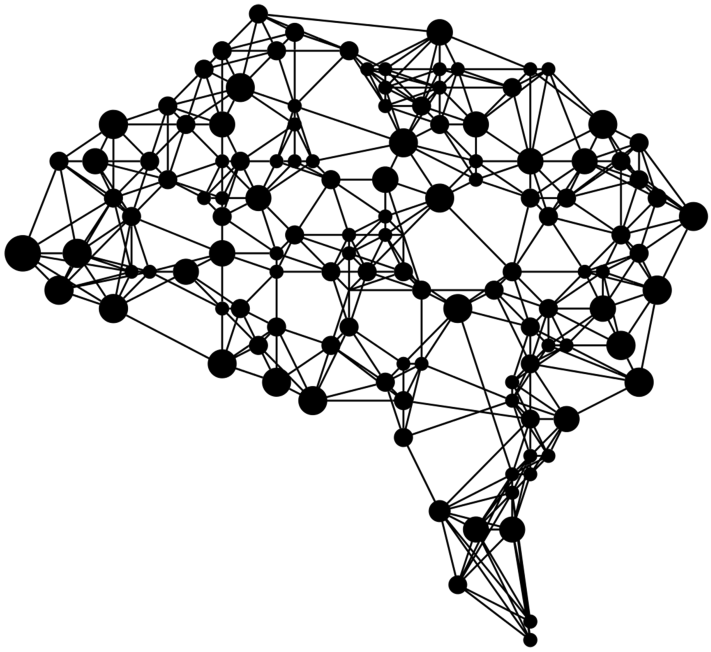
Machine Learning in Smartphone Apps

ASGSG Informatik, 2020/2021

Phillip Bronzel 2. Januar 2021



cleanpng 2020

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst, dass ich keine anderen Quellen und Hilfsmittel als die angegebenen benutzt und die Stellen der Arbeit, die anderen Quellen dem Wortlaut oder Sinn nach entnommen sind, in jedem einzelnen Fall unter Angabe von Quellen kenntlich gemacht habe.

Inhaltsverzeichnis

1 Einführung

Im Enterprise und Forschungsbereich spielt Machine Learning schon seit vielen Jahren eine bedeutende Rolle. Doch wie kann es dem Endnutzer, zum Beispiel in mobilen Apps, weiterhelfen?

1.1 Wahl des Themas

Seitdem ich im Jahr 2017 meinen ersten richtigen Kontakt mit der Programmierung von Microcontrollern (Arduinos) hatte, habe ich mich stark für die Entwicklung von Software interessiert. Dies war ein guter Einstieg, da man dort schnell und recht einfach Ergebnisse, wie zum Beispiel eine blinkende LED, erzielt.

Auch habe ich mich seitdem immer für die „neuen großen Technologien“ wie Blockchain oder Machine Learning interessiert. Zum Thema Machine Learning habe ich zuvor noch nicht viel gemacht, daher ergriff ich die Chance dieses Jahr meine Facharbeit über dieses Thema zu schreiben.

> AI is profound, and we are at a point—and it will get better and better over time—where the GPU is getting so powerful there's so much capability to do unbelievable things. What all of us have to do is to make sure we are using AI in a way that is for the benefit of humanity, not to the detriment of humanity.[1]

Ich persönlich finde dieses Zitat sehr wichtig; es ist jetzt über 3 Jahre alt und bis heute hat sich enorm viel in diesem Bereich getan. Wir haben nun GPU's, welche speziell auf mathematische Berechnungen mit Tensoren optimiert sind und so das Trainieren von Neuronalen Netzen um ein Vielfaches beschleunigen.[2]

Des weiteren ist es mir, genauso wie Cook, wichtig, diese mächtige Technologie nicht zu missbrauchen[3], sondern gute Dinge mit ihr zu schaffen: wie beispielsweise im Bereich der Medizin. In diesem Bereich wurden schon viele beachtliche Anwendungszwecke gefunden, so hat Google's Tochterfirma DeepMind im Dezember 2020 eine Technologie[4] präsentiert, welche das Falten von Proteinen akkurat prognostizieren kann; dies war vorher nur sehr langsam und deutlich ungenauer möglich.[5]

1.2 Ziel der Arbeit

Mein persönliches Ziel ist es, mehr über den Aufbau von Neuronalen Netzen und die Funktionsweise von Machine Learning zu lernen. Außerdem möchte ich auch ein

[1] Byrnes 2017, Tim Cook (CEO von Apple) In einem Interview mit MIT Technology Review
[2] Hebert 2020, NVIDIA Grafikprozessoren mit integrierten Tensor Kernen
[3] Beispiel: Autonome Waffen, wie Drohnen, welche Ziele autonom erfassen können oder auch „Deep-fakes“
[4] Künstliche Intelligenz
[5] Rettner 2020

praktisches Ergebniss haben, dafür habe ich im Kapitel !TODO! eine App entwickelt, welche dem Nutzer mehr Informationen über Produkte beim einkaufen liefern soll.

2 Neuronale Netzwerke

2.1 Wichtigste Ereignisse in der Geschichte

Im Jahr 1943 wurde die erste Arbeit darüber geschrieben, wie Neuronen im Gehirn funktionieren könnten und die Autoren Warren McCulloch und Walter Pitts experimentierten sogar damit diese mit elektronischen Schaltkreisen nachzubauen.[6]

In den 1950er Jahren haben Forscher von IBM daran gearbeitet ein Neuronales Netzwerk mit einem Computer zu simulieren. Der Versuch scheiterte allerdings.[7]

Immer wieder gab es kleinere Forschungsprojekte, ein sehr großer Durchbruch war aber 1975 die Entwicklung eines „Backpropagation“ Algorithmus durch den Wissenschaftler Paul Werbos. Ähnliche Algorithmen wurden wiederholt und unabhängig entwickelt, aber Werbos' Algorithmus war der erste mit großer Bedeutung.[8] Das Prinzip des Algorithmus wird auch heute noch verwendet, es ist dieser Algorithmus der dem Neuronalen Netzwerk das selbstständige Lernen ermöglicht.[9]

In 1998 veröffentlichte Yann LeCun und sein Team eine Arbeit über die Anwendung eines „Convolutional Neural Networks[10]“ zur Erkennung von geschriebenen Zeichen in einem Dokument.[11] Diese Arbeit gilt als Ursprung des, für beispielsweise Bilderkennungs Software gut geeignete, CNNs und Weiterentwicklungen werden auch heute noch verwendet.

Obwohl ein großes Potenzial erkannt wurde, war es über die nächsten Jahre wieder recht still. Der nächste große Durchbruch passierte in 2012 als Geoffrey Hinton ein Modell entwickelte, was die Fehlerquote in einer öffentlichen Challenge für Bilderkennung beinahe halbierte.[12] Der Grund dafür waren mehrere fundamentale Neuerungen aus dem Bereich Deep Learning; die wahrscheinlich größte Änderung: Starke Parallelisierung des Backpropagation-Prozesses, durch Verschiebung der Last von der CPU auf die GPU. Aufgrund der starken Überlegenheit eines Grafikprozessors in parallelisierten Prozessen, wie die benötigten Tensormultiplikationen durch die deutlich größere Anzahl an (dafür schwächeren) Kernen im Vergleich zu einer herkömmlichen CPU, kann ein Neuronales Netzwerk mehrere hundertmal schneller trainiert werden.

Heute gibt es (vergleichsweise) simple Frameworks, wie das im Jahr 2015 erschienende TensorFlow oder PyTorch aus 2016, welche das erstellen, trainieren und ver-

[6] Warren McCulloch 1943
[7] Roberts unbekanntes Jahr, Absatz 3
[8] Werbos 1975
[9] Genaueres in Kapitel 2.3
[10] Ab jetzt als CNN bezeichnet
[11] Yann LeCun und Haffner 1998
[12] Geoffrey E. Hinton 2012

wenden von Neuronales Netzwerk enorm vereinfachen. Ihr Funktionsumfang wächst durch die große Open-Source Community ständig.

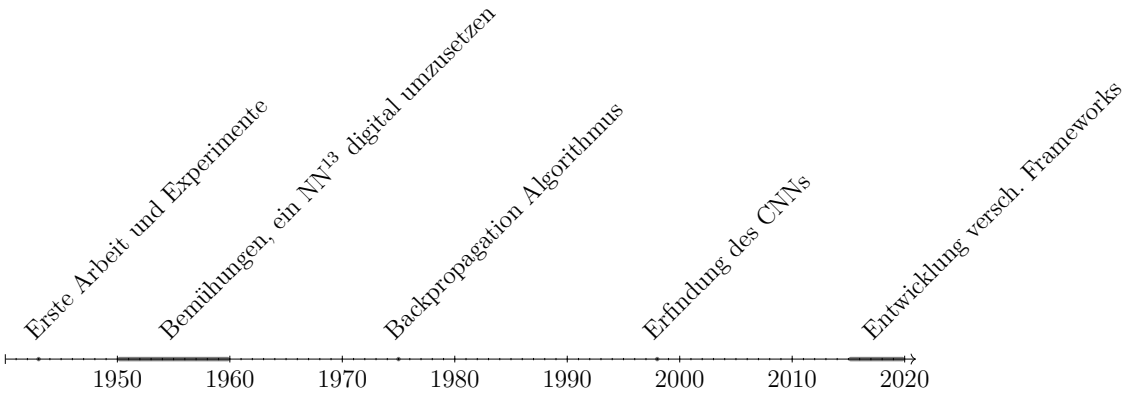


Abbildung 1: Zeitstrahl von 1940 bis 2020 mit den wichtigsten Ereignissen der Entwicklung künstlicher Neuronaler Netzwerke

2.2 Aufbau

In Abbildung 2 sieht man den Aufbau eines herkömmlichen künstlichen Neuronalen Netzwerks, so wie es noch vor 40 Jahren verwendet wurde. In der Grafik erkennt man drei Layer mit einer x-beliebigen Anzahl Neuronen, welche untereinander mit jeweils allen Neuronen der vorigen und nächsten Layer verbunden sind. Im Gegensatz zu einem biologischen Neuron, welches nur aktiv oder inaktiv sein kann, kann ein künstliches Neuron einen Zustand in Form eines Wertes von $0 \leq x \leq 1$ haben. Jede Verbindung hat einen Weight Paramter und auch jedes Neuron hat einen Bias. Die Anzahl der Hidden Layer kann an das Ziel angepasst und ausgewählt werden und auch die Anzahl der einzelnen Neuronen ist erstmal beliebig, als Faustregel für gute Ergebnisse gilt aber:

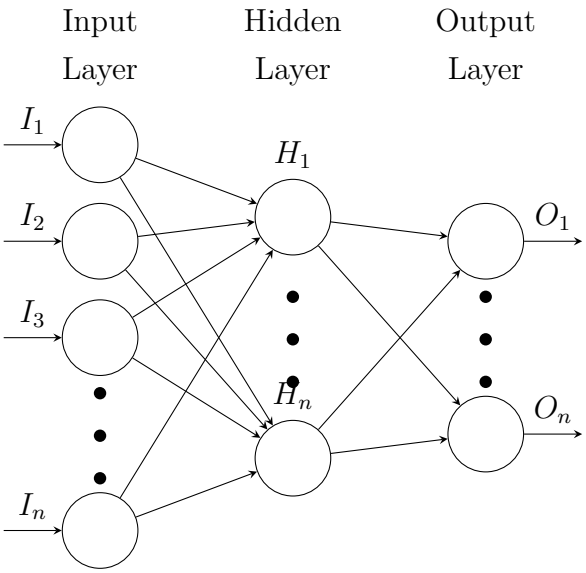


Abbildung 2: Vereinfachter Aufbau eines Neuronales Netzwerk

- Die Anzahl der Neuronen in dem Hidden Layer sollte zwischen der Größe des Input und Output Layers liegen.

- Die Anzahl der Neuronen in dem Hidden Layer sollte etwa $\frac{2}{3}$ der Größe des Input Layers plus der Größe des Output Layers entsprechen.
- Die Anzahl der Neuronen in einem Hidden Layer sollte weniger als die Hälfte der Größe des Input Layers sein.[14]

2.2.1 Erstellung eines Neuronalen Netzwerks anhand eines Beispiels

Als Beispiel für ein Neuronales Netzwerk, welches darauf ausgelegt ist, geschriebene Ziffern aus Bildern mit 24x24 Pixeln und nur Graustufen zu erkennen wäre dann: Ein Input Layer mit 24^2 Neuronen, jeweils für jeden Pixel, welche jeweils eine Aktivierung zwischen 0 (komplett weiß) und 1 (komplett schwarz) haben können, eines. Genau 10 Neuronen im Output Layer, für jedes Zahlzeichen eines. Schließlich muss die Anzahl der Hidden Layer und Neuronen festgelegt werden. Ich wähle als Beispiel 2 Layer mit jeweils 16 Neuronen, die Neuronen-Anzahl kann aber auch unterschiedlich sein. Auch die Weights und Biases werden zunächst zufällig ausgewählt, die Werte werden dann später im Trainingsprozess[15] angepasst. Die Eingabe in das Netzwerk ist also ein Zweidimensionaler Tensor, oder auch eine Matrix, mit den Bilddaten und die Ausgabe des Netzwerks ist ein eindimensionaler Tensor, oder auch ein Vektor, mit den Prognosen.

$$\begin{bmatrix} i_{0,0} & \dots & i_{0,23} \\ \vdots & \ddots & \vdots \\ i_{23,0} & \dots & i_{23,23} \end{bmatrix} \Longrightarrow \begin{bmatrix} o_0 \\ \vdots \\ o_9 \end{bmatrix} \tag{1}$$

2.3 Funktionsweise

Ein Neuronales Netzwerk kann man sich eigentlich als eine große Mathematische Funktion vorstellen. In dem zuvor genannten Beispiel wäre es eine Funktion mit 576 Variablen und 10 Ergebnissen. Gibt man dieser Funktion nun ein Bild, beziehungsweise 576 Werte als Input, so werden von links nach rechts alle Weights w und Biases b zusammen mit dem vorigen Aktivierungswerten a berechnet. Da ein Neuron aber nur Werte im Bereich $0 \leq x \leq 1$ haben kann[16], so wird das Ergebniss noch mithilfe einer Aktivierungsfunktion in diesen Bereich umgewandelt.[17] Eine

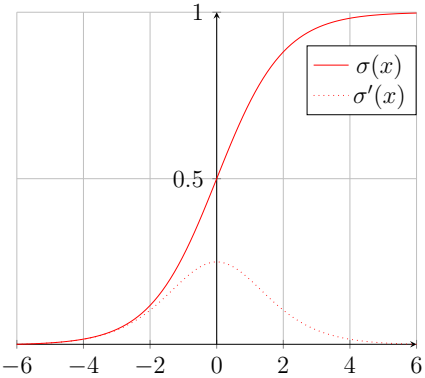


Abbildung 3: Die Sigmoidfunktion

[14] Heaton 2017, Alle drei Faustregeln
[15] siehe Kapitel 2.3.1
[16] Sanderson 2017
[17] Google 2020 Schlüsselwort: Activation Function

früher Häufig verwendete Funktion ist dabei die
Sigmoidfunktion, siehe Abbildung 3.[18] Es gibt aber auch noch eine Vielzahl weiterer Funktionen, wie die heute häufig verwendete ReLU Funktion[19], welche den Trainingprozess durch die einfachere Funktion beschleunigt.[20] Die daraus resultierende Funktion würde in etwa so aussehen:[21]

$$\sigma(w_1a_1 + w_2a_2 + w_3a_3 + \ldots + w_na_n + b) \tag{2}$$

Um mit dieser Formel alle Aktivierungen auf einmal berechnen zu können verwendet man folgende Funktion, in welcher alle Weights und Biases in Spalten-Vektoren zusammengefasst werden. Die Hochzeichen sind keine Exponenten sondern gelten als Bezeichnung für den Layer, hier beispielsweise 0 und 1. Das Ergebniss dieser Funktion ist ein Vektor mit allen Aktivierungen des darauf folgenden Layers.

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \ldots & w_{0,n} \\ w_{1,0} & w_{1,1} & \ldots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \ldots & w_{k,n} \end{bmatrix} \cdot \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_k \end{bmatrix}\right) = a^{(1)} \tag{3}$$

[22]

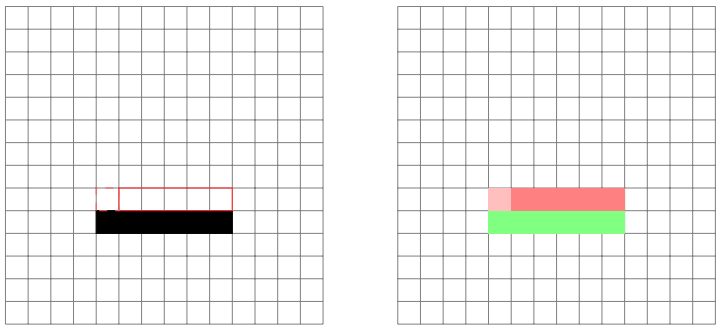
Auch diese Funktion kann wiederrum kompakter formuliert werden und diese Schreibweise wird auch für gewöhnlich verwendet:

$$a^{(1)} = \sigma(W \cdot a^{(0)} + b) \tag{4}$$

[20]

Theoretisch wenn ein Neuron einen hohen Aktivierungswert haben soll, wenn beispielsweise eine gerade Linie erkannt wird (um mit anderen Neuronen zusammen im späteren Verlauf aus den Mustern ganze Ziffern zu erkennen), so müssen die Weights der zu dem Neuron führenden Verbindungen alle möglichst niedrige Aktivierungen haben, ausser an den Stellen an denen die Linie sich befinden soll. Um sicherzustellen, dass es sich wirklich um eine gerade Linie handelt befindet sich direkt über dem Strich ein Bereich in dem keine Aktivierungen sein sollten, dieser ist rot markiert. Das erkennt man in Abbildung 4 sehr gut. In a erkennt man die zu erkennende Linie und in b sieht man die zugehörigen Weights der Input Nodes zu dem Neuron. Dabei stellt grün positive Weights da, rot negative und Weiß/Transparent ist 0. Der Bias des Neurons stellt eine Zusätzliche Hürde oder eine Verstärkung da, was auch in Formel 2 als b sichtbar ist.

[18] Sanderson 2017
[19] siehe Anhang A.1
[20] Harrison Kinsley 2020 Seite 76 folgende
[21] Harrison Kinsley 2020 Seite 185
[22] Gleichungen 3 und 4 von Sanderson 2017



(a) Zu erkennendes Bild (b) Benötigte Weights

Abbildung 4: Visualisierung der gewünschten Formen a und die dazugehörigen Weights b (jeweils abgeschnitten)

2.3.1 Trainieren des Neuronalen Netzwerks

Dieser Prozess ist der wichtigste. Durch das Trainieren erzielt ein Neuronales Netzwerk den Effekt des selbstständigen Lernens. Und da die Werte der Weights und Biases zunächst zufällig ausgewählt wurden, muss das Netzwerk trainiert werden um nicht völligen Unsinn auszugeben.[23]

Die Cost Function Um herauszufinden wie gut oder schlecht ein Neuronales Netzwerk arbeitet, also auf das Beispiel bezogen wie genau oder ungenau es Ziffern erkennen kann, gibt es die Cost Function. Es gibt verschiedene Arten und Möglichkeiten ähnliche Funktionen anzuwenden, hier werde ich mich allerdings auf die Minimierung der Cost Function beziehen. Als Ergebniss kommt eine einzige Zahl heraus welche hoch ist, wenn das Netzwerk schlechte Ergebnisse erzielt und gegen 0 läuft, wenn das Netzwerk sehr gute Ergebnisse liefert. Es gibt mehrere verschiedene Cost Functions, aber ich fokussiere mich erstmal auf die MSE Funktion. MSE steht für „Mean squared Error“, sie berechnet den Cost Wert[24] aus dem durchschnitt der Summe der Vorhersagen und den erwarteten Ergebnissen zum Quadrat:

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - y^{(i)})^2 \tag{5}$$

Wenn:

- i = Index der Trainingsdaten
- x = Vorhersage des Netzwerks
- y = Erwartetes (richtiges) Ergebniss
- m = Anzahl der Trainingsdaten[25]

[23] Ein Code Beispiel, wie man ein solches Netzwerk mit modernen Frameworks erstellen und trainieren würde befindet sich im Anhang A.2.

[24] Manchmal auch Loss genannt, meint das gleiche.

[25] Formel 5 und Erklärung vergleiche Krzyk 2018

Gradientenabstiegsverfahren Leider ist es bei solch großen Funktionen nicht mehr möglich (stimmt das? !TODO!) das Globale Minimum explizit zu bestimmen.

Daher berechnet man die Steigung ΔC[26] der Funktion und bestimmt anschließend die Richtung $-\Delta C$ in welche der Graph sinkt. In Abbildung 5 ist der Graph nur Zweidimensional und daher gibt es nur eine Richtung in welcher der Graph fallen kann. So wird die Eingabe immer weiter so verändert, dass sich die Cost Function minimiert. Dies passiert in mehreren Iterationen oder auch Epochen, in welchen die Veränderungen, also Schritte in Richtung $-\Delta C$, in Abhänigkeit von der Steigung, immer kleiner werden um einen Überschuss zu verhindern.[27] Das ist auch der Grund weswegen Trainingszeiten exponentiell zur Genauigkeit ansteigen. Die Größe dieser Schritte wird auch Lernrate / Learning Rate genannt.[28]

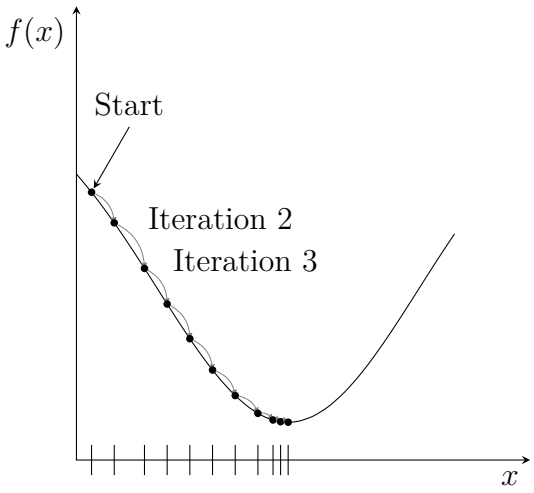


Abbildung 5: Vorgehen bei der Minimierung

Bei mehrdimensionalen Funktionen, wie auch den Neuronalen Netzwerken, gibt es mehr Möglichkeiten in welche Richtung der Graph am schnellsten Fallen könnte. Die oben beschriebene Technik die dafür verwendet wird nennt sich das Gradientenabstiegsverfahren. Da die Funktionen in echt allerdings deutlich komplizierter sind, gibt es sehr viele Extrema und da das Ziel ein möglichst tiefer Extrempunkt ist, versucht man zu verhindern, dass man in einem solcher hohen Minima „stecken bleibt“. Deswegen verwendet man meistens abgewandelte Formen des Gradientenabstiegsverfahrens, zum Beispiel mit einem Modifikator für Momentum. So fällt un-

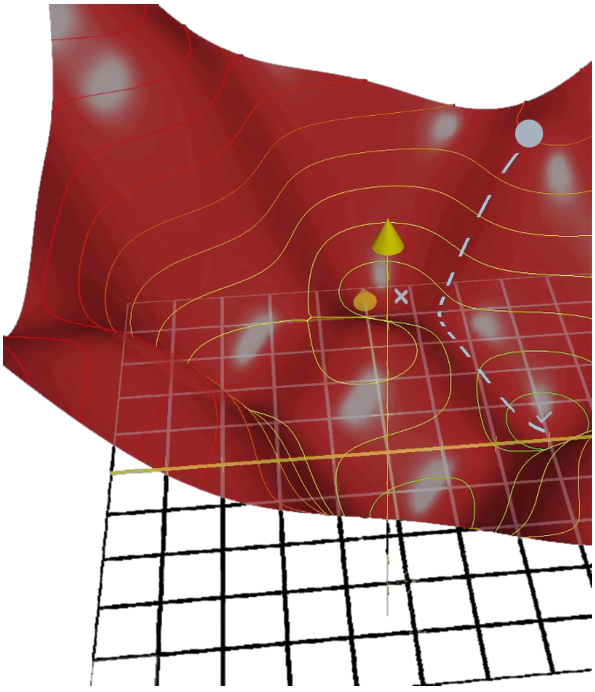


Abbildung 6: Visualisierung des Gradientenabstiegsverfahren im dreidimensionalen Raum Sanderson 2017)

[26] C bezieht sich hier auf die **C**ost function
[27] Sanderson 2017
[28] Fortuner 2017

ter Umständen „der Ball“ weiter in ein tieferes Minimum, was in Abbildung 6 zu erkennen ist. Desweiteren wird für Machine Learning fast ausschließlich „Stochastic gradient descent“ verwendet, da das Gradientenabstiegsverfahren allein viel zu aufwendig ist. Mit dieser Form verliert man etwas Genauigkeit, der Prozess geht aber ein vielfaches schneller vonstatten. Anstatt den Gradienten von allen Trainingsdaten zusammen zu suchen, teilt man die Trainingsdaten in mehrere „Batches“ auf und wendet auf diesen das Gradientenabstiegsverfahren an. So werden häufiger/schneller Schritte richtung Extrempunkt gemacht, diese sind dafür aber ungenauer (sie repräsentieren nicht den schnellsten Weg).[29]

Ketten Regel Die Kettenregel vereinfacht das Ableiten von komplizierten Funktionen mit Logarithmen und Wurzeln, E-Funktionen und auch Klammern. Wie man diese anwendet sieht man in Formel 6.[30]

$$\begin{aligned} f(x) &= u(g(x)) \\ f'(x) &= u'(g(x)) \cdot g'(x) \end{aligned} \tag{6}$$

Dieses Verfahren wird verwendet um die Ableitungen der Cost Function und des Netzwerks zu finden.

Backpropagation Jetzt ist klar was das Ziel des Trainingsprozesses ist, aber wie wird die Cost Function minimiert? Hier kommt der Backpropagation Algorithmus ins Spiel. In Kombination mit der Cost Function aus und den soeben besprochenen Verfahren bestimmt dieser Algorithmus welche und um wie viel die Weights und Biases des Neuronalen Netzwerks angepasst werden müssen.

2.4 Abwandlungen

Heute gibt es sehr viele verschiedene Abwandlungen von diesen Techniken und auch Neuronale Netzwerke, die zwar ähnlich aufgebaut sind, die besser für manche Zwecke sind als andere. Da gibt es zum Beispiel „Convolutional Neural Networks“[31], welche besonders gut zum erkennen von Objekten in Bildern geeignet sind und daher auch in (!TODO! app kapitel einfügen) verwendet wird. Oder „Long short Term memory Networks“[32], welche speziell auf das erkennen von Stimmen ausgelegt sind.

[29] Sanderson 2017 und Sra 2019
[30] Rudolph 2019
[31] auch CNN; deutsch: **faltendes** neuronales Netzwerk
[32] auch LSTMN; deutsch: Langes **Kurzzeitgedächtnis** Netzwerk

2.4.1 Convolutional Neural Networks

Convolutional Neural Networks versuchen ein Problem der normalen Neuronalen Netzwerke im Bereich der Bilderkennung zu lösen. Während in einem normalen Netzwerk die Position des zu erkennenden Objektes im Bild eine Rolle spielt, wird dies zum Großteil in einem Convolutional Neural Network durch die veränderte Funktionsweise behoben/verbessert.

3 Labelcheck als Smartphone App

In diesem Kapitel wird mithilfe von Python und Tensorflow ein Netzwerk erstellt und trainiert, sowie anschließend eine mobile App mit Dart und Flutter entwickelt, welche dann öffentlich für den Download bereit stehen soll.

3.1 Die Idee

Die Idee ist, dass die App es ermöglicht im Supermarkt die verschiedenen Label der Produkte zu scannen und dem Nutzer dann Auskunft über die Vertrauenswürdigkeit und generelle Aussage des Labels gibt.

Der Name der App „Labelcheck“ setzt sich ganz einfach aus den Wörtern „Label“ und „Check“ zusammen → „Label überprüfen“.

3.2 Erstellen des Models

Zum erstellen und trainieren des Modells werde ich die Sprache Python und das Framework Tensorflow verwenden.

3.2.1 Die Trainingsdaten

3.2.2 Trainieren des Modells mit Tensorflow und Python

3.3 Entwickeln der App

Zum entwickeln der App verwende ich die Sprache Dart und das zugehörige Framework Flutter. Im Gegensatz zu nativ geschriebenen Apps bietet Flutter die möglichkeit nur einmal den Code in Dart zu schreiben und anschließend kann die App für alle großen Platformen kompiliert werden, dazu zählen iOS, Android, aber auch Linux, Windows, MacOS und Web/Javascript. Nun muss die App ja Zugriff auf die Kamera haben und auch in die Supermärkte „mitgebracht“ werden, weshalb nur iOS und Android relevant sind.

3.3.1 Das Framework: Flutter

Flutter Apps funktionieren anders als Nativ entwickelte Apps. Herkömmliche native Apps verwenden die UI Komponenten des Betriebssystems und sehen daher auf jedem Gerät mit unterschiedlichen Betriebssystemversionen leicht unterschiedlich aus. Flutter hingegen stellt ein „Canvas“ Element bereit, welches als unterliegende Grafik-Engine Google's Skia nutzt. In Flutter stehen eine Menge UI Komponenten zur verfügung die entweder Googles Material Design guidelines oder Apples Human interface guidelines folgen. Der Dart Code stellt dann als Einzigen Eintrittspunkt die `main()` Methode bereit, aus welchem dann die App gestartet wird. Das Framework wird mit der Methode `runApp(Widget)` initialisert. In Flutter ist jedes UI Element ein „Widget“, wodurch sich dann in Kombination in einer App große Widgethierarchien erstellen lassen. Der Code einer simplen App, welche nur den Text „Hello World!“ in der Mitte des Bildschirms anzeigen würde, sehe demnach so aus:

```
1  import 'package:flutter/widgets.dart';
2
3  void main() => runApp(MyApp());
4
5  class MyApp extends StatelessWidget {
6    @override
7    Widget build(BuildContext context) {
8      return Center(
9        child: Text('Hello World!'),
10     );
11   }
12 }
```

Zeile 1: Importieren der Widgets aus dem Framework zum bereitstellen der Klassen, wie `Stateless`- und `StatefulWidget` und den Methoden, wie `runApp()`.

Zeile 3: Die `main()` Methode mit dem einzigen Aufruf `runApp()`, was die Klasse `MyApp` als Flutter App initialisert.

Zeile 5f.: `MyApp` erweitert die Klasse `StatelessWidget`, überschreibt die `build()` Methode und gibt eine Widgethierarchie zurück.

Zeile 8f.: Das `Center` Widget nimmt als einzigen benannten Parameter ein weiteres (child) Widget an, was in diesem Fall ein `Text` Widget ist.

3.3.2 Importieren des Models

3.3.3 Veröffentlichen der App

3.4 Testen der App

3.5 Fazit

A Anhang

A.1 Weitere Aktivierungsfunktionen

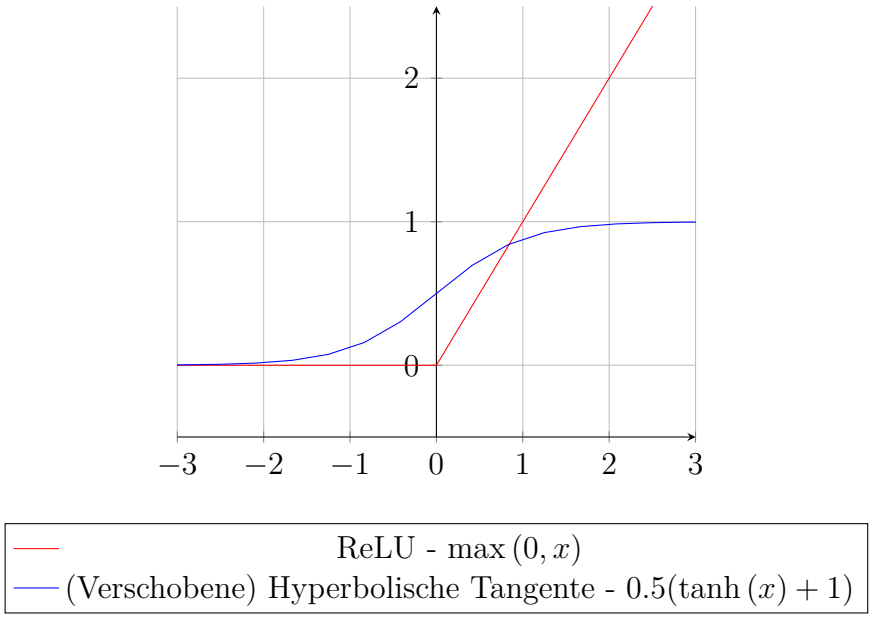


Abbildung 7: Weitere Aktivierungsfunktionen, ergänzend zu der Sigmoid Funktion aus Kapitel 2.3

Die ReLU (Rectified linear Unit) Funktion ist im Vergleich zu anderen Aktivierungsfunktionen, wie der Sigmoidfunktion oder der Hyperbolischen Tangente, deutlich simpler, was sich in Leistungsansprüchen des Trainingsprozesses wiederspiegelt.[33]

[33] Harrison Kinsley 2020

A.2 Code für das Beispiel aus 2.3

```python
import tensorflow as tf
import tensorflow.keras.layers as layers

numberOfNeuronsInFirstLayer = 16
numberOfNeuronsInSecondLayer = 16
numOfEpochs = 5

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data() # Laden des MNIST Datasets
# Und aufteilen in Trainigsdaten und Testdaten

x_train = tf.keras.utils.normalize(x_train, axis=1) # Normalisieren des Datasets
x_test = tf.keras.utils.normalize(x_test, axis=1)

model = tf.keras.models.Sequential() # Erstellen des Neuronalen Netzwerks
model.add(layers.Flatten())
model.add(layers.Dense(numberOfNeuronsInFirstLayer, activation=tf.nn.sigmoid))
# Hinzufügen der Layer
model.add(layers.Dense(numberOfNeuronsInSecondLayer, activation=tf.nn.sigmoid))
model.add(layers.Dense(10, activation=tf.nn.softmax))

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy']) # Kompilieren der Layer zu einem trainierfähigen Modell

model.fit(x_train, y_train, epochs=numOfEpochs)
# Trainieren des Modells mit den Trainingsdaten und x Epochen

model.save('beispielModel_MNIST') # Speichern des Modells
```

Listing 1: Umsetzung mit Python und Tensorflow

Ein interaktives Beispiel gibt es zusätzlich hier in meinem Colab Notebook: `https://bit.ly/34Ggfuh`[34]

[34] Ungekürzter Link: `https://colab.research.google.com/drive/1ty_QQlL038YT6KpBjSdqGvIGyH0YXwxW`

Literaturverzeichnis

Byrnes, Nanette (2017). *Tim Cook: Apple Isn't Falling Behind, It's Just Not Ready to Talk About the Future*. URL: https://www.technologyreview.com/2017/06/14/4525/tim-cook-apple-isnt-falling-behind-its-just-not-ready-to-talk-about-the-future/ (besucht am 05. 12. 2020).

cleanpng (2020). *Deep learning maschinelles lernen, Künstliche neuronale Netzwerk der informatik Convolutional neural network - Vernetzung*. URL: https://de.cleanpng.com/png-s07s8u (besucht am 17. 11. 2020).

Fortuner, Brendan (2017). *ML Glossary - Gradient Descent*. URL: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html (besucht am 28. 12. 2020).

Geoffrey E. Hinton Alex Krizhevsky, Ilya Sutskever (2012). *ImageNet Classification with Deep Convolutional Neural Networks*. URL: https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (besucht am 17. 12. 2020).

Google (2020). *Machine Learning Glossary*. URL: https://developers.google.com/machine-learning/glossary (besucht am 26. 12. 2020).

Harrison Kinsley, Daniel Kukiela (2020). *Neural Networks from Scratch (NNFS)*. URL: https://nnfs.io (besucht am 20. 11. 2020).

Heaton, Jeff (2017). *Heaton Research, The Number of Neurons in the Hidden Layers*. URL: https://www.heatonresearch.com/2017/06/01/hidden-layers.html (besucht am 17. 12. 2020).

Hebert, Chris (2020). *Accelerating WinML and NVIDIA Tensor Cores*. URL: https://developer.nvidia.com/blog/accelerating-winml-and-nvidia-tensor-cores/ (besucht am 05. 12. 2020).

Krzyk, Kamil (2018). *Coding Deep Learning for Beginners — Linear Regression (Part 2): Cost Function*. URL: https://towardsdatascience.com/coding-deep-learning-for-beginners-linear-regression-part-2-cost-function-49545303d29f (besucht am 27. 12. 2020).

Rettner, Rachael (2020). *AI system solves 50-year-old protein folding problem in hours*. URL: https://www.livescience.com/artificial-intelligence-protein-folding-deepmind.html (besucht am 05. 12. 2020).

Roberts, Eric S. (unbekanntes Jahr). *History: The 1940's to the 1970's*. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html (besucht am 14. 12. 2020).

Rudolph, Dennis (2019). *Kettenregel Ableitung*. URL: https://www.gut-erklaert.de/mathematik/kettenregel-ableitung.html (besucht am 02. 01. 2021).

Sanderson, Grant (2017). *But what is a Neural Network?* URL: https://www.3blue1brown.com/neural-networks (besucht am 18. 12. 2020).

Sra, Suvrit (2019). *25. Stochastic Gradient Descent*. URL: `https://www.youtube.com/watch?v=k3AiUhwHQ28` (besucht am 30. 12. 2020).

Warren McCulloch, Walter Pitts (1943). *A logical calculus of the ideas immanent in nervous activity.*

Werbos, Paul (1975). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.*

Yann LeCun Léon Bottou, Yoshua Bengio und Patrick Haffner (1998). *Gradient-Based Learning Applied to Document Recognition*. URL: `http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf` (besucht am 17. 12. 2020).