Technische
Hochschule
Georg Agricola

Einführung in die Objektdetektion mit Convolutional Neural Networks

---

Dennis Gardiner

dennis.gardiner@stud.thga.de

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Diese Arbeit stellt eine kurze Einführung in das maschinelle Lernen mittels faltender neuronaler Netzwerke (Convolutional Neural Networks, kurz CNN) dar. In der ersten Hälfte wird erläutert, was unter maschinellem Lernen zu verstehen ist, um dann die mathematischen Grundlagen vorzustellen und mögliche Lösungen für die mathematische Aufgabenstellung zu nennen.
Zum Abschluss der ersten Hälfte dieser Arbeit sollen einige einfache Modelle für verschiedene Anwendungen aufgezeigt werden, um ein Gefühl für mögliche Einsatzzwecke maschinellen Lernens zu vermitteln.

In der zweiten Hälfte wird der grundlegende Aufbau eines einfachen neuronalen Netzes erläutert. Neben der Funktionsweise werden auch mögliche Schwierigkeiten bei der Verwendung neuronaler Netzwerke und passende Gegenmaßnahmen genannt, um schließlich die Erweiterung zum Convolutional Neural Network zu zeigen. Den Abschluss bildet eine Erläuterung der Funktionsweise mit einigen Beispielen aus der Praxis, bei welchem ein Convolutional Neural Network zur Bilddetektion eingesetzt wird.

2 Definitionen und Erläuterungen

2.1 Erläuterung: was ist maschinelles Lernen

Öffentlich bekannt geworden ist das maschinelle Lernen sicherlich spätestens seit der Datenschutzdiskussionen um Firmen wie Google, Facebook oder Amazon. Tatsächlich jedoch findet maschinelles Lernen bereits deutlich länger in vergleichsweise kleinem Maßstab im Alltag statt. Dabei ist von maschinellem Lernen, etwas salopp formuliert, immer dann zu reden, wenn ein Computer ein Problem bzw. eine Aufgabe löst, deren Lösung nicht vorher explizit einprogrammiert wurde.

Ein Beispiel, welches kein maschinelles Lernen darstellt, wäre eine einfache `if-else` Schleife oder eine `switch-case`-Funktion, welche Aufgrund einer Eingabe eine explizit programmierte Ausgabe erzeugt. Ein MATLAB-Beispiel hierfür zeigt Abbildung 1.
Als explizit bekannte Lösung wird dabei auch eine explizit programmierte Funktion gewertet. Das schlichte Lösen einer fest programmierten mathematischen Funktion $f(x_1, x_2, \dots, x_n)$, zählt damit ebenfalls nicht als maschinelles Lernen. Abbildung 2 zeigt ein entsprechendes Beispiel.

```
1  function einfache_Ausgabe(y)
2
3
4  switch y
5      case 0
6          disp('y ist Null');
7      case 1
8          disp('y ist Eins');
9      otherwise
10         disp('FEHLER: y ist nicht binaer');
11 end
12
13 end
```

Abbildung 1: Negativbeispiel 1

```
1 function y = einfache_Funktion (x)
2
3 y = x^2 + 5*x + 10;
4
5 end
```

Abbildung 2: Negativbeispiel 2

Ein recht einfaches Beispiel für maschinelles Lernen stellt hingegen das bestimmen einer Regressionsgeraden durch einen Datensatz dar. Hierbei wird nicht einfach ein bestimmter Wert mittels einer mathematischen Funktion bestimmt, sondern das Programm ist in der Lage, aus vorhandenen Daten neue Daten zu erzeugen, ohne hierfür eine explizite Rechenvorschrift wie im zweiten Negativbeispiel erhalten zu haben.

Eine gut formulierte, formale Definition liefert beispielsweise Tom Mitchell:

> „Each machine learning problem can be precisely defined as the problem of improving some measure of performance P when executing some task T, through some type of training experience E"[1]

Ein Alltagsbeispiel hierfür ist der Wetterbericht. Dabei versucht ein Algorithmus anhand verschiedener Parameter wie Luftdruck und -feuchte, Temperatur, Windrichtung und -geschwindigkeit etc. aus den aktuellen Wetterbedingungen vorherzusagen, welche durchschnittliche Temperatur am nächsten Tag erwartet wird.
Die Aufgabe „T" ist in diesem Fall die Wettervorhersage für den nächsten Tag. Die Erfahrung „E", aus welcher der Algorithmus lernen soll, sind die Wetterdaten aus der Vergangenheit. Und das Maß um die Leistung zu messen ist, wie oft der Algorithmus korrekte Vorhersagen liefert.

Weiterhin kann maschinelles Lernen im Wesentlichen auf zwei verschiedene Arten durchgeführt werden. Die erste Variante ist durch sogenanntes überwachtes Lernen (supervised learning). Überwacht bedeutet dabei keine aktive menschliche Aufsicht. Vielmehr bezieht sich der Begriff auf die genutzten Daten: ist zu den Daten, mit welchen der Algorithmus lernt, eine Aufgabe zu lösen, die korrekte Lösung bekannt, handelt es sich um ein überwachtes Lernen.
In diesem Fall ist der Algorithmus in der Lage, eine Hypothese zur Lösung der Aufgabe solange zu verbessern, bis die Performance ein Maximum erreicht. Messbar ist dieses Maximum anhand des Fehlers zwischen der Ausgabe des Algorithmus und den bekannten, korrekten Werten. Dies führt zur so genannten Kostenfunktion (cost function), auf welche unten näher eingegangen wird.

Die Alternative ist das unüberwachte Lernen (unsupervised learning). Hierbei wird eine Aufgabe gestellt, deren korrekte Lösung nicht bereits bekannt ist. Dies lässt sich am besten anhand der beiden Standardfälle der Segmentierung (clustering) und der Datenkomprimierung bzw. Dimensionsreduktion (dimensionality reduction) erläutern.
Die Aufgabe bei der Segmentierung ist es, anhand von gegebenen Daten Muster oder Gruppen zu erkennen, welche dem Programmierenden vorher selbst nicht bewusst sind. Dies ist sogar in einem solchen Umfang möglich, dass für den Menschen nicht sichtbare Strukturen durch maschinelles Lernen gefunden werden können.
Die Dimensionsreduktion fasst redundante Daten zusammen, sodass der Informationsgehalt innerhalb gegebener Toleranzen erhalten bleibt, obwohl weniger Speicherplatz benötigt wird. Auch hier gibt es im Regelfall vor der Reduktion keine bekannten Ergebnisse.

2.2 Mathematische Grundlagen und Definitionen

Die Grundaufgabe beim maschinellen Lernen besteht darin, ein Modell aufzustellen, wie Daten zu interpretieren oder zu verarbeiten sind. Dieses Modell kann dabei durch den Lernalgorithmus selbst aufgestellt werden, oder es wird durch den Menschen vorgegeben.

[1] Mitchell, 2017

Das dafür geschriebene Programm muss nun die Parameter dieses Modells erlernen, um die so genannte Hypothese zu bestimmen. Anders formuliert: Das Programm arbeitet solange daran, die Parameter der Hypothese zu verbessern, bis die Hypothese das bestmögliche Ergebnis liefert. Hierzu bedarf es einer Möglichkeit, die Performance der Hypothese zu bestimmen.

Für Anwendungen im Bereich des überwachten Lernens ist dies recht intuitiv: Wenn die Hypothese bestimmte Werte voraussagt, können diese mit den bekannten realen, korrekten Werten verglichen werden. Je größer dabei die Abweichung ist, umso schlechter ist die Hypothese.

Zur mathematischen Beschreibung werden folgende Definitionen eingeführt:

- m – die Anzahl der Trainingsdaten
- n – Die Anzahl der Features
- $X_j^{(i)}$ – Die i'te Eingangsvariable, $i = 1,2,3,\dots,m$, zu Feature $j = 1,2,3,\dots,n$
- $y^{(i)}$ – Die i'te Ausgangsvariable, auch i'ter Wert, $i = 1,2,3,\dots,m$
- $\left(x_j^{(i)}, y^{(i)}\right)$ – Das i'te Trainingsdatum zu Feature j
- θ_j – Der j'te Parameter, $j = 1,2,3,\dots,n$
- $h_\theta(x)$ – Die Hypothese, abhängig von den Eingangsvariablen $x_j^{(i)}$ und den in ihr statischen Werten der Parameter θ_j
- $J(\theta)$ – Die Kostenfunktion, **unabhängig** von $x_j^{(i)}$

Sofern die Variablen oder Parameter mit einem Index versehen sind ist ein einzelner Eintrag eines Vektors oder einer Matrix gemeint. $X_j^{(i)}$ repräsentiert somit einen bestimmten Zahlenwert. Der Einfachheit halber werden Vektoren ohne expliziten Vektorpfeil dargestellt, die fehlende Hochstellung oder der fehlende Index zeigt einen Vektor an. Somit gilt

$$x^{(i)} = \begin{pmatrix} x_1^{(i)} & \cdots & x_n^{(i)} \end{pmatrix}. \tag{1}$$

Damit ist ein einzelnes Datum von der Dimension $(1 \times n)$ dargestellt. Analog wird ein Spaltenvektor in der Form

$$x_j = \begin{pmatrix} x_j^{(1)} \\ \vdots \\ x_j^{(m)} \end{pmatrix} \tag{2}$$

dargestellt. Die bekannten Daten werden im Vektor

$$\mathrm{y} = \begin{pmatrix} \mathrm{y}_1 \\ \vdots \\ \mathrm{y}_\mathrm{m} \end{pmatrix} \tag{3}$$

abgelegt. Matrizen werden durch Großbuchstaben kenntlich gemacht. Die Gesamtheit der Eingangsvariablen wird damit durch

$$X = \begin{pmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix} \tag{4}$$

dargestellt. Die Matrix X besitzt also die Dimension $(m \times n)$. In dieser Schreibweise befindet sich jeweils ein Datum $x^{(i)}$ in einer Zeile der Matrix, wobei jedes Datum $x^{(i)}$ über n verschiedene Features verfügt.
Diese Features sind die verschiedenen Größen, welche zusammen ein Datum ergeben. Um beim Wetterbeispiel zu bleiben entspricht $x^{(i)}$ den Wetterbedingungen eines Tages, bestehend aus Luftdruck, Luftfeuchte, Temperatur, Windrichtung und Geschwindigkeit, es gilt $n = 5$.

Eine einfache Form der Hypothese wäre dann durch

$$h_\theta(X) = X \cdot \theta \tag{5}$$

gegeben. Das Ergebnis ist ein Vektor mit m Einträgen, welcher jedoch wertemäßig nicht mit dem Vektor der Ausgangsvariablen übereinstimmen muss. θ ist dabei von der Dimension $(n \times 1)$. Im obigen Beispiel des Wetterberichts ist $n = 5$. Für ein einzelnes Datum i gilt damit die Hypothese:

$$h_\theta\left(x^{(i)}\right) = x_1^{(i)} \cdot \theta_1 + x_2^{(i)} \cdot \theta_2 + x_3^{(i)} \cdot \theta_3 + x_4^{(i)} \cdot \theta_4 + x_5^{(i)} \cdot \theta_5 \tag{6}$$

Für viele Probleme kann eine solche Hypothese jedoch keine guten Ergebnisse erzielen, da alle Parameter θ_j mit den Eingangsvariablen gekoppelt sind. Dadurch fehlt die Möglichkeit, einen Offset darzustellen. Bei der Berechnung einer Temperatur könnte dies bedeuten, dass eine Umrechnung zwischen Kelvin und Grad Celsius nicht auf einfache Art berücksichtigt werden kann.

Dieses Problem lässt sich durch eine einfache Ergänzung lösen: Es wird ein weiteres Feature definiert, dessen Wert fest ist, und allen Daten vorangestellt wird:

$$x_0^{(i)} = 1 \tag{7}$$

Damit die Bestimmung der Hypothese mit nun $n + 1$ Features weiter gelingt, muss auch die Dimension von θ um eins vergrößert werden und es wird analog ein neuer Parameter θ_0 vor den bisherigen Parametervektor vorangestellt. Die Hypothese nach (6) wird damit für ein einzelnes Datum zu

$$h_\theta\left(x^{(i)}\right) = \theta_0 \cdot x_0^{(i)} + x_1^{(i)} \cdot \theta_1 + x_2^{(i)} \cdot \theta_2 + x_3^{(i)} \cdot \theta_3 + x_4^{(i)} \cdot \theta_4 + x_5^{(i)} \cdot \theta_5 \tag{8}$$

Der zusätzliche Term x_0 wird auch als Bias-Term bezeichnet.

Die Hypothese dient nun zur Erfüllung der Aufgabe T, die benötigte Erfahrung E findet sich in den vorhandenen Daten (X, y). Der letzte fehlende Faktor, die Performance P, lässt sich über die Kostenfunktion darstellen.

Einfach ausgedrückt ist es die Aufgabe der Kostenfunktion, den Fehler zu quantifizieren, welchen die Hypothese $h_\theta(X)$ bei der Vorhersage von y erzeugt. Dabei hängt dieser Fehler **nicht** von den Eingangsvariablen X ab, sondern von den Parametern θ, da der Algorithmus daran arbeitet, θ für gegebene Daten X zu optimieren.
Die genaue Ausgestaltung der Kostenfunktion ist im Übrigen nur von minderer Wichtigkeit, solange sie für das Problem und die Hypothese geeignet ist. Eine häufig verwendete Kostenfunktion stellt die quadratische Abweichung dar. Konkret wird die Kostenfunktion damit durch

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 \tag{9}$$

beschrieben.

Eine vektorielle Darstellung der Kostenfunktion lässt sich für die einfache Hypothese (5) durch

$$J(\theta) = \frac{1}{2}(X\theta - y)^T(X\theta - y) \tag{10}$$

erreichen. Dies ist insbesondere für große Datenmengen wichtig, da die Zeit zur Berechnung der Matrixmultiplikation deutlich unter der benötigten Zeit zur Berechnung der Summe liegt. Außerdem ist hiermit eine Implementierung in nur einer einzelnen Zeile möglich, wie Abbildung 3 zeigt.

```
1 function J = costfunction(X, y, theta)
2
3 J=(1 / (2*m)) * ( X*theta - y )' * ( X*theta - y );
4
5 end
```

Abbildung 3: Implementierung der Kostenfunktion nach (10) in MATLAB

Da das Ziel des maschinellen Lernens ist, den Wert dieser Kostenfunktion zu minimieren, ist der Vorfaktor $\frac{1}{2}$ prinzipiell nicht nötig, da das Minimum für die gleichen Werte von θ vorliegt.
Die Lösung eines solchen Minimierungsproblems lässt sich mathematisch durch verschiedene Verfahren beschreiben, welche dazu jedoch häufig den Gradienten des Kostenfaktors benötigen. Der Gradient ist definiert als

$$\nabla J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}. \tag{11}$$

Der Vorfaktor wird also in der Ableitung der Kostenfunktion eliminiert. Unter Berücksichtigung von (6) bzw. (8) ergibt sich mit der Kettenregel sofort die partielle Ableitung der Kostenfunktion (9) für einen einzelnen Parameter zu

$$\frac{\partial \mathrm{J}(\theta)}{\partial \theta_j} = \sum_{i=1}^{m} x_j^{(i)} \cdot \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \tag{12}$$

Auch hierzu lässt sich eine vektorielle Darstellung ermitteln:

$$\frac{\partial \mathrm{J}(\theta)}{\partial \theta_\mathrm{j}} = \mathrm{x_j^T} \cdot (\mathrm{X}\theta - \mathrm{y}) \tag{13}$$

Die Dimensionsgleichung hierzu ergibt $(1 \times m) \cdot [(m \times n)(n \times 1) - (m \times 1)] = 1$. Der gesamte Gradient ist damit gemäß (11) und (12) als Vektor berechenbar:

$$\nabla J(\theta) = X^T(X\theta - y) \tag{14}$$

Dies ist ein Vektor der Dimension $(n \times 1)$. Streng genommen müsste in allen n −Dimensionen noch der Bias-Term berücksichtigt werden, womit der Gradient von der Dimension $((n+1) \times 1)$ ist. In der Literatur wird dieser mandatorische Term häufig nicht explizit genannt, er sollte aber stets im Hinterkopf des Anwenders präsent sein.

3 Praktische Lösungen des Minimierungsproblems

Im Folgenden werden drei verschiedene Möglichkeiten zur Bestimmung der optimalen Parameter θ vorgestellt. Eine detaillierte Betrachtung wird jedoch aufgrund des gewünschten Umfangs dieser Ausarbeitung nicht stattfinden.

3.1 Das Gradientenverfahren

Bei diesem Verfahren werden zunächst Startwerte für die Parameter θ erzeugt. Eine häufige Vorgehensweise ist es, $\theta = \vec{0}$ zu setzen. Ausgehend von diesem Startpunkt wird nun mit einer definierten Schrittweite, der Lernrate α, solange ein Schritt in die durch den Gradienten vorgegebene Richtung gemacht, bis das (lokale) Minimum der Kostenfunktion erreicht wird. Abbildung 4 zeigt ein Beispiel für eine Kostenfunktion mit zwei Features. Der Startpunkt ist in diesem Fall auf den roten Hügel im linken Bereich gelegt.

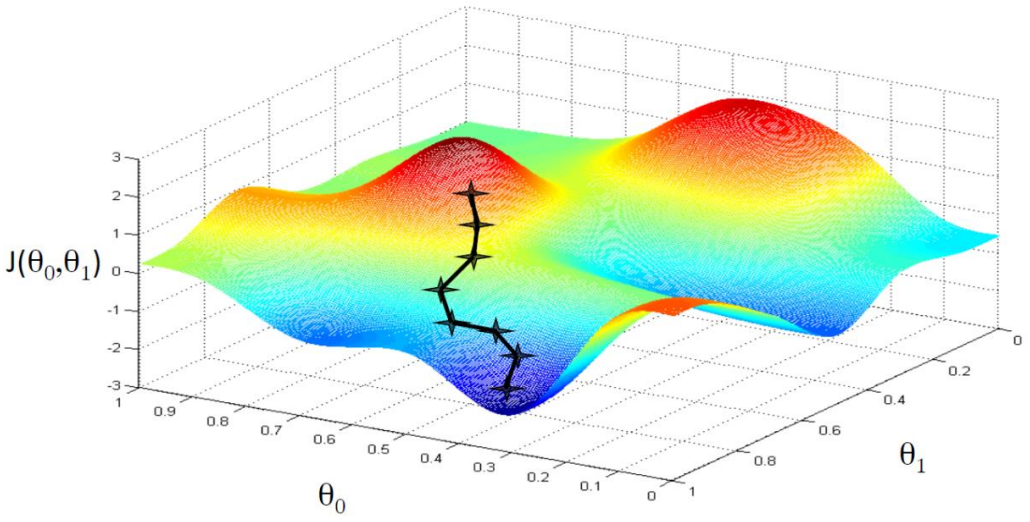

Abbildung 4: Visualisierung des Gradientenverfahrens für ein Feature[2]

Das Gradientenverfahren läuft dann in einer Schleife ab, welche solange läuft, bis die Verkleinerung der Kostenfunktion durch eine Iteration eine Abbruchbedingung erfüllt.

Eine übliche Abbruchbedingung lautet:

$$\|J_{k-1}(\theta) - J_k(\theta)\| < \varepsilon \tag{15}$$

Die in (17) verwendete Formulierung $\|f(x)\|$ stellt dabei die euklidische Norm dar. Der Parameter ε wird als threshold bezeichnet und wird um so kleiner gewählt, je höher die gewünschte Genauigkeit ist.

[2] NG, 2020

Eine weitere Abbruchbedingung sollte außerdem eine maximale Anzahl an Iterationen sein, sodass das Gradientenverfahren nicht zu lange läuft, falls die Lernrate oder die Konvergenzgrenze unglücklich gewählt wurde. In diesem Fall empfiehlt es sich natürlich, programmiertechnisch weitere Ausgaben zu erzeugen und den Anwender auf das Problem hinzuweisen.

3.2 Die Normalengleichung

Ausgehend von der Kostenfunktion in Gleichung (10) lässt sich auch eine analytische Methode zur Lösung des Minimierungsproblems herleiten. Zur Vereinfachung soll der Vorfaktor nicht weiter beachtet werden. Die so genannte Normalengleichung lautet dann:

$$\theta = (X^T X)^{-1} X^T y \tag{16}$$

Unter der Voraussetzung, dass $X^T X$ invertierbar ist, steht so mittels der Normalengleichung (16) eine analytische Methode zur Verfügung, die gesuchten Parameter θ zur Minimierung der Kostenfunktion $J(\theta)$ zu bestimmen.

3.3 fminunc in Octave/ MATLAB

Sowohl das Gradientenverfahren als auch die Normalengleichung stellen anschauliche Verfahren dar, welche leicht nachzuvollziehen und zu implementieren sind. Gerade mit steigender Anzahl an zu berücksichtigenden Features n sind beide Algorithmen jedoch eher langsam, weshalb in der Praxis fortgeschrittene Algorithmen wie `fminunc` genutzt werden. Da diese Algorithmen in der Regel ebenfalls den Gradienten als Eingabeparameter benötigen stellt das Gradientenverfahren einen guten Einstieg in die Thematik dar.

Der Befehl `fminunc` steht für „Find minimum of unconstrained multivariable function“.[3] Die untersuchte Funktion darf also keine Randbedingungen aufweisen. Eine Beschränkung wie $\theta_1 \in [5; 150]$ wäre nicht zulässig. Der Funktionsaufruf folgt der Syntax `x = fminunc(fun, x0, options)`. Abbildung 5 zeigt einen einfachen maschinellen Regressionsalgorithmus unter Verwendung von `fminunc`.

```
function [theta] = trainLinearReg(X, y, lambda)

% Initialize Theta
initial_theta = zeros(size(X, 2), 1);

% Create "short hand" for the cost function to be minimized
costFunction = @(t) linearRegCostFunction(X, y, t, lambda);

% Set Options for fminunc
options = optimset('MaxIter', 200, 'GradObj', 'on', 'Display', 'off');

% Minimize using fminunc
theta = fminunc(costFunction, initial_theta, options);

end
```

Abbildung 5: fminunc-Aufruf in MATLAB

Es handelt sich somit um einen vollständigen, wenn auch einfachen Fall maschinellen Lernens. Die Definition der Kostenfunktion sowie des Gradienten ist ausgelagert. Für Interessierte sei für die Erläuterungen zu `fminunc` auf die exzellenten MATLAB-Dokumentationen verwiesen.

[3] The MathWorks, Inc, 2021

4 Modellbildungen für ausgewählte Anwendungen

4.1 Regressionsmodelle

Ein klassischer Anwendungsfall bereits für sehr wenige Daten ist die Regressionsanalyse. Dabei wird in der Regel durch äußere Bedingungen wie physikalische Gesetzmäßigkeiten ein Modell vorgegeben, welches durch maschinelles Lernen zu einer konkreten Hypothese angepasst werden soll. Häufig sind diese Modelle einfache Polynome, exponentielle oder trigonometrische Funktionen. Entsprechende Implementationen finden sich in zahlreichen Anwendungen, insbesondere auch in allen gängigen Tools zur Tabellenkalkulation.

4.2 Logistische Regression

Der Begriff logistische Regression kennzeichnet die Vorhersage von binären Werten, also die Ausgabe einer 0 oder einer 1. Häufig wird dabei eine 0 auch als negatives Ergebnis bezeichnet, wohingegen eine 1 als positive Ausgabe beziffert wird – unabhängig von der zugrunde liegenden Anwendung. In Diagrammen wird eine 0 in der Regel als o dargestellt, eine 1 meistens als x.

Das Modell zur Vorhersage solcher Werte wird mit Hilfe einer Sigmoidfunktion erzeugt. In der Regel ist dabei mit Sigmoidfunktion die logistische Funktion

$$\mathrm{sig}(t) = \frac{1}{1 + e^{-t}} \tag{17}$$

gemeint. Kennzeichnend ist, dass die Kurve für $t = 0$ den Wert $sig(0) = 0.5$ annimmt. Außerdem ist die Kurve symmetrisch um die Y-Achse und nimmt (unter)über einem Schwellenwert nahezu sofort den Wert (0)1 an, wie Abbildung 6 zeigt:

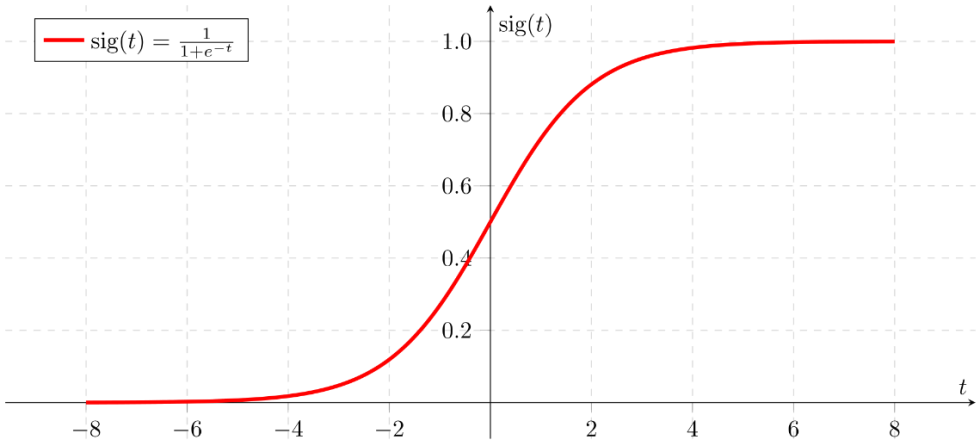

Abbildung 6: Die logistische Funktion, die häufigste genutzte Sigmoidfunktion[4]

Ein einfaches Modell mit der Sigmoidfunktion hat dann die vektorisierte Form

$$h_\theta(X) = \frac{1}{1 + e^{-X\theta}} \tag{18}$$

und erzeugt bei Minimierung der Kostenfunktion

[4] Thoma, 2021

$$J(\theta) = -y^T \cdot log\big(\mathrm{sig}(X\theta)\big) - (1-y)^T \cdot log\big(1-\mathrm{sig}(X\theta)\big) \tag{19}$$

eine Art Grenzlinie, wobei alle Daten auf einer Seite der Linie einen Wert $h_\theta(X) < 0.5$ und Daten auf der anderen Seite entsprechend $h_\theta(X) > 0.5$ erzeugen. Die Form dieser Grenze hängt dabei von X und θ ab. Für hinreichend viele und komplexe Parameter sind auch sehr freie Formen möglich, auch eine Hypothese gemäß Abbildung 7 kann leicht bestimmt werden:

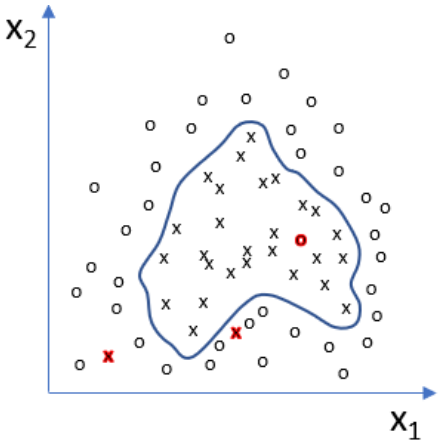

Abbildung 7: Beispiel für komplexe kogistische Regression mit zwei Features

In Rot markiert sind Ausreißer, welche von dem Algorithmus falsch klassifiziert werden.

Eine Erweiterung für mehr als zwei mögliche Zustände wird mit der One-vs-all-Methode realisiert.

Die Sigmoidfunktionen bilden außerdem häufig eine Grundlage für Klassifikationen mittels neuronaler Netze, welche alternativ zur One-vs-all-Methode zur Multi-Klassifikation genutzt werden können.

4.3 Clustering mit k-means

Beim Clustering handelt es sich um einen unüberwachten Lernprozess. Hierbei soll ein Algorithmus Häufungen (Cluster) erkennen und eine Zuteilung von Daten zu einem Cluster erarbeiten. Im Unterschied zur Regressionsanalyse und zur logistischen Regression sind hierbei nur die Eingangsvariablen X bekannt, es gibt jedoch keine Werte y zu den Eingangsvariablen.

Solche Algorithmen lösen gleich zwei Probleme auf einmal: Es muss einerseits die Anzahl an Clustern bestimmt werden, andererseits aber auch die Position und Größe bzw. Form der Cluster. Dabei kann die Anzahl von außen vorgegeben oder durch den Lernprozess bestimmt werden.[5]

4.4 Datenkompression

Hierbei handelt es sich um verschiedene Anwendungen, welche jedoch die gleiche Konzeption verfolgen. Bei einer großen Zahl an Features werden wahrscheinlich einige Features Redundant sein. Der Algorithmus kann solche Redundanzen erkennen und die redundanten Daten zusammenfassen, wobei vorher definiert werden muss, wie groß der dadurch eventuell entstehende Fehler sein darf.

Genutzt werden solche Algorithmen einerseits, um den Speicherbedarf der Daten zu reduzieren, andererseits können dadurch beispielsweise Daten, welche im Dreidimensionalen eine Ebene aufspannen würden, in 2D dargestellt werden.

[5] Kodinariya & Makwana, 2013

5 Künstliche neuronale Netzwerke

5.1 Motivation und Aufbau

Das menschliche Gehirn ist in der Lage, mit ein und denselben Zellen im selben Hirnbereich verschiedene Aufgaben zu erlernen und zu lösen. Ein bekanntes Beispiel stellt das Umkehrbrillen-Experiment dar. Hierbei wird durch eine Brille das Sehfeld des Trägers auf den Kopf gestellt. Nach einigen Tagen Gewöhnungszeit (sprich: Lernzeit) ist das Gehirn jedoch in der Lage, die Störung auszugleichen und dem Träger ein gewohntes Verhalten zu ermöglichen.

Inspiriert von der Lernfähigkeit des menschlichen Gehirns wird daher versucht, diese Lernprozesse in künstlichen neuronalen Netzwerken nachzubilden. Dabei ist die Grundidee recht einfach: Das Netzwerk besitzt mehrere Spalten an Knoten, welche über Ein- und Ausgänge verfügen. Diese Knoten sind untereinander verknüpft und jeder Knoten erzeugt einen Ausgabewert abhängig von seinem Eingangswert, genau wie die Neuronen im menschlichen Gehirn. Entsprechend werden die Knoten als künstliche Neuronen (artificial neurons) bezeichnet. Das gesamte Netzwerk wird oft mit ANN (artificial neural network) abgekürzt.

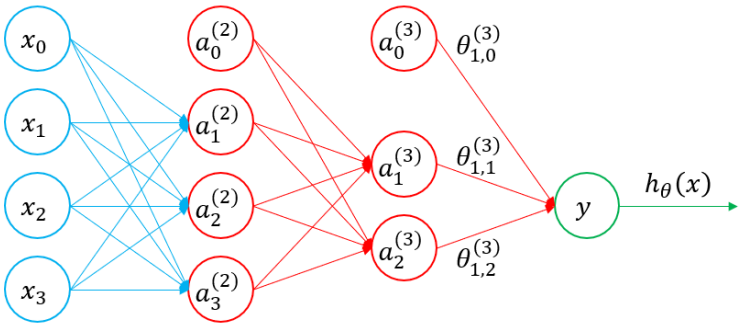

Abbildung 8: Beispiel für ein künstliches neuronales Netzwerk

Wie Abbildung 8 zeigt, wird auch bei neuronalen Netzen ein zusätzlicher Bias-Term mit dem festen Wert 1 benötigt. Die Knoten in diesem Netzwerk werden spaltenweise angeordnet. Eine komplette Spalte wird als Schicht (layer) bezeichnet.

Die erste Schicht (blau in Abbildung 8) ist die sogenannte Eingangsschicht, hier werden alle n Features eines Datums abgelegt und das neuronale Netz beginnt die Verarbeitung mit den Daten dieser Schicht. Analog dazu ist die letzte Schicht (grün in Abbildung 8) die Ausgabeschicht. Diese beiden Schichten sind normalerweise die einzigen Schichten, deren Werte sichtbar sind, weshalb alle Schichten dazwischen (rot in Abbildung 8) auch als versteckte Schichten (hidden layer) bezeichnet werden.

Bei Verwendung von ANNs müssen zwei wesentliche Ergänzungen zu den Definitionen im Abschnitt 2.2 gemacht werden. Die erste betrifft die Parameter θ. So ist bei neuronalen Netzen einerseits die Rede von Gewichten, andererseits werden die Parameter deutlich komplexer. Wie Abbildung 8 zeigt wird jede Schicht durch ihre Gewichte mit der nächsten Schicht verbunden, weshalb sich eine Nomenklatur der Form

$$\theta_{i,j}^{(l)} \tag{20}$$

anbietet. Die Terme sind wie folgt zu verstehen:

- (l) spezifiziert die Schicht, zu welcher das Gewicht gehört, $1,2,3,\dots,L$

- i spezifiziert das Zielneuron in der nächsten Schicht $(l+1)$
- j spezifiziert das Quellneuron in der aktuellen Schicht (l)

Insgesamt existieren dann $L-1$ Gewichtsmatrizen. Diese Matrizen müssen in den Dimensionen nicht unbedingt übereinstimmen, da auch die Anzahl der Neuronen von Schicht zu Schicht variieren kann.

Die zweite Ergänzung ist die Bezeichnung der einzelnen Neuronen. Diese werden analog mit einem Superskript versehen, um die zugehörige Schicht anzuzeigen. Außerdem besitzen die Neuronen einen Index um anzuzeigen, welche Position das Neuron innerhalb seiner Schicht belegt.

Die Bezeichnung einzelner Neuronen mit a weist darauf hin, dass die Ausgabe eines Neurons eine so genannte Aktivierung ist. Sie wird durch die sogenannte Aktivierungsfunktion erzeugt, deren Variablen durch eine Übertragungsfunktion festgelegt werden. Dabei bildet die Übertragungsfunktion Σ die Eingaben auf einen reellen Wert ab, welcher üblicherweise durch ein Skalarprodukt bestimmt wird:

$$\Sigma = \mathrm{x}^{\mathrm{T}}\theta \tag{21}$$

Die Aktivierungsfunktion kann prinzipiell jede Funktion sein, in der Regel werden hier monoton steigende Funktionen genutzt. Dabei sollten insbesondere keine Linearen Funktionen verwendet werden, da die entstehende Gesamtfunktion auch direkt als einzele Linearkombination dargestellt werden könnte – in diesem Fall wäre eine lineare Regression das bessere Modell. Eine häufig genutzte Aktivierungsfunktion ist daher eine Sigmoidfunktion wie die logistische Funktion (18).

Zusammengefasst gibt damit jedes Neuron abhängig von seinen Eingangswerten eine 0 oder eine 1 aus, welche von einer nachfolgenden Schicht genutzt wird, um ebenfalls wieder binäre Ausgaben zu erzeugen. Damit ist die Aktivierung eines einzelnen Neurons vergleichbar mit dem Ergebnis einer logistischen Regression.

Im Unterschied zur logistischen Regression kann ein neuronales Netzwerk jedoch für verschiedene Neuronen verschiedene Aktivierungsfunktionen besitzen, außerdem Bedarf ein neuronales Netzwerk nicht einer aufwendigen Erzeugung von komplexen Features, um auch komplexe Zusammenhänge wie in Abbildung 7 darstellen zu können.
Insbesondere multiple Klassifikationen sind durch mehrere Ausgabeneuronen sehr leicht realisierbar.

Zuletzt sei darauf hingewiesen, dass auch andere Verbindungen zwischen den Neuronen, beispielsweise Rückkopplungen innerhalb einer einzelnen Schicht und sogar Neuronen, möglich sind. Die Komplexität eines Netzes kann damit recht einfach aber schnell gesteigert und an hochkomplexe Aufgaben angepasst werden, jedoch zum Preis eines steigenden Rechenaufwands und der potentiellen Gefahr einer Überanpassung, wie in Kapitel 6 erläutert wird.

5.2 Forward Propagation

Die Forward Propagation entspricht der Vorhersage mittels der aktuellen Hypothese, vergleichbar zum Einsetzen von Werten in eine Funktion. Dabei wird ein Datum in die Eingabeschicht des ANN gegeben und die Neuronen der darauffolgenden Schicht berechnen ihre Ausgabe aus den Eingaben und den zugehörigen Gewichten. Diese Ausgaben werden dann wieder entsprechend gewichtet an die nächste Schicht übergeben, solange noch weitere Schichten vorhanden sind. Die Eingaben pflanzen sich also im neuronalen Netz „nach vorne“ fort.

Die Berechnung einer einzelnen Schicht kann wieder vektorisiert dargestellt werden, wobei an dieser Stelle nochmal auf den Unterschied zwischen dem Kleinbuchstaben θ für Vektoren oder skalare Einträge und dem Großbuchstaben Θ für Matrizen hingewiesen werden soll.

Die Ausgabe einer Schicht eines neuralen Netzwerks mit der auf Matrizen anwendbaren Aktivierungsfunktion $f^{(l)}(x)$ lautet dann

$$a^{(l+1)} = f^{(l+1)}\left(A^{(l)}\left(\Theta^{(l)}\right)^T\right) \tag{22}$$

Die Transponierung von Θ soll am Beispiel von Abbildung 8 und dem Übergang von der zweiten in die Dritte Schicht erläutert werden. Da in der zweiten Schicht 3 Features und der Bias vorhanden sind hat $A^{(2)}$ die Dimension $(m \times 4)$:

$$A^{(2)} = \begin{pmatrix} 1 & a_{1,1}^{(2)} & a_{1,2}^{(2)} & a_{1,3}^{(2)} \\ 1 & a_{2,1}^{(2)} & a_{2,2}^{(2)} & a_{2,3}^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & a_{m,1}^{(2)} & a_{m,2}^{(2)} & a_{m,3}^{(2)} \end{pmatrix} \tag{23}$$

Θ besteht aus insgesamt 8 Einträgen, welche sich als Produkt aus den 2 Neuronen in der dritten Schicht mit den 4 Neuronen in der zweiten Schicht ergeben. Damit ist

$$\Theta^{(2)} = \begin{pmatrix} \theta_{1,0}^{(2)} & \theta_{1,1}^{(2)} & \theta_{1,2}^{(2)} & \theta_{1,3}^{(2)} \\ \theta_{2,0}^{(2)} & \theta_{2,1}^{(2)} & \theta_{2,2}^{(2)} & \theta_{2,3}^{(2)} \end{pmatrix} \tag{24}$$

von der Dimension (2×4). Das Matrizenprodukt $A^{(2)}\left(\Theta^{(2)}\right)^T$ hat demnach die Dimension $(m \times 2)$.

Für das Netzwerk in Abbildung 8 ergibt sich damit das Modell, also die Forward Propagation Vorschrift

$$h_\Theta(x) = f^{(4)}\left(f^{(3)}\left(f^{(2)}\left(X\left(\Theta^{(1)}\right)^T\right)\cdot\left(\Theta^{(2)}\right)^T\right)\cdot\left(\Theta^{(3)}\right)^T\right) \tag{25}$$

Der Blaue Term ist die Übertragungsfunktion aus den Eingangsvariablen und deren Gewichten. In Grün hervorgehoben ist die Ausgabe der zweiten Schicht, die dritte Schicht ist entsprechend rot markiert und in Schwarz ist die Ausgabe der vierten Schicht, der Ausgabeschicht, gekennzeichnet.

Gleichung (25) gilt dabei insbesondere ohne die auf den ersten Blick naheliegende Einschränkung, dass alle Neuronen einer Schicht dieselbe Aktivierungsfunktion besitzen. Programmiertechnisch ist es über Funktionen mit vektoriellen Ausgaben leicht umsetzbar, in jedem Neuron eine andere Aktivierungsfunktion zu hinterlegen.

5.3 Backward Propagation

Bei der Backward Propagation handelt es sich um die Minimierungsaufgabe bezüglich der Kostenfunktion für neuronale Netze. Die Kostenfunktion selbst ist zunächst nicht anders motiviert als die Kostenfunktion der logistischen Regression (19). Sie muss jedoch noch erweitert werden, da neuronale Netzwerke in der Lage sind, mehr als eine Klassifikation vorzunehmen und einen binären Zustandsvektor zurückzugeben.

Eine direkte vektorielle Darstellung ist für diesen Fall leider nicht mehr möglich, ein teilweise vektorisierter Algorithmus ist aber möglich und beschleunigt die Berechnung. Die nicht-vektorisierte Kostenfunktion für multiple Ausgaben ergibt sich zu

$$J(\Theta) = \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \, log\left[\left(h_\Theta\left(x^{(i)}\right)\right)_k\right] + \left(1 - y_k^{(i)}\right) log\left[1 - \left(h_\Theta\left(x^{(i)}\right)\right)_k\right] \tag{26}$$

Der Faktor K in der zweiten Summe stellt die Anzahl der möglichen Ausgaben dar, entspricht also der Dimension des Vektors y. Das konkrete Modell ergibt sich gemäß der Beschreibung in Abschnitt 5.2 abhängig vom verwendeten neuronalen Netzwerk und soll daher nicht weiter spezifiziert werden.

Anschaulich summiert die Kostenfunktion die falsch positiven und die falsch negativen Zuordnungen der Hypothese für jeden der K möglichen Zustände zusammen.

Die nächste Besonderheit besteht in der hohen Verkettungs- und Verschachtelungstiefe der Hypothese und damit der Kostenfunktion, welche sich aus den zahlreichen Neuronen und deren Verknüpfungen untereinander ergibt. Es ist damit für gewöhnlich nicht mehr trivial, die Gradienten der Kostenfunktion zu bestimmen. Zusätzliche Komplexität entsteht durch den Umstand, dass für neuronale Netzwerke $L - 1$ Parametermatrizen $\Theta_{i,j}^{l}$ vorliegen.

Das Minimierungsproblem lässt sich jedoch durch Rückführung des „absoluten Fehlers“ am Ausgang des neuronalen Netzes auf die vorherigen Neuronen lösen, woher auch der Name Backward Propagation oder kurz Backpropagation für das Verfahren stammt. Dabei wird die Kettenregel verwendet, um ausgehend von der Ausgabe den lokalen Gradienten für ein Neuron zu bestimmen:

$$\frac{\partial \mathrm{h}_\Theta(x_j)}{\partial x_j} = \frac{\partial \mathrm{h}_\Theta(x_j)}{\partial a_j} \frac{\partial a_j}{\partial x_j} \tag{27}$$

Für ein einfaches Beispiel mit drei Eingaben der Form

$$f(x_1, x_2, x_3) = (x_1 + x_2)x_3 \tag{28}$$

kann ein einfaches Netz gemäß Abbildung 9 erstellt werden. Auf den Bias-Term sowie die Übertragungsfunktion wurde der Einfachheit halber verzichtet.

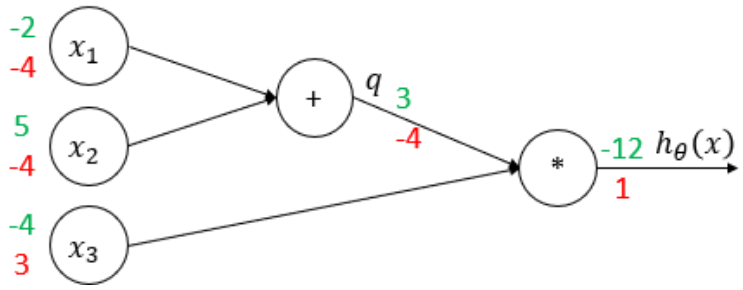

Abbildung 9: Einfaches Beispiel für Backpropagation[6]

In Grün dargestellt sind die Eingabewerte, welche mittels Forward Propagation zum versteckten Neuron $q = x_1 + x_2$ weitergeleitet werden und dort die Aktivierung 3 ergeben. In der Ausgabeebene

[6] Angelehnt an Li, Johnson, & Yeung, 2017

wird mittels $h_\Theta(x) = x_3 \cdot q$ das Gesamtergebnis -12 bestimmt. Für die Backpropagation wird das Netzwerk nun von rechts nach links durchlaufen. Dabei ist am Ausgang der lokale Gradient gemäß

$$\frac{\partial h_\Theta}{\partial h_\Theta} = 1 \tag{29}$$

trivial.

Im nächsten Schritt werden die lokalen Gradienten vor dem letzten Neuron gebildet. Für das versteckte Neuron gilt

$$\frac{\partial \mathrm{h}_\Theta}{\partial q} = \frac{\partial x_3 \cdot q}{\partial q} = x_3 = -4 \tag{30}$$

und für die dritte Eingabe

$$\frac{\partial \mathrm{h}_\Theta}{\partial \mathrm{x}_3} = \frac{\partial x_3 \cdot q}{\partial \mathrm{x}_3} = q = 3 \tag{31}$$

Beide Werte müssen nun noch mit dem Gradienten des Nachfolgers -1 multipliziert werden. Zuletzt werden die beiden lokalen Gradienten

$$\frac{\partial q}{\partial x_1} = \frac{\partial x_1 + x_2}{\partial x_1} = 1 \quad \text{und} \quad \frac{\partial q}{\partial x_2} = \frac{\partial x_1 + x_2}{\partial x_2} = 1 \tag{32}$$

bestimmt. Multipliziert mit dem Gradienten des Nachfolgers folgt damit der gesamte Gradient zu

$$\nabla = \begin{pmatrix} -4 \\ -4 \\ 3 \end{pmatrix} \tag{33}$$

Nach diesem Prinzip kann für jedes auch komplexe Netzwerk stückweise der Gradient bestimmt werden. Abbildung 10 zeigt das Beispiel der Backpropagation für die logistische Funktion. Dieses Beispiel verdeutlicht nochmal die Multiplikation des lokalen Gradienten eines Neurons mit dem Wert des lokalen Gradienten des Nachfolgers.

Abbildung 10: Backpropagation für eine Sigmoidfunktion[7]

In der Praxis besitzt ein Neuron natürlich mehrere Nachfolger. In diesem Fall wird die Summe der lokalen Nachfolgergradienten gebildet und mit dem aktuell betrachteten lokalen Gradienten multipliziert.

Dieser Ansatz kann auf die oben beschriebenen ANNs übertragen werden. Zunächst wird eine Forward Propagation mit zufällig initialisierten Gewichten durchgeführt – die Vorgehensweise wird hiernach erläutert. Nun kann der Fehler des Neurons j in Schicht l, bezeichnet als $\delta_j^{(l)}$, berechnet werden. Für die Ausgangsschicht ergibt sich damit der Fehler zu

$$\delta^{(L)} = a^{(L)} - y \tag{34}$$

[7] Li, Johnson, & Yeung, 2017

Einzelne Gewichte werden nach der Vorschrift

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} - \frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}} \tag{35}$$

aktualisiert. Da alle Änderungen simultan erfolgen sollen werden die neu berechneten Gewichte in der Backpropagation meist in zusätzlichen, hier mit Delta betitelten Variablen gespeichert. Die jeweiligen Gradienten werde stückweise per Backpropagation bestimmt. Für die logistische Funktion als Aktivierungsfunktion aller Neuronen lässt sich der Fehler für eine vorgelagerte Schicht direkt zu

$$\delta^{(l)} = \left(\Theta^{(l)}\right)^T \delta^{(l+1)} .* a^{(l)} .* \left(1 - a(l)\right) \tag{36}$$

bestimmen. Damit kann auch der gesuchte Gradient berechnet werden:

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \tag{37}$$

Eingesetzt in (36) kann so die Aktualisierungsvorschrift definiert werden zu:

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} - a_j^{(l)} \delta_i^{(l+1)} \tag{38}$$

Die mathematischen Grundlagen zum Trainieren eines neuronalen Netzwerks sind hiermit vorhanden. Die nötige zufällige Initialisierung der Gewichte erfolgt als Faustregel in dem Bereich $[-\varepsilon, \varepsilon]$, wobei

$$\varepsilon^{(l)} = \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}} \tag{39}$$

gilt. Damit wird für jede Schicht ein eigener Bereich definiert, wobei n_{in} die Anzahl der eingehenden Verbindungen in diese Schicht ist. Analog entspricht n_{out} der Anzahl an ausgehenden Verbindungen aus der betrachteten Schicht.

Mit dieser Vorarbeit kann eine Berechnungsvorschrift für die Backpropagation für ein ANN aufgestellt werden:

- Die zu berechnenden Anpassungen an die Gewichte werden zu $\Delta_{i,j}^{(l)} := 0$ initialisiert
- Die Gewichte werden zufällig initialisiert
- In einer Schleife über alle Trainingsdaten:
  - Setzen der Eingangsschicht zu $a^{(1)} = x^{(i)}$
  - Alle Aktivierungsfunktionen $a^{(l)}$ mittels Forward Propagation bestimmen
  - Die Fehler $\delta^{(L)}$, $\delta^{(L-1)}$, …, $\delta^{(2)}$ bzw. die lokalen Gradienten mittels Backpropagation bestimmen
  - Die nötigen Anpassungen berechnen: $\Delta^{(l)} := \Delta^{(l)} - \delta_i^{(l+1)} \left(a_j^{(l)}\right)^T$
  - Nach durchlauf eines kompletten Trainingsdatums die Gewichte anpassen

Diese Vorschrift minimiert den Fehler abhängig von allen Θ, wobei das neuronale Netzwerk mit jedem Durchlauf eines Datums durch den Algorithmus seine Gewichte modifiziert. Für eine abweichende Aktivierungsfunktion muss entweder eine Verallgemeinerung der Gradienten wie in (37) und (38) gefunden werden oder tatsächlich eine stückweise Backpropagation durchgeführt werden.

Bei Bedarf kann die Anpassung außerdem durch eine Lernrate angepasst werden. Weiterhin kann für etwas bessere Ergebnisse statt einem Aktualisieren der Gewichte nach jedem Durchlauf auch ein beliebig großer Batch an Trainingsdaten durchlaufen werden. Die jeweiligen Anpassungen werden anschließend akkumuliert, was das Netzwerk weniger anfällig für statistische Ausreißer macht.

In der Praxis bietet sich für das Anlernen eines ANN oft eine Rekursion als effektive Lösung an, welche zunächst die Forward Propagation durchführt und bei der Rückabwicklung der Rekursionen die Backpropagation durchführt und die neuen Gewichte bestimmt, um diese auf oberster Ebene zu aktualisieren.

5.4 Vor- und Nachteile künstlicher neuronaler Netzwerke

Neuronale Netzwerke sind häufig sehr flexibel auf unterschiedlichste Anwendungen anlernbar, ohne große Änderungen vornehmen zu müssen. Sie können daher mit geringem Anpassungsaufwand eine Vielzahl von Problemen lösen.

Durch mehr oder größere Schichten können schnell und unkompliziert Adaptionen an komplexere Problemstellungen vorgenommen werden und auch schwierige Aufgaben wie eine Handschrift-erkennung gelöst werden. Eine einfache Bilderkennung zur Identifizierung der Ziffern von Null bis Neun ist bereits in einem Netz mit nur vier Schichten realisierbar.

Gleichzeitig ist der Rechenaufwand aufgrund der Komplexität sehr hoch, weshalb neuronale Netzwerke erst seit den Zweitausendzehnerjahren dank der gesteigerten Computerleistung effektiv genutzt werden.

Außerdem ist es oft schwierig bis unmöglich zu verifizieren, ob die Ausgabe eines komplexen neuronalen Netzwerks die tatsächlich gewünschten Features der Eingaben erkannt hat, bzw. den Entscheidungsfindungsprozess eines neuronalen Netzwerks nachzuvollziehen. Ein Problem an dieser Stelle zeigt sich leider erst bei Anwendung des ANN auf neuen Daten durch plötzlich deutlich höhere Fehler.

Zuletzt neigen neuronale Netzwerke aufgrund ihrer hohen Komplexität dazu, sich zu stark an ihre Daten anzupassen. Mehr hierzu im nächsten Abschnitt.

6 Typische Probleme und Lösungsansätze

Bei der Anwendung maschineller Lernalgorithmen können im Wesentlichen zwei verschiedene Fehlertypen auftreten: Eine Überanpassung (overfit oder high variance) oder eine Unteranpassung (underfit oder high bias) an die vorliegenden Daten. Diese Probleme können bei allen Lernalgorithmen auftreten, nicht nur bei neuronalen Netzwerken.

Abbildung 11 zeigt im ersten Fall eine Unteranpassung. Augenscheinlich können die Datenpunkte sehr gut durch eine quadratische Funktion wie im zweiten Fall dargestellt werden, eine einfache Gerade kann dem Verlauf der Punkte jedoch nicht folgen und verursacht damit einen sehr hohen Fehler.
Im Gegensatz dazu zeigt der dritte Fall eine Überanpassung. Hierbei wird der Fehler auf den Trainingsdaten (nahezu) verschwinden, da der Algorithmus alle Datenpunkte quasi exakt trifft. Es erscheint jedoch wahrscheinlich, dass weitere Datenpunkte deutlich von der Kurve abweichen werden und damit in der Anwendung ebenfalls hohe Fehler erzeugt werden.

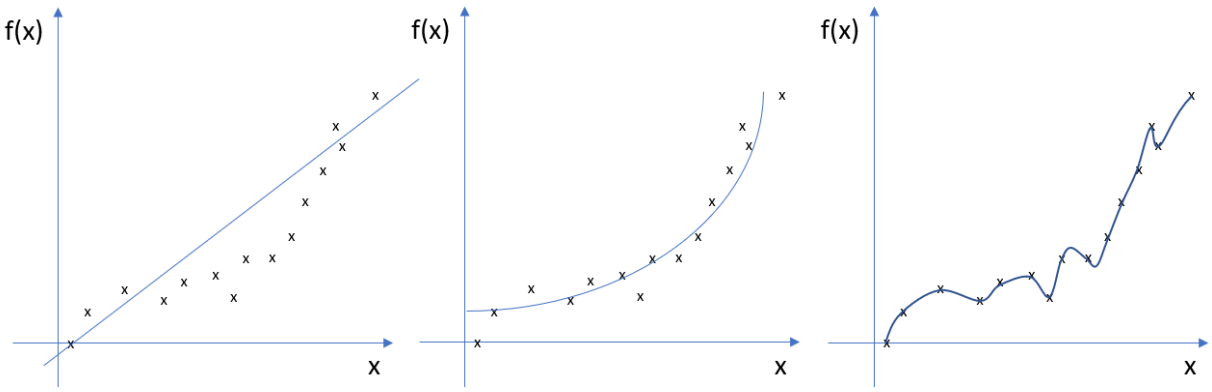

Abbildung 11: Unteranpassung, gute Anpassung und Überanpassung

Die Lösungsansätze sollen nur der Vollständigkeit halber, aber ohne nähere Details erklärt werden:

Für das Problem der Unteranpassung hilft es, das Modell komplexer zu gestalten, entweder durch Hinzuziehen weiterer Features, oder durch Erstellung von neuen Features aus vorhandenen Daten, z.B. durch Berechnung höherer Polynome oder Verrechnung mehrerer Features. Bei neuronalen Netzen kann eine weitere versteckte Schicht hinzugefügt oder die Anzahl an Neuronen in den vorhandenen Schichten erhöht werden.

Bei einer Überanpassung können in einem neuronalen Netzwerk zwar recht einfach Schichten entfernt werden, bei anderen Algorithmen können jedoch nicht immer Features vernachlässigt werden, um eine geringere Komplexität zu erreichen, da hierdurch Informationen verloren gehen, die aber einen realen Einfluss auf die betrachteten Daten haben.

Eine Alternative zum Entfernen von Features oder Neuronen bzw. ganzen Schichten stellt die Regulierung (regularization) dar. Dabei wird der Einfluss der Parameter θ bei der Minimierung über einen Vorfaktor abgedämpft, wodurch das Problem der Überanpassung minimiert wird.

Hierzu wird in der Kostenfunktion einfach ein weiterer Faktor hinzugefügt, welcher die Parameter bei stärkeren Anpassungen entsprechend kostspieliger macht, weshalb zu starke Anpassungen bei der Minimierung vermieden werden.

7 Die Erweiterung zum Convolutional Neural Network

7.1 Motivation und Anwendungsgebiet

Ein Convolutional Neural Network ist ein spezialisiertes Netz zur Verarbeitung von Bild- und Audiodateien. Der Begriff Convolutional Neural Network, kurz CNN, lässt sich als gefaltetes neuronales Netzwerk übersetzen. Namensgebend ist dabei die erste und wichtigste der drei Schichten, in die sich das CNN aufteilen lässt, wie im nächsten Abschnitt näher erläutert wird.

Bei der Verarbeitung von Audio- und Bilddaten wird ein maschineller Lernalgorithmus vor eine große Herausforderung durch die Menge an Features gestellt. Wird beispielsweise ein Bild mit einer Auflösung von 5MP RGB-kodiert gespeichert, so verfügt ein einzelnes Bild bereits über rund 15 Millionen Features. Dies würde für ein einfaches Netzwerk wie in Kapitel 5 vorgestellt bedeuten, dass das Netzwerk über 15 Millionen Eingänge und entsprechend viele Verbindungen pro folgendem Neuron verfügen muss. Bei ebenfalls 15 Millionen Neuronen in der folgenden Schicht wären alleine dies bereits $2.25 \cdot 10^{14}$ Verbindungen.

Die Bearbeitung ist in der Theorie natürlich möglich, bedarf aber einer entsprechend hohen Rechenleistung und ist sehr Zeitintensiv. Für die Praxis ist so ein Ansatz daher in der Regel nicht brauchbar, weshalb eine Weiterentwicklung unabdingbar ist. Eine dieser Weiterentwicklungen stellen CNNs dar.

Ein weiteres Problem bei der Verarbeitung typischer Bildgrößen in ANNs stellt eine rapide steigende Wahrscheinlichkeit der Überanpassung dar. Um aus der hohen Zahl an Eingangswerten überhaupt sinnvolle Ausgaben erzeugen zu können bedarf das Netzwerk vieler versteckter Schichten, welche nach und nach in ihrer Größe reduziert werden können, bis eine Ausgabe möglich ist. Wie in Kapitel 6 beschrieben neigt ein Netzwerk mit vielen und großen Schichten jedoch stark zur Überanpassung.

7.2 Aufbau eines CNN

Der wesentliche Unterschied zum einfachen ANN liegt darin, dass ein CNN mit dreidimensionalen Matrizen als Eingangsgröße arbeitet. Ein RGB-Bild mit der Auflösung 64×64 Pixel erzeugt eine Eingabe der Größe $64 \times 64 \times 3$. Diese Dimensionen werden auch als Höhe, Breite und Tiefe bezeichnet. Zum Vergleich: ein ANN benötigt immer einen Vektor als Eingabe, welcher in diesem Beispiel die Eingangsdimension 12288×1 besitzen müsste.

Ein CNN verfügt darüber hinaus über drei unterschiedlich arbeitende Schichten:

1. Die Faltungs-Schicht (convolutional layer)
2. Die Pooling-Schicht, auch Subsampling genannt
3. Die vollständig vernetzte/ vermaschte Schicht (fully connected layer)

Dabei folgt die Pooling-Schicht immer einer oder mehrerer Faltungsschichten. Kombinationen beider Schichten können außerdem mehrfach hintereinandergeschaltet werden, analog zum Einfügen mehrerer versteckter Schichten in einem einfachen neuronalen Netzwerk. Den Abschluss bildet jedoch immer die vollständig vernetzte Schicht.
Faltungs- und Pooling-Schicht werden dabei durch lokal vermaschte Teilnetze realisiert, sodass die Gesamtzahl an Verbindungen auch bei großen Eingabemengen im für praktische Anwendungen nutzbarem Bereich verbleiben.

7.2.1 Die Faltungs-Schicht

Die Faltungsschicht hat die Aufgabe, verschiedene Merkmale wie horizontale oder vertikale Linien, Kanten, Farbmuster etc. aus einer Eingabe zu filtern. Dazu wird ein so genannter Kernel als Filtermatrix mit geringer Größe, beispielsweise 3×3, für jedes Feature angelernt. Diese Kernel werden dabei genau wie Θ vom Algorithmus selbst erlernt.

Jeder dieser Kernel wird nun Schrittweise von links nach rechts über das Bild geführt und springt anschließend Zeilen entsprechend der Schrittweite nach unten. Dabei werden die Werte des aktuell betrachteten Bildausschnitts elementweise mit dem Kernel multipliziert und summiert, das Ergebnis ist also ein einzelner skalarer Wert.

Im Randbereich findet dabei häufig ein sogenanntes Padding statt, um die Dimensionen der gefilterten Daten nicht zu verändern. Hierbei ist eine häufige Methode, einen Rand aus Null-Einträgen um das Bild herum zu legen, oder den Rand zu duplizieren.

Eingabe

0	0	0	0	0	0	0	0
0	4	8	2	6	4	2	0
0	1	8	4	7	1	7	0
0	3	5	7	0	4	4	0
0	2	7	1	2	0	3	0
0	2	9	6	6	4	5	0
0	4	5	0	3	8	1	0
0	0	0	0	0	0	0	0

Kernel

-1	-1	2
2	-1	2
2	0	-1

Ausgabe bei Schrittweite 1

4	2	35	13	19	8
22	-7	38	13	17	-3
15	17	17	15	11	-4
14	13	12	-7	19	5
18	-4	12	10	4	12
16	1	-3	-4	0	-9

Ausgabe bei Schrittweite 2

4	35	19
15	17	11
18	12	4

Abbildung 12: Faltung mit 0-Padding und unterschiedlichen Schrittweiten bei einer Tiefe von 1

Die Filter werden dabei auf die gesamte Tiefe D der Eingabe angewendet, es werden also D Ausgabenmatrizen pro Kernel erzeugt. Diese werden in der Regel einfach addiert und erzeugen damit eine zweidimensionale Ausgabematrix.

Die Ausgabematrix, auch Feature Map genannt, entspricht mathematisch gesehen einer Faltung, was der Schicht und damit dem gesamten Netzwerk den Namen gibt. Wie viele Kernel pro Schicht eingesetzt werden hängt wieder vom Problem ab und kann sich von Netz zu Netz unterscheiden.

Im nächsten Schritt wird eine Aktivierungsfunktion auf alle Ausgaben der Faltung angewendet, wie bei einem einfachen Neuronalen Netz auch. Für ein CNN ist die Standardaktivierungsfunktion die Rectified Linear Unit ReLU:

$$\mathrm{ReLu}(x) = \ln(1 + e^x) \tag{40}$$

Eine Sigmoidfunktion sollte hier explizit **nicht** genutzt werden. Häufig werden außerdem mehrere Faltungsschichten hintereinandergeschaltet, bevor eine Pooling-Schicht geschaltet wird.

7.2.2 Die Pooling-Schicht

Nach den Faltungsschichten folgt eine Pooling-Schicht, welche zur Datenreduktion dient. Dabei wird in der Regel in einem 2×2 Bereich der maximale Wert ausgewählt und in eine neue Matrix überführt, dies ist das sogenannte Max-Pooling. Die Feature-Map wird dabei mit Schrittweite 2 durchlaufen, sodass insgesamt eine Datenreduktion um 75% pro Pooling-Schicht erfolgt.

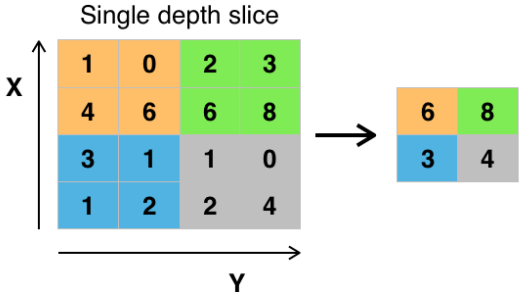

Abbildung 13: Veranschaulichung des Max-Pooling-Prozesses[8]

[8] Aphex34, 2015

Auch wenn auf den ersten Blick dadurch unzulässig viele Daten verloren gehen stellt sich heraus, dass die Performance des CNN sogar steigt. Zum einen wirkt die deutliche Datenreduktion einer potentiellen Überanpassung entgegen. Andererseits ermöglicht die deutliche Reduktion eine Erzeugung tieferer Netze, welche dadurch deutlich komplexere Probleme behandeln können, ohne an Leistung einzubüßen.

Und zuletzt ist bereits durch die Faltung der Umstand berücksichtigt worden, dass benachbarte Bereiche in einer Eingangsmatrix in der Regel korrelieren und damit eine gewisse Redundanz aufweisen. Durch das Pooling werden also in der Regel keine relevanten Informationen vernichtet, vielmehr wird das Neuron, welches im aktuell betrachteten Bereich ein bestimmtes Merkmal mit der höchsten Sicherheit erkannt hat, weiter berücksichtigt.
In der Praxis geht damit nicht die Tatsache verloren, dass beispielsweise eine Kante erkannt wurde, es wird jedoch die Information über die exakte Position der Kante in eine ungefähre Position reduziert, was zur Objekterkennung aber vollkommen ausreicht. Vergleichbar ist dies mit dem Erkennen eines großen und eines kleinen Hauses – die Kanten wie an Mauern, Fenster, Türen etc. werden nicht an derselben Stelle sein, jedoch wird ein Objekt mit Mauern, Fenstern und Türen höchst wahrscheinlich ein Haus sein.

7.2.3 Die vollständig vernetzte Schicht

Den Abschluss einer hinreichend reduzierten Eingabe bildet die vollständig vernetzte Schicht. Die Eingabe dieser Schicht besteht aus einer größeren Zahl an Feature Maps, welche durch mehrere Faltungs- und Poolingschichten erzeugt wurden. Diese Feature Maps sind von der Dimensionierung eher klein, beispielsweise 8×8, sodass auch bei einer großen Anzahl von beispielsweise 256 Feature Maps gerade einmal 16,388 Eingabewerte vorliegen. Einige Tausend Werte kann ein heutiger Prozessor jedoch in Sekundenschnelle verarbeiten, womit das ursprüngliche Problem der zahlreichen Eingabeparameter bei hochauflösenden Bildern eliminiert ist.

Die vollständig vernetzte Schicht ist daher ein gewöhnliches ANN, welches die durch Faltung und Pooling aufgearbeiteten Daten wie in Kapitel 5 beschrieben auswertet und eine Kategorisierung vornimmt. Um eine Eingabe in dieses ANN zu ermöglichen werden vorher alle Feature Maps in einen Vektor ausgerollt, dieser Vorgang wird auch als flatten bezeichnet. Der große Vorteil besteht darin, dass nun das Vorhandensein bestimmter Features durch das ANN überprüft werden kann, unabhängig von der Lage ihres Auftretens im Originalbild.

Das Erlernen von Parametern und Kernels geschieht wieder durch Backpropagation, indem der Fehler auf alle Parameter und die Kernel zurückgeführt wird.

7.3 Objektdetektion mit CNNs in der Praxis

In der Praxis wird die Arbeit mit einem CNN im Wesentlichen durch zwei Aspekte beeinflusst. Zunächst müssen natürlich Kernel gefunden werden, welche die gewünschten Features extrahieren können. Dabei werden die erkannten Merkmale durch jede Faltung immer komplexer, gleichzeitig aber werden die Ausgaben immer schwieriger zu interpretieren. Häufig werden daher sogenannte vorgelernte CNNs eingesetzt, welche dann durch sogenanntes transferlearning auf die eigenen Aufgaben abgestimmt werden.

Die Abbildungen Abbildung 14 und Abbildung 15 beispielsweise zeigen zwei bereits vorgelernte Kernel, welche im Python-Framework TensorFlow hinterlegt sind.

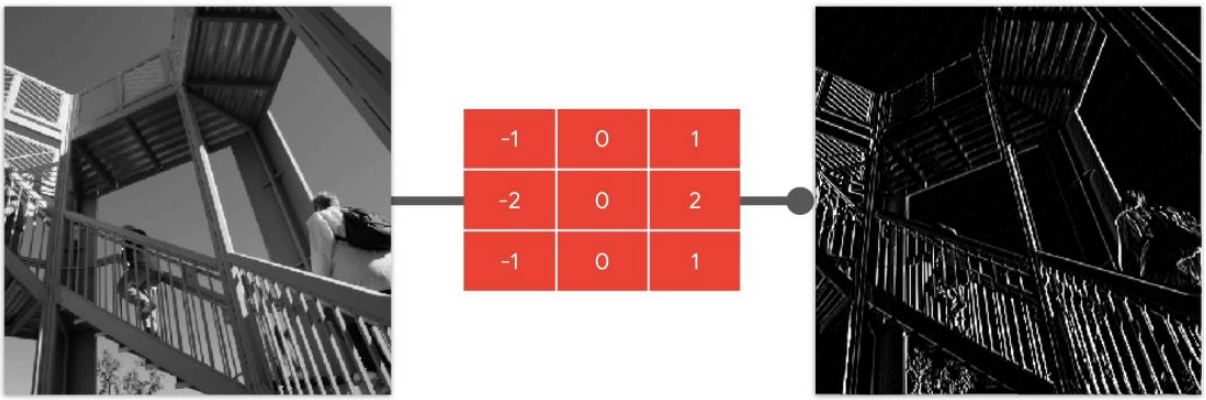

Abbildung 14: Kernel zum Erkennen senkrechter Linien mit Originalbild und Ausgabebild des Kernels[9]

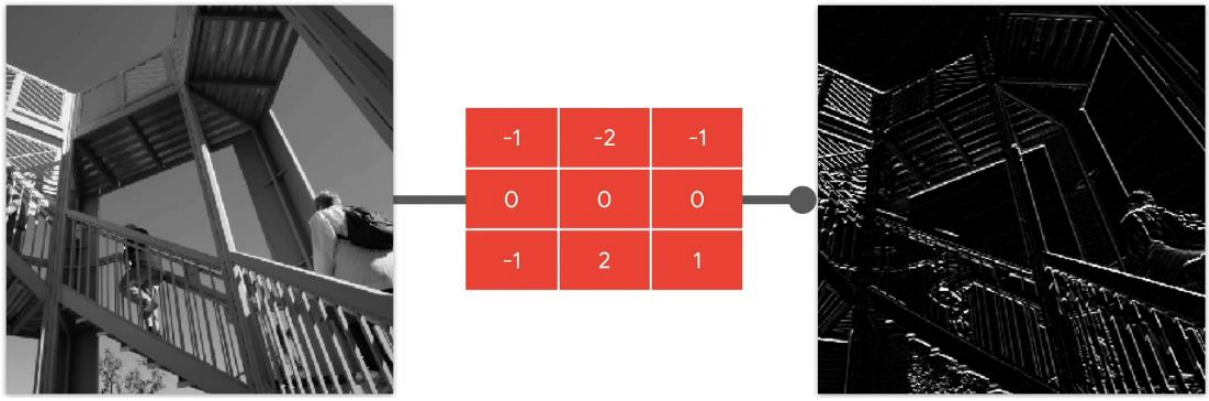

Abbildung 15: Kernel zum Erkennen waagerechter Linien mit Originalbild und Ausgabebild des Kernels[9]

In der Praxis werden häufig genau solche vordefinierten Filter genutzt, während die Erstellung neuer Filter eher eine Aufgabe für die Entwicklungsabteilungen darstellt. Vorgelernt bedeutet also, dass die Kernel bereits vorhanden sind und in der weiteren Nutzung kaum bis gar nicht verändert werden.

Ein anschließendes Pooling wird in

Abbildung 16 für die Ausgabe des Kernels gemäß Abbildung 14 dargestellt. Dabei handelt es sich um ein 2×2 Max Pooling, wie in Abschnitt 7.2.2 erläutert.

Abbildung 16: 2×2 *Max Pooling von Abbildung 14*[9]

Für diese einfachen Merkmale lässt sich die Ausgabe einer Faltung durch den Menschen gut nachvollziehen. Außerdem zeigt das anschließende Pooling, dass die erkannten Linien deutlich stärker hervortreten, als direkt nach der Filterung.

Dies entspricht dem in Abschnitt 7.2.2 erläuterten Effekt und zeigt, dass trotz einer Datenreduktion um 75% keine relevanten Informationen verloren gegangen sind.

Mit jeder weiteren Faltung werden die erkannten Features nun komplexer, so werden aus einfachen Linien geometrische Formen wie Ecken, Kurven etc. Diese werden in den weiteren Faltung dann zu

[9] TensorFlow, 2020

noch komplexeren Formen und nach einigen Faltungen erkennt der Algorithmus beispielsweise Türen, Fenster, Mauersteine etc. und ist damit in der Lage ein Haus auf einem Bild zu detektieren,
Aus einfachen Linien könnten auch Kurven werden, anschließend runde Formen wie Kreise und Ovale später Zylinder, welche letztlich als Arme, Beine, Körper und Köpfe einen Menschen kennzeichnen. Die wirkliche Herausforderung ist dann die Unterscheidung zwischen ähnlichen Objekten wie Hunden oder Katzen – beides lässt sich grob durch dieselben Formen darstellen. Aus diesem Grund werden auch Filter zur Erkennung von Farben, Farbmustern etc. und entsprechend viele Kernel in der ersten Schicht eingesetzt.

Die zweite Schwierigkeit bei der Anwendung eines CNN besteht in der Auswahl der Geometrie. Leider lässt sich schwer bis gar nicht vorhersagen, welche Kombinationen von Faltungs- und Poolingschichten mit welcher Größe tatsächlich in der Lage sind, die gesuchten Objekte mit hinreichender Genauigkeit zu identifizieren.

Vielmehr bedarf es hier einer Mischung aus Erfahrung, Intuition und leider auch schlichtweg Ausprobieren, um eine gute Lösung zu finden, weshalb häufig auch die Struktur bereits vorhandener Netzwerke genutzt wird und die eigentliche Anpassung somit tatsächlich nur im ANN der vollständig vermaschten Schicht stattfindet.

Bekannte CNNs wie das GoogLeNet, welches bereits 2014 den IMAGENET[10] Wettbewerb Large Scale Visual Recognition Challenge in einigen Kategorien gewinnen konnte, bestehen aus vielen komplexen Schichten. Das GoogLeNet besteht alleine aus 22 Faltungsschichten. Diese werden in Abbildung 17 in Blau dargestellt.

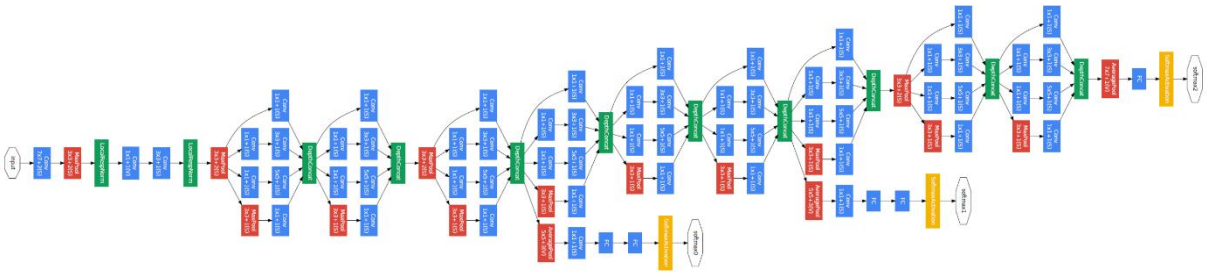

Abbildung 17: Visualisierung des GoogLeNet CNN[11]

Für viele Anwendungen werden daher vorhandene Netzwerke vollständig übernommen und lediglich die Auswertungen der vom CNN gefunden Features in der letzten Phase durch das ANN angelernt. Aus diesem Grund bieten aktuelle Versionen spezialisierter Programmiersprachen wie MATLAB oder Frameworks wie TensorFlow bereits vollständige Implementationen zur Nutzung an.

Gleichzeitig bedeutet dies, dass die Entwicklung von Netzarchitekturen und Kernels aktuell vor allem Teil der universitären und privaten Forschung ist.

[10] http://www.image-net.org/challenges/LSVRC/
[11] Szegedy, et al., 2015

8 Zusammenfassung und Ausblick

Viele moderne Lernalgorithmen, so auch die neuronalen Netzwerke, sind in der Theorie schon deutlich länger bekannt, waren jedoch aufgrund der nicht vorhandenen Trainingsdaten und der fehlenden Rechenleistung sowie Speicherkapazität lange Zeit nicht für die Praxis relevant.

Dank der stetig ansteigenden Leistung der verfügbaren Hardware sind maschinelle Lernalgorithmen nicht mehr aus dem heutigen Alltag wegzudenken, sei es bei der Filmempfehlung in der Mediathek, den empfohlenen Beiträgen in sozialen Netzwerken oder auch einfach nur der Spamschutz im E-Mailprogramm.

Gleichzeitig steigt durch die Digitalisierung sowohl im beruflichen als auch im privaten Umfeld die verfügbare Menge an Trainingsdaten exponentiell an, was immer besser trainierte und komplexere Algorithmen erlaubt.

Auch in der Forschung in anderen Fachbereichen wird maschinelles Lernen immer wichtiger. Die fortschreitende Entschlüsselung des menschlichen Genoms wäre von Hand ebenso undenkbar wie die Klassifikation von Milliarden Galaxien, welche immer leistungsstärkere Teleskope Nacht für Nacht enthüllen.

Die hier vorgestellten Convolutional Neural Networks stellen dabei eine spezialisierte Software für die Auswertung von Bild- und Videodaten dar. Sie sind hoch effizient und in der Lage, komplexe Probleme zu lösen. Mit ihnen ist es möglich, Gesichtserkennungen durchzuführen oder menschliche Sprache zu erfassen.
Nutzer, die ihr Smartphone per Gesichtserkennung entsperren nutzen damit höchst wahrscheinlich jeden Tag vollkommen unbemerkt ein Convolutional Neural Network, genauso wie Besitzer von Smart Home Geräten, wenn sie per Sprachbefehl das Licht oder das Radio steuern.

Das Themengebiet des maschinellen Lernens ist sehr interessant und stellt eine immer wichtigere Unterkategorie der Informatik dar. Dabei ist die Architektur von Netzen und das Erarbeiten von Filtern hoch aktuelle Forschung, welche derzeit wie Hardware nach Moore‘s Law rasant zu immer Leistungsstärkeren Konzepten führt.
Aufgrund dessen ist das Thema jedoch auch entsprechend komplex, weshalb auf viele wichtige und interessante Aspekte rund um maschinelles Lernen in dieser Ausarbeitung leider nicht eingegangen werden kann.
Beispielsweise ist ein wesentlicher Aspekt, der noch offenbleibt, die Quantisierung der Qualität einer Hypothese, welche durch maschinelles Lernen erzeugt wird. Und so interessant neuronale Netze und deren Weiterentwicklungen auch sind, so reichen für viele Probleme doch einfachere aber oft umso elegantere Algorithmen, welche in Kapitel 4 nur sehr kurz angeschnitten wurden.

Allen interessierten Lesern seien daher beispielsweise der Einstiegskurs „Machine Learning“[12] oder auch der Blog von „towards data science“ [13] empfohlen, welche in der Regel mit guten Zusammenfassungen und interessanten Visualisierungen überzeugen. Zu einiger Berühmtheit haben es auch die CS231-Kurse der Stanford University School of Engineering geschafft, welche größtenteils kostenlos online auffindbar sind und denen einige Abbildungen entnommen wurden.

[12] https://www.coursera.org/learn/machine-learning
[13] https://towardsdatascience.com/tagged/tds-explore

Technische Hochschule Georg Agricola

Literaturverzeichnis

Aphex34. (16. December 2015). *wikimedia.* Von https://commons.wikimedia.org/wiki/File:Max_pooling.png abgerufen

Gross, D. (03. Januar 2017). *JAXenter.* Von https://jaxenter.de/big-data-bildanalyse-50313# abgerufen

Kodinariya, T., & Makwana, P. (November 2013). *International Journal of Advance Research inComputer Science and Management Studies.* Von https://d1wqtxts1xzle7.cloudfront.net/34194098/V1I6-0015.pdf?1405312820=&response-content-disposition=inline%3B+filename%3DReview_on_determining_number_of_Cluster.pdf&Expires=1610120759&Signature=G2GQ-IUBAaNGl-ZnI6~2XKb7wR9kf1pvjX4KPc7R3cdLynw1CzZMZjISSi- abgerufen

Li, F.-F., Johnson, J., & Yeung, S. (13. April 2017). Von Stanford CS231N: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf abgerufen

Lopez, C., Tucker, S., Salameh, T., & Tucker, C. (September 2018). Von ScienceDirect: https://www.sciencedirect.com/science/article/pii/S1532046418301308 abgerufen

Mitchell, T. (04. Dezember 2017). *Key Ideas in Machine Learning.* Von http://www.cs.cmu.edu/%7Etom/mlbook/keyIdeas.pdf abgerufen

Nanda, J. (Oktober 2016). Von https://www.cc.gatech.edu/classes/AY2016/cs4476_fall/results/proj4/html/jnanda3/index.html abgerufen

NG, A. (28. 12 2020). Von https://www.coursera.org/learn/machine-learning/supplement/ExY6Z/lecture-slides abgerufen

S, S. (03. October 2017). Von machinelearningmedium: Für den nächsten Schritt wird die Ableitung f‘ der Aktivierungsfunktion benötigt. Die Operation Geben Sie hier eine Formel ein. abgerufen

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Magic Leap Inc. (15. October 2015). Von https://static.googleusercontent.com/media/research.google.com/de//pubs/archive/43022.pdf abgerufen

TensorFlow. (17. Januar 2020). Von https://www.youtube.com/watch?v=ZdTang-aIP4 abgerufen

The MathWorks, Inc. (07. Januar 2021). *MathWorks Help Center*. Von https://de.mathworks.com/help/optim/ug/fminunc.html abgerufen

Thoma, M. (07. Januar 2021). *wikimedia.* Von https://upload.wikimedia.org/wikipedia/commons/5/53/Sigmoid-function-2.svg abgerufen