Chancen nutzt, die Deep Learning bietet, um das Allgemeinwohl im hohen Maße zu fördern. Welche sozialen Auswirkungen die sich rasant entwickelnde Künstliche Intelligenz hat, dies kann man in dem folgenden Wikipedia-Artikel nachlesen.

https://de.wikipedia.org/wiki/Künstliche_Intelligenz#Soziale_Auswirkungen

Ein Beitrag zu den Gefahren der Künstlichen Intelligenz liefert der folgende Link mit dem Titel: *Wie gefährlich ist künstliche Intelligenz?* Videodauer: 5 Minuten

https://www.youtube.com/watch?v=i5D1jHQ-vNk

Künstliche Intelligenz ermöglicht eine ganz neue Dimension von Fakenews. Die Juraprofessorin Danielle Citron warnt hiervor in ihrem TED-Vortrag „Wie Deepfakes die Wahrheit untergraben und die Demokratie bedrohen“. Videodauer: 13 Minuten

https://www.ted.com/talks/danielle_citron_how_deepfakes_undermine_truth_and_threaten_democracy?language=de

Lesenswert ist auch der Artikel von Alexander Armbruster in der Frankfurter Allgemeine. Er trägt den Titel: *Elon Musk und Co. warnen vor Killer-Robotern*

https://www.faz.net/aktuell/wirtschaft/kuenstliche-intelligenz-elon-musk-warnt-vor-killer-robotern-15161436.html

2 Deep Learning

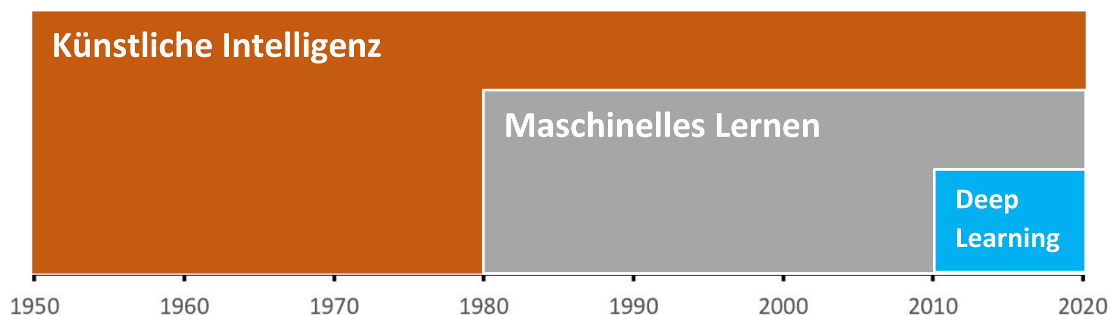

Abbildung 2.1: Die zeitliche Entwicklung der Künstlichen Intelligenz

Schon ab 1950 versuchte man mit vorgegebenen Regeln Computern das Denken beizubringen, um so eine Künstliche Intelligenz zu erschaffen. Man verbuchte erste bescheidene Erfolge, wie zum Beispiel Schachprogramme. Ein Paradigmenwechsel fand dann in den 80er Jahren statt. Beim **Maschinellen Lernen** gibt man Daten und zugehörige Ergebnisse ein und die Software lernt selbstständig die Regeln, die von den Daten zu den entsprechenden Ergebnissen führen. Mit diesen Regeln kann die Software dann für andere, aber ähnliche, Daten eigenständig die zugehörigen Ergebnisse finden. Für eine Revolution im Bereich des Maschinellen Lernens sorgte dann ab 2010 das **Deep Learning**. Mittels Deep Learning, als Teilbereich des Maschinellen Lernens, wurden nicht nur entscheidende Durchbrüche in den Bereichen Bilderkennung, Spracherkennung und Sprachsynthese erzielt, sondern auch bei der Übersetzung von Fremdsprachen, der Suche im Web und der Steuerung von Maschinen. Der Erfolg von Deep Learning beruht auf der Weiterentwicklung von Algorithmen, der Verfügbarkeit großer Datenmengen, steigender Rechenleistung und der Möglichkeit, die parallelen Rechenarchitekturen von **GPU**s (Graphics Processor Unit) auf preiswerten Grafikkarten zu nutzen. Später haben Programme wie CUDA, TensorFlow, Keras, diese Entwicklung beschleunigt.

Bevor wir nun etwas tiefer in das maschinelle Lernen und Deep Learning einsteigen, schauen wir uns das Video „Wie funktioniert eigentlich **Machine Learning**?“ von dem YouTuber Doktor Whatson (Cedric Engels) an.

https://www.youtube.com/watch?v=ya_6I9IVMzY Videodauer: 8 Minuten

Der im Video und auch in vielen populärwissenschaftlichen Veröffentlichungen aus historischen Gründen verwendete Begriff **Neuron** verleitet einen dazu, sich ein neuronales Netz wie ein Gehirn vorzustellen. Auch wenn in Sachen Bilderkennung die Architektur des Gehirns Pate gestanden hat, so haben neuronale Netze, auch wenn sie so heißen, eine gänzlich andere Funktionsweise als ein Gehirn. Gehirne funktionieren elektrisch und biochemisch. In neuronalen Netzen werden nur Heerscharen von Einsen und Nullen mit großem mathematischen Aufwand in differenzierbaren, hochdimensionalen Räumen in Beziehungen gesetzt und nach den Rechnungen in für Menschen verständliche Aussagen überführt. Der Prozess selbst kann von Menschen in der Regel jedoch nicht mehr im Einzelnen verstanden werden. Aber keine Panik! Wir starten bei $y = f(x)$.

Ein **Neuronales Netz (NN)** kann man stark vereinfacht als eine Ansammlung von Funktionen betrachten, wie zum Beispiel $y = f(x)$. Man gibt x ein und die Funktionen liefern y zurück. Gibt man z.B. ein Foto in ein NN ein, so versuchen die Funktionen im NN anhand der Bildpixelwerte zu

erkennen, was auf dem Foto dargestellt ist. Nach einer gewissen Rechenzeit gibt das NN dann das Ergebnis seiner Berechnungen bekannt, zum Beispiel: „Hund“.

Die folgende Abbildung 2.2 stellt die Struktur eines sehr kleinen NNs dar. Es besteht in dieser Abbildung aus zwei Schichten (h_1, h_2), die jeweils drei **Neuronen** enthalten, sowie der Eingabeschicht x und der Ausgabeschicht y. Solche Schichten nennt man **Layer**. Die Neuronen in den beiden Schichten sind in der Abbildung durch Kreise dargestellt. In der Biologie sind Neuronen auf Erregungsübertragung spezialisierte Nervenzellen. So ähnlich kann man sich auch die Neuronen in einem NN vorstellen. Jedes Neuron in der Schicht eines NN ist typischerweise, aber nicht immer mit allen Neuronen in der Vorgängerschicht verbunden. Das unten dargestelltes NN besteht also aus einer Verschachtelung von Funktionen.

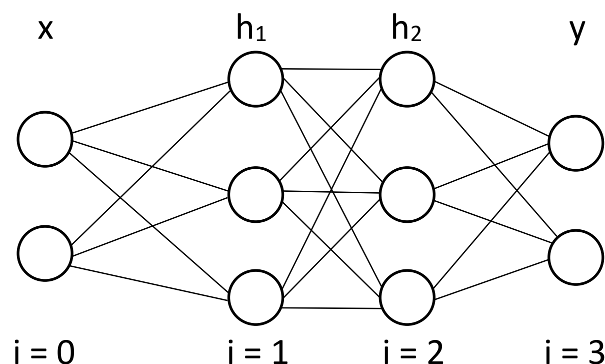

Abbildung 2.2: Ein einfaches neuronales Netz

Wie rechnet nun ein einzelnes Neuron? Schauen wir uns ein Neuron in der Schicht h_2 näher an. Wie in der Abbildung 2.3 dargestellt, erhält das Neuron die drei Eingangsinformationen x_1, x_2 und x_3 von den drei Neuronen aus der Schicht h_1. Diese Eingangsinformationen werden jedoch jeweils unterschiedlich gewichtet (w_1, w_2, w_3). Das Neuron gibt entsprechend dieser Eingangsinformationen einen Wert a aus. Den Ausgabewert berechnet das Neuron als gewichtete Summe der Eingangswerte x mit den **Wichtungswerten** w und einem **Biaswert** b wie folgt: $a = w \cdot x + b$. Zu beachten ist, dass das x in der Gleichung ein Eingabevektor $x = (x_1, x_2, x_3)$ und w ein Wichtungsvektor $w = (w_1, w_2, w_3)$ ist. Der Biaswert b in der Gleichung ist nur eine Zahl, in unserem Beispiel beträgt er 1.

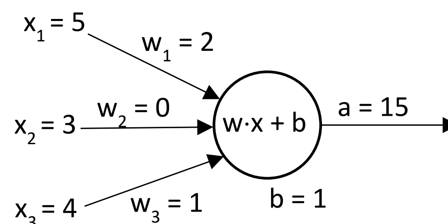

Abbildung 2.3: Ein Neuron mit Eingabewerten, Wichtungswerten, Aktivierungsfunktion und Ausgabewert
$a = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + b = 5 \cdot 2 + 3 \cdot 0 + 4 \cdot 1 + 1 = 15$

Wer die Programmiersprache Python beherrscht, der kann die in Abbildung 2.3 durchgeführte Rechnung auch mittels eines kleinen Programms durchführen (siehe folgende Seite). Mehr zu Python findet man in Kapitel 3.

```python
import numpy as np
w = np.array([2, 0, 1])
x = np.array([5, 3, 4])
a = np.dot(w, x) + 1
print(a)
# Ergebnis: 15
```

Die Ausgabe nach einer ganzen Schicht geschieht durch eine Funktion $h_i = f_i(x_{i-1})$.

Diese Funktion lautet $\mathbf{h_i = W_i \cdot x_{i-1} + b_i}$

Sie liefert als Ausgabe einen Vektor $h_i = (a_1, a_2, \dots a_n)$

Hierbei ist b_i ein Biasvektor $b_i = (b_1, b_2, \dots b_n)$

und W_i eine Wichtungsmatrix $$W_i = \begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{pmatrix}$$

Da $h_i = W_i \cdot x_{i-1} + b_i$ eine lineare Funktion ist, können mit solch linearen Netzwerken auch nur einfache Aufgaben gelöst werden. Die Abbildung 2.4 links zeigt als Beispiel, wie ein NN zwischen Punkten und Kreuzen unterscheiden kann, indem es bildlich gesprochen die Punkte und Kreuze mittels einer Linie, also einer linearen Funktion, voneinander trennt.

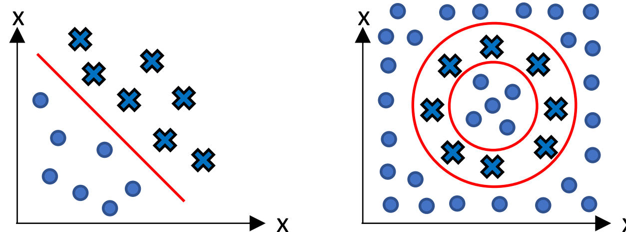

Abbildung 2.4: Lineare Aufgabenstellung (links) und nichtlineare Aufgabenstellung (rechts)

Nichtlineare Aufgabenstellungen (Abb. 2.4, rechts) benötigen aber nichtlineare Funktionen, wie zum Beispiel $a = \tanh(w \cdot x + b)$.

Der Ausgabewert des bisher betrachteten Neurons wird dann mittels einer nichtlinearen Funktion verrechnet. Solche Funktionen nennt man **Aktivierungsfunktionen**.

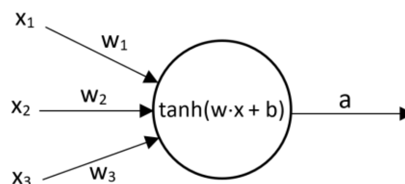

Abbildung 2.5: Neuron mit nichtlinearer Aktivierungsfunktion

Heute verwendet man jedoch anstelle von tanh andere Aktivierungsfunktionen (Abb. 2.6). Hier drei Beispiele.

Sigmoid-Funktion $y = \frac{1}{1+e^{-x}}$

ReLU-Funktion, für die gilt: x < 0 → y = 0 und x > 0 → y = x

PReLU-Funktion für die gilt: x < 0 → y = a·x und x > 0 → y = x. Bei der PReLU-Funktion ist a nicht konstant, sondern wird vom neuronalen Netz beim Lernprozess angepasst.

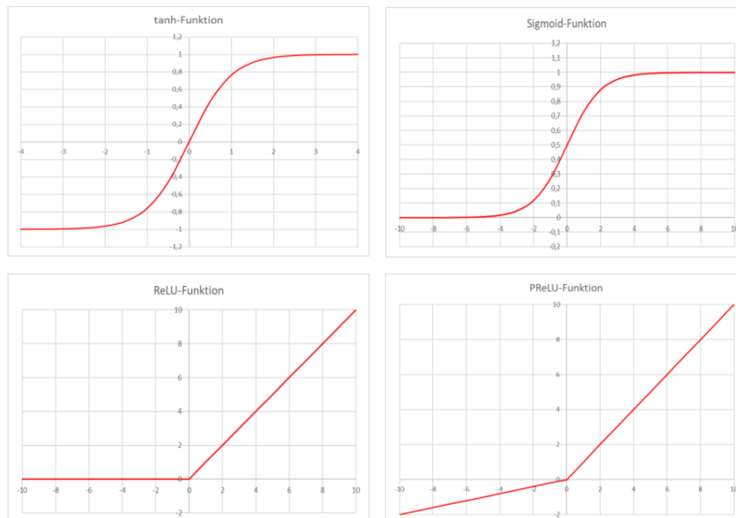

Abbildung 2.6: Unterschiedliche Aktivierungsfunktionen

Bevor ein NN Entscheidungen treffen kann, muss es mit sehr vielen bekannten Daten trainiert werden. Mit Hilfe einer **Verlustfunktion** (Fehlerfunktion), zum Beispiel $\varepsilon\,(y_w,\ y_a) = |y_w - y_a|$ kann dann, nach Eingabe eines Beispiels aus den bekannten Daten in das NN, der aus den Daten bekannte wahre Wert y_w mit dem ausgegebenen Wert y_a verglichen werden. In nahezu allen Fällen weicht der ausgegebene Wert vom wahren Wert ab. Dann ändert das NN die einzelnen Wichtungen und Biaswerte mit einem sogenannten **Optimierer** und hofft, dass der nächste ausgegebene Wert dem wahren Wert etwas näherkommt. Die Abweichung des ausgegebenen Wertes vom wahren Wert kann sich durch die Änderung der Wichtungen aber auch vergrößern. Also probiert das NN es erneut mit einem anderen Wert. Trägt man nun alle ε-Werte gegen die möglichen Parameter X_z des NNs auf, so erhält man das folgende Diagramm (Abb. 2.7).

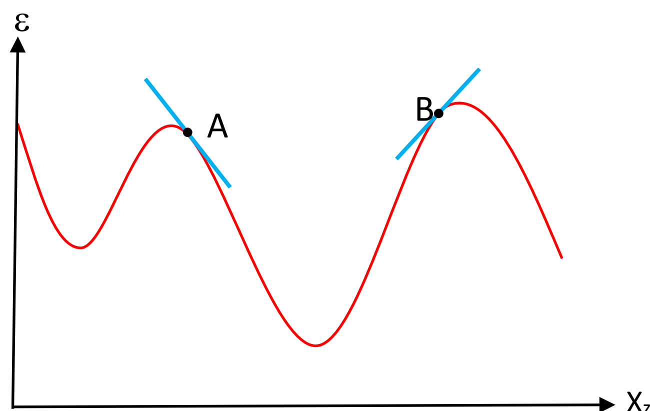

Abbildung 2.7: Wert der Fehlerfunktion mit zwei eingezeichneten Steigungen in Bezug auf die möglichen Parameter (Wichtungsmatrizen und Biasvektoren) eines NNs

Um den Fehler zu minimieren, kann man den oben dargestellten Graphen differenzieren. Dieser Prozess wird als Backpropagation bezeichnet. D.h., man bestimmt die Steigung zum Beispiel für den Punkt A oder B (siehe Abb. 2.7). Ist die Steigung negativ, dann muss man den Wert von X_z etwas erhöhen, also entsprechend der obigen Abbildung nach rechts verschieben. Ist die Steigung positiv, so muss man den Wert von X_z etwas verkleinern, also nach links verschieben. Da man mit dieser Methode jedoch meist mehrere Minima findet, muss man jetzt noch das Minimum für den kleinsten Fehler suchen. Den ganzen beschriebenen Vorgang muss man jedoch nicht per Hand durchführen, sondern dies erledigt, wie oben schon erwähnt, das NN mittels der Verlustfunktion und dem Optimierer selbst. Diese führen mittels Differenzierung und **Gradientenabstiegs-verfahren** (X_z nach links oder rechts verschieben) einen Fehlerminimierungsprozess durch. Dies gelingt durch die Änderung der **Wichtungsmatrizen** und der **Biasvektoren**. Schülerinnen und Schüler wären mit solchen Rechenoperationen überfordert, da diese in einem mehrdimensionalen Raum erfolgen. Doch keine Panik! Diese Rechnungen erledigt Python für uns.

Die Abbildung 2.8 fasst das oben Beschriebene noch einmal zusammen.

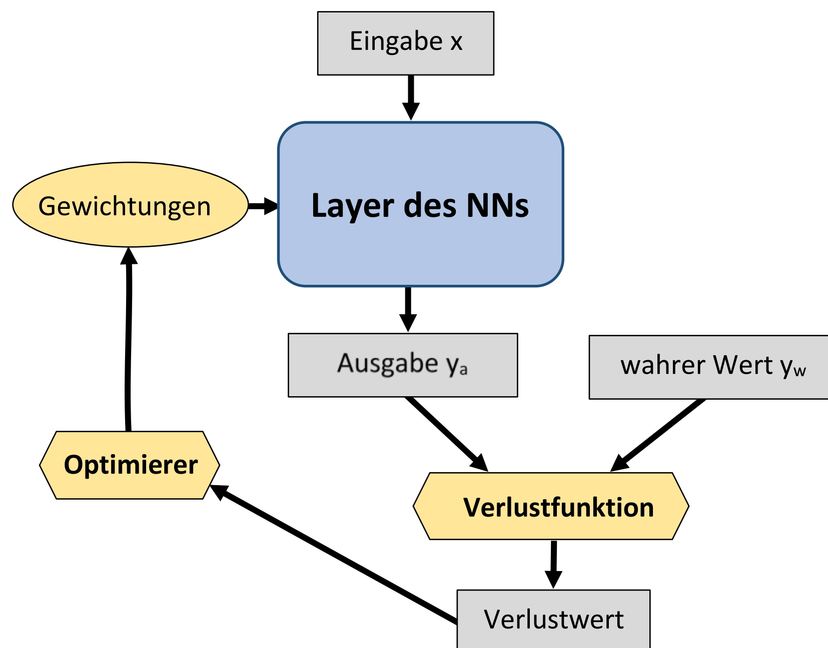

Abbildung 2.8: Überblick über die Funktionsweise eines neuronalen Netzes

Zur Festigung und zur Vertiefung des Gelernten lese man den Artikel „Wie lernen künstliche Neuronale Netze?" von Roland Becker.

https://jaai.de/machine-deep-learning-529/

3 Python und Jupyter-Notebook

In Kapitel 2 habe ich die folgenden furchterregenden Sätze geschrieben: *„In neuronalen Netzen werden nur Heerscharen von Einsen und Nullen mit großem mathematischen Aufwand in differenzierbaren, hochdimensionalen Räumen in Beziehungen gesetzt und nach den Rechnungen in für Menschen verständliche Aussagen überführt. Der Prozess selbst kann von Menschen in der Regel jedoch nicht mehr im Einzelnen verstanden werden."* Auch von den Menschen nicht, die sich mit Differential- und Tensorrechnung auskennen. Und als Schülerin und Schüler versteht man diese Rechnungen erst recht nicht. Vielleicht haben diese das Wort Tensor in der Schule schon mal gehört und gelernt, dass dies ein Oberbegriff für Skalare, Vektoren und Matrizen ist. Aber hiermit Rechnungen in hochdimensionalen Räumen durchzuführen wäre jedoch zu viel verlangt.

Wenn solche Rechnungen für Deep Learning aber notwendig sind, wie soll es denn eine Schülerin oder ein Schüler schaffen, hierzu ein funktionsfähiges Computerprogramm zu schreiben? Hier hilft die Programmiersprache **Python** im Zusammenspiel mit dem Framework **TensorFlow** und der Python-Deep-Learning-Bibliothek **Keras** (Mehr zu TensorFlow und Keras in Kapitel 4). So können auch ohne fundierte Mathematikkenntnisse funktionsfähige Programme geschrieben werden. Um TensorFlow und Keras zu nutzen, ist es jedoch unbedingt notwendig, Grundlagen in der Programmiersprache Python zu erwerben. Erfreulicherweise ist Python eine sehr leicht zu erlernende und zudem eine weitverbreitete Programmiersprache. Mit der zugehörigen Programmbibliothek **NumPy** können Vektoren, Matrizen und mehrdimensionale Arrays so auch von Schülerinnen und Schülern ohne großartige Mathematikkenntnisse berechnet werden.

Zum Erlernen der Programmiersprache Python findet man im Internet zahlreiche Tutorials. Besonders gut gefallen hat mir das Online-Tutorial der Universität Heidelberg für das physikalische Anfängerpraktikum. Siehe:

https://www.physi.uni-heidelberg.de/Einrichtungen/AP/Python.php

Klickt man hier auf *Interaktive Kursmaterialien starten*, dann öffnet sich ein interaktives **Jupyter-Notebook**. Nun kann man sich entscheiden, ob man den Kurs offline oder online absolviert. Alle hier in diesem Buch erwähnten Programme wurden von mir online verwendet. So musste ich nichts installieren. Aber dies muss jeder für sich selbst entscheiden. Als Schülerin oder Schüler wird man den Kurs wahrscheinlich nicht auf dem Schulcomputer speichern dürfen.

Ein ebenfalls empfehlenswerter, interaktiver und kostenloser Online-Kurs für Schüler/innen und Lehrer/innen findet sich unter:

https://cscircles.cemc.uwaterloo.ca/using-website-de/

Am Ende dieses Kapitels soll noch kurz erklärt werden, was ein Jupyter-Notebook ist. Ein Jupyter-Notebook ist eine Open-Source-Webanwendung, mit der man in seinem Browser nicht nur Dokumente erstellen und freigeben kann, sondern auch Code in Python und anderen Programmiersprachen schreiben kann. Seine Oberfläche ist intuitiv zu bedienen, da sie einem Notebook ähnelt. Jupyter-Notebooks werden von Studenten und Wissenschaftlern in zunehmendem Maße genutzt. Sie sind auch bestens für die Erstellung von Programmen zum Thema Deep Learning geeignet. Wer mehr zum Thema Jupyter-Notebook erfahren will, der klicke auf diesen Link: https://jupyter.org

4 Google Colaboratory, TensorFlow und Keras

Bevor wir nun im nächsten Kapitel mit der konkreten Programmierung beginnen, müssen wir eine grundsätzliche Frage klären. Wollen wir die aufwendigen Rechnungen bei Deep Learning offline auf unserem eigenen Computer ausführen oder lieber online in der Cloud? Wenn man offline arbeiten will, dann braucht man eine leistungsstarke Grafikkarte, da die Berechnungen mittels einer **CPU** oft Stunden bis Tage dauern. Weiterhin muss man Python, die Deep-Learning-Plattform TensorFlow, CNTK oder Theano sowie die Deep-Learning-Bibliothek Keras und weitere Programme auf seinem Computer installieren. Anleitungen hierzu gibt es im Internet oder für Ubuntu in dem empfehlenswerten Buch *Deep Learning mit Python und Keras* von François Chollet. Die Online-Version des Buches findet man unter dem folgenden Link.

https://livebook.manning.com/book/deep-learning-with-python/about-this-book/

Eine Anleitung zur Installation von TensorFlow 2 findet man unter

https://www.tensorflow.org/install

Ich habe mich für die Online-Lösung mit **Google Colaboratory** entschieden, da ich glaube, dass dies auch im Sinne der meisten Schüler/innen und Lehrer/innen ist. Bei Google Colaboratory, kurz **Colab** genannt, müssen keine Programme installiert werden. Man benötigt nur einen aktuellen Webbrowser wie Chrome, Firefox oder Safari und einen Google Account (Google-Konto). Dem Nutzer entstehen keine Kosten, obwohl Google dem Nutzer für die Ausführungen der von ihm geschriebenen Programme eine Tesla P100 oder sogar eine Tesla T4 GPU zur Verfügung stellt, welche extrem leistungsstarke Deep Learning GPUs sind, die normalerweise ca. 5000 € kosten! Seine Programme schreibt man bei Colab in der Programmiersprache Python unter Einbeziehung von TensorFlow und Keras in einer Jupyter-Notebook-Umgebung. Diese Programme werden in **Google Drive**, also in der Google Cloud gespeichert. Ein Einführungsvideo zu Google Colaboratory findet man unter dem folgenden Link.

https://www.youtube.com/watch?v=inN8seMm7UI

Weitere Informationen zu Google Colaboratory findet man unter:

https://colab.research.google.com/notebooks/intro.ipynb

Klickt man auf den obigen Link und anschließend auf *More Resources*, so erhält man vielfältige Informationen zu Google Colaboratory und TensorFlow (Abb. 4.1).

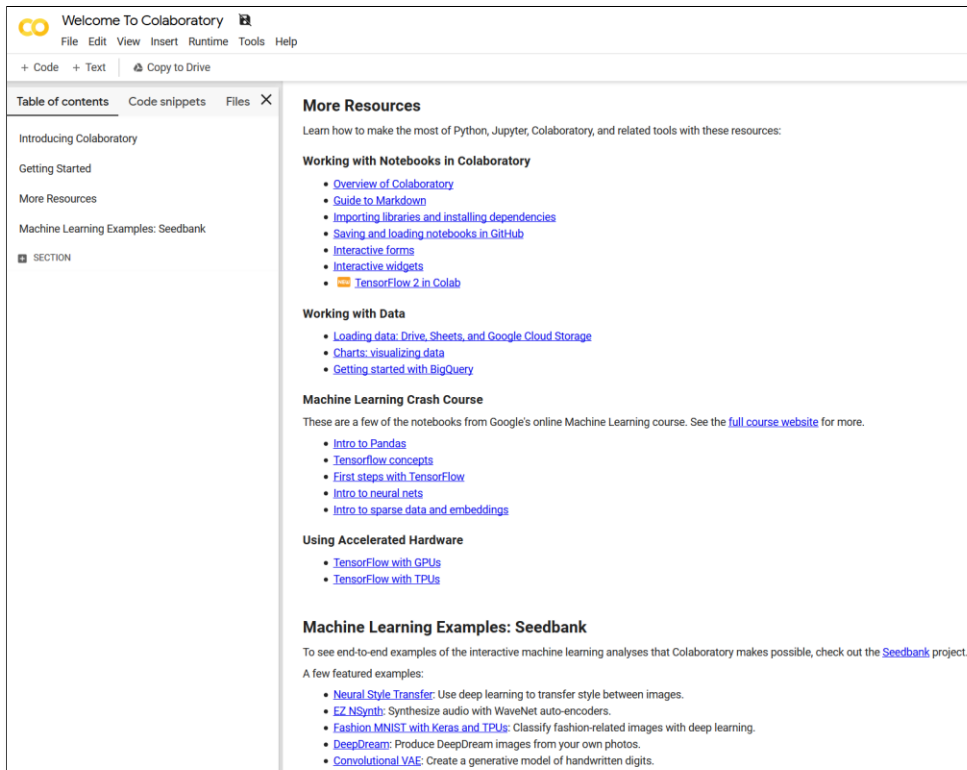

Abbildung 4.1: Willkommens- und Informationsseite von Google Colaboratory

Will man ein Programm in Colab erstellen, so klickt man in dem Welcome-Fenster (Abb. 4.1) auf *File* und anschließend auf *New Python 3 notebook* (Abb. 4.2 links). Nun kann man damit beginnen, seinen Code zu schreiben (Abb. 4.2, rechts). Mehr dazu jedoch anhand konkreter Beispiele im nächsten Kapitel.

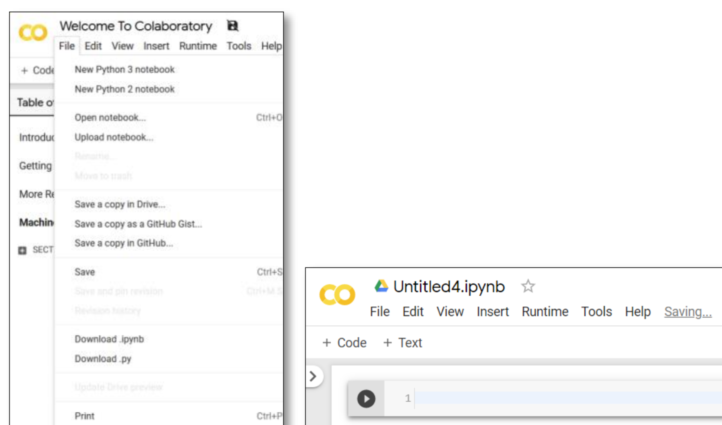

Abbildung 4.2: Nach dem Klick auf File kann man sein Python-Notebook für die eigene Programmentwicklung öffnen

Wenn man sein Programm geschrieben hat, dann startet man es, indem man auf das kleine graue Dreieck im schwarzen Kreis klickt. Möchte man zudem noch Zugriff auf seine eigenen Dateien in Google Drive haben, so muss man wie in Abb. 4.3 dargestellt, die Codezeilen 2 und 3 eingeben und ausführen. Nun wird man aufgefordert einen Autorisierungscode einzugeben. Den erhält man, wenn man auf den blauen Link klickt (Abb. 4.3). Nachdem man den Autorisierungscode eingegeben hat, hat man für den Rest der „Sitzung“ Zugriff auf Google Drive.

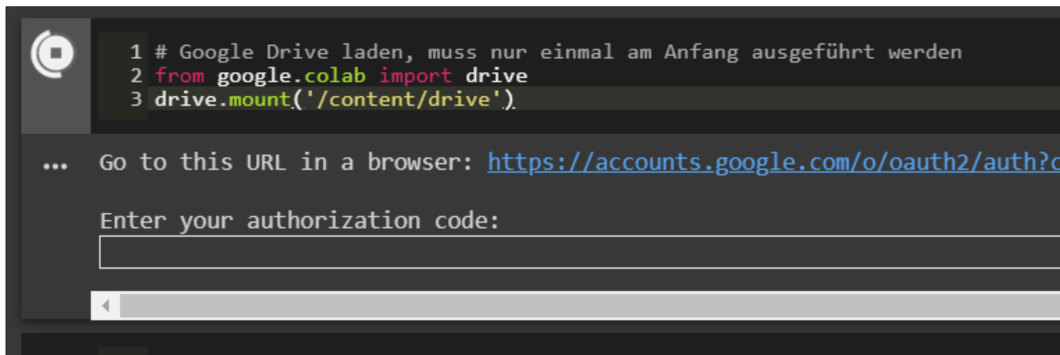

Abbildung 4.3: Google Colab verlangt einen Autorisierungscode

In Abbildung 4.3 sieht man einen Ausschnitt des Python-Notebooks mit schwarzem Hintergrund. Die ansonsten in meinem Buch wiedergegebenen Programme besitzen jedoch alle einen weißen Hintergrund. Dies spart Druckerfarbe, wenn man sich die eine oder andere Seite ausdrucken möchte. Während des Programmierens sollte man aber einen schwarzen Hintergrund verwenden, da dieser augenschonender ist. Im Python-Notebook stellt man einen schwarzen Hintergrund wie folgt ein: Tools → Preferences → Theme → dark

In Python-Notebook von Colab sollte man sicherstellen, dass der selbsterstellte Code nach dem Start nicht von einer CPU, sondern von einer GPU abgearbeitet wird. Der Geschwindigkeits-unterschied ist beachtlich. So erfolgt die Einstellung: Oben in der Menüleiste bei Runtime → Change Runtime Type → Hardware Accelerator auf GPU klicken.

Zum Schluss des Kapitels noch je ein Link zu TensorFlow und Keras. TensorFlow von Google ist das am weitesten verbreitete Machine-Learning-Framework und wird in Google Colaboratory verwendet. In TensorFlow enthalten ist die Deep-Learning-Bibliothek Keras. Keras wurde von François Chollet entwickelt, um Programmierern die Erstellung von Deep Learning-Modellen zu erleichtern.

Einen kurzen Überblick über TensorFlow in Google Colab findet man unter dem Link:

https://colab.research.google.com/notebooks/mlcc/tensorflow_programming_concepts.ipynb#scrollTo=NzKsjX-ufyVY

Ausführliche Informationen zu Keras findet man unter: https://keras.io

Fassen wir kurz zusammen: Wir erstellen unsere Programme online in Colab mittels Python, TensorFlow und Keras.

5 Praktische Übungen

5.1 Einfache Zahlenfolgen

Beginnen wir unsere praktischen Übungen noch nicht mit Deep Learning, sondern mit einem ganz einfachen Beispiel für maschinelles Lernen. Unser erstes neuronales Netz soll eine Beziehung zwischen zwei Zahlenfolgen erkennen. Nachdem es trainiert ist, soll es uns die richtige Zahl für die zweite Zahlenfolge anzeigen, wenn wir einen beliebigen ganzzahligen Wert für die erste Zahlenfolge eingeben.

Schauen wir uns hierzu das folgende YouTube-Video an, welches von Jacob Beautemps in Zusammenarbeit mit Philip Häusser erstellt wurde. In diesem Video behauptet Jacob, dass man seine erste KI ganz ohne Vorkenntnisse innerhalb von fünf Minuten erstellen kann. Erfreulicherweise hat er seinen Code mit ausführlichen Kommentaren versehen und auch zum Download zur Verfügung gestellt.

Hier ist der Link zu diesem Video: https://www.youtube.com/watch?v=8Qc2fG3ZbTg

Wenn man sich das Programm mit seinen Kommentaren gründlich angeschaut hat, dann hat man schon viel gelernt und kann mit dem Programm auch etwas „spielen". D.h., gezielt Größen ändern und schauen was passiert. Da es ein sehr kurzes Programm ist, macht das Abtippen keine große Mühe. Beim Abtippen muss man genau hinschauen und deshalb lernt man auch hierbei etwas.

Im ursprünglichen Programm wurden die Zahlen der ersten Folge mit dem Faktor Zwei multipliziert, um die Zahlen der zweiten Folge zu erhalten. Passen wir das Programm dem neuen TensorFlow 2.x an und testen, ob es auch mit dem Faktor Drei funktioniert?

```
 1  try:
 2    %tensorflow_version 2.x
 3  except Exception:
 4    pass
 5
 6  from tensorflow import keras
 7
 8  model = keras.Sequential()
 9  model.add(keras.layers.Dense(units=1, input_shape=[1]))
10  model.compile(optimizer='sgd', loss='mean_squared_error')
11
12  inputs=[1, 2, 3, 4, 5, 6, 7, 8, 9]
13  expected_outputs=[3, 6, 9, 12, 15, 18, 21, 24, 27]
14
15  model.fit(inputs, expected_outputs, epochs=200)
16  print(model.predict([11]))
```

Wir geben die Zahl 11 ein (Zeile 16 im Programm) und erhalten als Ergebnis die Zahl 32,760254 ≈ 33. Also ein brauchbares Ergebnis, denn $3 \cdot 11 = 33$.

Funktioniert das NN auch mit diesen Zahlenfolgen? Einfach ausprobieren!

```
inputs = [1, 2, 3, 4, 5, 6, 7, 8, 9]
expected_outputs = [1, 3, 5, 7, 9, 11, 13, 15, 17]
```

Wo liegen die Grenzen dieser sehr einfachen KI?

5.2 Logische Operatoren

Schauen wir uns ein weiteres Beispiel zum „Spielen“ an. Allgemein bekannt sind die logischen Operatoren AND, OR, XOR und NOT. Ob man mittels maschinellen Lernens logische Operationen richtig lösen kann, dies kann man zum Beispiel für XOR mit dem folgenden Programm testen, wenn man die Programmzeilen nach Google Colab überträgt.

```
1   # Neueste Version von TensorFlow laden
2   try:
3     %tensorflow_version 2.x
4   except Exception:
5     pass
6
7   # Keras aus Tensorflow und Numpy importieren
8   from tensorflow import keras
9   import numpy as np
10
11  # XOR-Datenset definieren
12  inputs = np.array([[0, 0], [1, 0], [1, 1], [0, 1]])
13  expected_outputs = np.array([0, 1, 0, 1])
14
15  # Netzwerk erstellen mit zwei Dense-Layern (5 Neuronen, 1 Neuron)
16  model = keras.Sequential()
17  model.add(keras.layers.Dense(units = 5, activation = 'tanh', input_shape=(2,)))
18  model.add(keras.layers.Dense(units = 1, activation = 'sigmoid'))
19
20  # Trainingsparameter festlegen und Netzwerk trainieren
21  model.compile(loss='mean_squared_error', optimizer='adam')
22  model.fit(inputs, expected_outputs, epochs = 10000, verbose = 0)
23
24  # Trainiertes Netzwerk testen
25  print(model.predict(inputs))
```

Das obige Programm liefert für das ausschließende Oder XOR die folgenden Ergebnisse.

XOR
[0.0041326] ≈ 0
[0.9959012] ≈ 1
[0.00588012] ≈ 0
[0.9931472] ≈ 1

Tabelle 1: Ergebnisse des neuronalen Netzes für XOR

Die logische Operation XOR wird also richtig gelöst. Gelingt dies auch mit AND und OR? Einfach die entsprechenden Änderungen in dem obigen Programm vornehmen und ausprobieren! Hier als Hilfe die unten aufgeführte Wahrheitstabelle für AND und OR. Die Zahl 1 steht für „wahr“ und die Zahl 0 steht für „falsch“.

x	y	AND	OR
0	0	0	0
1	0	0	1
1	1	1	1
0	1	0	1

Tabelle 2: Wahrheitstabelle für AND und OR

Wenn man alles richtig gemacht hat, dann liefert das neuronale Netz die folgenden, richtigen Ergebnisse. Die roten Zahlen ergeben gerundet 1, also „wahr“ und die blauen Zahlen ergeben gerundet 0, also „falsch“.

AND	OR
[[3.8623810e-05]	[[0.00497425]
[3.3634305e-03]	[0.9963947]
[9.9528790e-01]	[0.99990237]
[3.1374395e-03]]	[0.99654454]]

Tabelle 3: Ergebnisse des neuronalen Netzes für AND und OR

Nachdem man das Programm außer mit XOR auch mit den Operatoren AND und OR getestet hat, sollte man nun auch einige andere Größen ändern. Dadurch lernt man die Bedeutung dieser Programmelemente kennen.

Klären wir zuerst die Bedeutung einzelner, im Programm vorkommenden, Begriffe. Als einfaches Modell für das neuronale Netz wir das Modell ***Sequential*** verwendet, welches aus einem Stapel von Layern (Schichten) besteht. Die zwei Layer in dem Modell tragen den Namen ***Dense***. Schauen wir uns dazu die Zeilen 17 und 18 in dem Code an:

```
model.add(keras.layers.Dense(units = 5, activation = 'tanh', input_shape=(2,)))

model.add(keras.layers.Dense(units = 1, activation = 'sigmoid'))
```

Die Zahlen 5 und 1 hinter *units* geben die Zahl der Neuronen an, die die jeweiligen Layer enthalten. Mit *input_shape=(2,)* teilt man dem Modell mit, dass es als Eingabeform einen Tensor, der zwei Werte enthält, erwarten kann. Diese Angabe muss nur im ersten Layer des Modells erfolgen. Als Aktivierungsfunktionen für die einzelnen Neuronen wurden die Funktionen *tanh* und *sigmoid* gewählt. Jeder *Dense*-Layer nimmt also einen Tensor an, bearbeitet ihn mittels seiner Aktivierungsfunktionen, sowie mittels Wichtungsmatrix und Biasvektoren und gibt den so bearbeiteten Tensor wieder aus.

Beim Kompilieren des Modells wurde die Verlustfunktion (loss) ***mean_squared_error*** und der Optimierer (optimizer) ***adam*** verwendet (Zeile 21).

Nachdem das Modell kompiliert ist, muss es nun noch trainiert werden (Zeile 22). Hier werden dem Modell die Eingangs- und Ausgangswerte übergeben. In 10000 Durchläufen (*epochs=10000*) versucht das Modell mit Hilfe des Optimierers den wirklichen Werten so nahe wie möglich zu kommen. Damit nicht jeder Durchlauf nach dem Start des Programms als einzelne Zeile im Programmfenster angezeigt wird, steht in der Klammer *verbose=0*. Möchte man jedoch die Entwicklung der genannten Werte beobachten, dann muss man *verbose=1* schreiben, oder die Funktion ***verbose*** ganz weglassen. Die Auflistung der Werte bei 10000 Durchläufen dauert jedoch recht lange.

Mit ***print()*** zeigt das Modell dann das Ergebnis seiner Rechnungen an (siehe obige Tabellen).

Nach der Erläuterung der einzelnen Programmteile kann man nun mit ihnen „spielen“. Hier einige Anregungen. Um die Auswirkungen der „Spielereien“ zu verfolgen, muss man, wie oben schon angemerkt, die Funktion *verbose* im Code entfernen.

- Wird das Ergebnis wesentlich schlechter, wenn man die Anzahl der Durchläufe von 10000 auf 1000 reduziert?
- Liefert die Aktivierungsfunktion *relu* bei 10000 Durchläufen bessere Ergebnisse als *tanh*?
- Wird das Ergebnis besser, wenn man die Anzahl der Neuronen in dem ersten *Dense*-Layer von 5 auf 32 erhöht?
- Wird das Ergebnis besser, wenn man einen weiteren *Dense*-Layer mit 5 Neuronen einfügt?
- Ersetze den Optimierer *adam* durch einen anderen Optimierer, z.B. *RMSprop* und schaue dir das Ergebnis an. Welcher der beiden Optimierer erzeugt die besseren Ergebnisse? Einen Überblick über die in Keras zur Verfügung stehenden Optimierer findet man unter: https://keras.io/optimizers/

Da in der digitalen Welt die Entwicklung mit Riesenschritten voranschreitet, wird auch TensorFlow und Keras immer weiter verbessert. Wenn bei der Ausführung eines Programms also Warnhinweise derart auftauchen, dass man nicht die neueste Version verwendet, so muss man nicht sofort in Panik geraten, sondern sich im Internet an geeigneter Stelle die notwendigen Informationen besorgen.

5.3 Handgeschriebene Ziffern erkennen

Nach den ersten beiden „Spielereien“ wollen wir nun in Deep Learning einsteigen. Meine ehemalige Schülerin Lena Maxine Lenkeit hat dazu ein Programm geschrieben, welches die folgenden handgeschriebenen Zahlen erkennen kann (Abb. 5.1). Diese Zahlen wurden mit der Maus in einem Grafikprogramm gezeichneten und dann nach **Google Drive** exportiert.

Abbildung 5.1: Handgeschriebene Zahlen

Unten findet man nun das zugehörige Programm für ein sogenanntes **CNN (Convolutional Neural Network)**. Ein CNN eignet sich mit seinen Convolutional Layers besonders gut zur Bilderkennung, da die Neuronen untereinander ein für die Bilderkennung vorteilhaftes Verschaltungsmuster besitzen. Es erreicht für die gegebene Aufgabenstellung eine Trefferwahrscheinlichkeit von über 99%. Mit einem neuronalen Netz aus **Dense Layer**, bei dem sämtliche Bildpixel betrachtet werden, erreicht man bei dem hier gestellten Problem der Zahlenerkennung „nur“ eine Trefferwahrscheinlichkeit von 97%. CNNs beschäftigen sich nicht mit allen Bildpixeln, sondern sie suchen nach lokalen Mustern und Kombinationen von Mustern. Besonders informativ sind hier die Randbereiche der im Bild enthaltenen Objekte (Portrait des Mannes in Abb. 5.2). In unserem konkreten Beispiel bieten sich dazu die Randbereiche der Zahlen an. Der schwarze Hintergrund und die weißen Innenflächen der Zahlen tragen schließlich wenig bis nichts zur Bilderkennung bei.

Abbildung 5.2: Die Randbereiche von Bildobjekten liefern die wesentlichen Bildinformationen

Das unten aufgeführte Programm zur Zahlenerkennung wurde in einem Jupyter-Notebook von Google Colaboratory geschrieben und besteht aus drei sogenannten „**code cells**“. Die Aufteilung eines Programms in einzelne „code cells“ erleichtert die Fehlersuche deutlich. Man kann eine neue Code-Zelle erstellen, indem man mit dem Mauszeiger auf den Rand einer schon vorhandenen Code-Zelle geht (Abb. 5.3).

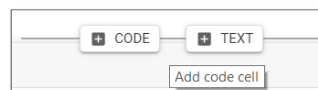

Abbildung 5.3: Einfügen einer neuer Code-Zelle

Kommen wir nun zum Programm. In der ersten Codezelle wird Google Drive in Colab geladen, da sich hier die Bilder unserer handgeschriebenen Zahlen befinden.

```
1  # Google Drive laden, muss nur einmal am Anfang ausgeführt werden
2  from google.colab import drive
3  drive.mount('/content/drive')
```

In der zweiten Codezelle (s.u.) werden die Bibliotheken aufgerufen und das neuronale Netz wird mit den notwendigen Daten gefüttert, um es anschließend mit diesen Daten zu trainieren.

```
# Die neueste Version von TensorFlow laden
try:
   %tensorflow_version 2.x
except Exception:
   pass

# Bibliotheken importieren
from tensorflow import keras
from keras.datasets import mnist
import numpy as np

# Datenbank mit Trainings- und Test-Datenset importieren
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Daten in das richtige Format bringen (float32, 28 * 28 Pixel in XY Format, 0 bis 1, 1 Farbkanal)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train.reshape((-1, 28, 28, 1))
x_test = x_test.reshape((-1, 28, 28, 1)) # -1 repräsentiert die Anzahl der Bilder im Set
# (Die -1 ist ein Trick von Numpy: So findet Numpy die Anzahl der Bilder selbst heraus)
x_train /= 255
x_test /= 255

# Modelstruktur definieren
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(filters = 16, kernel_size = (3, 3), padding = 'same', input_shape=(28, 28, 1,)))
model.add(keras.layers.PReLU())
model.add(keras.layers.Conv2D(filters = 16, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.Conv2D(filters = 16, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
model.add(keras.layers.Conv2D(filters = 32, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.Conv2D(filters = 32, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.Conv2D(filters = 32, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
model.add(keras.layers.Conv2D(filters = 64, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.Conv2D(filters = 64, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.Conv2D(filters = 64, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
model.add(keras.layers.Conv2D(filters = 128, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.Conv2D(filters = 128, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.Conv2D(filters = 128, kernel_size = (3, 3), padding = 'same'))
model.add(keras.layers.PReLU())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(50))
model.add(keras.layers.PReLU())
model.add(keras.layers.Dense(10, activation='softmax')) # Letzte Schicht des Netzwerkes, gibt die Klasse an

# Parameter für die Optimierung definieren
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['sparse_categorical_accuracy'])
# Model an die Daten anpassen bzw. trainieren
model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size = 256, epochs = 10)
```

Die einzelnen Zeilen dieses Programmteils bedürfen jedoch der näheren Erläuterung. In den Zeilen 2 bis 5 sorgen wir dafür, dass die neueste Version von TensorFlow geladen wird. In Zeile 8 importieren wir mittels TensorFlow das Deep-Learning-Framework Keras. In Zeile 9 importieren wir die in Keras enthaltene Datensammlung **MNIST**. Sie enthält 60000 Graustufenbilder von handgeschriebenen Zahlen von 0 bis 9 im Format von 28x28 Pixel und die dazu gehörigen 10000

Testbilder, ebenfalls handgeschriebene Zahlen von 0 bis 9 im Format von 28x28 Pixel. In Zeile 10 wird die Programmbibliothek NumPy importiert. Sie eignet sich besonders gut zur Verarbeitung von Vektoren, Matrizen und mehrdimensionalen Arrays.

In der Zeile 13 werden die Trainings- und Testdaten geladen und anschließend in das richtige Format gebracht (Zeile 16 bis 22). Da neuronale Netze bevorzugt mit einer 32-Bit-Genauigkeit arbeiten, werden in den Zeilen 16 und 17 mit **astype('float32')** die Bilder in ein 32-Bit-Floatzahlenformat konvertiert. Mit **reshape(-1, 28, 28, 1)** in den Zeilen 18 bis 19 sorgt man dafür, dass die Bilder in eine Form gebracht werden, die die später verwendete Funktion **Conv2D()** erwartet (Zeile 26). Was die Zahl -1 in der Klammer von *reshape* bedeutet, wird in den Zeilen 19 und 20 erklärt. Natürlich hätten wir bei *x_train* anstelle der -1 auch 60000 und bei *x_test* 10000 schreiben können. Aber wenn man die Anzahl der Bilder nicht kennt, dann ist -1 schon die bessere Wahl. Als letzte Zahl in der Klammer von *reshape* steht eine 1 ohne Minuszeichen. Diese Zahl gibt die Anzahl der Farbkanäle an. Graustufenbilder haben nur einen Farbkanal. Farbbilder haben drei Farbkanäle (RGB). Bleibt noch die Bedeutung der beiden Zahlen 28 und 28 zu klären. Hiermit wird die Bildgröße auf 28x28 Pixel festgelegt. Dies ist gewiss ein sehr kleiner Wert für ein Bild. Doch bei 60000 zu bearbeitenden Bildern ist dies schon eine sinnvolle Größe, um die Bearbeitungszeit in einem angemessenen Rahmen zu halten.

In den Zeilen 21 und 22 teilen wir die Trainings- und Testwerte durch 255. Da ein Pixel bei einem Graustufenbild die Werte 0 (schwarz) bis 255 (weiß) annehmen kann, sorgen wir mittels der Division durch 255, dass sich die Farbwerte nur zwischen 0 und 1 bewegen. Dies erleichtert dem CNN die Arbeit.

Kommen wir nun zum größten Abschnitt des Programms innerhalb Codezelle 2, der Erstellung der Modellstruktur. Hier bietet Keras zwei Möglichkeiten an, die Modellstruktur **Sequential** oder die Modellstruktur mittels der funktionellen **API**. Zeile 25 zeigt, dass für unser Bilderkennungsprogramm die Modellstruktur *Sequential* gewählt wurde. Von Zeile 26 bis 56 fügen wir unserem Modell aus der Keras-Bibliothek mehrere, sich wiederholende Schichten hinzu.

In Zeile 26 taucht zum ersten Mal die ganz entscheidende Funktion **Conv2D()** auf, welche einen Convolutional Layer darstellt, der mit zweidimensionalen Bilddaten arbeitet; mit vielen Argumenten innerhalb der Klammer.

```
model.add(keras.layers.Conv2D(filters = 16, kernel_size = (3, 3), padding = 'same', input_shape=(28, 28, 1,)))
```

Das Argument **kernel_size = (3, 3)** besagt, dass ein 3x3 Pixel großes Fenster (**Kernel**) mit seinen Filterwerten pixelweise über den vorhergehenden Layer gleitet. Bei der Filterung werden bestimmte Merkmale und Muster herausgearbeitet. **filters = 16** bedeutet, dass 16 Kernels die Eingabewerte filtern sollen, dass pro Pixel also 16 Filter angewendet werden. **padding = 'same'** sorgt dafür, dass das 3x3 Pixel große Fenster auch über den Rand des 28x28 Pixel großen Bildes hinausragen darf (siehe Abb. 5.4). Eine gute Simulation, ähnlich der Abbildung 5.4, findet man unter dem folgenden Link.

https://de.wikipedia.org/wiki/Convolutional_Neural_Network

Das Argument **input_shape=(28, 28, 1,)** taucht nur in Zeile 26 auf. Hier erhält unser Modell die Information, dass Bilder in der Größe von 28x28 Pixel mit nur einem Farbkanal zu erwarten sind.

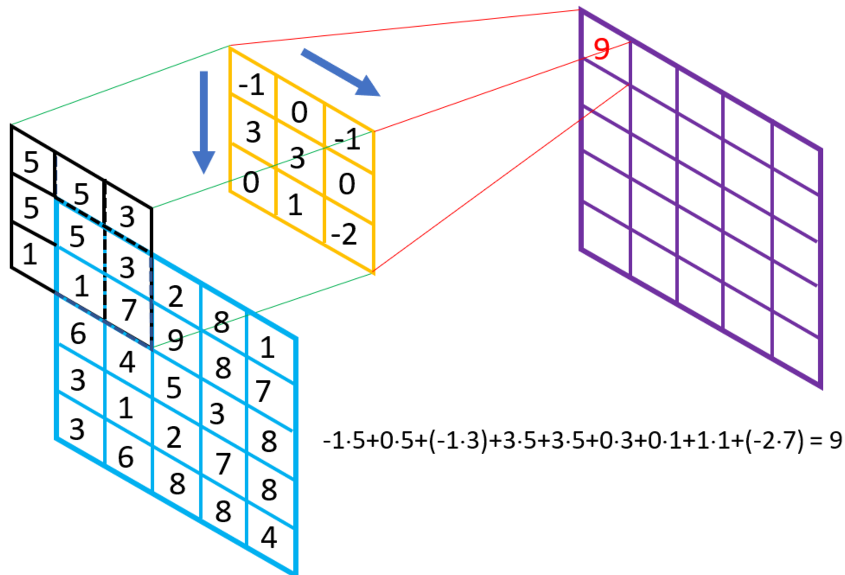

Abbildung 5.4: Ein Filter (Kernel) gleitet über einen Layer

In Zeile 27 sehen wir, dass als Aktivierungsfunktion die **PReLU-Funktion** benutzt wird. Diese wurde schon im Kapitel 2 vorgestellt. Die Zweierkombination Conv2D und PReLu wird mit steigender Filterzahl ($2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$) im Programm wiederholt, um ein möglichst gutes Endergebnis zu erzielen.

In etwas größeren Abständen erscheint die Funktion **MaxPooling2D()** mit ihren Argumenten, welche eine MaxPooling-Schicht repräsentiert.

```
model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
```

Was bewirkt eine MaxPooling-Schicht? Auch sie tastet den vorausgehenden Layer mit einem Filter ab. Dieser Filter, der sich in Zweierschritten über den Layer schiebt, hat in unserem Fall die Größe 2x2 = 4 Pixel **(pool_size = (2, 2))**. Von den vier Werten, die der Filter ermittelt, gibt er nur den größten und damit wichtigsten Wert weiter. Dadurch reduziert sich auch die Datenmenge und damit die Rechenzeit für das CNN (Abb. 5.5).

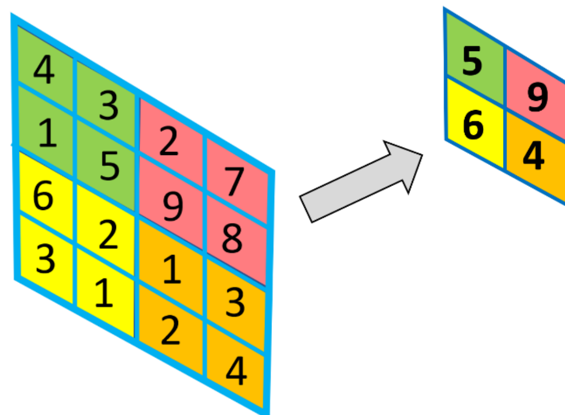

Abbildung 5.5: MaxPooling mit einem 2×2-Filter und der Schrittgröße = 2.
Die Schrittgröße gibt an, um wie viele Pixel der Filter pro Operation verschoben wird.

Am Ende der Codeblocks, in Zeile 56 von Codezelle 2, steht:

```
model.add(keras.layers.Dense(10, activation='softmax'))
```

Hier ermittelt ein **Dense-Layer** aus den Zahlenwerten, die der vorgeschaltete Layer liefert, mit Hilfe der Aktivierungsfunktionen ***softmax*** die Wahrscheinlichkeit für die 10 Zielklassen (Zahlen 0 bis 9). Also für jede der 10 Zahlen die Wahrscheinlichkeit, wie sicher sich das CNN ist, ob diese Zahl im Bild vorhanden ist. Die Summe dieser Wahrscheinlichkeiten ist 1.

Nachdem das Modell erstellt ist, kann es kompiliert werden. Hierbei kann die Art der Verlustfunktion (loss), die Art des Optimierers (optimizer) und die Art der Metrik-Funktion (metrics) gewählt werden. Mit der Loss- und der Metrik-Funktion wird die Leistung des im Programm erstellten Modells bewertet. In Zeile 59 steht:

```
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=
    ['sparse_categorical_accuracy'])
```

Als Verlustfunktion wurde hier die Kreuzentropie gewählt, die den Unterschied zwischen den von dem Netzwerk vorhergesagten Wahrscheinlichkeiten und den wirklichen Werten für die einzelnen der 10 Klassen ermittelt. Diese Werte werden an den Optimierer übergeben.

Nach der Kompilierung kann das Modell gefittet werden (Zeile 61). Hierbei werden die vom Modell gelieferten Daten mit den Testdaten verglichen und das Modell an die Daten angepasst. Bei diesem Vergleich wird das Modell in dem vorliegenden Programm 10mal dem Trainingsdatensatz ausgesetzt (**epochs = 10**). **batch_size = 256** besagt, dass dem Modell immer 256 Trainingsbeispiele gleichzeitig gezeigt werden.

Wenn wir nun den Trainingsvorgang starten, dann sehen wir (Abb. 5.6), dass der Fehler immer kleiner wird und die Genauigkeit auf über 0,99, also über 99% steigt. Das Modell zur Zahlenerkennung funktioniert also scheinbar sehr gut. Scheinbar deshalb, weil hohe Prozentwerte auch durch Überanpassung entstehen können. D.h., das Modell hat sich hervorragend an die Trainingsdaten angepasst und dabei die Eigenschaft verloren, auch gänzlich unbekannte Zahlen zu erkennen.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 27s 446us/step - loss: 0.3617 - sparse_categorical_accuracy: 0.8739 - val_loss: 0.0584 - val_sparse_categorical_accuracy: 0.9827
Epoch 2/10
60000/60000 [==============================] - 18s 304us/step - loss: 0.0631 - sparse_categorical_accuracy: 0.9804 - val_loss: 0.0485 - val_sparse_categorical_accuracy: 0.9855
Epoch 3/10
60000/60000 [==============================] - 18s 300us/step - loss: 0.0423 - sparse_categorical_accuracy: 0.9873 - val_loss: 0.0452 - val_sparse_categorical_accuracy: 0.9870
Epoch 4/10
60000/60000 [==============================] - 18s 301us/step - loss: 0.0320 - sparse_categorical_accuracy: 0.9900 - val_loss: 0.0348 - val_sparse_categorical_accuracy: 0.9893
Epoch 5/10
60000/60000 [==============================] - 18s 301us/step - loss: 0.0259 - sparse_categorical_accuracy: 0.9920 - val_loss: 0.0354 - val_sparse_categorical_accuracy: 0.9894
Epoch 6/10
60000/60000 [==============================] - 18s 301us/step - loss: 0.0219 - sparse_categorical_accuracy: 0.9931 - val_loss: 0.0360 - val_sparse_categorical_accuracy: 0.9890
Epoch 7/10
60000/60000 [==============================] - 18s 302us/step - loss: 0.0209 - sparse_categorical_accuracy: 0.9934 - val_loss: 0.0239 - val_sparse_categorical_accuracy: 0.9913
Epoch 8/10
60000/60000 [==============================] - 18s 301us/step - loss: 0.0185 - sparse_categorical_accuracy: 0.9943 - val_loss: 0.0338 - val_sparse_categorical_accuracy: 0.9910
Epoch 9/10
60000/60000 [==============================] - 18s 302us/step - loss: 0.0157 - sparse_categorical_accuracy: 0.9948 - val_loss: 0.0305 - val_sparse_categorical_accuracy: 0.9912
Epoch 10/10
60000/60000 [==============================] - 18s 302us/step - loss: 0.0161 - sparse_categorical_accuracy: 0.9951 - val_loss: 0.0329 - val_sparse_categorical_accuracy: 0.9909
<keras.callbacks.History at 0x7fd21ad04a20>
```

Abbildung 5.6: Das erstellte Modell wird mit den Testdaten trainiert.
Nach 10 Epochen erreicht es eine Genauigkeit von 99,09%.

Testen wir also das Modell auch mit unseren selbsterstellten Bildern. Dazu benutzen wir den Code von Codezelle 3 (s.u.). Wir importieren mit Hilfe der Python-Bibliothek **PIL** aus unserem Google-Drive-Ordner eines von unseren Bildern. Zum Beispiel das Bild von der Zahl 4 (Zeile 3, Zahl4.png). Nun lassen wir das Modell eine Vorhersage erstellen, welche Zahl in dem Bild vorhanden ist. Mittels der Funktion **print** wird die Zahl angezeigt, für die unser Modell die größte Wahrscheinlichkeit ermittelt hat. Erfreulicherweise erhalten wir bei allen Zahlen ein richtiges Ergebnis.

```
 1  # Bild importieren
 2  from PIL import Image
 3  image_path='/content/drive/My Drive/Bilder/Zahlen_28x28/Zahl4.png'
 4  image = Image.open(image_path, 'r')
 5  image = image.resize((28, 28), Image.ANTIALIAS)
 6  pixels = list(image.getdata(0))
 7  np_pixels = np.asarray(pixels, dtype = np.float32).reshape((1, 28, 28, 1))
 8  np_pixels /= 255
 9
10  # Klasse voraussagen und ausgeben
11  predicted_class = np.argmax(model.predict_on_batch(np_pixels))
12  print(predicted_class)
```

Auf Seite 24 wurde die Modellstruktur für das Programm zur Zahlenerkennung Schritt für Schritt aufgelistet. Mit einer doppelten for-Schleife kann man sich, wenn man möchte, bei diesem Programmteil einiges an Schreibarbeit sparen (s.u.).

```
# Modelstruktur definieren
model = keras.models.Sequential()
model.add(keras.layers.Input(shape=(28, 28, 1,)))
for i in range(4):
  for j in range(3):
    model.add(keras.layers.Conv2D(filters = 16 * 2**i, kernel_size = (3, 3), padding = 'same'))
    model.add(keras.layers.PReLU())
  model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(50))
model.add(keras.layers.PReLU())
model.add(keras.layers.Dense(10, activation='softmax'))
```

Nun wird es Zeit, mal wieder einige Blicke ins Internet zu werfen, um das Gelernte zu wiederholen und zu vertiefen. Als Erstes solltet man sich das Video von Johannes Wenisch ansehen. Johannes Wenisch erklärt hier am Beispiel der Zahlenerkennung sehr ausführlich und anschaulich, wie ein Convolutional Neural Network (CNN) funktioniert. Hier ist der Link:

https://www.youtube.com/watch?v=Kd5jCvXvAu8

Weiterhin ist der Artikel *Einstieg in Convolutional Neuronale Netze mit Keras* von Johannes Höhne zur Vertiefung des Gelernten sehr gut geeignet.

https://www.mt-ag.com/einstieg-in-convolutional-neuronale-netze-mit-keras/

Und Irhum Shafkat erklärt anhand von zahlreichen animierten Grafiken in seinem Artikel *Intuitively Understanding Convolutions for Deep Learning* sehr anschaulich die Funktionen von Convolutional Neural Networks.

https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

5.4 Tiere erkennen

Als weitere Übung wollen wir unser Programm zur Zahlenerkennung nun auch für die Erkennung von Tieren benutzen. Dazu benötigen wird natürlich einen entsprechend großen Datensatz mit passenden Trainings- und Testbildern. Hierzu verwenden wir den Datensatz **CIFAR-10**, der auch in Keras enthalten ist. Der CIFAR-10-Datensatz besteht aus 60000 32x32-Farbbildern (50000 Trainingsbilder und 10000 Testbilder), die in 10 Klassen aufgeteilt sind (Abb. 5.7). Infos zu dem CIFAR-10 Datensatz findet man unter: https://www.cs.toronto.edu/~kriz/cifar.html

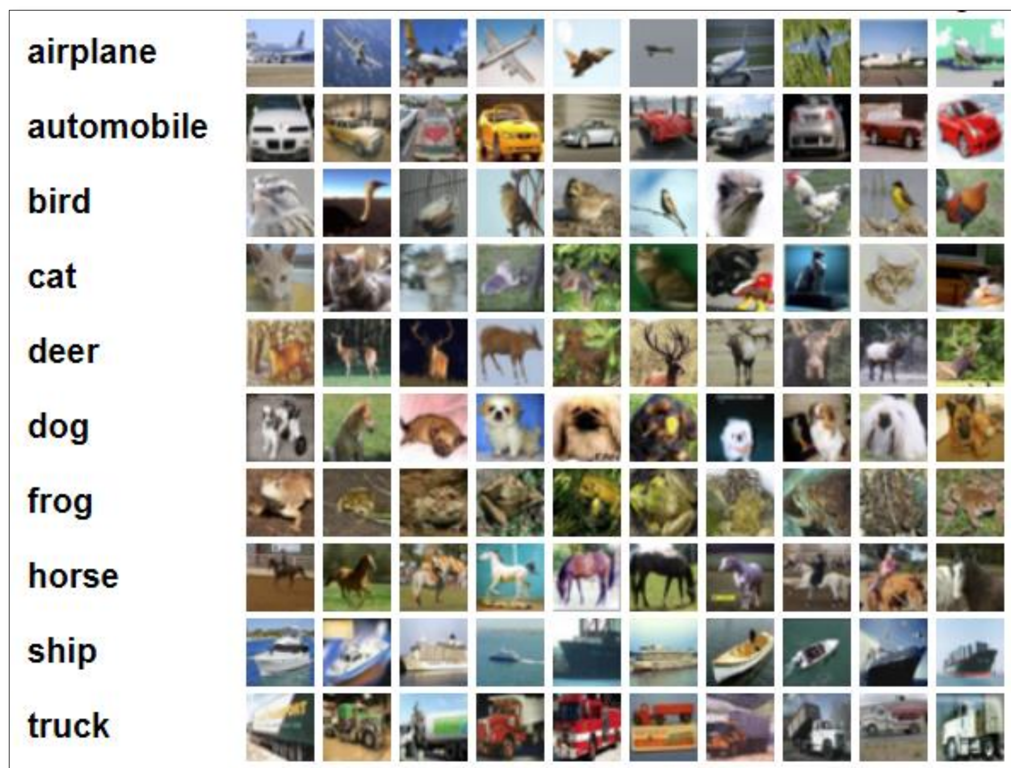

Abbildung 5.7: Bildbeispiele für die 10 Klassen des Datensatzes CIFAR-10
Quelle: https://www.cs.toronto.edu/~kriz/cifar.html
Alex Krizhevsky, 2009

Welche Änderungen müssen wir nun in dem Programm zur Zahlenerkennung vornehmen, damit es sich für die Erkennung unserer Tierbilder (Abb. 5.10 und 5.11) eignet? In der ersten Codezelle ändert sich nichts. Hier wird wieder Google Drive importiert.

```
1  # Google Drive laden, muss nur einmal am Anfang ausgeführt werden
2  from google.colab import drive
3  drive.mount('/content/drive')
```

In der zweiten Codezelle, also im Hauptteil des neuen Programms (s.u.), werden gegenüber dem Programm zur Zahlenerkennung jedoch einige Änderungen vorgenommen. Im zweiten Codefenster wird in Zeile 9 anstelle von *mnist* das Datenset *cifar10* importiert. Da es sich bei den Bildern um Farbbilder handelt, steht in den Zeilen 17, 18 und 24 statt einer 1 eine 3, da Farbbilder drei Farbkanäle (RGB) besitzen. Des Weiteren wird die Anzahl der Filter auf 256 erhöht (Zeile 45 – 49), um die Genauigkeit des Modells zu verbessern.

```
1  # Die neueste Version von TensorFlow laden
2  try:
3     %tensorflow_version 2.x
4  except Exception:
5     pass
6
7  # Bibliotheken importieren
8  from tensorflow import keras
9  from keras.datasets import cifar10
10 import numpy as np
11
12 # Datenbank mit Trainings- und Test-Datenset importieren
13 (x_train, y_train), (x_test, y_test) = cifar10.load_data()
14 # Daten in das richtige Format bringen (float32, 32 * 32 Pixel in XY Format, 0 bis 1, 3 Farbkanal)
15 x_train = x_train.astype('float32')
16 x_test = x_test.astype('float32')
17 x_train = x_train.reshape((-1, 32, 32, 3))
18 x_test = x_test.reshape((-1, 32, 32, 3)) # -1 repräsentiert die Anzahl der Bilder im Set
19 x_train /= 255
20 x_test /= 255
21
22 # Modelstruktur definieren
23 model = keras.models.Sequential()
24 model.add(keras.layers.Conv2D(filters = 32, kernel_size = (3, 3), padding = 'same', input_shape=(32, 32, 3,)))
25 model.add(keras.layers.PReLU())
26 model.add(keras.layers.Conv2D(filters = 32, kernel_size = (3, 3), padding = 'same'))
27 model.add(keras.layers.PReLU())
28 model.add(keras.layers.Conv2D(filters = 32, kernel_size = (3, 3), padding = 'same'))
29 model.add(keras.layers.PReLU())
30 model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
31 model.add(keras.layers.Conv2D(filters = 64, kernel_size = (3, 3), padding = 'same'))
32 model.add(keras.layers.PReLU())
33 model.add(keras.layers.Conv2D(filters = 64, kernel_size = (3, 3), padding = 'same'))
34 model.add(keras.layers.PReLU())
35 model.add(keras.layers.Conv2D(filters = 64, kernel_size = (3, 3), padding = 'same'))
36 model.add(keras.layers.PReLU())
37 model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
38 model.add(keras.layers.Conv2D(filters = 128, kernel_size = (3, 3), padding = 'same'))
39 model.add(keras.layers.PReLU())
40 model.add(keras.layers.Conv2D(filters = 128, kernel_size = (3, 3), padding = 'same'))
41 model.add(keras.layers.PReLU())
42 model.add(keras.layers.Conv2D(filters = 128, kernel_size = (3, 3), padding = 'same'))
43 model.add(keras.layers.PReLU())
44 model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
45 model.add(keras.layers.Conv2D(filters = 256, kernel_size = (3, 3), padding = 'same'))
46 model.add(keras.layers.PReLU())
47 model.add(keras.layers.Conv2D(filters = 256, kernel_size = (3, 3), padding = 'same'))
48 model.add(keras.layers.PReLU())
49 model.add(keras.layers.Conv2D(filters = 256, kernel_size = (3, 3), padding = 'same'))
50 model.add(keras.layers.PReLU())
51 model.add(keras.layers.Flatten())
52 model.add(keras.layers.Dense(50))
53 model.add(keras.layers.PReLU())
54 model.add(keras.layers.Dense(10, activation='softmax')) # Letzte Schicht des Netzwerkes. Sie gibt
55 # die Klasse an
56
57 # Parameter für die Optimierung definieren: Fehler-Funktion und Optimier-Algorithmus mit eine
58 model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['sparse_categorical_accuracy'])
59
60 # Model an die Daten anpassen bzw. trainieren
61 model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size = 256, epochs = 10)
```

Testen wir nun das neue Programm. Nach 10 Durchläufen erhalten wir eine Genauigkeit von 72,49% (Abb. 5.8).

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 30s 608us/sample - loss: 2.1017 - sparse_categorical_accuracy: 0.1906 - val_loss: 1.7285 - val_sparse_categorical_accuracy: 0.3565
Epoch 2/10
50000/50000 [==============================] - 28s 561us/sample - loss: 1.6053 - sparse_categorical_accuracy: 0.3992 - val_loss: 1.6448 - val_sparse_categorical_accuracy: 0.4077
Epoch 3/10
50000/50000 [==============================] - 28s 562us/sample - loss: 1.3568 - sparse_categorical_accuracy: 0.5018 - val_loss: 1.3801 - val_sparse_categorical_accuracy: 0.4947
Epoch 4/10
50000/50000 [==============================] - 28s 562us/sample - loss: 1.1796 - sparse_categorical_accuracy: 0.5704 - val_loss: 1.1662 - val_sparse_categorical_accuracy: 0.5870
Epoch 5/10
50000/50000 [==============================] - 28s 562us/sample - loss: 1.0289 - sparse_categorical_accuracy: 0.6293 - val_loss: 1.0214 - val_sparse_categorical_accuracy: 0.6322
Epoch 6/10
50000/50000 [==============================] - 28s 562us/sample - loss: 0.9029 - sparse_categorical_accuracy: 0.6792 - val_loss: 0.9353 - val_sparse_categorical_accuracy: 0.6733
Epoch 7/10
50000/50000 [==============================] - 28s 563us/sample - loss: 0.7976 - sparse_categorical_accuracy: 0.7159 - val_loss: 0.9043 - val_sparse_categorical_accuracy: 0.6858
Epoch 8/10
50000/50000 [==============================] - 28s 562us/sample - loss: 0.7074 - sparse_categorical_accuracy: 0.7501 - val_loss: 0.8601 - val_sparse_categorical_accuracy: 0.7048
Epoch 9/10
50000/50000 [==============================] - 28s 563us/sample - loss: 0.6065 - sparse_categorical_accuracy: 0.7869 - val_loss: 0.8511 - val_sparse_categorical_accuracy: 0.7089
Epoch 10/10
50000/50000 [==============================] - 28s 563us/sample - loss: 0.5325 - sparse_categorical_accuracy: 0.8134 - val_loss: 0.8578 - val_sparse_categorical_accuracy: 0.7249
<tensorflow.python.keras.callbacks.History at 0x7f55618f5cc0>
```

Abbildung 5.8: 10 Durchläufe des neuen Programms

Lässt sich die Erkennungsgenauigkeit von 72,49 % nicht noch übertreffen, wenn man die Anzahl der Epochen zum Beispiel von 10 auf 15 erhöht? Probieren wir es aus.

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/15
50000/50000 [==============================] - 35s 696us/sample - loss: 2.0562 - sparse_categorical_accuracy: 0.2222 - val_loss: 1.6978 - val_sparse_categorical_accuracy: 0.3721
Epoch 2/15
50000/50000 [==============================] - 28s 559us/sample - loss: 1.5349 - sparse_categorical_accuracy: 0.4288 - val_loss: 1.4094 - val_sparse_categorical_accuracy: 0.4760
Epoch 3/15
50000/50000 [==============================] - 28s 560us/sample - loss: 1.2493 - sparse_categorical_accuracy: 0.5419 - val_loss: 1.2208 - val_sparse_categorical_accuracy: 0.5458
Epoch 4/15
50000/50000 [==============================] - 28s 561us/sample - loss: 1.0709 - sparse_categorical_accuracy: 0.6117 - val_loss: 1.0398 - val_sparse_categorical_accuracy: 0.6354
Epoch 5/15
50000/50000 [==============================] - 28s 560us/sample - loss: 0.9259 - sparse_categorical_accuracy: 0.6686 - val_loss: 1.0034 - val_sparse_categorical_accuracy: 0.6519
Epoch 6/15
50000/50000 [==============================] - 28s 561us/sample - loss: 0.8146 - sparse_categorical_accuracy: 0.7112 - val_loss: 0.9271 - val_sparse_categorical_accuracy: 0.6807
Epoch 7/15
50000/50000 [==============================] - 28s 561us/sample - loss: 0.7254 - sparse_categorical_accuracy: 0.7447 - val_loss: 0.9006 - val_sparse_categorical_accuracy: 0.6887
Epoch 8/15
50000/50000 [==============================] - 28s 560us/sample - loss: 0.6374 - sparse_categorical_accuracy: 0.7745 - val_loss: 0.8703 - val_sparse_categorical_accuracy: 0.7020
Epoch 9/15
50000/50000 [==============================] - 28s 559us/sample - loss: 0.5490 - sparse_categorical_accuracy: 0.8072 - val_loss: 0.8600 - val_sparse_categorical_accuracy: 0.7132
Epoch 10/15
50000/50000 [==============================] - 28s 558us/sample - loss: 0.4745 - sparse_categorical_accuracy: 0.8336 - val_loss: 0.8836 - val_sparse_categorical_accuracy: 0.7195
Epoch 11/15
50000/50000 [==============================] - 28s 558us/sample - loss: 0.4128 - sparse_categorical_accuracy: 0.8566 - val_loss: 0.9452 - val_sparse_categorical_accuracy: 0.7181
Epoch 12/15
50000/50000 [==============================] - 28s 558us/sample - loss: 0.3555 - sparse_categorical_accuracy: 0.8740 - val_loss: 0.9366 - val_sparse_categorical_accuracy: 0.7320
Epoch 13/15
50000/50000 [==============================] - 28s 558us/sample - loss: 0.3012 - sparse_categorical_accuracy: 0.8945 - val_loss: 0.9998 - val_sparse_categorical_accuracy: 0.7200
Epoch 14/15
50000/50000 [==============================] - 28s 559us/sample - loss: 0.2639 - sparse_categorical_accuracy: 0.9076 - val_loss: 1.0203 - val_sparse_categorical_accuracy: 0.7211
Epoch 15/15
50000/50000 [==============================] - 28s 563us/sample - loss: 0.2145 - sparse_categorical_accuracy: 0.9255 - val_loss: 0.9983 - val_sparse_categorical_accuracy: 0.7215
<tensorflow.python.keras.callbacks.History at 0x7f55b00a5518>
```

Abbildung 5.9: 15 Durchläufe des neuen Programms

Wie man der Abbildung 5.9 entnehmen kann, hat sich die Genauigkeit nach 15 Epochen nicht erhöht, sondern ist mit 72,15 % ungefähr gleichgeblieben. Vergrößert hat sich aber der Wert von *sparse_categorical_accuracy* von 81,34 % auf 92,55 %. Dies bedeutet, dass unser NN bei 15 Durchläufen für die Trainingsdaten eine bessere Genauigkeit erzielt hat als bei 10 Durchläufen, aber die Erkennungsgenauigkeit für die Testdatenmenge hat sich nicht erhöht. Sie beträgt nicht 92,55 %, sondern nur 72,15 %. Man spricht in einem solchen Fall von Überanpassung. Belassen wir es also bei 10 Epochen und laden nun nacheinander unsere eigenen Bilder hoch (Codefenster 3 siehe folgende Seite). Einmal testen wir unser NN nur mit Katzen (Abb. 5.10) und ein weiteres Mal mit Katze, Hund, Pferd und Rehbock (Abb. 5.11).

```python
# Bild importieren
from PIL import Image
image_path='/content/drive/My Drive/Bilder/Tiere_32x32/Katze_02.jpg'
image = Image.open(image_path, 'r')
image = image.resize((32, 32), Image.ANTIALIAS)
pixels = list(image.getdata())
np_pixels = np.asarray(pixels, dtype = np.float32).reshape((1, 32, 32, 3))
np_pixels /= 255

# Klasse voraussagen und ausgeben
labels = []
labels.append("airplane")
labels.append("automobile")
labels.append("bird")
labels.append("cat")
labels.append("deer")
labels.append("dog")
labels.append("frog")
labels.append("horse")
labels.append("ship")
labels.append("truck")

predicted_class = labels[np.argmax(model.predict_on_batch(np_pixels))]
print(predicted_class)
```

Abbildung 5.10: Katzenbilder

Das CNN liefert uns zu unseren Katzenbildern die folgenden Angaben.

cat cat cat truck

Die ersten drei Katzen werden also richtig erkannt. Bei der liegenden Katze liefert das CNN jedoch eine falsche Aussage.

Bei den folgenden vier Bildern werden alle Tiere richtig erkannt.

Katze_02 Hund_01 Pferd_01 Rehbock_01

Abbildung 5.11: Bilder von unterschiedlichen Tieren

cat dog horse deer

Dies sind doch schon ganz gute Ergebnisse. Natürlich kann man durch gezielte Anpassung des CNNs noch bessere Ergebnisse erzielen, aber wer mehr erwartet hat, dem sei zum Trost gesagt, dass auch die KIs von Profis keine 100%ige Vorhersagegenauigkeit erreichen. Davon kann man sich anhand des folgenden Links überzeugen. Wenn man den Link in seinem Browser aufruft, dann versucht ein Convolutional Neural Network der Universität Stanford vorgegebene Objekte zu erkennen. Als Datensatz dient hier auch CIFAR-10. Die Rechnungen laufen mittels Java-Script live im eigenen Browser ab (siehe Abb. 5.12). Hier ist der Link: http://cs231n.stanford.edu/

Abbildung 5.12: Eine Vorhersage der CNNs der Universität Stanford live im Browser

Zum Schluss des Kapitels ein sehr, sehr langer Link. Er erlaubt es, im Browserfenster mit Layers, Neuronen usw. zu „spielen“ (siehe Abb. 5.13). Dies ist nützlich zur Vertiefung des bisher Gelernten.

http://playground.tensorflow.org/#activation=sigmoid®ularization=L2&batchSize=10&dataset=gauss®Dataset=reg-plane&learningRate=0.0001®ularizationRate=0&noise=0&networkShape=&seed=0.55381&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false&showTestData_hide=false&learningRate_hide=true®ularizationRate_hide=true&percTrainData_hide=true&numHiddenLayers_hide=true&discretize_hide=true&activation_hide=false&problem_hide=true&noise_hide=true®ularization_hide=true&dataset_hide=true&batchSize_hide=true&playButton_hide=false

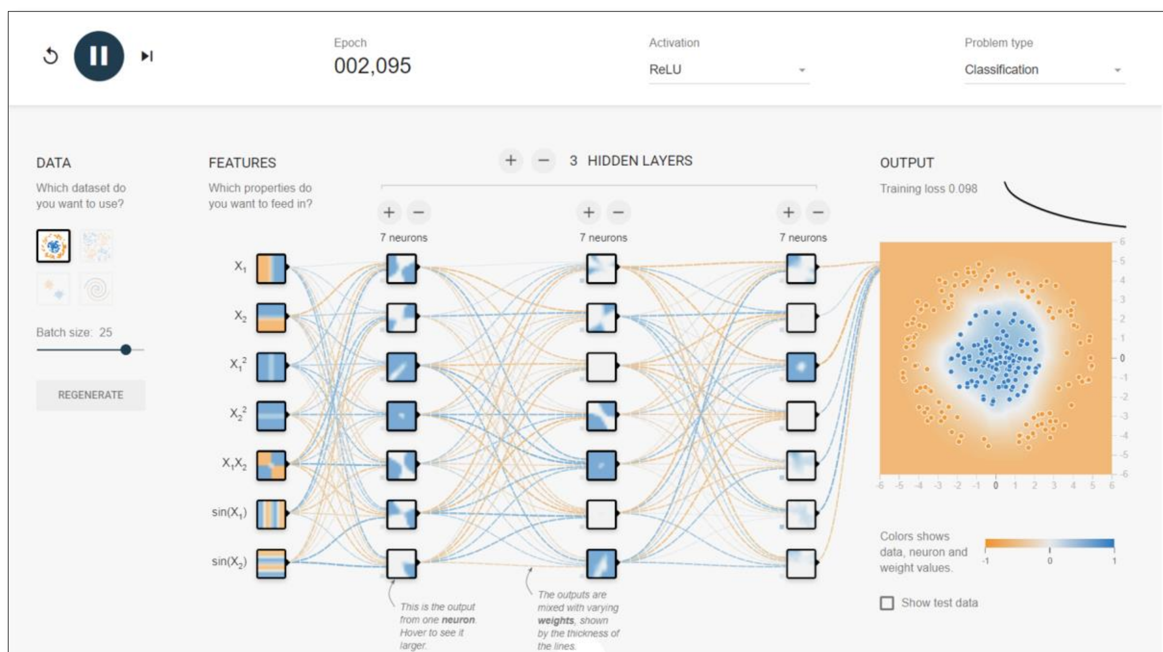

Abbildung 5.13: "Spielereien" im Browser mit einem CNN

5.5 Kleidungsstücke erkennen

Nun wird es aber Zeit für eine eigene Übung. Keras enthält außer den beiden bisher verwendeten Datensätzen noch weitere Datensätze. Diese findet man unter: https://keras.io/datasets/ . Für die eigene Übung soll der Datensatz ***Fashion-MNIST*** verwendet werden. Dieser Datensatz besteht aus 60000 Trainingsbildern und 10000 Testbildern von verschiedenen Kleidungsstücken, die in 10 Klassen aufgeteilt sind (siehe Abb. 5.14) Diesen Datensatz importiert man wie folgt: *from keras.datasets import fashion_mnist*.

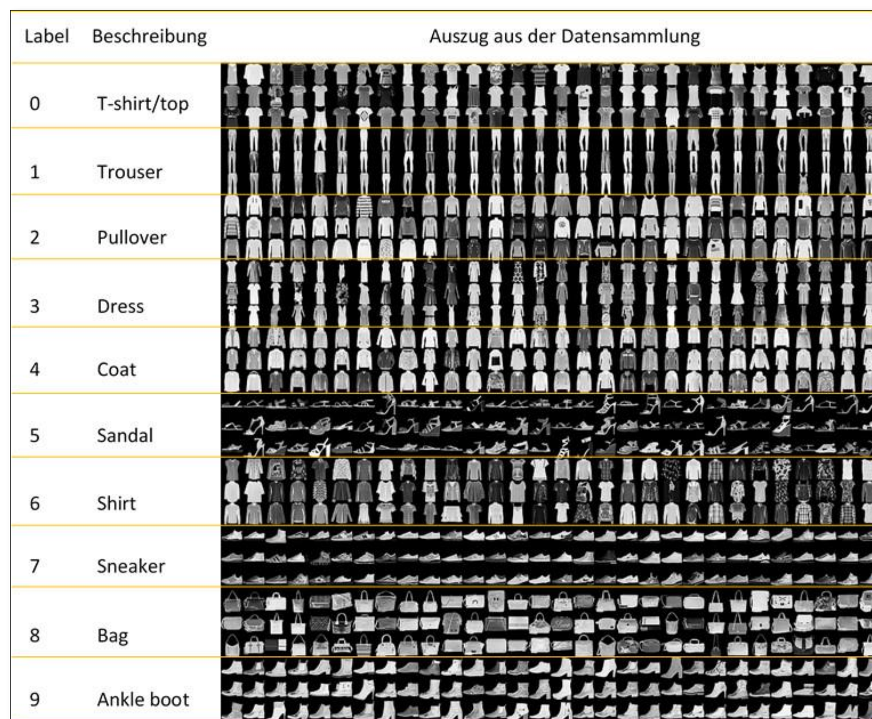

Abbildung 5.14: Auszug aus der Fashion-MNIST-Datensammlung
Quelle: https://github.com/zalandoresearch/fashion-mnist

Hier ein Tipp zu dieser Aufgabe. Ich habe vier Kleidungsstücke vor einem weißen Hintergrund fotografiert, diese Aufnahmen zuerst in Graustufenbilder und anschließend in Negativbilder umgewandelt (Abb. 5.15 obere Reihe). Danach habe ich ihre Größe auf 28x28 Pixel reduziert (Abb. 5.15 untere Reihe) und sie in Google Drive hochgeladen. Alle vier Kleidungsstücke wurden von meinem trainierten CNN ohne Probleme erkannt. Bei weißem Hintergrund war die Erkennungsrate deutlich schlechter.

Abbildung 5.15: Eigene Fotos von Kleidungsstücken

5.6 Weitere Datensammlungen

Neben den bisher verwendeten Datensammlungen gibt es auch weitere Datensammlungen, die man als Lehrerinn oder Lehrer unter den Bedingungen der Princeton University und der Stanford University kostenlos herunterladen kann. Hier ist der entsprechende Link.

http://www.image-net.org

Informationen zu ImageNet findet man auch unter:

https://de.wikipedia.org/wiki/ImageNet

Viele Schülerinnen und Schüler werden an einer Schule unterrichtet. Kann man mit ihrer Hilfe auch eine eigene Datensammlung für ein Neuronales Netz erstellen? Lena Maxine Lenkeit hat hierzu ein Programm geschrieben (siehe folgende Seite). Dieses Programm haben wir mit jeweils 116 Fotos von Ananas, Apfelsinen und Bananen getestet (Abb. 5.16). Obwohl das Programm nur über sehr wenige Trainings- und Testdaten verfügte, wurden die drei Obstsorten stets richtig erkannt. 116 Fotos pro Objekt sind für ein funktionierendes NN eigentlich zu wenig, doch Lenas Programm wendet einen Trick an. Die Bilder werden automatisch mit Hilfe von Keras importiert (Codezelle 2, Zeile 12). Dies erlaubt es, das Datenset durch automatische Bildbearbeitung zu vergrößern. Aus den importierten Bildern werden so durch Drehen, Verzerren, Spiegeln, etc. neue Bilder generiert, die auch zum Training benutzt werden, was das Problem kleiner Datenmengen löst (Codezelle 2, Zeile 14-16). All dies geschieht, wie auch die Aufteilung in Trainings- und Testdaten, automatisch durch Keras. Selbst muss man nur den Hauptordner angeben, in dem sich die Unterordner mit den verschiedenen Obstbildern befinden. Nicht nur mit Obst, sondern auch mit Gemüse (jeweils 100 Gurken, Pilze und Tomaten) sowie mit Besteck (jeweils 100 Gabeln, Löffel und Messer) wurde Lenas Programm erfolgreich getestet.

Wenn viele fleißige Schülerinnen und Schüler bei der Erstellung eines Datensets helfen, dann verfügt man schnell als Lehrerin und Lehrer über ein Datenset, mit dem eine sinnvolle Bilderkennung möglich ist.

Abbildung 5.16: Drei Objekte aus der Gruppe „Obst“ für ein eigenes Neuronales Netz

Um das Programm auf der folgenden Seite zu nutzen, muss man für die Trainings- und Testdaten einen Ordner *Obst_32x32* mit den Unterordnern *Ananas*, *Apfelsinen* und *Bananen* erstellen. Jeden dieser Ordner füllt man dann mit einer ausreichenden Anzahl von 32x32 Pixel großen, selbst-erstellten Bildern. Für den Ordner *Obst_02* in Codezelle 3, mit seinen Unterordner *Ananas*, *Apfel-sinen* und *Bananen*, benötigt an nur wenige Bilder (32x32 Pixel). Da diese Bilder nicht Bestandteil

des Trainings- und Testdatensets sind, kann mit ihnen die Güte der Aussagewahrscheinlichkeit des Neuronalen Netzes überprüft werden.

In Zeile 12 von Codezelle 2 werden die Bilder (32x32 Pixel) von Ananas, Apfelsinen und Bananen aus den Unterordnern von Ordner *Obst_32x32* importiert, um sie als Trainings- und Testdaten zu nutzen. In Zeile 4 von Codezelle 3 ruft man aus den Unterordnern von Ordner *Obst_02* einzeln die Bilder (32x32 Pixel) von Ananas, Apfelsinen und Bananen auf, um so das NN zu überprüfen.

Codezelle 1

```
1  # Google Drive laden, muss nur einmal am Anfang ausgeführt werden
2  from google.colab import drive
3  drive.mount('/content/drive')
```

Codezelle 2

```
1   # Die neueste Version von TensorFlow laden
2   try:
3     %tensorflow_version 2.x
4   except Exception:
5     pass
6
7   # Bibliotheken importieren
8   from tensorflow import keras
9   import numpy as np
10
11  # Datenbank mit Trainings- und Test-Datenset mittels Keras importieren (10% der Bilder werden als Test-Datenset verwendet)
12  image_main_folder='/content/drive/My Drive/Bilder/Obst_32x32'
13
14  datagen = keras.preprocessing.image.ImageDataGenerator(rotation_range=180, width_shift_range=0.25, height_shift_range=0.25,
15                                                         shear_range=0.25, zoom_range=0.25, horizontal_flip = True, vertical_flip = True,
16                                                         rescale = 1.0/255.0, validation_split=0.1, fill_mode='nearest')
17  train_generator = datagen.flow_from_directory(image_main_folder, target_size = (32, 32), class_mode='sparse', interpolation='bilinear', subset='training')
18  test_generator = datagen.flow_from_directory(image_main_folder, target_size = (32, 32), class_mode='sparse', interpolation='bilinear', subset='validation')
19
20  num_classes = len(train_generator.class_indices) # Anzahl der Klassen
21
22  # Modelstruktur definieren
23  model = keras.models.Sequential()
24  model.add(keras.layers.Input(shape=(32, 32, 3,)))
25  for i in range(4):
26    for j in range(2):
27      model.add(keras.layers.Conv2D(filters = 64 * 2**i, kernel_size = (3, 3), activation = 'elu', padding = 'same'))
28    model.add(keras.layers.MaxPooling2D(pool_size = (2, 2), padding = 'same'))
29  model.add(keras.layers.Flatten())
30  for i in range(3):
31    model.add(keras.layers.Dense(50, activation = 'elu'))
32  model.add(keras.layers.Dense(num_classes, activation = 'softmax')) # Letzte Schicht des Netzwerkes, gibt die Klasse an
33
34  # Parameter für die Optimierung definieren: Fehler-Funktion und Optimier-Algorithmus mit einer Metrik
35  model.compile(loss='sparse_categorical_crossentropy',optimizer='sgd',metrics=['sparse_categorical_accuracy'])
36  # Model an die Daten anpassen bzw. trainieren
37  model.fit_generator(train_generator, steps_per_epoch=256, epochs = 5, validation_data=test_generator, validation_steps=4)
```

Codezelle 3

```
1   from PIL import Image
2
3   # Bild importieren
4   image_path='/content/drive/My Drive/Bilder/Obst_02/Apfelsine_02.jpg'
5   image = Image.open(image_path, 'r')
6   image = image.resize((32, 32), Image.ANTIALIAS)
7   pixels = list(image.getdata())
8   image.close()
9   np_pixels = np.asarray(pixels, dtype = np.float32).reshape((1, 32, 32, 3))
10  np_pixels /= 255
11
12  # Klasse voraussagen und ausgeben
13  inverted_dict = {v: k for k, v in train_generator.class_indices.items()}
14  predicted_class = inverted_dict[np.argmax(model.predict_on_batch(np_pixels))]
15  print(predicted_class)
```

6 Unterrichtsmaterialien und Schülerwettbewerbe

In meinem kleinen Buch hat man nur eine kurze Einführung in die Themen *Künstliche Intelligenz* und *Deep Learning* erhalten. Möchte man als Lehrerin oder Lehrer sich diesen Themen in seinem Unterricht widmen, so benötigt man hierzu passende Unterrichtsmaterialien. Gute und kostenlose Hilfe findet man dazu im Internet. Hier sind zwei entsprechende Links.

https://appcamps.de/unterrichtsmaterial/kuenstliche-intelligenz/

https://ki-macht-schule.de

Besonders interessierte Schülerinnen und Schüler, die schon Programmierkenntnisse besitzen, kann man ermutigen, an entsprechenden Wettbewerben teilzunehmen. So gab es zum Beispiel im Jahre 2019 einen Schülerwettbewerb speziell zum Thema Künstliche Intelligenz. Siehe:

https://bw-ki.de/

Auch auf dem Bundeswettbewerb Jugend forscht 2019 in Chemnitz konnte man sich von Schülerinnen und Schülern beeindruckende Arbeiten zum Thema Künstliche Intelligenz erläutern lassen. Eine leicht verständliche Kurzfassung ihrer Forschungsarbeiten findet man in der Projektdatenbank von **Jugend forscht**.

https://www.jugend-forscht.de/projektdatenbank.html

Gibt man hier in die Suchmaske die folgenden Daten ein (Abb. 6.1), so wird man über die Themen von sechs Jungforscherinnen und Jungforschern informiert, die im Jahre 2019 am Bundeswettbewerb teilgenommen und sich mit dem Thema Künstliche Intelligenz beschäftigt haben. Zwei von ihnen, Tara Moghiseh und Constantin Tilman Schott, wurden sogar mit dem Bundessieg ausgezeichnet.

Jahr	2019 ▼
Bundesland	Alle ▼
Fachgebiet	Alle ▼
Preis	Alle ▼
Stichwort	Künstliche Intelligenz

Abbildung 6.1: Suchmaske Jugend forscht

2019 | Mathematik/Informatik | Hamburg | Felix Petersen

AlgoNet – algorithmische neuronale Netzwerke

2019 | Mathematik/Informatik | Bayern | Thomas Sedlmeyr, Philip Haitzer

AnnI, eine künstliche Intelligenz für jeden

2019 | Arbeitswelt | Rheinland-Pfalz | Tara Moghiseh

CELLnet: automatisierte Leukozytendifferenzierung für die Leukämiediagnostik mit KI

2019 | Mathematik/Informatik | Hessen | Vinh Phuc Tran

Deep Learning trifft AOI: Automatische Optische Inspektion von Leiterplatten

2019 | Mathematik/Informatik | Niedersachsen | Constantin Tilman Schott

Einsatz von Methoden künstlicher Intelligenz in der kephalometrischen Röntgendiagnostik

2019 | Physik | Nordrhein-Westfalen | Carolin Kohl

Neuronale Netze auf der Suche nach dunkler Materie

7 Ausblick

In diesem Buch haben wir nur sehr einfache Programme zum Thema Deep Learning kennengelernt. Wer tiefer einsteigen will, der kann sich an ***Neural Style Transfer*** oder ***DeepDream*** versuchen. Ausführliche Informationen hierzu gibt es in dem Buch *Deep Learning mit Python und Keras* von François Chollet. Die Online-Version findet man unter dem folgenden Link.

https://livebook.manning.com/book/deep-learning-with-python/about-this-book/

Bei *Neural Style Transfer* ändert man mittels Deep Learning ein Foto so, als hätte es ein berühmter Künstler gemalt (Abb. 7.1).

Abbildung 7.1: Die Häuserfront an der Erft in Bad Münstereifel wurde mittels Style Transfer im Stile von Vincent van Gogh verändert

Bei *DeepDream* werden Bilder mittels eines Convolutional Neural Network so verändert, dass man auf den Gedanken kommt, der Computer würde träumen. Die Abbildung 7.2 ist ein Beispiel hierzu. Eingegeben wurde ein Bild von der Stadt Toledo (Abb. 7.2, links) und der Computer hat wohl von den Hunden in der Stadt geträumt (Abb. 7.2, rechts).

Abbildung 7.2: Mittels eines DeepDream-Algorithmus verändertes Bild

Hilfen für den Einstieg in *Neural Style Transfer* und *DeepDream* findet man auch im Python-Notebook von Colab unter *Machine Learning Examples: Seedbank* (Abb. 7.3). Hier erhält man auch tiefergehende Informationen zu Programmen, die in diesem Buch erstellt wurden.

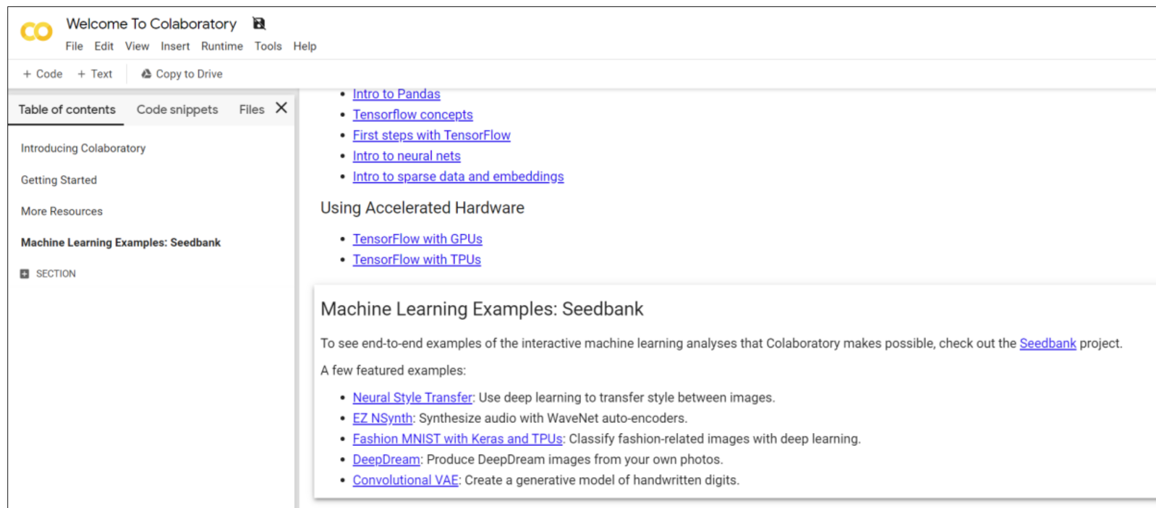

Abbildung 7.3: Links zu Beispielprogrammen findet man im Jupyter-Notebook von Colab
Quelle: https://colab.research.google.com/notebooks/welcome.ipynb#scrollTo=P-H6Lw1vyNNd

Einen wirklich beeindruckenden Überblick über alle Einsatzmöglichkeiten von TensorFlow erhält man unter

https://www.tensorflow.org/

Ein Besuch dieser Seite lohnt sich für alle, die tiefer ins maschinelle Lernen einsteigen wollen.

Wenn man sich privat, im Informatikunterricht oder einer vergleichbaren Arbeitsgemeinschaft offline mit Künstlicher Intelligenz beschäftigen will, dann kann man in Erwägung ziehen, den ca. 100 € teuren **Jetson Nano von NVIDIA** anzuschaffen. Dies ist ein kleiner leistungsstarker 5-Watt-Computer mit einer 64-Bit-ARM-Quad-Core-CPU und einer integrierten NVIDIA-GPU mit 128 Kernen. Hiermit gelingen zahlreiche Offline-Anwendungen für Deep Learning. Objekterkennung mittels Jetson Nano in Kombination mit einer Raspberry-Pi-Kamera ist ein Beispiel hierfür. Dies wird in dem folgenden YouTube-Video überzeugend demonstriert.

https://www.youtube.com/watch?v=k5pXXmTkPNM

Weitere Informationen zum Jetson Nano findet man auf der Homepage von NVIDIA.

https://developer.nvidia.com/embedded/jetson-nano-developer-kit

8 Anmerkung

Dieses Buch enthält Links zu externen Webseiten Dritter, auf deren Inhalte ich keinen Einfluss habe. Deshalb kann ich für diese fremden Inhalte auch keine Gewähr übernehmen. Für die Inhalte der verlinkten Seiten ist stets der jeweilige Anbieter oder Betreiber der Seiten verantwortlich. Die verlinkten Seiten wurden zum Zeitpunkt der Verlinkung auf mögliche Rechtsverstöße überprüft. Rechtswidrige Inhalte waren zum Zeitpunkt der Verlinkung nicht erkennbar. Eine permanente inhaltliche Kontrolle der verlinkten Seiten ist jedoch ohne konkrete Anhaltspunkte einer Rechtsverletzung nicht zumutbar. Bei Bekanntwerden von Rechtsverletzungen werde ich derartige Links umgehend entfernen.

Alle aufgeführten Links habe ich zuletzt am 29.01.2021 überprüft.

9 Index

adam ... 20
Aktivierungsfunktionen ... 7
API ... 25
astype ... 25
batch_size ... 27
Biasvektoren ... 9
Biaswert ... 6
CIFAR-10 ... 29
CNN ... 23
code cells ... 23
Colab ... 13
Conv2D ... 25
Convolutional Neural Network ... 23
CPU ... 13
Deep Learning ... 5
DeepDream ... 41
Dense ... 20
Dense Layer ... 23
epochs ... 27
Fashion-MNIST ... 35
filters ... 25
Google Colaboratory ... 13
Google Drive ... 23
GPU ... 5
Gradientenabstiegsverfahren ... 9
input_shape ... 25
Jetson Nano ... 42
Jugend forscht ... 39
Jupyter-Notebook ... 11
Keras ... 11
Kernel ... 25
kernel_size ... 25
Layer ... 6
Machine Learning ... 5
Maschinelles Lernen ... 5
MaxPooling2D ... 26
mean_squared_error ... 20
MNIST ... 24
Neural Style Transfer ... 41
Neuron ... 5
Neuronales Netz ... 5
Neuronen ... 6
NumPy ... 11
Optimierer ... 8
padding ... 25
PIL ... 27
pool_size ... 26
PReLU-Funktion ... 8, 26
print ... 20, 27
Python ... 11
ReLU-Funktion ... 8
reshape ... 25
Sequential ... 20, 25
Sigmoid-Funktion ... 8
softmax ... 27
verbose ... 20
Verlustfunktion ... 8
Wichtungsmatrizen ... 9
Wichtungswerte ... 6