Fachhochschule Dortmund

Studienarbeit

Erkennung kinematischer Symbole durch maschinelles Lernen

Kai Lawrence

8. Oktober 2020

Inhaltsverzeichnis

1 Einleitung

Die erste Arbeit, die heute dem Bereich der künstlichen Intelligenz (KI) zugeordnet wird, wurde 1943 von McCulloch und Pitts geleistet [MP43]. Sie ließen sich in ihrer Arbeit von Neuronen im Gehirn inspirieren und erstellten erste mathematische Formulierungen welche später großen Einfluss in der Entwicklung neuraler Netze hatten. Die Neuronen werden dabei als Schalter gesehen, dessen Aktivierung durch andere Neuronen beeinflusst wird. Auf diese weise lassen sich komplexe Systeme erstellen, welche in der Lage sind unterschiedliche Aufgaben zu lösen. Der Begriff der künstlichen Intelligenz selbst wurde 1956 von McCarthy geprägt, als er einen zweimonatigen Workshop in Dartmouth [McC55] organisierte.

Nach den anfänglich großen Erwartungen an die künstliche Intelligenz, ohne die versprochenen Ergebnisse zu liefern, erhielt das Gebiet weniger Aufmerksamkeit. Ein berühmtes Zitat von Herbert Simon aus dem Jahr 1957 ist:

> Es ist nicht mein Ziel, Sie zu überraschen oder zu schockieren - aber ich kann am einfachsten zusammenfassen, dass es jetzt in der Welt Maschinen gibt, die denken, die lernen und die schaffen. Darüber hinaus wird ihre Fähigkeit, diese Dinge zu tun, rasch zunehmen, bis - in einer sichtbaren Zukunft - die Bandbreite der Probleme, die sie bewältigen können, mit der Bandbreite, auf die der menschliche Verstand angewandt wurde, übereinstimmen wird.

Entgegen dieser Übertreibungen wurden im Laufe der Jahrzehnte allerdings dennoch Fortschritte erzielt. Nach einer weiteren Periode hoher Investitionen in den 1980er Jahren, ohne die ehrgeizigen Ziele zu erreichen, setzte der so genannte "KI-Winterëin, der sich durch mangelndes akademisches und wirtschaftliches Interesse und mangelnde Finanzierung bemerkbar machte.

In jüngster Zeit jedoch finden Forschung und Anwendung dieser Techniken wieder mehr Aufmerksamkeit. Dies ist auf einige in der heutigen Zeit vorherrschende Faktoren zurückzuführen. Zum einen ist es die schiere Menge an Rechenleistung, die moderne Computern haben können. Selbst Heimcomputer sind heute in der Lage, Modelle von bemerkenswerter Komplexität zu trainieren.

Des Weiteren erleichtert die Menge an Daten, die durch das Internet zur Verfügung steht, das Training der KIs für verschiedene Aufgaben. Um ein Modell für eine bestimmte Aufgabe zu trainieren, werden geeignete Daten benötigt, um korrekte Vorhersagen über nie gesehene Daten zu treffen (Generalisierung). Da Daten in großer Menge zur Verfügung stehen, gibt es entsprechend mehrere Datenbanken für Forschung, Training

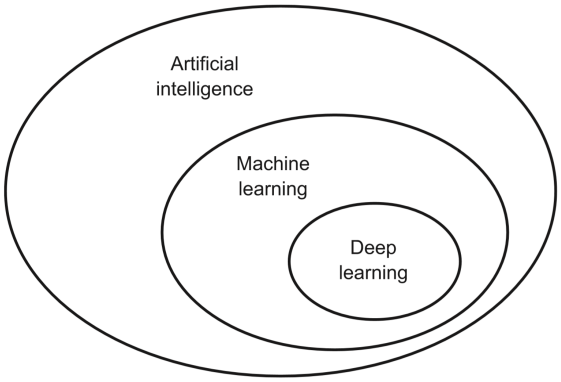


Abbildung 1: Künstliche Intelligenz, Maschinelles Lernen und Deep Learning [Cho17, p.4]

und Evaluation. Eine Liste von Datenbanken kann unter `https://aka.klawr.de/srp#1` eingesehen werden.

Deshalb wird der künstlichen Intelligenz in der Forschung immer mehr Aufmerksamkeit geschenkt, und immer mehr praktische Bereiche betrachten diese Technik als Lösung für verschiedenste Problemstellungen. Ob der jüngste Enthusiasmus nur übertriebene Erwartungen sind, oder ob die "letzte Erfindung der Menschheit"[Goo65] nahe ist, bleibt abzuwarten.

Der Begriff künstliche Intelligenz wird verwendet, um maschinelle Vorgänge zu beschreiben, die Aufgaben ausführen, die sonst charakteristisch für die menschliche Intelligenz sind. Diese Arbeit soll sich vor Allem mit einem Teilgebiet der künstlichen Intelligenz befassen, dem so genannten "deep learning". Dieser Zusammenhang ist in Abbildung 1 dargestellt.

In diesem Projekt möchte ich maschinelles Lernen in das Gebiet der Kinematik für zweidimensionales Skizzieren und der Prototypentwicklung mit `deepmech` vorstellen. Die erste für dieses Projekt implementierte Lösung nimmt Bilder als Input und gibt den jeweiligen Typ handgeschriebener mechanischer Symbole aus. Die mit dem vorgestellten Ansatz erstellten Modelle sind klein und meist unabhängig von der genutzten Programmiersprache, was durch die Verwendung der populären Keras [Cho19]-Bibliothek auf TensorFlow [Goo19b] ermöglicht wird.

Einfache Demonstrationen werden mit JavaScript durchgeführt, um eine Verbindung zu einem aufstrebenden Bereich der Kinematik mit Hilfe von Web-Technologien wie `mec2` [Gös19c] und `mecEdit` [Uhl19a] herzustellen. Für das Training des aktuellen Modells wird die Python-Implementierung von TensorFlow verwendet, da durch die bessere Ausnutzung der GPU mit CUDA [nvi19] als Backend ein schnelleres Training der Keras-

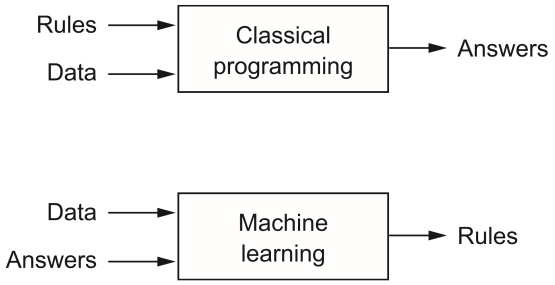


Abbildung 2: Maschinelles Lernen, ein neues Programmierparadigma, von [Cho17, S.5]

Modelle möglich ist. Dank der Konvertierbarkeit der Modellbeschreibungen zur Nutzung in unterschiedlichen Programmiersprachen ermöglicht dies nahtlose Übergänge zwischen verschiedenen Ansätzen.

Bevor gezeigt wird, wie deepmech entwickelt wurde, sollen die Grundlagen des maschinellen Lernens behandelt werden, wobei einige einfache Beispiele als Einführung gezeigt werden und dabei die in dieser Arbeit verwendete allgemeine Terminologie eingeführt wird. In Kapitel 3 werden die Konzepte durch die Einführung neuronaler Netze erweitert, die der Schlüssel zur Umsetzung der Funktionsweise des maschinellen Lernens in praktische Modelle sind. Ein weiteres Thema, das behandelt werden muss, ist die Datengenerierung. Daten sind ein wichtiger Teil des Trainings statistischer Modelle, daher wird die Datenerzeugung und -augmentierung in Kapitel 4 behandelt. Daraufhin wird das erste funktionierende Modell gezeigt. Alle Hauptkomponenten des Prozesses werden detailliert untersucht, um ein besseres Verständnis für jede Funktion zu erhalten. Im 6. Kapitel geht es um die Optimierung des Modells, um die Genauigkeit zu verbessern und die Größe des Modells, das in zukünftigen Anwendungen geladen wird, zu minimieren. Abschließend wird eine Schlussfolgerung erläutert, in der die Ergebnisse evaluiert werden.

2 Was ist maschinelles Lernen?

2.1 Maschinelles Lernen allgemein

Normalerweise besteht die Aufgabe der Programmierung darin, Regeln auf bestimmte Probleme anzuwenden, um Lösungen zu erhalten. Mit Hilfe des maschinellen Lernens wollen wir ein Modell erstellen, das Lösungen für bestimmte Probleme verwendet, um die entsprechenden Regeln herauszufinden. Das Verhältnis wird in Abbildung 2 dargestellt.

Das Versprechen des maschinellen Lernens ist, dass die gelernten Regeln wesentlich

komplexer sein können, als es möglich ist, sie von Hand zu programmieren. Ein gutes Beispiel dafür ist die Bilderkennung (die im Mittelpunkt dieses Projekts stehen wird), bei der der Inhalt von Bildern klassifiziert werden soll. Die Beziehungen zwischen den Pixeln müssen herausgefunden werden, was zu einem ziemlich komplexen Modell führt. Dies ist selbst für die einfachsten Aufgaben nicht auf traditionelle Weise zu programmieren.

Das universelle Approximationstheorem [Cyb89] [HSW89] besagt, dass es für eine beliebige Eingabe x eine Funktion h gibt, die sich der Zuordnung y annähert, während die Zuordnung nur die Eingabe zur jeweils richtigen Ausgabe (oft von einem Menschen beschriftet) ist. Diese Beziehung wird in Gleichung (1) gezeigt.

$$\forall \epsilon > 0 : \exists h(\theta, x) : \forall x \in I : |h(\theta, x) - y(x)| < \epsilon \tag{1}$$

Die mathematische Beschreibung von h wird als Modell bezeichnet (z.B. $h(x) = \sum_{i=1}^{n} \lambda(\theta^T * x)$). $\lambda \in \mathbb{R}$ ist eine Konstante und θ ist ein Tensor, der die Parameter von h definiert, die sich aus so genannten Weights und Biases zusammensetzen, die später näher erläutert werden.

Das universelle Approximationstheorem macht keine Aussagen über die Form von θ; daher wird h als Hypothese bezeichnet, weil es besagt, dass für eine bestimmte Zusammensetzung von θ diese Gleichung zutrifft. Dieser Begriff wird in der Gleichung (2) zusammengefasst.

$$\forall \epsilon > 0 : \exists \theta \in X : \forall x \in I : |h_\theta(x) - y(x)| < \epsilon \tag{2}$$

Zur Modellbewertung wird eine Verlustfunktion verwendet; sie vergleicht das Ergebnis der Hypothese mit den bereitgestellten Daten. Die wohl prominenteste Verlustfunktion ist der `Mean Square Error`. Sie nimmt die Summe der Abweichung von n-Beispielen zum Quadrat und dividiert sie durch $2 * n$.

$$L_{mse}(\theta, x) = \frac{1}{2n} \sum_{i=1}^{n} (h_\theta(x) - y(x))^2 \tag{3}$$

Die Aufgabe des maschinellen Lernens besteht darin, ein θ für ein Modell zu bestimmen, das die Gleichungen 1 und 2 bestätigt, was durch iterative Aktualisierung des θ geschieht. Vor dem Training wird θ zufällig initiiert und dann iterativ angepasst, um ϵ gegen null konvergieren zu lassen. Dadurch wird der Wert von L minimiert.

Damit der Wert der Verlustfunktion allmählich abnimmt, wird θ mit Hilfe einer Optimierungsfunktion angepasst. Dies geschieht meist mit einem Gradientenabstiegsalgorithmus oder einer Variante davon. Der Gradientenabstieg wird durch Berechnung des

Gradienten der Verlustfunktion und Subtraktion von den entsprechenden Parametern durchgeführt, die in Gleichung (4) dargestellt sind.

$$\theta_{i+1} := \theta_i - \eta \nabla_\theta L(\theta, x) \tag{4}$$

Dabei ist η eine Konstante, die als Lernrate bezeichnet wird und die dazu beiträgt, dass die Verlustfunktion mit Hilfe des Gradientenabstiegsalgorithmus auf 0 konvergiert. Sie ist einer von vielen Hyperparametern, die nicht automatisch während des Trainings angepasst, sondern vorher bestimmt werden. Eine optimale Einstellung von Hyperparametern ist eine wichtige Aufgabe, die schwer zu automatisieren ist und daher im Mittelpunkt vieler Forschungsarbeiten steht[1] und ist das Hauptthema des Kapitels 6. Wenn die Gleichung (2) für das jeweilige Modell zutrifft, ist sie anschließend für den gegebenen Input ausreichend geeignet.

Nachfolgend werden diese Konzepte anhand eines einfachen Beispiels vorgestellt.

2.2 Einfaches lineares Beispiel

Als illustratives Beispiel wird ein Modell erstellt, das Fahrenheit in Celsius übersetzt[2]. Die ursprüngliche Gleichung ist durch die lineare Funktion $y = mx+b$ als $F = C*1,8+32$ gegeben, mit F als Grad Fahrenheit und C als Grad Celsius.

Ein Modell, das diese Beziehung erlernen soll, ist in der Auflistung 1 definiert [3] .

Listing 1: Celsius zu Fahrenheit

```
const model = {
    w: [Math.random(), Math.random()],
    h(x) {
        return this.w.reduce((pre, cur, idx) => pre + x ** idx * cur, 0);
    }
}

function y(C) { return C * 1.8 + 32; }

function mse(h, y) { return (h - y) ** 2 / 2; }

function sgd(model, x) {
    model.w =  model.w.map((weight, idx) =>
        weight - (model.h(x) - y(x)) * x ** idx);
}
```

[1] Es gibt Algorithmen zur Anpassung der Lernrate, nämlich `AdaGrad` [DHS10] und abgeleitete Algorithmen.

[2] Bitte beachten Sie, dass die Verwendung eines Lernalgorithmus hier sehr ineffizient ist und im Gegensatz zu Maslows Hammer steht`https://aka.klawr.de/srp#2`

[3] Auch verfügbar unter `https://aka.klawr.de/srp#3`

```
for (i = 0; i < 200; ++i) {
    const C = Math.random() * 100;
    sgd(model, C / 100);
    if (!(i % 20)) console.log('loss: ', mse(model.h(C), y(C)));
}

console.log('weights: ', model.w);
```

Das beschriebene Modell ist durch `w` und die Hypothesenfunktion durch `h` implementiert. Da bekannt ist, dass die Beziehung linear ist, stellt das Modell ein eindimensionales Polynom dar, wobei der erste Index dimensionslos ist (und wie bereits erwähnt den Bias emuliert) und der zweite als Input x verwendet wird. Wir werden später eine Dimension hinzufügen, um das Verhalten bei Polynomen zu demonstrieren, die nicht unmittelbar die Zielfunktion darstellen.

`y` liefert die richtige, vorgegebene Lösung. Bitte beachten Sie, dass `y` nur während des Trainings verwendet wird und weggelassen wird, wenn die Modellparameter nach dem Training richtig eingestellt sind.

In diesem Beispiel wird die Verlustfunktion durch den *mean square error* beschrieben [4]

Zur Minimierung der in Gleichung (4) beschriebenen Verlustfunktion wird der Gradientenabstieg verwendet.

In listing 1 wird Gleichung (4) als `sgd` mit $\theta_0 =$ `b` und $\theta_1 =$ `m` implementiert, wie in Gleichung (5) gezeigt.

`sgd` ist die Abkürzung für *stochastic gradient descent*(zu deutsch: stochastischer Gradientenabstieg)[5]. Sie stellt die Optimierungsfunktion dar, die die Parameter des Modells aktualisiert, indem sie den Gradienten des Verlusts für jeden einzelnen Parameter berechnet und sie anschließend aktualisiert.

$$\begin{aligned} \theta_0 &:= \theta_0 - \frac{\partial L}{\partial \theta_0} = \theta_0 - (h(x) - y(x)) \\ \theta_1 &:= \theta_1 - \frac{\partial L}{\partial \theta_1} = \theta_0 - (h(x) - y(x)) * x \end{aligned} \tag{5}$$

Die jeweilige Ausgabe sieht dann in etwa so aus:

Listing 2: Output des C to F Konverters

[4] abgekürzt als *mse*, welche in Gleichung (3) gezeigt wird.

[5] Es wird unterschieden zwischen Batch, Mini-Batch und stochastischem Gradientenabstieg; unter Verwendung eines vollständigen Datensatzes, eines definierten Teilsatzes oder einzelner Daten auf die einzelnen Iterationen des Trainingsprozesses.

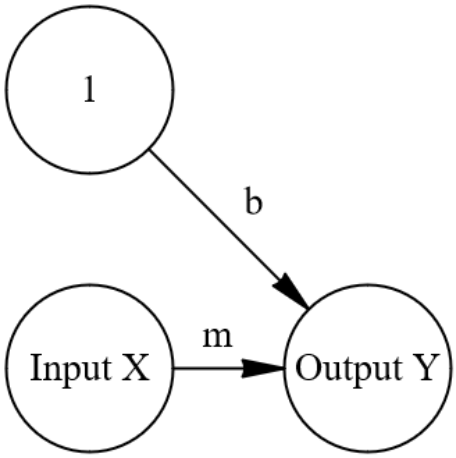


Abbildung 3: Das vorliegende Beispiel kann wie hier dargestellt visualisiert werden. Der Bias kann als zusätzlicher Input interpretiert werden, welcher stets den Wert 1 hat. Die Eingabe wird mit den Parametern multipliziert, die mit den entsprechenden Pfeilen gekennzeichnet sind, die wiederum aufsummiert werden (daraus resultiert $y = mx + b$).

```
loss:  343041.666256673
loss:  16242.7707476548
loss:  88.27595280422442
loss:  10.74432403839251
loss:  1.1607137779951606
loss:  0.252186283500194
loss:  0.00040450626910586374
loss:  0.00004247033303206121
loss:  7.158429555712516e-7
loss:  1.6948019668728757e-10
weights:  [ 31.99999973758735, 1.8000003977458756 ]
```

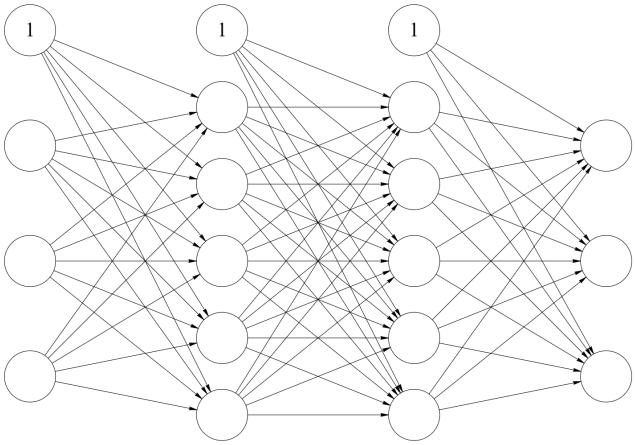
Das erwartete Verhalten für die Verlustfunktion ist, bei jeder Iteration abzunehmen. In diesem Beispiel ist dies der Fall und damit wird gezeigt, dass die Abweichung der Hypothese vom tatsächlichen Ergebnis abnimmt. Nach 200 Iterationen werden die Parameter `m` und `b` des `Modells` ausgegeben und zeigen, dass sie tatsächlich nahe am erwarteten Ergebnis liegen. Durch Erhöhen der Iterationszahl kann das Ergebnis bis zu dem Punkt erhöht werden, an dem sie vom JavaScript-Compiler gerundet werden, um exakte Ergebnisse darzustellen (unter Verwendung von Node.js[6]).

Vorausgesetzt, dass diese Hypothese anfangs gut für die Aufgabe geeignet ist, ist es wichtig zu beachten, dass durch die Modifizierung der Hypothese zur Darstellung eines Polynoms höheren Grades (z.B. $y = nx^2 + mx + b$) die zusätzlichen Parameter gegen 0 konvergieren, was effektiv das gleiche Ergebnis wie zuvor liefert [7]. Das Experiment kann

[6] Soweit bei der Berechnung exakter Ergebnisse unter Verwendung von Fließkommazahlen eben angenommen werden können.

[7] Aber um ähnliche Ergebnisse zu erzielen, muss die Iterationszahl mindestens um den Faktor 20 erhöht werden.

Abbildung 4: Ein neuronales Netzwerk mit drei Eingangsknoten, zwei verborgenen Schichten mit je fünf Knoten und einer Ausgangsschicht mit drei Knoten. Der Bias wird jeder Schicht hinzugefügt, indem ein Knoten mit dem Wert eins vorangestellt wird. Jeder Knoten wird, wie im vorigen Beispiel, durch die Summe des Produkts der vorherigen Schicht und ihres jeweiligen Gewichts berechnet.



unter `https://aka.klawr.de/srp#4` nachvollzogen werden.

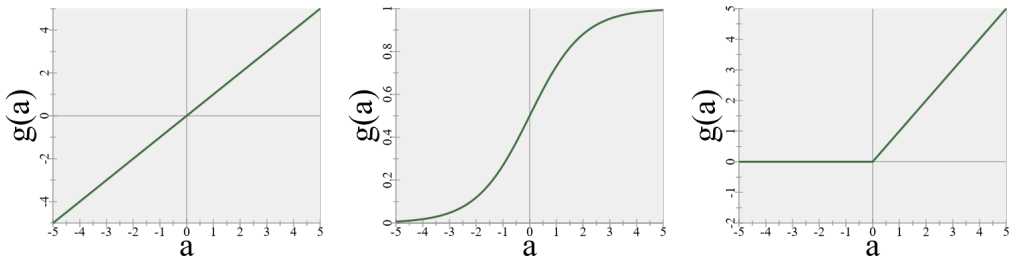
3 Neuronale Netzwerke

Wie bereits erwähnt, hat sich der Name Neuronales Netzwerk durchgesetzt, obwohl es weder neuronal noch ein Netzwerk ist. Sie werden verwendet, um komplexere Zuordnungen zu erstellen als die bisherigen Modelle in der Lage waren. Im Folgenden wird erörtert, was die Motivation hinter diesem Konzept ist und wie Netzwerke trainiert werden.

3.1 Deep Learning

Auch wenn das universelle Approximationstheorem zeigt, dass alle kontinuierlichen Funktionen mit nur einer Schicht beschrieben werden können, ist es in der Praxis viel sinnvoller, mehrere Schichten zu unserem Modell hinzuzufügen, um die erforderlichen Ressourcen gering zu halten. Rolnick und Tegmark [RT17] zeigen, dass bei einschichtigen Modellen die Anzahl der Neuronen exponentiell mit der Anzahl der Variablen des Inputs wächst, während mehrschichtige neuronale Netzwerke lediglich linear wachsen.

Jeder Knoten in Abbildung 4 kann durch die Notation z_i^l angesprochen werden. l ist die Nummer der Schicht, in welcher der Knoten liegt, und i ist der Index des Knotens. Jedes Gewicht $\theta_{i,j}^l$ kann ebenfalls adressiert werden; l ist die Schicht, auf welche die



(a) Linear (b) Logistischer Sigmoid (c) ReLU

Abbildung 5: Aktivierungsfunktionen

Gewichtung verweist, i ist der Index des Knotens, der mit der Gewichtung multipliziert wird, um den Wert des Knotens in der Schicht l mit dem Index j zu beeinflussen. So können wir jeden Knoten als Resultat folgender Gleichung beschreiben[8]:

$$z_j^l = \sum_{i=0}^{n} z_i^{l-1} \theta_{i,j}^l \tag{6}$$

n ist die Anzahl der Knoten in der jeweiligen Schicht. Gleichung (6) kann vektorisiert werden zu:

$$z^l = \theta^l z^{l-1} \tag{7}$$

3.2 Aktivierungsfunktionen

Ein Problem bei den Werten der einzelnen Zellen ist, dass sie in beliebiger Größe erfolgen können. Numerisch gesehen ist dies sehr schwer zu kontrollieren, so dass die Knoten selbst wiederum zuerst an eine andere Funktion gegeben werden; diese Funktion wird passenderweise als Aktivierungsfunktion bezeichnet.

Die Aktivierungsfunktionen werden auf unterschiedliche Weise genutzt und bieten weitere Möglichkeiten der Anpassung durch weitere Parameter, um die Modellleistung zu erhöhen. Bisher wurde die in Abbildung 5a gezeigte lineare Funktion mit der Abbildung $g(a) = a$ angenommen, in welcher der Eingang und Ausgang äquivalent sind. Der logistische Sigmoid wird in der Praxis oft verwendet, da es beliebige Eingaben in das Intervall $[0, 1]$ verteilt. Die "rectified linear unit"(ReLU) ist rechnerisch erheblich simpler und wird für die Verwendung mit den meisten vorwärts gerichteten neuronalen Netzen [GBC17,

[8] In der Literatur wird der Bias oft separat hinzugefügt. Gemäß der Konvention, dem Eingangsvektor x eine 1 voranzustellen, wird er immer als Index 0 jeder Ebene aufgenommen. Bitte beachten Sie, dass dies mathematisch gesehen dasselbe ist.

S.169][GBB11] empfohlen. Mathematische Darstellungen werden in den Gleichungen (8) gezeigt.

$$\begin{aligned} Sigmoid : f(x) &= \frac{1}{1+\exp^{-x}} \\ ReLU : f(x) &= max(0,x) \end{aligned} \tag{8}$$

Die Verwendung der Aktivierungsfunktionen kann ebenfalls variieren; z.B. ist es üblich, dass das logistische Sigmoid einen Schwellenwert hat, um überhaupt nicht zu feuern, wenn ein bestimmter Wert (z.B. 0,5) nicht erreicht wird.

Durch die Einführung von Aktivierungsfunktionen belaufen sich die Gleichungen (6) und (7) zu:

$$a_j^l = \sigma(z_j^l) = \sigma(\sum_{i=0}^{n} z_i^{l-1}\theta_{i,j}^l) \tag{9}$$

Mit der entsprechenden vektorisierten Gleichung:

$$a^l = \sigma(z^l) = \sigma(\theta^l z^{l-1}) \tag{10}$$

Es gibt eine Vielzahl verschiedener Aktivierungsfunktionen, wobei diese beiden am prominentesten sind. Die Auswahl der Aktivierungsfunktion und eventuell ein entsprechender Schwellenwert sind ein weiterer Hyperparameter, der vor dem Training gewählt werden muss.

3.3 Backpropagation

Beim Gradientenabstieg (4) können nicht alle Parameter in allen Ebenen aktualisiert werden, da er nur die Aktualisierung der Parameter der letzten Ebene berücksichtigt, wenn sie als unabhängig vorangegangener Ebenen betrachtet wird. Da die letzte Schicht eine Funktion der jeweils vorhergehenden Schicht ist, kann die Kettenregel angewendet werden und so der berechneten Fehler der letzten Schicht auf die Vorangegangene übertragen und die Gewichte bzw. den Bias aktualisieren. Dieser Umstand gilt für jede Schicht, mit Ausnahme der Eingangsschicht (die schließlich nicht aktualisiert werden muss). Die folgenden Gleichungen wurden mit [Stu18, S.733], [GBC17, S.197] und [Nie15, ch.2] zusammengestellt:

$$\Delta^L = \nabla_a L \odot \sigma'(z^L) \tag{11}$$

$$\Delta^l = ((\theta^{l+1})^T \Delta^{l+1}) \odot \sigma'(z^l) \tag{12}$$

$$\theta_{i+1} := \theta_i - \eta\Delta \tag{13}$$

Gleichung (11) beschreibt den in der Ausgabeschicht berechneten Fehler. Wir definieren unser Netzwerk mit L-Schichten, daher wird die Ausgabeschicht immer durch den Index L beschrieben. $\odot$ ist der Hadamard-Operator. Er wird verwendet, um Tensoren elementweise zu multiplizieren, was in diesem Fall nützlich ist, da er verhindert, dass die Vektoren in Diagonalmatrizen transformiert werden müssen. Gleichung (11) ist ein direktes Ergebnis der Anwendung der multivariaten Kettenregel auf die Ableitung der in der letzten Schicht durchgeführten Verlustfunktion. Wie bereits erwähnt wird der Gradientenabstieg durch die Ableitung der Verlustfunktion in Gleichung (5) durchgeführt. Da zwischen dem in der Verlustfunktion verwendeten Eingang und dem Ausgang der vorhergehenden Schicht eine Aktivierungsfunktion implementiert ist, muss die Kettenregel auch auf diese angewendet werden.

$$\frac{\partial L}{\partial z_j^L} = \frac{\partial L}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \tag{14}$$

Indem man darauf hinweist, dass $\frac{\partial L}{\partial a_i^L}$ nur eine nicht vektorisierte Form von $\nabla_a L$ und $\frac{\partial a_i^L}{\partial z_i^L}$ von $\sigma'(z^L)$ ist, wird die Äquivalenz deutlich.

Gleichung (12) kann folgendermaßen umgeschrieben werden:

$$\Delta_i^l = \frac{\partial L}{\partial z_i^l} = \sum_{j=0}^{n} \left(\frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l}\right) \tag{15}$$

n ist die Anzahl der Knoten in der nachfolgenden Schicht. Aus $\frac{\partial L}{\partial z_i^{l+1}} = \Delta_i^{l+1}$ folgt:

$$\frac{\partial L}{\partial z_i^l} = \sum_{j=0}^{n} \left(\Delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_i^l}\right) \tag{16}$$

Die folgende Gleichung ist ebenfalls bekannt:

$$\begin{aligned} z_j^{l+1} &= \sum_{i=0}^{n} \theta_{i,j}^{l+1} a_i^l = \sum_{i=0}^{n} \theta_{i,j}^{l+1} \sigma(z_i^l) \\ \frac{\partial z_j^{l+1}}{\partial z_i^l} &= \theta_{i,j}^{l+1} \sigma'(z_j^l) \end{aligned} \tag{17}$$

So können wir die Begriffe in Gleichung (16) substituieren:

$$\frac{\partial L}{\partial z_i^l} = \sum_{j=0}^{n} (\Delta_j^{l+1} \theta_{i,j}^{l+1} \sigma'(z_j^l)) \tag{18}$$

Was wiederum nur eine nicht vektorisierte Form der Gleichung (12) ist.

Gleichung (12) wird anschließend auf jede Schicht ausgeführt, bis $l = 0$ erreicht ist und ein Δ für jeden Parameter in θ berechnet wird, die durch Gleichung (13) angewendet werden. Diese Art des Rückwärtsschreiten gab diesem Algorithmus den prominenten Namen der "backpropagation"(kurz für "backward propagation of error"[RHW86]).

3.4 Generalisierung

Die Aufgabe jedes Algorithmus für maschinelles Lernen ist es, zu generalisieren. Nachdem das Modell mit Daten trainiert wurde, sollte es auch mitDaten, die es noch nie gesehen hat, gute Resultate erzielen. Dies unterscheidet das maschinelle Lernen von Optimierungsproblemen (die z.B. mit der Newton-Raphson-Methode gelöst werden könnten).

Eine Generalisierung ist jedoch nicht gewährleistet. Insbesondere die Phänomene der Über- und Unteranpassung[9] stellen Probleme dar.

Beispielsweise könnte ein Modell, das für eine Klassifizierungsaufgabe trainiert wurde, korrekte Vorhersage für Trainingsbeispiele treffen, allerdings bei neuen Daten nicht mehr zuverlässig funktionieren (Überanpassung). Dies deutet darauf hin, dass das Modell die Trainingsbeispiele so zu sagen auswendig gelernt hat. Es gibt einige Dinge, die getan werden können, um dieses Problem zu lösen. Dem Modell mehr Daten zum Trainieren zu geben, ist eine Lösung, die von Banko und Brill [BB01] gezeigt und von Halevy, Norvig und Pereira in ihrem Artikel "The Unreasonable Effectiveness of Data"[HNP09] kommentiert wird. Das Hinzufügen einer Vielzahl von Daten ist nicht immer möglich, da die Generierung von Daten kostspielig sein kann, so dass möglicherweise einige Kompromisse eingegangen werden müssen, um dieses Problem zu lösen.

[9] Auch bekannt als Överfittingünd Ünderfitting"

Eine andere Möglichkeit ist die Reduzierung von Merkmalen der Trainingsdaten; scheinbar unintuitiv auf den ersten Blick wird deutlich, dass eine Fülle von Merkmalen einem Bild nicht viel Information hinzufügen. Um z.B. eine handgeschriebene Ziffer zu erkennen, müssen sehr viel mehr Parameter angepasst werden, wenn ein hochauflösendes Bild mit hunderttausenden von Pixeln in das Modell eingespeist wird, wobei eine Pixelanzahl von unter tausend bereits ausreichen würde [Nie15].

Auf der anderen Seite ist das Problem der Unteranpassung ebenso problematisch wie die Überanpassung. Wenn das Modell bereits keine ausreichende Ergebnisse bei den Trainingsdaten erhält, wird das Problem als Unteranpassung bezeichnet. Wenn dies der Fall ist, muss vermutlich das Modell oder Parameter angepasst werden, oder die Daten sind für das Problem unpassend. Des weiteren gilt, dass damit die Gleichungen (1) und (2) für ein vernünftiges Modell zutreffen, gilt als Faustregel, dass eine Abbildung des Inputs auf den erwarteten Output zumindest nachvollziehbar von einem Menschen sein sollte.

Damit anschließend mit diesen Techniken ein Modell auf die Erkennung mechanischer Symbole trainiert werden kann, müssen nun zunächst die entsprechenden Daten generiert werden.

4 Datengenerierung

Bevor Modelle für die Erkennung von mechanischen Symbolen erstellt werden können ist es notwendig, für diese Aufgabe geeignete Daten zu sammeln.

Ziel ist es, handgeschriebene mechanische Symbole zu lesen. Die Daten zum Trainieren des Modells sind an dieser Stelle notwendigerweise von mir selbst erstellt. Da die Datengenerierung zeitaufwendig sein kann, musste ein Weg gefunden werden die Daten möglichst zeiteffizient zu erstellen. Glücklicherweise hatte ich Zugang zu einem Tablet, welches es erlaubt, direkt auf dem Display zu zeichnen. Um die Daten schnell zu generieren, mussten einige Anforderungen berücksichtigt werden:

1. Ein Zeichenkontext wird erstellt.
2. Der Kontext ist in Größe und Farbe variabel.
3. Der Kontext soll in der Lage sein, Mausereignisse zu erkennen.
4. Das Skript sollte Zugriff auf das Dateisystem haben, um Daten auf der Festplatte zu sichern, ohne dass zusätzliche Arbeit erforderlich ist.

Ein Python-Skript ist in der Lage, diese Anforderungen zu erfüllen. Die Bibliothek `OpenCV` [Ope19] ermöglicht die Erstellung eines Fensters mit entsprechenden Anforderungen. Die Bibliothek `opencv-python` [Hei19] wird hier Implementierung von `OpenCV` verwendet. Es erfüllt all diese Anforderungen, da es die Erstellung eines Fensters mit programmierbarem Kontext und Funktionen zur Behandlung von Mausinteraktionen bereitstellt[10].

Da das Modell in der Lage sein soll, Symbole beliebiger Farbe auf beliebigem Hintergrund zu erkennen, ist der Kontext-Hintergrund ein zufälliger Wert in Graustufen, ebenso wie die Farbe der Linie mit der auf dem Hintergrund gezeichnet wird. Die Dicke der Linie, die zum Zeichnen verwendet wird, ist ebenfalls zufällig und imitiert damit das Erscheinungsbild eines Symbole unterschiedlicher Größen[11].

Nachdem die für das Zeichnen des Kontexts verwendeten Parameter festgelegt wurden, wird mit der Funktion `cv2` `namedWindow` die Funktion `cv2` erstellt. Für die jeweilige Interaktion mit der Zeichenfläche wird die Funktion `setMouseCallback` auf den erzeugten `namedWindow` angewendet, wobei eine `draw`-Funktion aufgerufen wird, die es ermöglicht, Mausereignisse zu erkennen und darauf zu reagieren.

Das Zeichnen erfolgt dann durch Ereignisse mit den Namen `cv2.EVENT_LBUTTONDOWN` und `cv2.EVENT_LBUTTONUP`, die ein boolesches `drawing`-Flag entweder auf `True` oder `False` setzen. Das Speichern des gezeichneten Bildes erfolgt mit `cv2.Event_RBUTTONUP`, der das Bild nach der Anzahl der zuvor automatisch gezeichneten Bilder benennt.

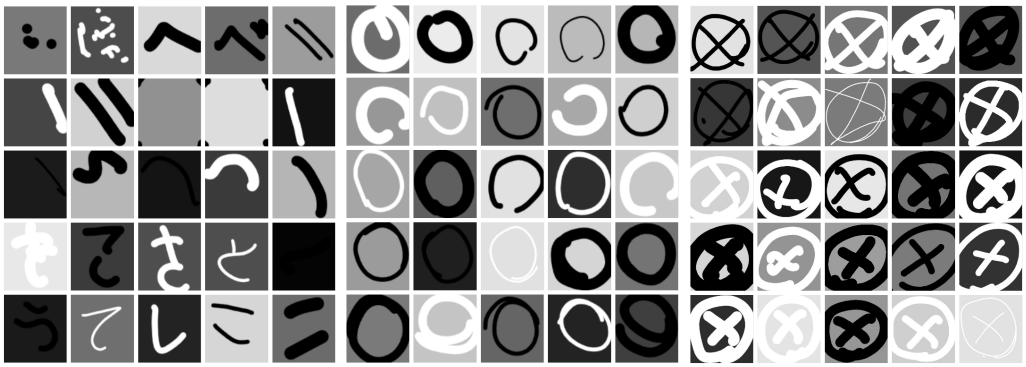
Die Bezeichnung für die verschiedenen Klassen wird am Anfang des Skripts gesetzt, wobei eine interaktive Eingabeaufforderung den Benutzer fragt, welche Klasse als nächstes gefüllt werden soll, mit den Optionen *"x"für Basis, ö"für Link, "n"für Nicht-Treffer*. Die Berücksichtigung von *Nicht-Treffer* ist besonders wichtig, da später bei der Suche nach einem beliebigen Bild die meisten Rückmeldungen voraussichtlich gar kein Symbol enthalten werden.

Mit diesem Skript werden 500 Symbole für jede Klasse erstellt. Es wird erwartet, dass diese Bilder später augmentiert werden, um einer Überanpassung entgegenzuwirken, aber für die Aufgabe der Unterscheidung von drei verschiedenen Klassen sollte diese Menge ausreichend sein[12] eingesehen werden.

[10] Der Datengenerierungsprozess sollte zunächst mit JavaScript im Webkontext durchgeführt werden, aber die letzte Anforderung zur Sicherung von Daten ist scheinbar nicht so einfach zu erfüllen. Node.js verfügt über einen Zugang zum Dateisystem, ist aber von Natur aus "headlessünd lässt daher keinen Kontext zu, auf den man zum Zeichnen zurückgreifen kann.

[11] Dies soll dazu dienen in einem späteren Prozess es auch zu ermöglichen Bilder unterschiedlicher Auflösungen zu klassifizieren

[12] Der Code kann unter `https://aka.klawr.de/srp#5`



(a) Keine Treffer (b) Nodes (c) Bases

Abbildung 6: Einige Beispiele für die mit der beschriebenen Methode erstellten Daten. Für die ersten Tests wurden nur diese drei Klassen erstellt. Die Bilder werden in einem Quadrat zentriert erstellt, so dass später beliebige Bilder gescannt werden können, wobei Quadrate unterschiedlicher Größe als Vorlagen verwendet werden können.

5 Erstes Modell

Das Jupyter-Notebook zu diesem Kapitel kann über `https://aka.klawr.de/srp#6` eingesehen werden.

5.1 Projektstruktur

Die Projektstruktur ist ein wichtiger Teil, um Modelle effizient zu testen und sie gegen andere Modelle zu evaluieren, um das beste Modell zu bestimmen. `deepmech` ist unter Verwendung einer modifizierten Version der Cookiecutter Data Science [dri19] Struktur aufgebaut. Sie kann unter `klawr.github.io/deepmech` eingesehen werden.

Der Stammordner enthält Dateien wie die Lizenz des gesamten Projekts. Das Projekt ist unter der MIT-Lizenz lizenziert, die jedem Menschen die Möglichkeit gibt, es für den privaten und kommerziellen Gebrauch zu verändern, zu verteilen oder zu nutzen, aber unter Ausschluss jeglicher Haftung oder Gewährleistung meinerseits. Im Stammordner befindet sich eine weitere Datei `requirements.txt`, die Teil des Installationsprozesses ist und es erlaubt, den gesamten in diesem Projekt behandelten Code zu replizieren.

Die "Cookiecutter Data ScienceStruktur ermöglicht eine klare Verteilung der Daten in Rohdaten (die nie angefasst werden dürfen), Zwischendaten (das sind die Rohdaten, die aber in irgendeiner Weise modifiziert, ergänzt oder verändert werden) und verarbeiteten Daten (die über einen Trainingsalgorithmus in das Modell eingespeist werden sollen). Diese Daten werden im Verzeichnis `data` gespeichert.

Logs werden während des Trainings mit `TensorBoard`[13] erstellt. Diese Protokolle werden durch einen Zeitstempel des jeweiligen Trainingslaufs unterschieden und befinden sich im Verzeichnis `logs`.

Die derzeit verwendeten Modelle befinden sich im Verzeichnis `models`. Die Modelle in diesem Verzeichnis können sich in verschiedenen Phasen des Projektes ändern, so dass die jeweiligen Berichte ihre eigenen Modellverzeichnisse haben, um die Zwischenmodelle zu speichern.

Im `reports` Verzeichnis befindet sich jeweils einem Ordner für jedes in diesem Projekt erstellten Berichten, einschließlich dieses Artikels[14]. Der `notebook`-Ordner in diesem Bericht enthält die Jupyter-Notebooks [Jup19], die in den unterschiedlichen Kapiteln genutzt wurden. Die Modelle werden ebenfalls innerhalb von `Jupyter` Notebooks trainiert. Jupyter notebooks sind interaktive Python-Code-Editoren, die eine einfache Möglichkeit bieten, effizient Code zu schreiben, zu testen und (mittels Markdown) zu kommentieren. Alle Notebooks mit entsprechendem Code befinden sich im Verzeichnis `notebooks` im Stammverzeichnis dieses Projekts[15]. Das jeweilige Verzeichnis der Ausarbeitungen enthält auch den verwendeten Code, die trainierten Modelle, Bilder und die für die Erstellung dieses Berichts erstellten Dokumente.

`src` enthält den gesamten Code (mit Ausnahme der in den Berichten verwendeten Demos), der bei der Entwicklung und dem Training der Modelle verwendet wurde. Sie reichen von Skripten, die geeignete Umgebungen zum Trainieren der Modelle schaffen, bis hin zur Datengenerierung und -augmentierung.

5.2 Laden von Daten

Bevor das Training des Modells beginnen kann, müssen die Daten geladen werden. Wie erwähnt werden die Daten in diesem Projekt im Verzeichnis `raw` des Verzeichnisses `data` gespeichert. Es ist eine gute Praxis, nie direkt mit den Rohdaten zu arbeiten, daher enthält der Beginn jeder Trainingseinheit eine Vorbereitungsphase, in der die Rohdaten in das `processed` Datenverzeichnis kopiert werden (eventuell mit Erweiterungen oder Änderungen in Zwischenschritten).

Demnach beginnt das Training eines Modells in diesem Projekt stets mit folgendem Code:

Listing 3: Laden der Daten.

[13] TensorBoard ist das Visualisierungs-Toolkit von `TensorFlow`. Die mit `TensorBoard` erzeugten Visualisierungen werden später gezeigt.
[14] Versehen mit dem Namen `srp`, für Student Research Project
[15] `https://aka.klawr.de/srp#7`

```
from os.path import join

raw = join('data', 'raw')
interim = join('data', 'interim')
processed = join('data', 'processed')

from src.utils import reset_and_distribute_data

reset_and_distribute_data(raw, processed, [400, 0, 100])
```

Für das erste Modell reichen die Rohdaten selbst aus; daher werden sie einfach mit einer vordefinierten Funktion `reset_and_distribute_data` aus `src/utils.py` in das Verzeichnis der verarbeiteten Daten kopiert. Der dritte Parameter von `reset_and_distribute_data` ist ein Array mit drei Zahlen, die die Verteilung der Trainings-, Validierungs- und Testdaten beschreiben. In diesem Beispiel werden die Validierungsdaten weggelassen, da keine Anpassungen an den Hyperparametern vorgenommen werden.

Listing 4: Laden der Daten mittels Generator-Funktion.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

def create_generator(data_dir, batch_size):
    datagen = ImageDataGenerator(rescale=1./255)
    full_path = join(processed, data_dir)
    return datagen.flow_from_directory(
        full_path,
        target_size=(32, 32),
        batch_size=batch_size,
        class_mode='binary')

train_generator = create_generator('train', 20)
test_generator = create_generator('test', 10)
```

Nachdem die entsprechenden Daten auf die verschiedenen Verzeichnisse verteilt sind, wird der `ImageDataGenerator`[16] verwendet, um dem Modell zu ermöglichen, die Daten später während des Trainingsprozesses zu laden. Die Generatoren definieren auch die Größe des geladenen Bildes, die auf 32 Pixel Breite und 32 Pixel Höhe[17] (im Gegensatz zur ursprünglichen Bildgröße von 512x512) und eine Batch-Größe[18] von 20 eingestellt. Der `class_mode` wird auf `'binary'` gesetzt, da die Eingabe aus 1D Numpy-Arrays be-

[16] `https://aka.klawr.de/srp#8`

[17] Die Größe der Bilder wird manuell bestimmt, indem man davon ausgeht, dass der Algorithmus, solange ein Mensch in der Lage ist, die charakteristischen Merkmale des Bildes zu erkennen, diese erlernen kann.

[18] Die Batch-Größe ist die Anzahl der Bilder, die zwischen jedem Backpropagationszyklus verwendet werden. Wenn die Batch-Größe mit der Datensatzgröße übereinstimmt, spricht man von einem Batch-Gradientenabstieg, wenn die Batch-Größe eine ist, von einem stochastischen Gradientenabstieg gesprochen. Alles dazwischen wird als Mini-Batch-Gradientenabstieg bezeichnet.

steht. Diese Generatoren erkennen auch die Struktur der Rohdaten (verteilt in jeweiligen Verzeichnissen unter Verwendung ihres Labels als Verzeichnisname) und weisen ihnen dadurch die entsprechenden Labels zu.

5.3 Modellkonstruktion

Listing 5: Definition des ersten Modells.

```
from tensorflow.keras import layers
from tensorflow.keras import models

model = models.Sequential()
model.add(layers.Flatten(input_shape=(32, 32, 3)))
model.add(layers.Dense(32,'relu'))
model.add(layers.Dense(32,'relu'))
model.add(layers.Dense(3, 'softmax'))

model.summary()
```

Alle in diesem Projekt verwendeten Modelle sind sequentielle Modelle. Sequentielle Modelle propagieren das Ergebnis jeder Schicht in einer Richtung auf die nächste Schicht.

`Keras` erfordert, dass die erste Schicht eine `input_shape` hat. Die Eingabeform des ersten Modells muss der tatsächlichen Größe der Eingabedaten entsprechen. Alle anderen Schichten sind in der Lage, die Form der vorherigen Schicht herzuleiten, so dass keine weiteren Definitionen erforderlich sind.

Schichten können durch die Verwendung einer bereitgestellten `add`-Funktion hinzugefügt werden, die Schichten als Parameter akzeptiert, die anschließend dem Modell hinzugefügt werden.

Es werden zwei Schichten hinzugefügt, die beide ”Dense Layers”mit jeweils 32 Knoten sind. Die ”Dense Layers”werden definiert, indem alle Knoten der vorherigen Schicht über Gewichte und Biases mit den Knoten ihrer eigenen Schicht verbunden werden und sich wie die in diesem Projekt verwendeten ”traditionellenSSchichten verhalten. Die beiden verborgenen Schichten werden mit der `'ReLU'`-Aktivierung aktiviert, wie in [GBC17, S.168] vorgeschlagen.

Die letzte Schicht hat drei Knoten in Bezug auf drei Klassen, denen die Daten zugeordnet werden können. Es wird die Aktivierung `'softmax'` verwendet, die die Werte der Schicht so anpasst, dass sie sich zum Wert eins summieren. Daher kann der Wert des Knotens in der Ausgangsschicht als ein Maß für die Zuversicht der Bewertung des Modells für eine bestimmte Eingabe angesehen werden. Der Knoten mit dem höchsten Wert wird entsprechend als das angenommene korrekte Ergebnis angenommen.

Eine Zusammenfassung des Modells wird gegeben als:

Listing 6: Summary des ersten Modells.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0
_________________________________________________________________
dense (Dense)                (None, 32)                98336
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 99
=================================================================
Total params: 99,491
Trainable params: 99,491
Non-trainable params: 0
_________________________________________________________________
```

Hier sehen wir, dass das θ des Modells 99491 Parameter hat, die angepasst werden können. Diese Anzahl ist die Summe der Parameter, die durch die 3 Schichten des Modells gegeben sind. Die erste Schicht hat die abgeflachten 3072 ($32 \times 32 \times 3$) Knoten. Die zweite Schicht multipliziert diese Anzahl ($+1$ für den Bias) mit 32 und hat daher zusätzliche 98336 Knoten. Die dritte Schicht ist das Produkt aus $32 + 1$ Knoten in der zweiten Schicht und 32 Knoten in der dritten Schicht, was uns 1056 Parameter zur Anpassung liefert. Die letzte Schicht hat wiederum $(32 + 1) \times 3$ Knoten.

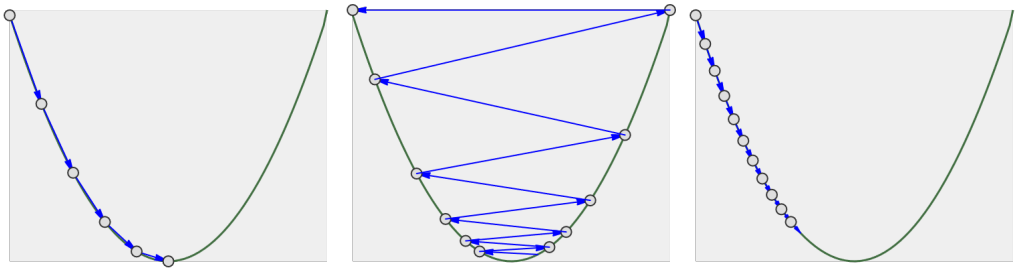
5.4 Optimierer

Um unser Modell zu trainieren, muss ein Optimierer definiert werden. Wir werden mit einem SGD-Optimierer beginnen, was die Abkürzung für `Stochastic Gradient Descent` ist. Stochastischer Gradientenabstieg wird bereits in Kapitel 2.2 erwähnt, wo er sehr rudimentär implementiert wurde. In dem eigentlichen Training des Modells wird die Keras-Implementierung von `SGD` verwendet. Die Keras-Implementierung von SGD unterscheidet sich etwas von der allgemeinen Definition, da sie nicht von Natur aus eine Batch-Größe von eins annimmt, sondern die im jeweiligen Generator definierten Batch-Größen, so dass der Anwender entscheiden kann, ob er Full Batch, Mini Batch oder stochastischen Gradientenabstieg verwendet[19].

Listing 7: Definition einer Optimierungsfunktion in Keras.

```
optimizer = SGD(lr=0.01, momentum=0.9, nesterov=True)
```

[19] Die hier verwendete Batch-Größe beträgt 20.



(a) Lernrate richtig (b) Lernrate zu hoch (c) Lernrate zu niedrig

Abbildung 7: Wenn die Lernrate richtig ist, nähert sich der Wert der Verlustfunktion in einer akzeptablen Anzahl von Schritten einem lokalen Optimum an. Ist er zu hoch, divergiert er. Ist die Lernrate zu niedrig, konvergieren die Werte, aber es sind zu viele Schritte nötig, um dies zu erreichen. Bitte beachten Sie, dass diese Bilder den Parameterraum sehr vereinfacht darstellen, da die dargestellte Funktion vollständig konvex und nur zweidimensional ist.

Die Funktion `SGD` benötigt drei Parameter: Die Lernrate `lr`, `momentum` und `nesterov`. Diese Werte sind wiederum Hyperparameter, die vor dem Training bestimmt werden müssen.

5.4.1 Lernrate

Die Lernrate ist einer der einflussreichsten Hyperparameter und wohl auch der bedeutendste, da er in fast jedem Optimierer verwendet wird. Sie ist ein skalarer Wert, der die Schrittweite der durchgeführten Schritte bestimmt. Die Aktualisierungsfunktion des Parameters θ ist wie bereits erwähnt:

$$\theta_{i+1} := \theta_i - \eta\Delta \qquad \text{(13 wiederholt)}$$

.

Wobei η der Wert der Lernrate ist.

Wenn die Lernrate auf einen zu kleinen Wert gesetzt wird, benötigt die Verlustfunktion viele Iterationen, um auf einen ausreichend kleinen Wert zu konvergieren. Andererseits kann eine auf einen hohen Wert gesetzte Lernrate überhaupt nicht konvergieren.

5.4.2 Momentum

Eingeführt 1964 von Polyak [Pol64] wird Momentum genutzt, um den Gradienten über mehrere Stufen hinweg konstant zu halten. Das Momentum bekommt seinen Namen durch eine Analogie zum physikalischen Effekt, nach dem zweiten Newtonschen Bewe-

gungsgesetz [New87, S.12]. Die Richtung des Gradienten sollte über mehrere Schritte konsistent sein, und es wäre merkwürdig, wenn die Richtung während des Trainings plötzliche Sprünge macht. Dies ist nützlich, wenn die Daten "noisyßind oder einige der Trainingsbeispiele falsch sind, da ihr negativer Effekt durch das Momentum entgegengewirkt wird. Daher wird ein weiterer Parameter ν in die Gleichung (13) eingeführt:

$$\begin{aligned} \nu_i &:= \alpha\nu_{i-1} - \eta\Delta \\ \theta_{i+1} &:= \theta_i + \nu_i \end{aligned} \tag{19}$$

Dagegen ist α ein weiterer Parameter, der eingestellt werden kann, um den Einfluss von η auf die Parameter der nächsten Iteration zu regulieren. Wenn also ein Trainingsbeispiel die "Richtung"des nächsten Schrittes mit einer hohen Abweichung vom Median der letzten Schritte ändern würde, würde die Auswirkung reduziert und die Konvergenzrate möglicherweise verbessert.

Im Jahr 2013 führten Sutskever, Martens, Dahl und Hinton [Sut+13] eine weitere Variante des Momentums ein, die von Nesterovs Accelerated Gradient [Nes83] inspiriert wurde. Diese Variante aktualisiert die zuvor diskutierte Gleichung, um den Parameter zur Berechnung von Δ unter Verwendung von ν zu ändern. Bitte beachten Sie, dass Δ in Gleichung (19) und (13) selbst eine Funktion des Parameters θ ist (siehe Gleichung (12)).

$$\begin{aligned} \nu_i &:= \alpha\nu_i - \eta\Delta(\theta_i + \alpha\nu_i) \\ \theta_{i+1} &:= \theta_i + \nu_i \end{aligned} \tag{20}$$

Die Idee des Momentums ist, dass der Richtungsvektor in die richtige Richtung zeigt, wobei die Verwendung des Nesterov-Beschleunigten Gradienten wahrscheinlich genauer ist, wenn man mit der Messung des nächsten Fehlers beginnt, der leicht in die jeweilige Richtung verschoben ist [Gér19, S.353] [GBC17, S.291].

5.4.3 Das tatsächliche Training

Nachdem der Optimierer definiert ist, wird das Modell durch den Aufruf der Modellkompilierfunktion kompiliert.

Listing 8: Kompilierung des Modells.

```
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=optimizer,
```

```
        metrics=['acc'])
```

`sparse_categorical_crossentropy` wird als Verlustfunktion verwendet, wie es für Mehrklassen-Kategorisierungsprobleme empfohlen wird, die durch ganze Zahlen [Cho17, S.84] gekennzeichnet sind. Sie misst den Abstand zweier Wahrscheinlichkeitsverteilungen, die durch die letzte Schicht des Modells, gegeben durch eine `softmax` Aktivierung und die beschrifteten Daten, gegeben sind. Die SoftMax-Aktivierung wird zur Normalisierung der Ausgabe verwendet, indem jeder Wert durch die Summe aller Werte in der jeweiligen Schicht geteilt wird. Dies ist nützlich, weil die beschrifteten Daten dem richtigen Knoten den Wert eins und allen anderen den Wert null zuweisen, wodurch sie numerisch vergleichbar werden und dadurch der Verlust für ein sinnvolles Ergebnis minimiert werden kann.

Das eigentliche Training erfolgt dann durch eine `Fit`-Funktion im Modell. Die Trainingsdaten werden durch Generatoren bereitgestellt; die Funktion ist in diesem Fall `fit_generator`. Daher wird das Training durch den Aufruf der folgenden Funktion durchgeführt:

Listing 9: Initialisierung des Trainings.

```
    history = model.fit_generator(
        train_generator,
        steps_per_epoch=20,
        epochs=20,
        callbacks=callbacks)
```

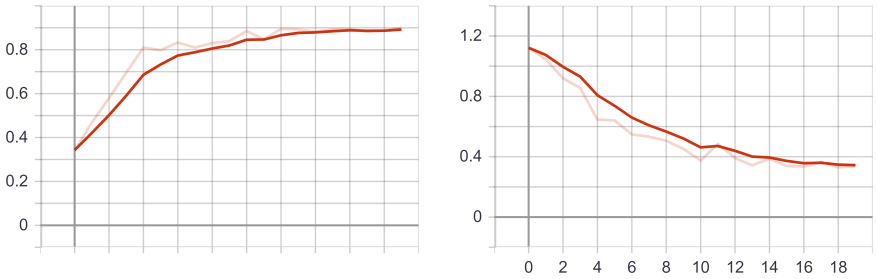
Hier wird definiert, dass bei jeder Iteration 20 Bilder als Batch in das Modell eingespeist werden, was für 20 Epochen wiederholt wird. Nach jeder Epoche aktualisiert der Optimierer die Parameter (die θ des Modells), was in diesem Durchlauf entsprechend 20 Mal geschieht.

5.4.4 Results

Wie erwartet, beginnt das Modell mit einer Genauigkeit von etwa 33%, indem es versucht, die Trainingsdaten in drei Klassen zu klassifizieren. Wie die Daten nahelegen, lernt das Modell schnell einige Beziehungen zwischen den Eingabedaten und dem jeweiligen Label, da es bereits nach nur vier Epochen eine Genauigkeit von fast 70% hat.

Listing 10: Resultate des Trainings (gekürzt).

```
    Epoch 1/20
    20/20 [=====...=====] - 2s 79ms/step - loss: 1.1220 - acc: 0.3425
    Epoch 2/20
    20/20 [=====...=====] - 1s 55ms/step - loss: 1.0457 - acc: 0.4675
```



(a) Accuracy (b) Loss

Abbildung 8: Die ”callbacks”des Trainingsalgorithmus liefern Logs, die von TensorBoard verwendet werden können, um nützliche Grafiken zur Bewertung des Modells bereitzustellen. Die vertikale Achse stellt den Wert der jeweiligen Epoche dar (auf der horizontalen Achse dargestellt).

```
Epoch 3/20
20/20 [=====...=====] - 1s 55ms/step - loss: 0.9199 - acc: 0.5800
Epoch 4/20
20/20 [=====...=====] - 1s 46ms/step - loss: 0.8548 - acc: 0.6950
...
Epoch 17/20
20/20 [=====...=====] - 1s 45ms/step - loss: 0.3340 - acc: 0.8950
Epoch 18/20
20/20 [=====...=====] - 1s 48ms/step - loss: 0.3648 - acc: 0.8825
Epoch 19/20
20/20 [=====...=====] - 1s 52ms/step - loss: 0.3291 - acc: 0.8875
Epoch 20/20
20/20 [=====...=====] - 1s 47ms/step - loss: 0.3388 - acc: 0.9000
```

Nach 20 Epochen hat das Modell eine Genauigkeit von etwa 90%, was bereits ein sehr guter Anfang für ein erstes Modell mit nicht optimierten Hyperparametern ist. Wichtig für ein funktionierendes Modell ist, wie gut das Modell Daten klassifiziert, die es noch nie zuvor gesehen hat. Um dies zu überprüfen, wird ein zweiter Generator erstellt: der `test_generator`, der zur Auswertung des Modells verwendet wird. Durch die Ausgabe von `model.evaluate_generator(test_generator)` erhalten wir `[0.6071663084129493, 0.7866667]`, der anzeigt, dass der Verlust bei `0.60` liegt, im Gegensatz zu `0.33` bei den Trainingsdaten und hat und eine Genauigkeit von 78% bei den Daten noch nie gesehen hat.

Dieses Ergebnis ist ziemlich bemerkenswert, aber offensichtlich schneidet das Modell bei neuen Daten schlechter ab, was darauf hindeutet, dass es bei den Trainingsdaten överfitted”. Ungeachtet dessen zeigt das Ergebnis, dass durch die Anpassung der Hyper-

parameter das Ergebnis auf ein zufriedenstellendes Niveau gesteigert werden kann.

Nachdem das Modell trainiert wurde, wird es in `models/symbol_classifier/first_model.h5` gespeichert und kann zur weiteren Überprüfung in Keras geladen werden.

6 Hyperparameter-Abstimmung

Leider gibt es nicht für jedes Problem ein offensichtlich optimales Modell. Ein Modell bildet den Input auf den Output ab, um jedoch zuverlässig zu bestimmen, welche Informationen in den Daten verworfen werden sollten (und somit nicht zur Generalisierung beitragen) und welche hervorgehoben werden sollten, müssen entsprechende Annahmen getroffen werden.

1996 wies David Wolpert darauf hin, dass es keinen Grund gibt, ein Modell einem anderen vorzuziehen, wenn keine Annahmen über den Datensatz getroffen werden. Diese Argumentationslinie ist heute als *No Free Lunch Theorem* bekannt [Wol96]. Einige dieser Annahmen werden bereits im Vorfeld getroffen (z.B. die Verwendung von Feed-Forward-Neuronalen Netzen), andere sind jedoch Gegenstand von Optimierungen.

Die Hyperparameter-Optimierung ist ein Optimierungsproblem, bei dem der Verlust (oder andere Metriken wie z.B. die Genauigkeit) des Modells für die gegebenen Trainingsdaten minimiert werden soll. Die im ersten Modell verwendeten Hyperparameter, die Gegenstand der Optimierung sein könnten, sind unter anderem:

1. Training data:
   a) Anzahl der Trainingsdaten
   b) Bildgröße
   c) Batch-Größe
   d) Reduzierung der Anzahl der Merkmale
2. Model:
   a) Anzahl der Schichten
   b) Typen der Schichten
   c) Größe der Schichten
   d) Wahl der Verlustfunktion
3. Optimizer:
   a) Wahl des Optimierers
   b) Lernrate
   c) Momentum
   d) Nesterov
4. Training:
   a) Schritte pro Epoche
   b) Anzahl der Epochen

Einige Hyperparameter müssen getestet werden und sind nicht offensichtlich, aber andere können ohne offensichtliche negative Nebenwirkungen verbessert werden, so dass sie hier zuerst behandelt werden. Danach werden Algorithmen zum Testen verschiedener Modelle mit unterschiedlichen Hyperparametern zur Verbesserung des Modells untersucht.

7 Augmentierung der Daten

Das erste, was verbessert werden kann, ist die Größe des Datensatzes, den wir für das Training des Modells verwenden. In den vorherigen Tests sahen die Modelle jedes Bild etwa 20 Mal während des Trainings (Batches der Größe 20, 20 Schritte pro Epoche und 20 Epochen geteilt durch 400 Trainingsproben), was höchstwahrscheinlich die Ursache für die auftretende Überanpassung ist. Die Größe des Trainingssets zu erhöhen, scheint also ein sinnvoller Ansatz zu sein.

Die Augmentierung der Daten funktioniert in diesem Fall über die Veränderung einiger Eigenschaften des Bildes, die für den jeweiligen Zweck sinnvoll sind. Im Falle der Symbolerkennung kann das Symbol gedreht oder gespiegelt werden, so dass Trainingsbeispiele hinzugefügt werden können, ohne dass der Datenbestand redundant wird und die Daten in diesem Fall in keiner Weise weniger gültig werden.

Die Datenerweiterung wird durch Anpassung der `create_generator`-Funktion durchgeführt:

Listing 11: Augmentierung der Trainingsdaten.

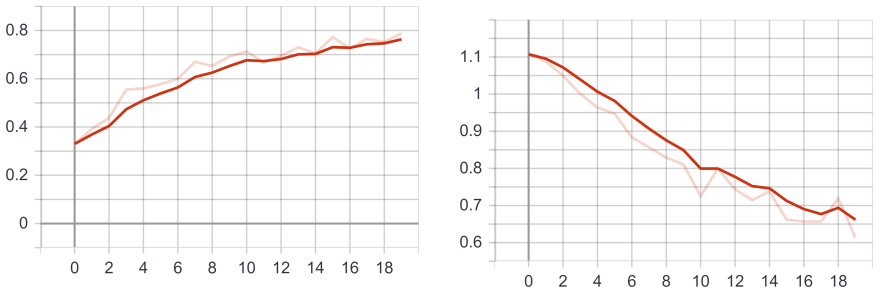
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

def create_generator(data_dir, batch_size, datagen):
    full_path = join(processed, data_dir)
    return datagen.flow_from_directory(
        full_path,
        target_size=(32, 32),
        batch_size=batch_size,
        class_mode='binary')

train_datagen = ImageDataGenerator(
        rescale = 1./255,
        rotation_range=360,
        horizontal_flip=True,
        vertical_flip=True)

test_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = create_generator('train', 20, train_datagen)
test_generator = create_generator('test', 10, test_datagen)
```



(a) Accuracy (b) Loss

Abbildung 9: Im Vergleich zu Abbildung 8 nehmen die Genauigkeit und der Verlust langsamer zu. Weitere Epochen können die Ergebnisse verbessern, da das Modell nach 20 Epochen noch nicht vollständig konvergiert zu sein scheint.

Anstelle von `datagen = ImageDataGenerator(rescale = 1./255)` wird dieses Objekt als Parameter angegeben, da das Trainingsset zufällige Änderungen an den geladenen Daten vornimmt. Nämlich eine zufällige Rotation und die Möglichkeit, horizontal und vertikal gespiegelt zu werden. Bitte beachten Sie, dass keine Änderungen auf die Testdaten angewendet werden.

Die Anwendung dieser Änderungen ergibt einen Verlust von 0,61 und eine 78,75%ige Genauigkeit der Trainingsdaten. Der Verlust und die Genauigkeit des Testsatzes betragen 0,63 bzw. 78%.

Es ist offensichtlich, dass die Genauigkeit der Trainingsdaten hier tatsächlich abnimmt. Dies ist nicht überraschend, da dem Modell keine Daten mehr zweimal gegeben werden, also keine Memorisierung stattfindet, aber da sich die Testgenauigkeit nicht geändert hat, scheint der gewünschte Effekt eingetreten zu sein. Die Modellgenauigkeit des Trainingssatzes scheint auch mit einer geringeren Rate zu steigen, so dass mehr Epochen oder größere Batches die Performanz bereits verbessern könnten.

Bitte beachten Sie, dass das Trainingsset praktisch um den Faktor 1440 (360 Grad + Drehungen in zwei Achsen) gewachsen ist, wodurch die Möglichkeit besteht, während des Trainings wesentlich mehr Daten in das Modell einzuspeisen, ohne mit einer zu großen Überanpassung rechnen zu müssen.

Das Ziel, die Überanpassung zu verringern, scheint in dieser Hinsicht gelöst zu sein; der Unterschied zwischen Trainings- und Testgenauigkeit ist vernachlässigbar klein[20].

[20]Link zum jeweiligen Notebook: `https://aka.klawr.de/srp#9`

7.1 Reduzierung der Merkmale

Eine weitere Sache, die die Leistung des Modells verbessern kann, ist die Reduzierung der Größe des Inputs. Das Phänomen, das als *Fluch der Dimensionalität* bekannt ist, beschreibt die Möglichkeiten unterschiedlicher Konfigurationen, die mit zunehmender Anzahl von Variablen möglich sind, und kommt von Richard Bellman, der sich zunächst auf Probleme in der dynamischen Programmierung [Bel57, p.ix] bezieht.

Die Daten werden bereits stark reduziert, indem die Daten als Bilder mit einer Breite und Höhe von 32 Pixeln eingeladen werden, obwohl die Daten roh als 512 mal 512 Bilder vorliegen. Diese Reduzierung wurde im Vorfeld manuell durchgeführt, um die Daten relativ klein zu halten, die Symbole aber noch gut zu unterscheiden sind.

Obwohl die Rohdaten nur Farben in Graustufen enthalten, werden sie mit drei Farbkanälen geladen (daher die Eingabegröße `[32, 32, 3]`). Eine Möglichkeit, die Eingabegröße zu reduzieren, besteht also darin, die Daten auf eine Farb-Dimension zu projizieren, wie von Géron [Gér19, S.215] vorgeschlagen. Der Aufruf des `flow_from_directory` wird also durch einem weiteren Parameter `color_mode='grayscale'` ergänzt um den 3-dimensionalen `color_mode='rgb'` (die als Standard eingestellt ist) auf eine Dimension zu reduzieren. Praktisch sollte sich also nichts ändern, bis auf die Eingabegröße, welche auf `[32, 32, 1]` angepasst wird.

Das resultierende Modell sieht entsprechend folgendermaßen aus:

Listing 12: Summary des verbesserten Modells.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 32)                32800
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 99
=================================================================
Total params: 33,955
Trainable params: 33,955
Non-trainable params: 0
_________________________________________________________________
```

Die Anzahl der trainierbaren Parameter wird fast durch den Faktor drei geteilt und damit die Größe des Modells von 807kb (von den Vorgängermodellen mit 99491 Parametern) auf 295kb reduziert. Hierbei geht dennoch keinerlei Genauigkeit verloren. [21].

[21] Link zum jeweiligen Notebook: `https://aka.klawr.de/srp#10`

7.2 Optimierer

Bisher wurde nur der Optimierer des stochastischen Gradientenabstiegs betrachtet. Der stochastische Gradientenabstieg hat mehrere Parameter, die sich optimieren lassen, z.B. die Lernrate, den Impuls und ob Nesterov verwendet wird oder nicht, wie bereits in Kapitel 5.4 behandelt.

Es wurden bereits mehrere Versuche unternommen, die Suche nach diesen Hyperparametern zu automatisieren. In diesem Projekt werden einige alternative Optimierer untersucht, aber nicht allzu ausführlich erläutert. Bitte beachten Sie die angegebenen Quellen für weitere Einblicke.

Beim stochastischen Gradientenabstieg ist die Lernrate für alle Parameter jeder Ebene des Modells gleich. Eine Anpassung für jeden Parameter oder sogar für einzelne Dimensionen der Schichten kann hilfreich sein, aber dies manuell zu tun würde einen unverhältnismäßig hohen Aufwand erfordern.

`AdaGrad` [DHS10] ändert die Lernrate für jeden Parameter. Dies geschieht durch Anpassung der Lernrate (beginnend mit einem einheitlichen Wert) für jeden einzelnen Parameter unter Verwendung der Größe des Gradienten für den jeweiligen Parameter. Während dieser Ansatz bei einigen Problemen gut funktioniert, ist er bei einigen Problemen auch nicht sinnvoll. Dennoch dient er als Grundlage für viele andere Optimierer, die nach diesem entwickelt wurden, von denen drei in dem in diesem Projekt verwendeten Tuning-Algorithmus getestet wurden.

`AdaDelta` [Zei12] ist ein modifizierter Algorithmus auf der Basis von `AdaGrad`, der versucht, AdaGrads Idee, die Lernrate konstant zu reduzieren, dadurch zu lösen, dass die Lernrate wieder durch einen exponentiell abklingenden Durchschnitt der Gradienten geteilt wird. Dies führt zu einem wesentlich stabileren Optimierer, der in der Praxis gut zu funktionieren verspricht.

`RMSProp` [Geo12] funktioniert ähnlich wie `AdaDelta` mit einer leicht abweichenden Aktualisierungsregel. Es wird als ïm Allgemeinen eine ausreichend gute Wahl, was auch immer Ihr Problem ist"[Cho17, S.77] betrachtet.

Adam [KB14][RKK18] ist der letzte in diesem Projekt enthaltene Optimierer, der eine weitere Methode zur Anwendung einer adaptiven Lernrate für jeden Parameter darstellt. Adam integriert die zuvor erwähnte Idee des Impulses in einen Algorithmus, der dem von `RMSProp` ähnlich ist.

Es gibt weitere erwähnenswerte Optimierer (nämlich AdaMax oder Nadam), die möglicherweise Gegenstand weiterer Untersuchungen sind.

Bemerkenswert ist, dass die hier erläuterten adaptiven Methoden möglicherweise nicht bei jedem Modell funktionieren, so dass es sich immer lohnt, den zugrundeliegenden

Gradientenabstieg mit Nesterov[Gér19, S.358] zu versuchen.

7.3 Convolutional Layers

"Convolutional Layersßind für die Bilderkennung weit verbreitet und es ist daher sinnvoll, sie in dieses Modell zu integrieren. Sie wurden von Yann LeCun 1998 in einer Arbeit [Lec+98] eingeführt, um handgeschriebene Ziffern zu erkennen.

In diesem Projekt werden nur zweidimensionale "Convolutional Layers"verwendet, da die Eingabe zweidimensional ist (nachdem die Bilder in Graustufen übertragen wurden). Der Unterschied der "Convolutional Layersßu den zuvor beschriebenen "Dense Layers"besteht darin, dass die trainierten Parameter nicht die Verbindungen aller Knoten der vorherigen Schicht zu einer nächsten darstellen, sondern einen Kernel, der über die Eingangsschicht gleitet und als Filter fungiert, der die Ausgangsschicht aufbaut. Dies bedeutet auch, dass die Eingangsschicht nicht flach gemacht werden muss, bevor sie in das Modell eingespeist wird.

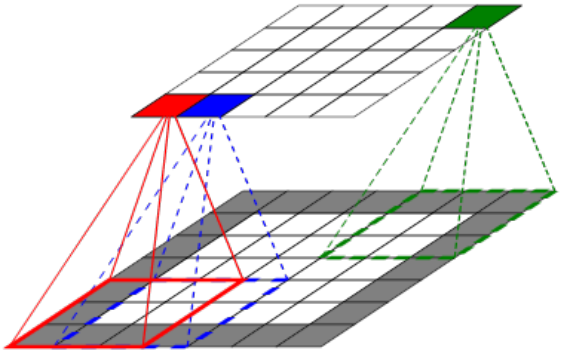
Beim Training eines "Convolutional Layersßind die trainierten Parameter die Filter, die eine vorgegebene Form haben. Eine zweidimensionales "Convolutional Layer"mit acht Filtern, einer Eingabetiefe von eins und einer Kernelgröße von vier mal vier (und einer für den Bias) hat 136 (8 x (1 x 4 x 4 + 1)) Parameter zu trainieren, was deutlich weniger als bei früheren Ansätzen ist und nicht von der Größe der Eingangsschicht abhängt, so dass es möglich ist, Muster in beliebig großen Bildern zu erkennen, ohne dass das Modell selber vergrößert werden muss.

Die entsprechende Gleichung für eine einzelne Aktivierung ist daher durch [Gér19, S. 453] gegeben:

$$z_{i,j,k} = b_k + \sum_{u=1}^{h} \sum_{v=1}^{w} \sum_{k'=1}^{n'} x_{i',j',k'} \times w_{u,v,k',k} \tag{21}$$

Dabei ist z der Aktivierungswert, wobei die Indizes i und j die Position des Knotens in der Ausgabeschicht und k den Index des Filters darstellen. Die Summe aller Eingabeknoten ergibt sich aus der Summe der Knoten in den jeweiligen Richtungen, die durch Iteration über drei Achsen unter Verwendung der Höhe (h), der Breite (w) und der Filtergröße (n') der vorhergehenden Ebene gegeben ist. Für jede dieser Kombinationen wird das jeweilige Verbindungsgewicht $w_{u,v,k',k}$ mit der Eingabe $x_{i',j',k'}$ an der jeweiligen Position in der vorhergehenden Schicht multipliziert, wobei $i' = imals_h + u$ bzw. $j' = jmals_w$ ist und s_h und s_w die Schritte sind, die bei jedem Schritt in jede Richtung unternommen werden. Die Schritte reduzieren dabei die Größe der nachfolgenden Schicht. Zusätzlich hat jeder Filter seinen eigenen Bias b_k, der zur Aktivierungsfunktion

Abbildung 10: Die unterste Ebene ist die Eingabe, die durch eine zweidimensionale Ebene in der Größe von fünf mal fünf (Filtergröße ist eins) repräsentiert wird, die mit einem Padding von eins (grau gezeichnet) versehen ist, um sicherzustellen, dass die Größe bei Verwendung eines Kernels der Größe drei mal drei (hervorgehoben durch die roten, grünen und blauen Rechtecke) gleich bleibt. Der verwendete Stride ist eins, d.h. das rezeptive Feld propagiert bei jedem Schritt einen Eingangsknoten. Hier beginnt der Prozess am roten Rechteck, blau ist der zweite Schritt und grün der letzte (insgesamt 25 Schritte).



hinzugefügt wird.

Damit die Ausgabe die gleiche Größe wie die Eingabe hat, müssen Zahlen zur Eingabe addiert werden, da ein Kernel, der in jeder Richtung größer als eins ist, eine Kante für die Eingabe nicht überqueren kann (und der Schritt 1 mal 1 sein muss). Daher wird das Padding zur Lösung dieses Problems verwendet, indem man (typischerweise) Nullen um die Eingabe herum anfügt, was zu einer Technik führt, die man Zero-Padding nennt.

Durch die Verwendung von "Convolutional Layersänstelle von "Dense Layers"wird die Modellgröße weiter reduziert (295kb bis 232kb) und die Genauigkeit um einige Prozente erhöht. Durch das Hinzufügen von Pooling wird die Größe weiter auf 149kb reduziert und die Genauigkeit wird weiter verbessert[22]. Pooling ist eine Technik, bei der ein Kernel (ähnlich wie der Kern von "Convolutional Layers") über die Eingabeschicht scannt. Im verknüpften Notizbuch wurde ein Kernel paarweise verwendet, der den maximalen Wert des Kernels nimmt und nur den maximalen Wert weiterleitet, wodurch die Ausgabegröße in Breite und Höhe halbiert wird[23].

[22] Verknüpfung mit den jeweiligen Notebooks: `https://aka.klawr.de/srp#11` und `https://aka.klawr.de/srp#12`

[23] Dies wird passenderweise `MaxPooling` genannt.

7.4 Tuning Algorithmen

Unter Berücksichtigung aller Möglichkeiten, das Modell vor dem Training zu optimieren, wurden einige Ansätze entwickelt, um die Hyperparameter automatisch zu optimieren. Eine Möglichkeit, die Hyperparameter einzustellen, besteht darin, jeden einzelnen mit verschiedenen Werten zu testen, wobei die Werte aller anderen Hyperparameter festgesetzt werden. Dies wird dann für alle Hyperparameter wiederholt. Dieses Verfahren stellt sicher, dass der optimale Satz von Hyperparametern verwendet wird, kann aber eine inakzeptable Zeit in Anspruch nehmen, da die Anzahl der erstellten Modelle exponentiell mit der Anzahl der anzupassenden Hyperparameter wächst. Bei zwei Hyperparametern wäre dies gleichbedeutend mit der Überprüfung jedes Wertes auf einem zweidimensionalen Gitter, daher heißt die Methode "grid search"[24].

Ein weiterer, in der Praxis häufig verwendeter Ansatz ist die Zufallssuche. Die Zufallssuche geht nach dem Prinzip, Hyperparameter nach dem Zufallsprinzip zuzuordnen und so verschiedene Modelle zu testen, ohne dass eine Kombination einen Vorteil bringt. Natürlich führt dies (höchstwahrscheinlich) nicht zu einem optimalen Ergebnis, aber in der Praxis wird dieser Ansatz häufig verwendet, da er in der Regel ein akzeptables Ergebnis liefert, indem er ein breites Spektrum möglicher Konfigurationen prüft. Ein Notizbuch wurde erstellt, um die Implementierung von `keras tuner` in diesem Projekt unter `https://aka.klawr.de/srp#13` zu zeigen.

Einige ausgefeiltere Methoden basieren auf dem Ansatz der Zufallssuche, bei dem die Ergebnisse der Zufallssuche wiederholt untersucht werden und dann die Hyperparameter zugunsten derjenigen abgestimmt werden, die bessere Modelle zu ergeben versprechen. Dieser Ansatz wird als Zoomen bezeichnet, da man nach jeder Iteration in den vielversprechendsten Bereich der Hyperparameter "hineinzoomen"kann.

Andere Ansätze versuchen, diesen Prozess vollständig zu automatisieren. Beliebte Algorithmen basieren auf der Bayes'schen Optimierung, aber diese werden in diesem Projekt nicht behandelt und daher ausgelassen.

Der in diesem Projekt verwendete Tuner heißt `Hyperband` [Li+18], der in der Bibliothek `keras-tuner` implementiert wird [Goo19a]. Im Wesentlichen trainiert `Hyperband` Modelle mittels Zufallssuche, schränkt aber das Training für ein breites Spektrum von Hyperparametern in einigen wenigen Epochen ein. Dies geschieht indem die Modelle für einige Epochen trainiert werden und nur die vielversprechendsten Modelle für weitere Iterationen behalten werden. Diese Methode verspricht eine wesentlich bessere Nutzung der Ressourcen und wird mit hoher Wahrscheinlichkeit gute Hyperparameter präsentie-

[24] Wenn nur ein Hyperparameter angepasst wird, nennt man das Verfahren "linear search".

ren können.

Der endgültige Modell-Tuner ist so konzipiert, dass er ein Hyperparameter-Objekt akzeptiert, das einer Funktion zugeführt wird, die das jeweilige Modell zurückgibt:

Listing 13: Funktion zum Suchen des besten Modells.

```
def create_model(hp):
    model = models.Sequential()
    model.add(layers.Conv2D(2**hp.Int('2**num_filter_0', 4, 6),
        (4,4) ,activation='relu', input_shape=(32, 32, 1)))

    for i in range(hp.Int('num_cnn_layers', 0, 3)):
        filter = 2**hp.Int('2**num_filter_' + str(i), 4, 7)
        model.add(
            layers.Conv2D(filter, (4,4), activation='relu',padding='same'))
        if hp.Boolean('pooling_' + str(i)):
            model.add(layers.MaxPooling2D(2, 2))

    model.add(layers.Flatten())
    for i in range(hp.Int('num_dense_layers', 1, 3)):
        nodes = 2**hp.Int('2**num_nodes_' + str(i), 4, 7)
        model.add(layers.Dense(nodes, activation='relu'))

    model.add(layers.Dense(3, 'softmax'))

    optimizers = {
        'adam': Adam(),
        'sgd': SGD(lr=hp.Choice(
            'learning_rate', [0.001, 0.003, 0.007, 0.01 0.03]),
            momentum=hp.Float('momentum', 0.6, 1, 0.1),
            nesterov=hp.Boolean('nesterov')),
        'rms': RMSprop(lr=hp.Choice(
            'learning_rate', [0.001, 0.003, 0.007,0.01, 0.03]))
    }

    model.compile(
        loss='sparse_categorical_crossentropy',
        optimizer=optimizers[hp.Choice('optimizer', list(optimizers.keys()),
        metrics=['acc'])

    return model
```

Diese Funktion erweitert 5 durch die Implementierung von Hyperparametern unter Verwendung des Hyperparameter von `keras tuner`. Dies wird später mit Hilfe einer benutzerdefinierten Klasse implementiert [25]:

Listing 14: Definition des Modell-Tuners.

```
tuner = customTuner(
```

[25] Die Ergebnisse wurden mit `TensorBoard` protokolliert, dessen API von `keras tuner` selbst noch nicht unterstützt wird, daher wird der Tuner von einer benutzerdefinierten Klasse vererbt

```
    create_model,
    hyperparameters=hp,
    objective='acc',
    max_trials=100,
    executions_per_trial=1,
    directory=log_dir,
    project_name=timestamp)
```

Das Training, das vorher durch die `fit`-Funktion durchgeführt wurde, wird nun durch die `search`-Funktion des Tuners durchgeführt:

Listing 15: Suchen des besten Modells.

```
tuner.search(
    train_dataset,
    validation_data=validation_dataset,
    epochs=30,
    steps_per_epoch=100,
    validation_steps=100,
    verbose=0,
    callbacks=callbacks)
```

Ein weiterer Hinweis, der hier zu beachten ist, ist, dass die in 7 verwendeten Generatoren durch eine manuelle Augmentierung der Daten und spätere Vorverarbeitung unter Verwendung von Protokollbuffern(`protobufs`) ersetzt wurden[26]. Durch die Vorverarbeitung der Bilddaten im Vorfeld wird die Ladezeit der Daten deutlich reduziert. Der dafür verwendete Code kann hier eingesehen werden: `https://aka.klawr.de/srp#15`.

Nach der Überprüfung der Daten konnte eine Genauigkeit von 96,7% auf die Testdaten bei einer Modellgröße von 4,3mb erreicht werden.

7.5 Vergrößerung des Trainingssets

Nachdem das Modell angepasst wurde, wurde noch ein anderer Ansatz versucht, um die Genauigkeit zu erhöhen. Die rohen Trainingsdaten wurden mehr als verdreifacht (auf 1000 Beispiele für jedes Symbol in den Trainingsdaten) und dann mit 32 Wiederholungen erweitert, was zu einem Datensatz von 96000 verschiedenen Bildern führte.

Bei Verwendung dieses Datensatzes wurde bei fast jedem zuvor getesteten Modell eine Genauigkeit von über 99% erreicht. Diese Situation präsentiert die Minimierung des Modells als wichtigstes Optimierungsziel. Das endgültige Modell wird manuell reduziert und abgestimmt, indem Schichten und Hyperparameter reduziert werden. Das endgültige Modell hat eine Größe von 207kb mit einer Genauigkeit von 99,3% und einem Verlust von 0.04 auf das Trainingsset, was zufriedenstellende Ergebnisse darstellt.

[26] Das endgültige Notizbuch, das diese verwendet, kann unter `https://aka.klawr.de/srp#14` eingesehen werden

Das Modell wird zusammengefasst als:

Listing 16: Definition des finalen Modells.

```
model = models.Sequential()
model.add(layers.Conv2D(16, (4,4), activation='relu', padding='same',
    input_shape=(32, 32, 1)))
model.add(layers.MaxPooling2D(2,2))
model.add(layers.Conv2D(32, (4,4), activation='relu', padding='same'))
model.add(layers.MaxPooling2D(2,2))
model.add(layers.Flatten())
model.add(layers.Dense(3, 'softmax'))

optimizer = Adam()
model.compile(loss='sparse_categorical_crossentropy',
    optimizer=optimizer,metrics=['acc'])
```

Der endgültige Code für die Modellerstellung kann unter `https://aka.klawr.de/srp#16` eingesehen werden.

8 Zusammenfassung

Nachdem die Möglichkeiten des maschinellen Lernens und ihre Auswirkungen auf die verschiedenen Technologiezweige untersucht wurden, scheint die Idee, dies auf den Bereich der Kinematik auszudehnen, vernünftig zu sein. In diesem Projekt wurden die Grundlagen für die Anwendung des maschinellen Lernens von Grund auf neu erlernt, wobei mit einem einfachen statistischen Modell begonnen wurde, das in der Lage ist, handgeschriebene mechanische Symbole zu klassifizieren.

Die Daten legen nahe, dass ein ausreichender Algorithmus gefunden wurde, der eine Genauigkeit von über 99% über die Testdaten aufweist. Nach dem Entfernen der Trainingskonfiguration hat diese eine Größe von 76kb als `Keras` Modell. Die Konvertierung des Modells in ein Format, das durch `tensorflow.js`[27] lesbar ist, und das Zusammensetzen in einen lesbaren `IOHandler`[28] erhöht die Größe nicht signifikant (81kb).

Anwendungen zum Testen der Modelle wurden im Repository zur Verfügung gestellt:

Ein Python-Skript zum Zeichnen von Bildern in die zum Erstellen von Daten verwendete Umgebung, das aber zur Vorhersage des gezeichneten Symbols verwendet wird, findet sich im Verzeichnis `code` als `symbol_classifier_test.py`[29].

Eine Webseite, die direkt in einen Web-Browser geladen werden kann[30]. Die Datei be-

[27] `https://www.tensorflow.org/js/`

[28] Die loader.js kann unter `https://aka.klawr.de/srp#15` gefunden werden

[29] `https://aka.klawr.de/srp#16`

[30] Der Test funktioniert auf Firefox und Google Chrome. Microsoft Edge funktioniert noch nicht, aber dies wird sich wahrscheinlich ändern, wenn Microsoft auf einen Chromium basierten Build umsteigt.

findet sich als `symbol_classifier_test.html`[31] ebenfalls im Verzeichnis `code`. Dieser Test kommt dem beabsichtigten Zweck nahe, indem Bilder in die Anwendung hochgeladen werden und dem Benutzer dann automatische Vorhersagen zur Verfügung gestellt werden.

Spätere Projekte werden folgen um die Funktionalität zur Erkennung der Position von erkannten Symbolen in beliebigen Bildern zu gewährleisten. Des weiteren sollen möglichen Verbindungen der Symbole zueinander in Form von "Constraintsërkannt werden. Nachdem vollständige Mechanismus-Darstellungen abgeleitet werden können, können sie in bestehende Web-Anwendungen wie `mec2` importiert werden [Gös19c; Gös19b; Gös19a] oder `mecEdit` [Uhl19a; Uhl19b].

[31] `https://aka.klawr.de/srp#17`

Abbildungsverzeichnis

Listings

Link Index - `https://aka.klawr.de/srp`

Literatur

[BB01] Michele Banko und Eric Brill. „Scaling to Very Very Large Corpora for Natural Language Disambiguation“. In: *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*. ACL '01. Toulouse, France: Association for Computational Linguistics, 2001, S. 26–33. DOI: 10.3115/1073012.1073017. URL: https://doi.org/10.3115/1073012.1073017.

[Bel57] Richard E. Bellman. *Dynamic Programming*. PRINCETON UNIV PR, 11. Okt. 1957. 342 S. ISBN: 069107951X. URL: https://www.ebook.de/de/product/34448612/richard_e_bellman_dynamic_programming.html.

[Cho17] Francois Chollet. *Deep Learning with Python*. Manning Publications, 28. Okt. 2017. 384 S. ISBN: 1617294438. URL: https://www.ebook.de/de/product/28930398/francois_chollet_deep_learning_with_python.html.

[Cho19] François Chollet. *Keras*. 2019. URL: https://keras.io/ (besucht am 09. 10. 2019).

[Cyb89] G. Cybenko. „Approximation by superpositions of a sigmoidal function“. In: *Mathematics of Control, Signals and Systems* 2.4 (Dez. 1989), S. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[DHS10] John Duchi, Elad Hazan und Yoram Singer. „Adaptive Subgradient Methods for Online Learning and Stochastic Optimization“. In: *Journal of Machine Learning Research 12* (2010). URL: http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[dri19] drivendata. *Cookiecutter Data Science*. 2019. URL: https://drivendata.github.io/cookiecutter-data-science/ (besucht am 23. 10. 2019).

[GBB11] Xavier Glorot, Antoine Bordes und Y. Bengio. „Deep Sparse Rectifier Neural Networks“. In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statisitics (AISTATS) 2011* 15 (Jan. 2011), S. 315–323.

[GBC17] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. The MIT Press, 3. Jan. 2017. 800 S. ISBN: 0262035618. URL: https://www.ebook.de/de/product/26337726/ian_goodfellow_yoshua_bengio_aaron_courville_deep_learning.html.

[Geo12] Kevin Swersky Geoffrey Hinton Nitish Srivastava. „Neural Networks for Machine Learning“. 2012. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[Gér19] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly UK Ltd., 1. Okt. 2019. 819 S. ISBN: 1492032646. URL: https://www.ebook.de/de/product/33315532/aurelien_geron_hands_on_machine_learning_with_scikit_learn_keras_and_tensorflow.html.

[Goo19a] Google. *Keras Tuner*. 2019. URL: https://keras-team.github.io/keras-tuner/ (besucht am 28. 11. 2019).

[Goo19b] Google. *Tensorflow*. 2019. URL: https://www.tensorflow.org/ (besucht am 09. 10. 2019).

[Goo65] Irvin John Good. *Speculations Concerning the First Ultraintelligent Machine*. 1965. URL: http://acikistihbarat.com/dosyalar/artificial-intelligence-first-paper-on-intelligence-explosion-by-good-1964-acikistihbarat.pdf (besucht am 16. 10. 2019).

[Gös19a] Stefan Gössner. „Ebene Mechanismenmodelle als Partikelsysteme - ein neuer Ansatz“. In: *13. Kolloquium-Getriebetechnik - Tagungsband*. Hrsg. von Burkhard Corves, Philippe Wenger und Mathias Hüsing. Cham: Springer International Publishing, 2019, S. 169–180. ISBN: 978-3-8325-4979-4.

[Gös19b] Stefan Gössner. „Fundamentals for Web-Based Analysis and Simulation of Planar Mechanisms“. In: *EuCoMeS 2018*. Hrsg. von Burkhard Corves, Philippe Wenger und Mathias Hüsing. Cham: Springer International Publishing, 2019, S. 215–222. ISBN: 978-3-319-98020-1.

[Gös19c] Stefan Gössner. *mec2*. 2019. URL: `https://github.com/goessner/mec2`.

[Hei19] Olli-Pekka Heinisuo. *opencv-python*. 2019. URL: `https://github.com/skvark/opencv-python` (besucht am 22. 10. 2019).

[HNP09] A. Halevy, P. Norvig und F. Pereira. „The Unreasonable Effectiveness of Data“. In: *IEEE Intelligent Systems* 24.2 (März 2009), S. 8–12. DOI: `10.1109/MIS.2009.36`.

[HSW89] Kurt Hornik, Maxwell Stinchcombe und Halbert White. „Multilayer feedforward networks are universal approximators“. In: *Neural Networks* 2.5 (Jan. 1989), S. 359–366. DOI: `10.1016/0893-6080(89)90020-8`.

[Jup19] Jupyter. *Jupyter Notebook*. 2019. URL: `https://jupyter.org/` (besucht am 23. 10. 2019).

[KB14] Diederik P. Kingma und Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: `1412.6980 [cs.LG]`.

[Lec+98] Y. Lecun u. a. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (1998), S. 2278–2324. DOI: `10.1109/5.726791`.

[Li+18] Lisha Li u. a. „Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization“. In: *Journal of Machine Learning Research* 18.185 (2018), S. 1–52. URL: `http://jmlr.org/papers/v18/16-558.html`.

[McC55] John McCarthy. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. 1955. URL: `http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html` (besucht am 09. 10. 2019).

[MP43] Warren S. McCulloch und Walter Pitts. „A logical calculus of the ideas immanent in nervous activity“. In: *The bulletin of mathematical biophysics* 5.4 (Dez. 1943), S. 115–133. ISSN: 1522-9602. DOI: `10.1007/BF02478259`. URL: `https://doi.org/10.1007/BF02478259`.

[Nes83] Y. E. Nesterov. „A method for solving the convex programming problem with convergence rate $O(1/k^2)$“. In: *Dokl. Akad. Nauk SSSR* 269 (1983), S. 543–547. URL: `https://ci.nii.ac.jp/naid/10029946121/en/`.

[New87] Isaac Newton. *Philosophiae naturalis principia mathematica*. Jussu Societatis Regiae ac Typis Josephi Streater. Prostat apud plures bibliopolas, 1687. DOI: 10.5479/sil.52126.39088015628399.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2015. URL: http://neuralnetworksanddeeplearning.com (besucht am 17. 10. 2019).

[nvi19] nvidia. *CUDA*. 2019. URL: https://www.geforce.com/hardware/technology/cuda (besucht am 09. 10. 2019).

[Ope19] OpenCV. *OpenCV*. 2019. URL: https://opencv.org/ (besucht am 22. 10. 2019).

[Pol64] Boris Polyak. „Some methods of speeding up the convergence of iteration methods“. In: *Ussr Computational Mathematics and Mathematical Physics* 4 (Dez. 1964), S. 1–17. DOI: 10.1016/0041-5553(64)90137-5.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. „Learning representations by back-propagating errors“. In: *Nature, Volume 323, Issue 6088, pp. 533-536 (1986)* 323 (Okt. 1986), S. 533–536. DOI: 10.1038/323533a0.

[RKK18] Sashank J. Reddi, Satyen Kale und Sanjiv Kumar. „On the Convergence of Adam and Beyond“. In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=ryQu7f-RZ.

[RT17] David Rolnick und Max Tegmark. „The power of deeper networks for expressing natural functions“. In: *Neural Networks* (16. Mai 2017). arXiv: http://arxiv.org/abs/1705.05502v2 [cs.LG].

[Stu18] Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach, Global Edition*. Addison Wesley, 28. Nov. 2018. ISBN: 1292153962. URL: https://www.ebook.de/de/product/25939961/stuart_russell_peter_norvig_artificial_intelligence_a_modern_approach_global_edition.html.

[Sut+13] Ilya Sutskever u. a. „On the importance of initialization and momentum in deep learning“. In: *Proceedings of the 30th International Conference on Machine Learning*. Hrsg. von Sanjoy Dasgupta und David McAllester. Bd. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 2013, S. 1139–1147. URL: http://proceedings.mlr.press/v28/sutskever13.html.

[Uhl19a] Jan Uhlig. „Entwicklung einer modularen Web-App zur interaktiven Modellierung und impulsbasierten Analyse beliebiger planarer Koppelmechanismen“. Magisterarb. Fachhochschule Dortmund, 2019.

[Uhl19b] Jan Uhlig. *mecEdit*. 2019. URL: https://mecedit.com (besucht am 11. 12. 2019).

[Wol96] David H. Wolpert. „The Lack of A Priori Distinctions Between Learning Algorithms“. In: *Neural Computation* 8.7 (1996), S. 1341–1390. DOI: 10.1162/neco.1996.8.7.1341. eprint: https://doi.org/10.1162/neco.1996.8.7.1341. URL: https://doi.org/10.1162/neco.1996.8.7.1341.

[Zei12] Matthew D. Zeiler. „ADADELTA: An Adaptive Learning Rate Method“. In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701.