Grundbegriffe des maschinellen Lernens

Neuronale Netze

Besondere Lernleistung im Fach Informatik

Clemens Dautermann

4. Februar 2020

Inhaltsverzeichnis

1 Was ist maschinelles Lernen?

Die wohl bekannteste und am häufigsten zitierte Definiton des maschinellen Lernens stammt von Arthur Samuel aus dem Jahr 1959. Er war Pionier auf diesem Gebiet und rief den Begriff „machine learning" ins Leben. So sagte er:

> [Machine learning is the] field of study that gives computers the ability to learn without being explicitly programmed[**1**].
>
> —Arthur Samuel, 1959

Beim maschinellen lernen werden Computer also nicht mit einem bestimmten Algorythmus programmiert um eine Aufgabe zu lösen, sondern lernen eigenständig diese Aufgabe zu bewältigen. Dies geschieht zumeist, indem das Programm aus einer großen, bereits „gelabelten", Datenmenge mit Hilfe bestimmter Methoden, die im Folgenden weiter erläutert werden sollen, lernt, gewisse Muster abzuleiten um eine ähnliche Datenmenge selber „labeln" zu können. Als Label bezeichent man in diesem Fall die gewünschte Ausgabe des Programmes. Dies kann beispielsweise eine Klassifikation sein. Soll das Programm etwa handgeschriebene Ziffern erkennen können, so bezeichnet man das (bearbeitete) Bild der Ziffer als „Input Verctor" und die Information welche Ziffer der Copmputer hätte erkennen sollen, als „Label ". Soll jedoch maschinell erlernt werden, ein simuliertes Auto zu fahren, so bestünde der Input Vector aus Sensorinformationen und das Label würde aussagen, in welche Richtung das Lenkrad hätte gedreht werden sollen, wie viel Gas das Programm hätte geben sollen oder andere Steuerungsinformationen. Der Input Vector ist also immer die Eingabe, die der Computer erhält um daraus zu lernen und das Label ist die richtige Antwort, die vom Programm erwartet wurde. Für maschinelles Lernen wird also vor allem eins benötigt: Ein enormer Datensatz, der bereits gelabelt wurde, damit das Programm daraus lernen kann.
Natürlich werden für maschinelles Lernen trotzdem Algorythmen benötigt. Diese Algorythmen sind jedoch keine problemspezifischen Algorythmen, sondern Algorythmen für maschinelles Lernen. Eine der populärsten Methoden des maschinellen Lernens ist das sogenannte „Neuronale Netz". Dies wird für die zwei Hauptproblemklassen, die man unterscheidet, verwendet. Klassifizierungs und Regressionsprobleme.

1.1 Klassifizierungsprobleme

Als Klassifizierfung bezeichnet man das Finden einer Funktion, die eine Menge von Eingabevariablen zu einer diskreten Menge von Ausgabevariablen, die auch als Klassen oder Labels bezeichnet werden, zuordnet. Dies kann beispielsweise das Erkennen von Mail Spam sein. Die Eingabevariablen sind die E-Mails und sie sollen den zwei Klassen „Spam" und „nicht Spam" zugeortnet werden.
Das in Dieser Arbeit gegebene Beispiel ist auch ein Klassifizierungsproblem. Die gegebenen Bilder von Ziffern sollen den zehn Klassen „0 bis 9" zugeordnet werden. Die Bilder sind hier die Eingabevariablen und die Klassen null bis neun beschreibt die endliche Menge diskreter Labels.
Das Erste Beispiel würde man als „Binärklassifizierung" bezeichnen, da zwei Klassen

unterschieden werden. Letzteres wird als „Multiklassenklassifizierung" bezeichnet, da mehr als zwei Klassen unterschieden werden. Die Binärklassifizierung ist in Abbildung 1 verbildlicht.

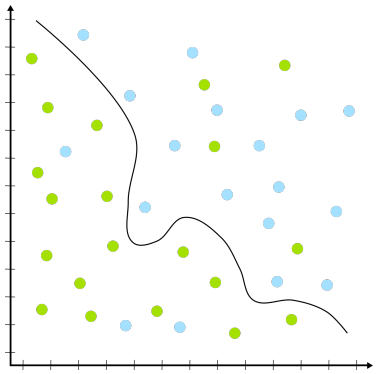

Abbildung 1: Binärklassifizierung

Die zwei Klassen wären hier „grün" und „blau". Die Linie stellt die Klassengrenze dar, die die zwei Klassen unterscheidet. Es sind außerdem einige Ausreißer in den Daten vorhanden.

1.2 Regressionsprobleme

Als Regressionsproblem hingegen bezeichnet man das Finden einer Funktion, die eine Menge von Eingabevariablen einer Stetigen Menge von Ausgabevariablen zuordnet. Wenn beispielsweise ein Bild eines Menschen gegeben ist, könnte ein Regressionsproblem sein, seine Höhe oder sein Gewicht zu bestimmen. Auch eine Wettervorhersage ist ein typisches Regressionsproblem. Ein Beispiel eines Regressionsproblemens wird in dieser Arbeit nicht behandelt werden. Wie Regression verbildlicht dargestellt werden kann, ist in Abbildung 2 gezeigt.

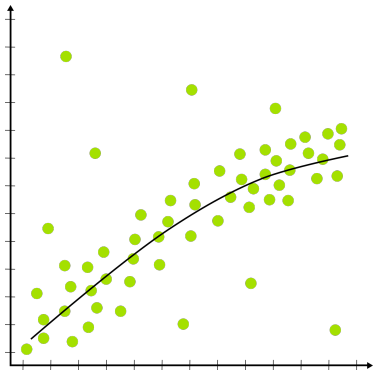

Abbildung 2: Regression

Die Kurve stellt hier Keine Grenze, sondern die Funktion, die die Werte approximiert, dar. Die Punkte repräsentieren die Eingabedaten, in denen auch hier einige Ausreißer erkennbar sind.

1.3 Gefahren von maschinellem Lernen

Maschinelles Lernen kann eine mächtige Technologie sein. Eine Vielzahl von Problemen lässt sich damit lösen, alle jedoch nicht. Man sollte sich bevor man maschinelles Lernen nutzt also Fragen: Lässt sich dieses Problem nicht einfacher auf konventionelle Weise lösen? Außerdem sollte man sich stets bewusst sein, dass maschinelles Lernen im Gegensatz zu den meißten Algorythmen, keine Technologie ist, die eine Treffsicherheit von 100% aufweist. In Systemen, wo eine korrekte Antwort kritisch ist, sollte man also nicht alleine auf maschinelles Lernen setzen.
Auch ist für maschinelles Lernen stets eine enorme Datenmenge nötig. Diese Daten müssen erst gesammelt werden. Hier stellt sich natürlich sofort eine ethische Frage: Welche Daten können guten Gewissens gesammelt und ausgewertret werden? Dabei sollte das Persönlichkeitsrecht und das Recht auf Privatsphäre eine zentrale Rolle spielen. Niemals sollte der Nutzen der Technologie über die Rechte der Nutzer gestellt werden. Betrachtet man hier beispielsweise den Flukhafen von Peking, sind erschreckende Tendenzen festzustellen. Dort wird beim Check-In via Gesichtserkennung die Identität der Person mit ihrem Gesicht verknüpft. Danach läuft alles vom Ticketkauf bis hin zum Duty-free-shop mit Hilfe von Gesichtserkennung ab [**4**].
Die zentralen Gefahren maschinellen Lernens sind also die eventuelle Unsicherheit im Ergebnis, der hohe Trainingsaufwand, der gegebenenfalls mit klassischen Algorythmen vermieden werden kann und die Verletzung von Rechten durch das Auswerten persönlicher Daten.

1.3.1 Die Daten

Wie bereits erwähnt sind die Datensätze oft der limitierende Faktor beim maschinellen Lernen. Das gravierendste Problem ist, überhaupt einen passenden Datensatz für das Problem zu finden oder generieren zu können. Dabei muss man beachten, dass man in den alle für das Problem relevanten Faktoren berücksichtigt. Möchte man beispielsweise Gesichter jeglicher Art erkennen, genügt es nicht den Algorythmus auf einem Datensatz von Gesichtern hellhäutiger Menschen zu trainieren, da dieser zum Erkennen von Gesichtern dunkelhäuitiger Menschen dann nutzlos wäre. Dass dies kein theoretisches, sondern auch ein praktisch auftretendes Phänomen ist, zeigt eine Studie des National Institute for Standards and Technology (NIST)[**5**]. Diese hat ergeben, dass beispielsweise ein in den USA entwickelter und dort sehr populärer Algorythmus eine extrem hohe Fehlerquote für afroamerikanische Frauen hat. Da dieses System unter anderem von der Polizei in den USA verwendet wird, haben afroamerikanische Frauen eine wesentlich höhere Chance fälschlicherweise einer Straftat beschuldigt zu werden.
Man sollte außerdem beachten, dass der Datensatz gut ausgeglichen ist. Das bedeutet, dass alle Trainingsdaten gleichmäßig verteilt sind. Möchte man beispielsweise Eingabedaten in 4 verschiedene Klassen klassifizieren, so sollten etwa 25% der Daten

zu Klasse A gehören, 25% zu Klasse B und so weiter. Der in dieser Arbeit später verwendete MNIST Datensatz hat einen Umfang von 60000 handgeschriebenben Ziffern von 0 bis 9. Den größten Anteil haben einsen mit rund 11%, den niedrigsten haben Fünfen mir 9%. Damit kann der Datensatz als ausgeglichen betrachtet werden. Eine genaue Definition, ab wann ein Datensatz nicht mehr ausgeglichen ist, existiert nicht.
Datensätze müssen ausgeglichen sein, da das Netz sonst den Fehler minimiert, indem es öfter die Klasse mit dem höhsten Anteil als Antwort liefert. Es erkennt, dass die höhste Trefferquote vorliegt, wenn es diese Antwort gibt, da so die Wahrscheinlichkeit eines Treffers maximal ist[1].

1.3.2 Overfitting

Overfitting ist ein häufig auftretendes Problem bei Klassifizierungsaufgaben. Die Klassengrenzen werden dabei zu genau aber falsch definiert. In Abbildung 3 ist dies dargestellt.

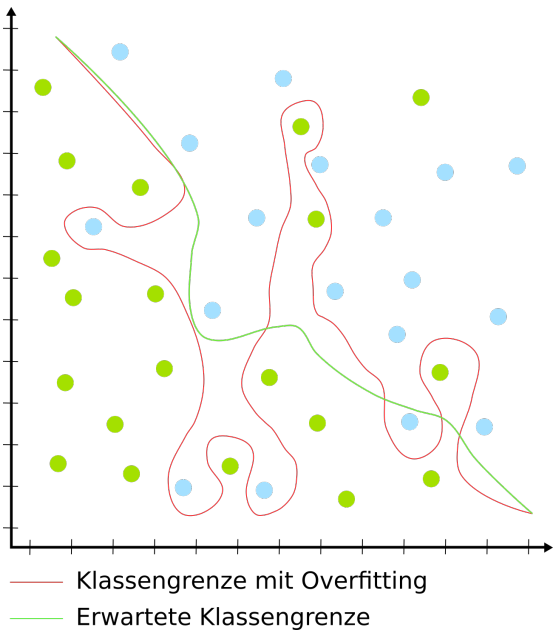

Abbildung 3: Overfitting

Overfitting tritt auf, wenn man ein neuronales Netz zu lange auf einem Datensatz trainiert. Das Netz lernt dann die Daten auswendig, da es so einen Fehler von 0 erreichen kann. Dadurch wurden aber keine wirklichen Klassengrenzen erlernt.
Um Overfitting entgegenzuwirken reicht es oftmals den Trainingsdatensatz in der

[1] In diesem Fall wird dann in der Fehlerfunktion ein lokales Minnimum gefunden. Dazu mehr unter Abschnitt 3.6, Gradientenverfahren und Backpropagation

Reihenfolge zu randomisieren. Dadurch kann das Netz diese gar nicht auswendig lernen.

2 Verschiedene Techniken maschinellen Lernens

Es gibt viele verschiedene Ansätze und Algorythmen um maschinelles Lernen zu implementieren. Der wohl häufigste ist das Neuronale Netz, von dem diese Arbeit handelt. Aber auch sogenannte „Support Vector machines" sind eine bekannte Technik. Neuronale Netze können In vielen verschiedenen Szenarien angewandt werden um unterschiedliche Ergebnisse zu erzielen. Beim Adversarial Learning lässt man mehrere Netze gegen einander antreten, sodass sie sich gegenseitig trainieren. Beim Q-Learning beginnt man mit zufälligen Reaktionen auf eine Eingabe und „belohnt" das Netz, falls es wie gewünmscht reagiert hat. Ein Beispiel hierfür ist die hide and seek AI von OpenAI[2]. Im groben unterscheidet man jedoch in überwachtes (supervised), unüberwachtes (unsupervised) und bestärkendes (reinforcement) Lernen.

2.1 Überwachtes Lernen

Beim überwachten Lernen ist ein Trainingsdatensatz vorhanden und die Eingabe sowie die gewünschte Ausgabe ist bekannt. Dies trifft sowohl auf klassifizierungs, alsauch auf Regressionsprobleme zu. Um Überwachtes Lernen nutzen zu können muss man also vor allem über einen großen Datensatz verfügen. Wie genau überwachtes Lernen innerhalb des neuronalen Netzes von Statten geht ist im nächsten Abschnitt unter „Neuronale Netze" beschrieben.

2.2 Unüberwachtes Lernen

Unüberwachtes Lernen erkennt automatisiert Muster in Datenmengen. Dies ist vergleichbar mit einem Kind, das zum ersten mal in seinem Leben einen Hund sieht und den nächsten Hund, den es sieht als einen solchen wiedererkennt. Das Kind hat nicht, wie es beim überwachten Lernen der Fall gewesen wäre, gesagt bekommen, dass es sich um einen Hund handelt. vielmehr hat es die Merkmale eines Hundes erkannt und sich diese gemerkt. Bei unüberwachtem Lernen erkennt werden also automatisch Muster erkannt, die dem Algorythmus zuvor nicht mitgeteilt wurden.

2.3 Bestärkendes Lernen

Bestärkendes Lernen ist die dritte Klasse von Lerntypen, die oft unterschieden werden. Hier ist kein Eingabedatensatz vorhanden, sondern ein bekanntes Ziel definiert. Der Algorythmus soll einen möglichst effizienten Weg finden dieses Ziel zu erreichen. Oft ist dabei ein Handlungsträger (agent) in einer Umgebung gegeben. Dies kann beispielsweise ein Auto sein, das lernen soll selbst zu fahren. Auch die oben erwähnte Q-Learning Methode aus open-AIs hide and seek AI fällt in die Kategorie des bestärkenden Lernens. Der agent beginnt mit zufälligen Handlungen und erhält wenn er ein

[2] https://openai.com/blog/emergent-tool-use/

Zwischenziel erreicht Belohnungen, oder Strafen wenn er einen Fehler macht. Diese Belohnungen und Strafen werden in Form von Zahlen vergeben. Das selbstfahrende Auto würde beispielsweise eine Belohnung erhalten wenn es einen Streckenabschnitt fehlerfrei zurückgelegt hat und eine Strafe, wenn es gegen eine Wand fährt. Das Ziel ist es diese kummulativen Belohnungen zu minimieren und dadurch das vordefinierte Ziel zu erreichen. Hierfür häufig verwendete Algorythmen sind das Q-learning oder sogenannte „Monte-Carlo Maschinen"

3 Neuronale Netze

bei Neuronalen Netzen handelt es sich um eine programminterne Struktur, die für das maschinelle Lernen genutzt wird. Wie der Name bereits vermuten lässt, ist diese Methode ein Versuch das menschliche Lernen nachzuahmen.

3.1 Maschinelles Lernen und menschliches Lernen

Das menschliche Gehirn ist aus sogenannten „Neuronen" aufgebaut. Ein Neuron ist eine Nervenzelle, die elektrische oder chemische Impulse annimmt, und gegebenenfalls einen elektrischen oder chemischen Impuls weitergibt. Die Nervenzellen berühren sich nicht direkt sondern sind nur über die sogenannten Synnapsen verbunden, über die diese Signale übertragen werden, sodass sich ein hoch komplexes Netzwerk von milliarden von Neuronen ergibt.[3] Ein neuronales Netz ist ähnlich aufgebaut. Es

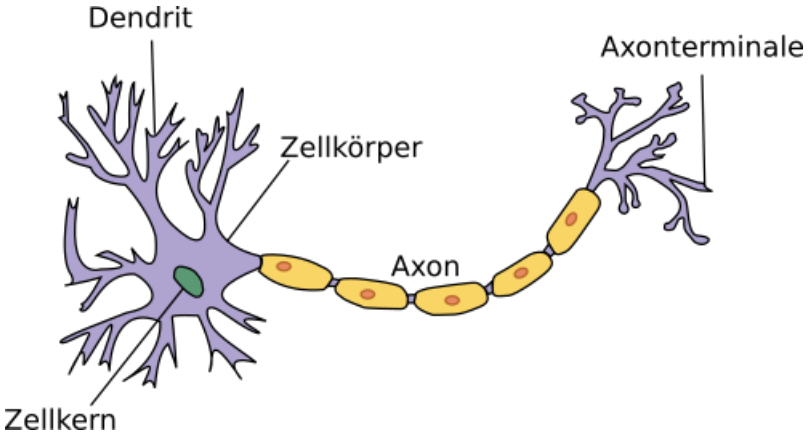

Abbildung 4: Ein Neuron wie es im Gehirn vorliegt

besteht aus „Neuronen", die eine theoretisch beliebige Anzahl von Eingaben annehmen können und mit einer entsprechenden Ausgabe reagieren, sowie Verbindungen zwischen den Neuronen. Auch das Lernprinzip entspricht dem eines Menschen. Das

[3]Diese Definition ist stark vereinfacht. Sie enthält ausschließlich die wesentlichen Komponenten um das menschliche Gehirn mit einem neuronalen Netz vergleichen zu können.

Netz nimmt immer Zahlen zwischen 0 und 1 als Eingabe an und berechnet eine entsprechende Ausgabe. Es erhält anschließend die Information, wie die richtige Lösung gelautet hätte und lernt dann aus seinen Fehlern, indem es gewisse Werte, die in die Berechnung einfließen, anpasst. Analog lernt ein Mensch, indem er ausprobiert, gegebenenfalls scheitert, anschließend die richtige Antwort durch eine externe Quelle erhält und somit aus seinem Fehler lernt. Im Menschlichen Gehirn verknüpfen sich Dabei oft genutzte neuronale Verbindungen stärker und weniger benutzte Verbindungen bauen sich ab[**2**]. Die Verstärkung und der Abbau entsprechen dem Ändern der Gewichtung einer Verbindung im neuronalen Netz. Die Gewichtung ist eine Eigenschaft der Verbindung, die eine zentrale Rolle in der Berechnung spielt und soll im folgenden weiter erläutert werden. Diese Ähnlichkeiten sind kein Zufall, sondern viel mehr Intention. Ein neuronales Netz ist nämlich der gezielte Versuch das menschliche Lernen nachzuahmen um maschinelles Lernen zu ermöglichen.

3.2 Der Aufbau eines neuronalen Netzes

Ein neuronales Netz besteht aus Neuronen und Verbindungen zwischen diesen. Es gibt einen sogenannten „Input Layer", der die Daten, den sogenannten „Input Vector", annimmt, eine beliebige Anzahl von sogenannten „Hidden Layers", in denen das eigentliche Lernen statt findet, und einen sogenannten „Output Layar", der für die Datenausgabe verantwortlich ist. Die Anzahl der Neuronen ist nach oben nicht begrenzt, wird jedoch zumeist der Aufgabe angepasst. Im Input Layer ist meißt ein Neuron pro Pixel des Eingabebildes vorhanden und im Output Layer ein Neuron pro möglicher Ausgabe. Sollen also 28×28 Pixel große Bilder handgeschriebener Ziffern klassifiziert werden, so gibt es 784 Eingabeneuronen, da jedes Bild 784 Pixel groß ist, und 10 Ausgabeneuronen, da es 10 Ziffern gibt. Jedes Neuron hat außerdem eine sogenannte Aktivierungsfunktion, die sich von Neuron zu Neuron unterscheiden kann, und jede Kante eine asoziierte Gewichtung und einen Bias. Ein neuronales Netz besteht also aus:

1. Neuronen mit gegebenenfalls verschiedenen Aktivierungsfunktionen, aufgeteilt in ein Input-, beliebig viele Hidden- und ein Output-Layer.
2. Verbindungen zwischen diesen Neuronen, die jeweils einen eigenen Bias und eine Gewichtung besitzen.

Sind alle Neuronen eines Layers jeweils mit allen Neuronen des nächsten Layers verbunden, wird das Layer als „fully connected layer" bezeichnet.

3.3 Berechnung des Ausgabevektors

Der Ausgabevektor wird berechnet, indem:

1. Alle Ausgaben aus der vorherigen Schicht mit der Gewichtung der korrespondierenden Kante multipliziert werden
2. Alle gewichteten Eingabewerte summiert werden

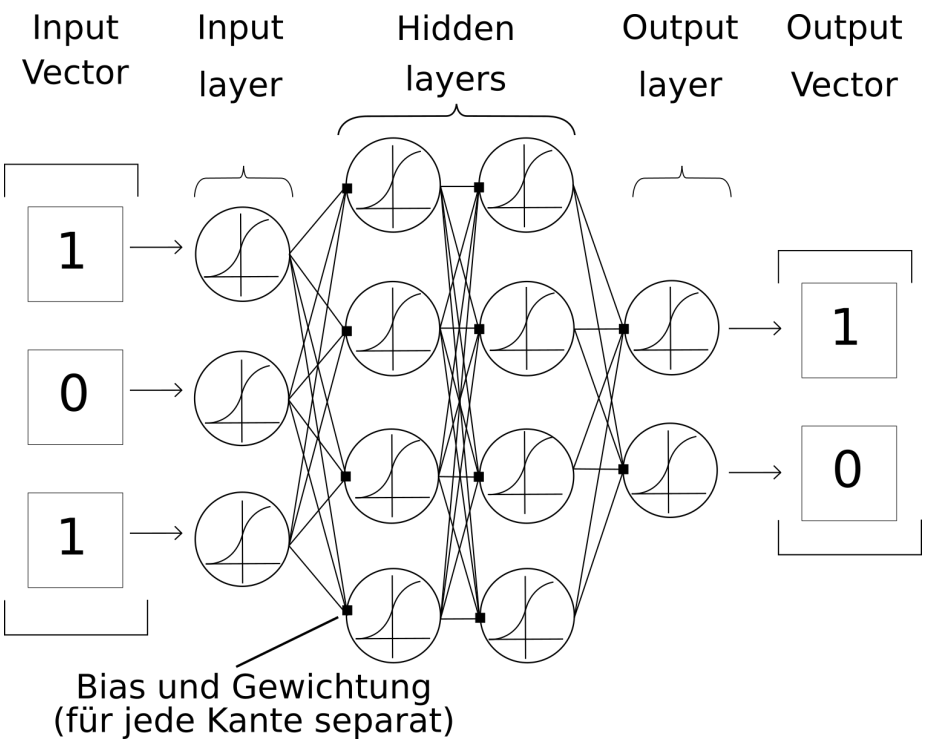

Abbildung 5: Ein einfaches neuronales Netz

3. Der Bias des Neurons hinzuaddiert wird
4. Die Aktivierungsfunktion auf diesen Wert angewandt wird

Die Aktivierungsfunktion hat dabei die Rolle die Werte zu normieren. Sie sorgt also dafür, dass alle Werte innerhalb des Netzes im Intervall $[0, 1]$ bleiben. Es gibt eine Vielzahl von Aktivierungsfunktionen. Die häufigste ist die in Abbildung 6 dargestellte „Sigmoid" Funktion.
Im Gegensatz dazu haben Gewichtungen typischerweise etwa den doppelten Wert der Eingaben. Alle Were werden jedoch automatisch im Lernprozess angepasst.
Der Begriff Eingabe- und Ausgabevektor lassen bereits vermuten, dass es sich bei Neuronalen Netzen um Objekte aus dem Bereich der linearen Algebra handelt. Daher wird im Folgenden auch die Notationsweise mit Hilfe von linearer Algebra verwendet. Betrachtet man eine Ausgabe eines Neurons wird diese als $a_{neuron}^{(layer)}$ bezeichnet. Den Ausgabevektor des Input Layers würde man also folgendermaßen schreiben:

$$\begin{bmatrix} a_0^0 \\ a_1^0 \\ a_2^0 \\ \vdots \\ a_n^0 \end{bmatrix}$$

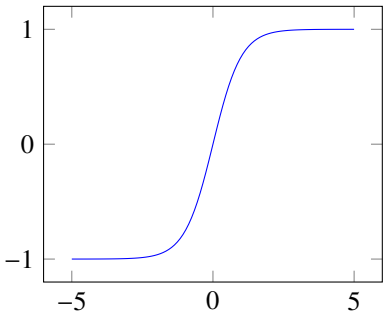

Abbildung 6: Der Plot der Sigmoid Funktion $\sigma(x) = \frac{e^x}{e^x+1}$

Die Gewichtungen w der jeweiligen Kanten werden notiert als $w^{(von\ Layer)}_{(zu\ Neuron, von\ Neuron)}$. „von Layer" bezeichnet dabei das Layer in dem das Neuron liegt, das die Information ausgibt. „zu Neuron" ist der Index des Neurons im nächsten Layer, das die Information annimmt und „von Neuron" der Index des Neurons, das die Information abgibt. Die Gewichtung der Kante, die das zweite Neuron im ersten Layer mit dem dritten Neuron im zweiten Layer verebindet würde also als $w^0_{3,2}$ bezeichnet werden. Dabei wird bei null begonnen zu zählen, sodass das erste Layer und das erste Neuron den Index 0 erhält.
Die Gewichtungen aller Verbindungen eines Layers zum nächsten können also als folgende Matrix geschrieben werden:

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix}$$

Dabei ist n hier die selbe Zahl wie n im Ausgabevektor, da genau so viele Ausgaben vorhanden sein müssen, wie Neuronen in diesem Layer vorhanden sind, da jedes Neuron einen Wert ausgibt.[4]Der Bias Vektor wird genau so wie der Ausgabevektor bezeichnet.

$$\begin{bmatrix} b^0_0 \\ b^0_1 \\ b^0_2 \\ \vdots \\ b^0_n \end{bmatrix}$$

Beachtet man jetzt noch, dass bei jedem Neuron die Aktivierungsfunktion angewandt werden muss ergibt sich folgende Gleichung für die Berechnung des Ausgabevektors $\vec{o}$ aus einem Einbagevektor $\vec{a}$ durch eine Schicht von Neuronen:

[4]Es existieren auch Neuronen, die Daten verwerfen. Diese kommen im hier betrachteten Typ von neuronalem Netz allerdings nicht vor und werden daher der Einfachheit halber außenvor gelassen.

$$\vec{o} = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^0 \\ a_1^0 \\ a_2^0 \\ \vdots \\ a_n^0 \end{bmatrix} + \begin{bmatrix} b_0^0 \\ b_1^0 \\ b_2^0 \\ \vdots \\ b_n^0 \end{bmatrix}\right)$$

Abbildung 7: Formel zur Berechnung eines Ausgabevektors aus einem Eingabevektor durch ein Layer Neuronen.

Zur Vereinfachung wurde die Funktion hier auf den gesamten Ausgabevektor angewandt. Dies ist korrekt, sofern alle Neuronen eines Layers die selbe Aktivierungsfunktion aufweisen. Dies muss natürlich nicht immer so sein. Sind die Aktivierungsfunktionen der Neuronen eines Layers verschieden, so wird die Aktivierungsfunktion des jeweiligen Neurones separat auf das korrespondierende Element des Vektors $W \cdot \vec{a} + \vec{b}$ angewandt.

3.4 Der Lernprozess

Der Lernprozess gliedert sich in wenige wesentliche Schritte. Zuerst wird unter Verwendung des oben beschriebenen Prozesses aus einem Eingabevektor ein Ausgabevektor berechnet. Diese Berechnung wird im Lernprozess extrem oft durchgeführt, weshalb sich neuronale Netze besonders schnell auf Grafikkarten trainieren lassen. Diese sind für mathematische Operationen im Bereich der linearen Algebra, wie Matritzenmultiplikation oder Addition optimiert und werden daher auch als Vektorprozessoren bezeichnet.
Dieser Ausgabevektor wird nun, mit Hilfe einer Fehlerfunktion, mit dem erwarteten Ausgabevektor verglichen. Je größer dabei die Differenz zwischen erwartetem Ausgabevektor und tatsächlichem Ausgabevektor ist, desto größer ist der Wert der Fehlerfunktion. Der Ausgabewert dieser Fehlerfunktion wird als „Fehler" oder auch als „Kosten" bezeichnet. Wenn also das Minimum dieser Fehlerfunktion bestimmt wird, wird der Fehler minimiert und die tatsächliche Ausgabe des Netzes nähert sich der korrekten Ausgabe immer weiter an.
Eine Methode, die hier erläutert werden soll, dieses Minimum zu finden ist das Gradientenverfahren. Nachdem mit Hilfe dieses Verfahrens der Fehler mimnimiert wurde, werden die Parameter, also die Gewichtungen und Biases, des neuronalen Netzes entsprechend angepasst. Diesen Prozess der Fehlerminimierung mittels des Gradientenverfahrens und der anschließenden Anpassung der Werte bezeichnet man auch als „Backpropagation". Es existieren auch noch andere Verfahren zur Fehlerminimierung, der Einfachheit halber soll hier aber nur Backpropagation erläutert werden.

3.5 Fehlerfunktionen

Es existiert eine Vielzahl von Fehlerfunktionen, die alle für unterschiedliche Anwendungsgebiete unterschiedlich passend sind. Im Groben lassen sich allerdings Fehler-

funktionen, die für Klassifizierungsprobleme geeignet sind von solchen unterscheiden, die für Regressionsprobleme geeignet sind.

3.5.1 MSE – Durchschnittlicher quadratischer Fehler

Der sogenannte durchschnittliche quadratische Fehler ist eine häufig genutzte Fehlerfunktion für Regressionsprobleme. Die englische Bezeichnung lautet „Mean squared error", woraus sich auch die Abkürzung „MSE loss" ergibt. Sie ist wie in Abbildung 8 dargestellt, definiert.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Abbildung 8: Die Gleichung für den durchschnittlichen quadratischen Fehler

Wie der Name vermuten lässt, gibt diese Fehlerfunktion den Durchschnitt der quadrierten Differenzen zwischen dem vorausgesagten und dem tatsächlichen Ergebnis an. Aufgrund der Quadrierung des Fehlers, werden durch diese Funktion stark abweichende Werte wesentlich stärker gewichtet, als weniger stark abweichende Werte. Ihr Gradient ist außerdem einfach berechenbar, was für das Gradientenverfahren später relevant ist.[**3**]

3.5.2 MAE – Durchschnitztlicher absoluter Fehler

Bei dem durchschnittlichen absoluten Fehler handelt es sich ebenfalls um eine Fehlerfunktion, die für Regressionsprobleme eingesetzt wird. Die englische Bezeichnung lautet „Mean absolute error". Sie ist ähnlich wie der durchschnittliche quadratische Fehler definiert.

$$MAE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n}$$

Abbildung 9: Die Gleichung für den durchschnittlichen absoluten Fehler

Auch hier wird die „Richtung" des Fehlers, in diesem Fall durch die Normierung, verworfen. Außerdem ist diese Fehlerfunktion nicht so anfällig gegenüber Ausreißern in den Daten, da dieser Fehler nicht quadriert wird. Ein Nachteil des durchschnittlichen absoluten Fehlers ist allerdings die höhere Komplexität zur Berechnung des Gradienten.[**3**]

3.5.3 Kreuzentropiefehler

Der Kreuzentropiefehler ist die am häufigsten verwendete Fehlerfunktion für Klassifizierungsprobleme. Sie gibt den Fehler für eine Klassifizierung an, die den gegebenen

Klassen Wahrscheinlichkeiten im Intervall $I = [0; 1]$ zuordnet. Dabei steigt der Fehler stärker, je weiter sich die Vorhersage vom tatsächlichen Wert entfernt. Wie aus Abbildung 10 hervorgeht, wird also sicheren, aber falschen Vorhersagen der höhste Fehlerwert zugeordnet.

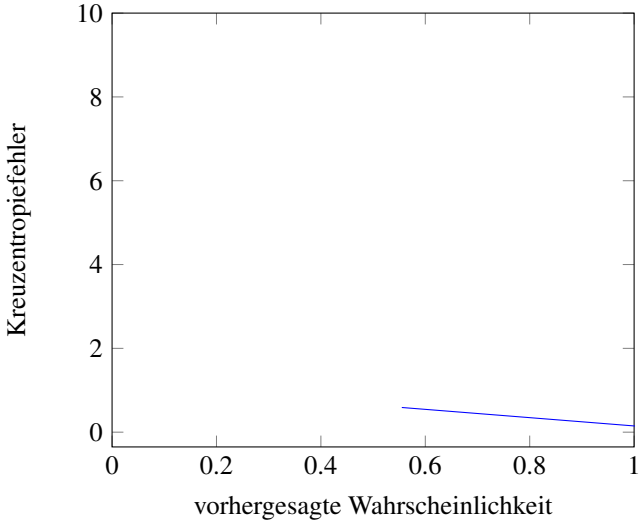

Abbildung 10: Der Graph der Kreuzentropie Fehlerfunktion wenn das tatsächliche Label 1 ist

Der Fehler steigt also mit zunehmender Abweichung der Vorhersage zum tatsächlichen Label rapide an.
Mathematisch ist der Kreuzentropiefehler nach der Funktion in Abbildung 11 definiert, wobei y einen Binärindikator darstellt, der angibt ob das zu klassifizierende Objekt tatsächlich zur Klasse gehört (dann ist er 1) und p die vorausgesagte Wahrscheinlichkeit ob das Objekt zur Klasse gehört, beschreibt.

$$CrossEntropyLoss = -(yln(p) + (1 - y)ln(1 - p))$$

Abbildung 11: Die Gleichung für den Kreuzentropiefehler

Hier fällt auf, dass, falls das Label 0 ist, der linke Teil der Gleichung weg fällt und falls es 1 ist, der Rechte. Wenn berechnetes und tatsächliches Label identisch sind, ist der Fehler stets 0.
Existieren mehr als 2 Klassen, handelt es sich also nicht mehr um eine Binärklassifizierung, müssen die Fehler nach der Gleichung in Abbildung 12 summiert werden.
Dabei gibt M die Anzahl der Klassen an, c das Label für die Klasse und o die berechnete Klassifizierung für diese Klasse.

$$CrossEntropyLoss(M) = -\sum_{c=1}^{M} y_{o,c} ln(p_{o,c})$$

Abbildung 12: Die Gleichung für den durchschnittlichen absoluten Fehler

3.6 Gradientenverfahren und Backpropagation

Das Gradientenverfahren ist ein Verfahren um das Minimum einer Funktion zu finden. Die Funktion, deren Minimum gefunden werden soll ist in diesem Fall die Fehlerfunktion. Diese ist von allen Gewichtungen und Biases des Netzwerkes abhängig, da sie direkt vom Ausgabevektor des Netzes abhängig ist. Der Gradient dieser Funktion ist in Abbildung 13 dargestellt.

$$\nabla C(w_1, b_1, \ldots, w_n, b_n) = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial b_1} \\ \vdots \\ \frac{\partial C}{\partial w_n} \\ \frac{\partial C}{\partial b_n} \end{bmatrix}$$

Abbildung 13: Die Gleichung für den Gradienten der Fehlerfunktion

Um also das Ergebnis „richtiger" zu machen, müssen alle Gewichtungen und Biases negativ zu diesem Gradienten angepasst werden, da der Gradient ja den Hochpunkt angibt. Diese Anpassung erfolgt, indem das Netz vom Ausgabelayer an, deshalb heißt das Verfahren Backpropagation, durchgegangen wird, und die Gewichtungen und Biases angepasst werden.
Oft wird zur Verbildlichung des Gradientenverfahrens die Analogie eines Balles verwendet, der einen Hügel hinunter rollt. Er findet den Tiefpunkt indem er hinab rollt und dabei immer automatisch eine Kraft nach unten wirkt.

3.6.1 Lernrate

Eine wichtige Rolle dabei spielt die sogenannte „Lernrate" η, mit der die Änderung nach der Formel in Abbildung 14 berechnet wird.

$$w^n_{neu} = w^n_{alt} - \eta \times \frac{\partial C}{\partial w^n}$$

Abbildung 14: Die Gleichung für die Anpassung eines einzelnen Parameters

Diese Lernrate ist notwendig um nicht über das Minimum „hinweg zu springen". Sollte sie zu groß sein, passiert genau dies, da die Anpassungen der Parameter in zu großen Schritten erfolgt. Sollte sie hingegen zu klein sein, lernt das Netz sehr langsam. Typische Werte sind abhängig von der zu erlernenden Aufgabe, liegen jedoch in

der Regel bei etwa 0.01 bis 0.0001 [5].

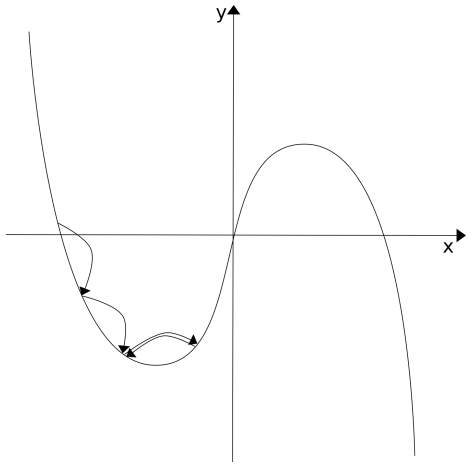

Abbildung 15: η ist hier zu groß gewählt

Abbildung 15 stellt dar, wieso das Minimum nicht erreicht werden kann, falls die Lernrate zu groß gewählt wurde. Es ist zu sehen, dass der Parameter immer gleich viel geändert wird und dabei das Minimum übersprungen wird, da die Lernrate konstant zu groß ist. Dieses Problem kann behoben werden indem eine adaptive Lernrate verwendet wird. Dabei verringert sich die Lernrate im Laufe des Lernprozesses, sodass zu Beginn die Vorzüge des schnellen Lernens genutzt werden können und am Ende trotzdem ein hoher Grad an Präzision erreicht werden kann.

3.7 Verschiedene Layerarten

Mit Hilfe von maschinellem Lernen lassen sich eine Vielzahl von Aufgaben bewältigen. Entsprechend komplex müssen Neuronale Netze aber auch sein. Demzufolge ist es notwendig, Neuronen zu entwickeln, die andere Fähigkeiten aufweisenl, als das einfache oben im sogenannten „Linear Layer" verwendete Neuron. Da man in der Regel nur eine Art von Neuron in einem Layer verwendet, wird das gesamte Layer nach der verwendeten Neuronenart benannt. Die unten beschriebenen Layerarten werden vor allem in einer Klasse von neuronalen Netzen verwendet, die als „Convolutional neural networks" bezeichnet werden. Sie werden meißt im Bereich der komplexen fragmentbasierten Bilderkennung eingesetzt, da sie besonders gut geeignet sind um Kanten oder gewisse Teile eines Bildes, wie zum Beispiel Merkmale eines Gesichtes, zu erkennen.

[5] Dies ist ein bloßer Erfahrungswert. Maschinelles Lernen erfordert oft sehr viele Versuche, weshalb nicht genau festgelegt werden kann, wann welche Lernrate optimal ist.

3.7.1 Convolutional Layers

Convolutional Layers weisen eine fundamental andere Funktionsweise als lineare Layers auf. Sie nehmen zwar ebenfalls rationale Zahlen an und geben rationale Zahlen aus [6], berechnen die Ausgabe jedoch nicht nur mit Hilfe einer Aktivierungsfunktion sondern unter der Verwendung sogenannter „Filter". Diese Filter sind eine $m \times n$ große Matrix, die auch als „Kernel" bezeichnet wird. Der Kernel wird dabei über die Eingabematrix bewegt (daher der Name convolution) und erzeugt eine Ausgabematrix. Dafür wird der betrachtete Abschnitt der Eingabematrix A und des Kernels B skalar multipliziert wobei das Skalarprodukt als Frobenius-Skalarprodukt, also als

$$\langle A, B \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} b_{ij}$$

definiert ist. Die Matritzen werden also Komponentenweise multipliziert und diese Produkte dann summiert.
Dies ist in Abbildung 16 verbildlicht.

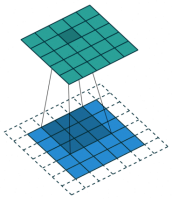

Abbildung 16: Eine Verbildlichung der Vorgänge in einem convolutional Layer. Das blaue Raster stellt die Eingabe dar, das grüne die Ausgabe.

[6] Im Folgenden werden 2 Dimensionale convolutional Layers betrachtet, da diese einfacher vorstellbar sind. Sie nehmen dann eine Matrix rationaler Zahlen an und geben auch eine Matrix rationaler Zahlen aus. Dies korrespondiert mit dem Anwendungsbereich der Erkennung von schwarz weiß Bildern.

Ein Filter kann ganz verschiedene Werte aufweisen. So können Filter der Form

$$\begin{bmatrix} -1 & -1 & -1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Abbildung 17: Erkennt obere horizontale Kanten

$$\begin{bmatrix} -1 & 1 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix}$$

Abbildung 18: Erkennt linke vertikale Kanten

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ -1 & -1 & -1 \end{bmatrix}$$

Abbildung 19: Erkennt untere horizontale Kanten

$$\begin{bmatrix} 0 & 1 & -1 \\ 0 & 1 & -1 \\ 0 & 1 & -1 \end{bmatrix}$$

Abbildung 20: Erkennt rechte vertikale Kanten

beispielsweise zur einfachen Kantenerkennung genutzt werden. Zur Veranschaulichung wurden diese Filter auf das Beispielbild in Abbildung 21 angewandt. Das Ergebnis ist in Abbildung 22 dargestellt.

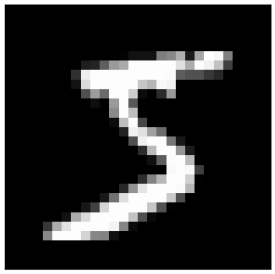

Abbildung 21: Das Beispielbild aus dem Mnist Datensatz

Abbildung 22: Die jeweils oben stehenden Filter wurden auf das Beispielbild angewandt.

Der jeweils dunkel dargestellte Bereich kann als das identifiziert werden, was vom convolutional Layer als Kante erkannt wurde. Hier werden eindeutige Limitationen deutlich: Es kann nur als Kante erkannt werden, was auch eindeutig senkrecht oder waagerecht ist. Außerdem kann es zu Fehlentscheidungen kommen.

Die Kernels werden natürlich nicht per Hand initialisiert und angepasst, sondern setzen sich aus Parameternm zusammen, die im Laufe des Lernprozesses durch das Netz anpassbar sind. Das Netz kann also die Filtermatrix selber verändern. Die Filter werden meißt mit Zufallswerten initialisiert und dann während des Lernens angepasst. Ferner muss ein Kernel auch nicht immer drei Einheiten breit sein, sondern kann jede Größe ≥ 2 annehmen. Je nachdem, wie sich der Kernel über die Eingabematrix bewegt, ist außerdem ein sogenanntes „Padding" nötig, da gegebenenfalls Werte betrachtet werden müssten, die nicht in der Eingabematrix liegen. In der Regel werden daher alle Werte, die nicht in der Eingabematrix vorhanden sind durch 0 ersetzt. Das Padding ist in Abbildung 16 als weiß in der Eingabematrix dargestellt. Es ist eine Art „Rand aus Nullen", der um dasd Bild gelegt wird.

Hintereinander können convolutional Layers auch ganze Elemente eines Bildes erkennen. Erkennt das erste Layer wie oben gezeigt beispielsweise Kanten, so kann das Layer darauf Kombinationen aus diesen, wie beispielsweise Ecken oder Winkel, erkennen. Wie gefilterte Bilder für sogenannte „High-Level-Features" aussehen können ist in Abbilldung 23 dargestellt. Die Ausgabebilder von Convolutional Layers werden als „Feature map" bezeichnet.

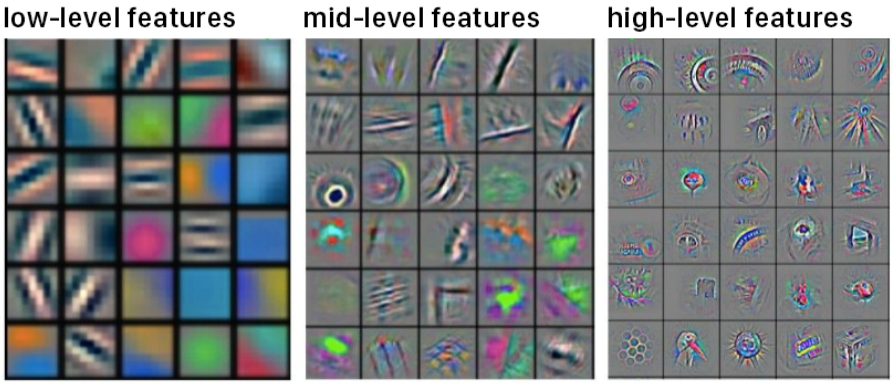

Abbildung 23: Beispiele für low- mid- und high-level Features in Convolutional Neural Nets

Das bemerkenswerte an Convolutional Layers ist vor allem, dass duch ähnliche Optimierungsalgorytmen auch hier maschinelles lernen möglich ist, dass sich ein neuronales Netz diese Filter also selbstständig beibringen kann.

3.7.2 Pooling Layers

Pooling Layers werden ebenfalls hauptsächlich in Convolutional Neural Networks verwendet. Sie werden nach Convolutional Layers genutzt um das Ausgabebild herunterzutakten, also verlustbehaftet zu Komprimieren. Dabei wird die Feature Map im wesentlichen zusammengefasst um die Datenmenge, die das Folgende Convolutional

Layer erhält zu reduzieren und sie Verschiebungen im Originalbild gegenüber immuner zu machen. Das ist deshalb notwendig, da die Convolutional Layers die Features lokal sehr begrenzt erkennen und daher eine kleine Verschiebung des Originalbildes zur Folge haben kann, dass im Folgenden Convolutional Layer die Kombination der Features gegebenenfalls nicht richtig erkannt wird. Das Pooling Layer kann diesem Effekt entgegenwirken, indem es die Features zusammenfasst. So kann aus einer ganzen Kante beispielsweise ein einziger Pixel werden.
Es werden im Wesentlichen zwei Techniken zum Pooling eingesetzt.

1. Max Pooling (Abbildung 24)
2. Average Pooling (Abbildung 25)

Sie unterscheiden sich darin, wie die zu komprimierenden Werte mit einander verrechnet werden, sind ansonsten jedoch identisch.
Beim Pooling wird die Eingabematrix in Submatritzen partitioniert[7]. Jede Submatrix stellt später einen Pixel in der Ausgabematrix dar. Hier unterscheiden sich jetzt Max- und Avarage- Pooling. Beim Max Pooling ist der neue Wert der höhste Wert aus dieser Submatrix, beim Avarage Pooling wird der Durchschnitt aller Werte Der Submatrix gebildet und als Neuer Wert verwendet.

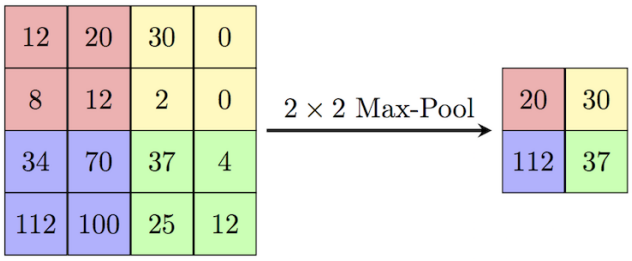

Abbildung 24: Max Pooling mit 2×2 großen Submatritzen

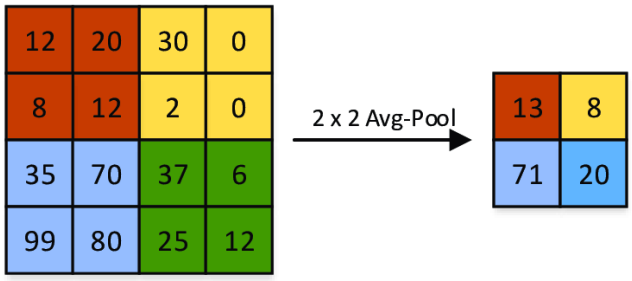

Abbildung 25: Average Pooling mit 2×2 großen Submatritzen

[7]Hier ist die Mengenteorethische Partitionierung gemeint. Eine Menge wird in nicht leere Teilmengen unterteilt, sodass jedes Element der Ausgangsmenge in genau einer der Teilmengen enthalten ist.

Die Dimension der Submatritzen beträgt meißt 2×2. In Abbildung 26 ist dargestellt, wie Pooling konkret auf das im letzten Abschnitt beschriebene Bild abgewandt aussieht. Dafür sind in der ersten Zeile die 28×28 Pixel großen Bilder dargestellt, die das Convolutional Layer mit Hilfe der Kantenerkennungsfilter berechnet hat. In Zeile zwei wurde auf die jeweils darüber stehenden Bilder Max Pooling angewandt, in Zeile drei auf die selben Bilder Average Pooling. Die Bilder sind nach dem Pooling 14×14 Pixel groß.

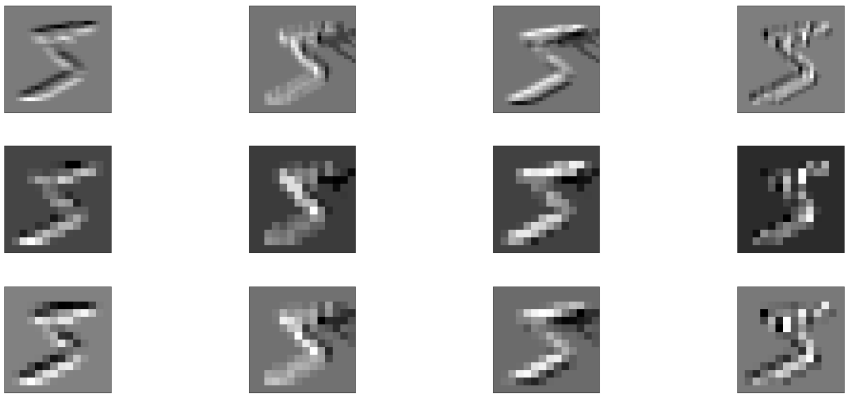

Abbildung 26: Gegenüberstellung von Max und Average Pooling

4 PyTorch

Pytorch ist ein von der Facebook Research Group entwickeltes Framework für machinelles Lernen in Python. Es ermöglicht Programmierern, maschinelles Lernen einfach und hochoptimiret umzusetzen. Dafür stellt es unter anderem eine Schnittstelle für Grafikkarten bereit und lieferet viele Funktionen, die oft benötigt werden. So muss beispielsweise die Gradientenberechnung oder die Berechnung der Fehlerfunktion nicht in jedem Projekt erneut implementiert werden. Die Grundlage der Pytorch Library ist der Datentyp „Tensor". Dabei handelt es sich im wesentlichen um eine Matrix, die optimierte Funktionen für maschinelles Lernen aufweist und auf Grafikkarten transferiert werden kann. Alle Daten werden in Form dieser Tensoren gespeichert und verarbeitet. Sollen also Bilder erkannt werden, müssen diese erst zu Tensoren konvertiert werden. Neben den Fehlerfunktionen und der Gradientenberechnung ist besonders die Einfachheit mit der ein Netz in Pytorch definiert werden kann bezeichnend. Pytorch ermöglicht es also, dass die Entwicklung auf die Logik selber fokusiert sein kann und trotzdem komplexe mathematische Funktionen verwendet werden können. Häufig genannte Alternativen zu Pytorch sind die Frameworks „Tensorflow" oder „Keras". Tensorflow wird von Google entwickelt und ist auch für andere Sprachen als Python verfügbar.

4.1 Datenvorbereitung

Wie bereits erwähnt, müssen die Daten erst vorbereitet werden. Dies kann unter Umständen das größte Problem bei einem Projekt, das maschinelles Lernen involviert, darstellen, da die Datenvorbereitung sehr komplex werden kann. In einem einfachen Fall liegt der Datensatz bereits in Pytorchs interner Datenlibrary „Torchvision" vor und muss nur noch geladen werden. Im komplexesten Fall, kann es allerdings notwendig werden, mehrere sogenannte „Transforms" auf die Daten anzuwenden. Das sind kleine Funktionen, die die Daten verändern. Sie schneiden beispielsweise das Eingabebild zu, normalisieren es oder wenden eine vollständig selbst definierte Funktion darauf an.
In ein neuronales Netz können immer nur Bilder der gleichen Größe gegeben werden, da die Größe der Eingabetensoren konstant ist und die Form des Eingabelayers definiert. Ist also ein Datensatz gegeben, in dem zum Beispiel einige Bilder im hoch und einige im Querformat vorliegen, müssen diese erst durch geschicktes Zuschneiden und verschieben auf eine Größe gebracht werden. Auch dafür lassen sich Transforms verwenden. Liegen die Daten allerdings nicht als Bilder vor, müssen gegebenenfalls angepasste Algorythmen angewandt werden um diese Daten zu Tensoren zu konvertieren.
Aufgrund der Vielseitigkeit von PyTorch ist es ebenfalls möglich andere Librarys einzubinden um vor der Konvertierung der Bilder zu einem Tensor zum Beispiel einen Kantenerkennungsalgorythmus darauf anzuwenden. Im einfachsten Fall stellt sich das Laden der Daten wie in Abbildung 27 dar.

```
1  from torchvision import transforms, datasets
2
3  train = datasets.MNIST('./datasets', train=True, download=True,
4  transform=transforms.Compose([
5          transforms.ToTensor()
6  ]))
7
8  test = datasets.MNIST('./datasets', train=False, download=True,
9  transform=transforms.Compose([
10         transforms.ToTensor()
11 ]))
12
13 trainset = torch.utils.data.DataLoader(train, batch_size=200, shuffle=True)
14 testset = torch.utils.data.DataLoader(test, batch_size=10, shuffle=False)
```

Abbildung 27: Der Code zum Laden des MNIST Datensatzes

Der Code lädt zwei Datensätze. Einen zum testen und einen zum trainieren. Anschließend wird aus diesen je ein DataLoader erstellt, um über die Daten iterieren zu können. In Zeile 1 werden dafür zunächst alle nötigen Funktionen importiert. Die Zeilen 3 bis 6 sind für das eigentliche Laden der Daten zuständig.
Die Funktion `datasets.MNIST()` nimmt dabei vier Parameter an:

1. `'./datasets'` Dieser Parameter gibt den Speicherort für den heruntergeladenen Datensatz an

2. `train` Dieser Booleanparameter gibt an, ob es sich bei diesem Datensatz um den Trainingsdatensatz oder um den Testdatensatz handeln soll.

3. `download` Mit diesem Parameter wird festgelegt ob der Datensatz heruntergeladen werden soll, oder jedes mal erneut aus dem Internet abgerufen werden soll.

4. `transform` Hier werden die Transforms angegeben, die auf die geladenen Daten angewandt werden sollen. In diesem Fall wurde der Ansatz `transforms.Compose([transforms.ToTensor()])` gewählt.
   `transforms.Compose()` ist dabei dafür verantwortlich die Transforms im Array zu kaskadieren, also nach einander auf die Eingabedaten anzuwenden.
   `transforms.ToTensor()` ist der häufigste Transform. Er wird eingesetzt um Bilddaten in einen Tensor umzuwandeln.

In Zeile 13 wird dann der DataLoader erstellt. Er nimmt folgende Parameter an:

1. Der erste Parameter ist der Datensatz aus dem der DataLoader erstellt werden soll

2. `batch_size` Ist ein Parameter, der eine ganz fundamentale Variable beim maschinellen Lernen durch neuronale Netze festlegt: die Batch size.
   Die Tensoren werden nämlich nicht einzeln, sondern in sogenannten minibatches in das Netz gegeben. Es wird mit Durchschnittswerten über diese Tensoren in einer Minibatch gerechnet. Dies dient der Reduktion der Rechenzeit. Batching kann veranschaulicht werden, indem man sich vorstellt, dass die „Eingabebilder" hinter einander geklebt und alle gleichzeitig betrachtet werden. Die Batch size gibt dann analog an, wie viele Bilder hinter einander geklebt werden. Je höher die Batch size, desto höher ist die Speicherauslastung auf der Grafikkarte und desto ungenauer ist das Ergebnis, da über mehr Werte der Durchschnitt gerechnet wird. Mit höherer Batch size sinkt allerdings auch die Rechenzeit massiv.

3. `shuffle` gibt lediglich an, ob die Reihenfolge der Daten randomisiert werden soll. Dies ist wie am Anfang bereits erwähnt ein sehr nützlicher Parameter um overfitting vorzubeugen.

Über den entstandenen DataLoader kann jetzt in einer konventionellen Schleife iteriert werden, da die Klasse DataLoader die MagicMethod `__iter__` implementiert. Der DataLoader gibt dabei Tupel der Form (Batch von Bildern als Tensoren, Batch von Labels als Klassenindices) zurück.

4.2 Definieren des Netzes

Das Definieren des Netzes ist in Pytorch bereits sehr einfach möglich, bietet jedoch dennoch extreme individuelle Anpassungsmöglichkeiten. Um ein Netz zu definieren

muss zunächst eine neue Klasse erstellt werden, die Subklkasse von `nn.Module` ist. Im Konstruktor wird dann angegeben, welche Layers das Netz haben soll. Es muss außerdem die Methode `forward(self, x)` implementiert werden. Diese spezifiziert, wie mit den Daten innerhalbn des Netzes verfahren wird. Eine möglichst einfache Definition eines Netzes ist in Abbildung 28 gegeben.

```
1  class Net(nn.Module):
2          def __init__(self):
3                  super().__init__()
4                  self.fc0 = nn.Linear(28 * 28, 64)
5                  self.fc1 = nn.Linear(64, 120)
6                  self.fc2 = nn.Linear(120, 10)
7
8          def forward(self, x):
9                  x = F.relu(self.fc0(x))
10                 x = F.relu(self.fc1(x))
11                 x = self.fc2(x)
12                 return F.log_softmax(x)
```

Abbildung 28: Code um ein einfaches Netz in Pytorch zu definieren

Dieses Netz hat nur drei Layers: Ein Eingabelayer (`fc0`), das genau die Größe der Eingabedaten (28×28) aufweist, ein hidden Layer (`fc1`), das 64 Skalere annimmt und 120 Skalare ausgibt und ein Ausgabelayer (`fc2`), das 120 Skalare annimmt und 10 ausgibt. Dass `fc2` 10 Ausgabeneuronen besitzt ist kein Zufall, sondern liegt darin begründet, dass dieses Klassifizierungsnetz genau 10 Klassen unterscheiden soll. Die Größe von `fc1` ist jedoch völlig frei gewählt. Es ist allerdings wichtig Acht zu geben, dass die Layers die Daten auch an einander weitergeben können. Ein Layer muss also stets so viele Ausgaben aufweisen, wie das Layer, an das die Daten weitergegeben werden sollen, Eingaben besitzt.
Die `forward(self, x)` Funktion definiert, wie die Daten innerhalb des Netzes weitergegeben werden sollen. Hier werden sie erst in `fc0` gegeben, dann wird auf die Ausgabe aus `fc0` die Aktivierungsfunktion „ReLu" (REctified Linear Unit) angewandt. Die Ausgabe daraus wird dann in das hidden Layer `fc1` gegebnen und die Aktivierungsfunktion wird erneut angewandt. Im Output Layer geschieht dies nicht. Abschließend wird die Ausgabe von `F.log_softmax` zurück gegeben. Dies wendet erst einen SoftMax und dann einen Logarythmus auf die Daten an [**6**] um diese zu normalisieren und ist in Klassifizierungsnetzwerken oft nötig um sogenannte „One hot Vectors" zu erstellen. Herkömmliche Ausgabevektoren haben einen Wert für jede Klasse und geben an wie wahrscheinlich das Netz die Eingabedaten dieser Klasse zuordnet. In One hot Vektoren ist immer der höhste Wert 1 und alle anderen Werte sind 0.
Da die Netze als Klassen definiert werden und der Interne Datenverkehr in der `forward(self, x)` Funktion abläuft, sind neuronale Netze in Pytorch also sehr anpassbar. So wäre es beispielsweise kein Problem zwischen Input und hidden Layer die Daten mit 2 zu multiplizieren (dafür würde man zwischen Zeile 9 und 10 den Co-

de „`x = x * 2`" einfügen), auch wenn dies in den meißten Anwendungsbereichen keinen Sinn hätte. Pytorch kombiniert mit dieser Art Netze zu definieren also eine umfangreiche Flexibilität mit einfacher Bedienbartkeit.

4.3 Trainieren des Netzes

Das Trainieren des Netzes erfolgt in der sogenannten „Training Loop". Also in einer Schleife, die über den Datensatz iteriert. Zumeißt steht diese noch in einer Schleife, die über die Epochenzahl iteriert. In der Training loop wird ein Element des Datensatzes gelesen, in das Netz gegeben, die Ausgabe wird mit dem Label verglichen und schließlich werden die Parameter des Netzes Angepasst. Der Code dafür ist in Abbildung 29 dargestellt.

```
1  net = Net()
2
3  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
4  print('runnning on %s' % device)
5
6  net = net.to(device)
7
8  loss_function = nn.CrossEntropyLoss()
9  optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)
10
11 for epoch in range(EPOCHS):
12         for data in trainset:
13                 X, y = data
14                 net.zero_grad()
15                 X = X.to(device)
16                 output = net(X.view(-1, n * n))
17                 output = output.cpu()
18                 loss = loss_function(output, y)
19                 loss.backward()
20                 optimizer.step()
21
22         net = net.cpu()
23         torch.save(net, './nets/net_' + str(epoch) + ".pt")
24         net = net.to(device)
```

Abbildung 29: Code um das Netz auf einem Datensatz zu trainieren

In Zeile 1 wird dafür zunächst das Netz instanziiert. In Zeile 3 und 4 Wird bestimmt ob eine Grafikkarte zum trainieren verfügbar ist und diese wird gegebenenfalls genutzt. Auch wird ausgegeben, auf welchem Gerät das Netz trainiert wird. Der Code in Zeile 6 verschiebt das Netz auf die Grafikkarte, um dort mit diesem rechnen zu können. In Zeile 8 wird die Fehlerfunktion definiert, hier der Kreuzentropiefehler. In der nächsten Zeile wird der sogenannte „Optimizer" instanziiert. Dieser berechnet

später die Gradienten und passt die Parameter des Netzes an. Der Optimizer, der hier gewählt wurde nennt sich „Adam". Es handelt sich um einen gradienten basierten Optimierungsalgorythmus, der von Diederik P. Kingma und Jimmy Ba entwickelt wurde [7]. Eine Liste aller Implementierten Optimierungsalgorythmen ist in der PyTorch Dokumentation unter https://pytorch.org/docs/stable/optim.html gegeben. In Zeile 11 folgt nun die Schleife, die Über die Anzahl der Epochen iteriert, in der nächste Zeile die Train loop. Sie iteriert pro Epoche ein mal über den Datensatz. In Zeile 13 werden die Daten entpackt, da diese als Tupel vorliegen. X stellt dabei eine batch von Eingabevektoren dar, y eine Batch von Features. Zeile 14 setzt die Gradienten auf 0 zurück. Dies ist notwendig, da diese sonsnt aufsummiert würden. Die Parameter würden also nicht nur den Gradienten aus diesem Schleifendurchgang entsprechend angepasst werden, sonndern entsprechend zu einer Suppe aus allen Gradienten. Das würde das Ergebnis verfälschen. Der Befehl in Zeile 15 verschiebt auch den Eingabevektor auf die Grafikkarte, da alle Daten, mit denen das Netz rechnen soll erst auf dieser sein müssen. In Zeile 16 wird dann die Ausgabe des Netzes aus den Daten berechnet. Hier wird eine weitere Einfachheit von PyTorch deutlich: Um eine Ausgabe eines Netzes zu berechnen reicht der Befehl `output = net(input)` völlig aus. `X.view(-1, n * n)` wandelt den 2-dimensionalen[8] Tensor des Eingabebildes in einen Eingabevektor um. Auch hier muss das batching wieder beachtet werden, daher die -1. $n \times n$ stellt nämlich die Größe der Eingabebilder dar und -1 passt den ersten Wert automatisch an den zweiten an, entspricht hier also immer der Batch size. Nachdem der Ausgabevektor berechnet wurde wird er zunächst auf die CPU verschoben, da dort der Fehler berechnet wird und anschließend mit Hilfe der Fehlerfunktion mit dem Label verglichen. Dies geschieht in Zeile 17 und 18. `loss.backward()` in Zeile 19 sorgt dann dafür, dass der Optimizer alle Alle Anpassungen berechnet um sie in Zeile 20 anzuwenden. Der weitere Code dient dazu jede Epoche das Netz in einer Datei abzuspeichern. Dafür muss es erst auf die CPU verschoben werden und danach wieder zurück auf die Grafikkarte.

4.4 Pytorch und weights and biases

In den im folgenden erläuterten Projekten wurde ein Framework namens weights and biases verwendet. Es erlaubt während des Trainingsprozesses auf einer Seite Parameter des Netzes und wichtige Characteristika des Lernprozesses sowie Eigenschaften der Hardware zu tracken. Da es mit Pytorch gut kompatibel ist sind hierfür nur wenige Zeilen Code notwendig. Zunächst wird mit
`wandb.init(project='Beispielprojekt')` das Projekt initialisiert. Danach muss angegeben werden, welches Netz betrachtet werden soll. Dafür verwendet man nachdem das Netz initialisiert wurde den Befehl `wandb.watch(model)`, wobei `model` der Instanz des Netzes entspricht. Danach kann einfach über
`wandb.log({'loss': loss})` Zum Beispiel in der Training loop jedes Mahl der Fehler mitgeschrieben werden.

[8] Streng genommen ist der Tensor der Bilder aus dem Datensatz nicht 2, sondern 4-dimensional. Er weist die Dimensionen *Batch_size* $\times$ *Kanalanzahl* $\times n \times n$ auf. Die Kanalanzahl entspricht bei Bildern den Farbkanälen, liegt bei schwarz-weiß Bildern also bei 1 und sonst bei 3 bis 4 für RGBA

5 Ein Klassifizierungsnetzwerk für handgeschriebene Ziffern

Die Klassifizierung handgeschriebener Ziffern aus dem MNIST Datensatz stellt etwa die „Hello world" Aufgabe des maschinellen Lernens dar. Sie ist gut zum Erlernen der verschiedenen Algorythmen geeignet, extrem gut Dokumentiert und daher leicht nachvollziehbar und ein Paradebeispiel eines Klassifizierungsproblemes. Wenn man sich mit maschinellem Lernen beschäftigt ist es also sehr wahrscheinlich, dass man als aller erstes ein Klassifizierungssystem für den MNIST Datensatz programmieren wird.

5.1 Aufgabe

Die Aufgabe besteht darin den die handgeschriebenen Ziffern des MNIST Datensatzes klassifizieren zu können. Das Ziel dabei ist es mit möglichst wenig Trainingsaufwand eine Genauigkeit von mindestens 97% zu erreichen. Um dies zu bewältigen soll ein neuronales Netz in PyTorch programmiert und trainiert werden.

5.2 Der MNIST Datensatz

Der MNIST Datensatz ist ein Datensatz aus 28×28 Pixel großen Graustufenbildern von handgeschriebenen Ziffern. Er weist 60000 Trainingsbilder und 10000 Testbilder auf und ist ein Teil des EMNIST Datensatzes vom National Institute for Standards and Technology, U.S. Department of Commerce. Der Datensatz ist frei unter http://yann.lecun.com/exdb/mnist/ verfügbar. Die Bilder sind bereits zentriert und normalisiert, was diesen Datensatz besonders geeignet für Einsteiger macht. Die meißten Bilder des Datensatzes sind auch von Menschen einfach zu erkennen, einige sind jedoch sehr schwierig einzuordnen und teilweise kaum als Zahl erkennbar. Aufgrund der Einfachheit der Daten sind durchaus hohe Erfolgsquoten zu erwarten. Diese liegen im Schnitt bei 98%.

5.3 Das Netz

Das Netz wurde für diese vergleichsweise unkomplizierte Aufgabe einfach gehalten. Es weist drei hidden Layers auf, besteht insgesamt also aus fünf Layers, die alle klassische Lineare Layers sind. Die Aktivierungsfunktion ist überall eine ReLu[9], deren Plot in Abbildung 31 dargestellt ist. Da es sich um eine Klassifizierungsaufgabe handelt hat das Ausgabelayer 10 Ausgabeneuronen, die die 10 Ziffern repräsentieren. Es ist wie in Abbildung 30 dargestellt, definiert.
Das erste Layer nimmt also einen Tensor der Größe 1×784 an, da die Bilder genau so groß sind, und gibt einen 1×64 großen Tensor aus. Dieser wird vom ersten hidden Layer angenommen, dass einen 1×120 großen Tensor ausgibt. Im zweiten hidden Layer bleibt die Größe mit 1×120 konstant und im dritten wird sie wieder auf 1×64

[9] ReLu steht für rectified linear unit. Diese Aktivierungsfunktion ist neben den Sigmoid Funktionen ebenfalls eine sehr populäre Aktivierungsfunktion. Sie ist als $f(x) = \max(0, x)$ definiert und ist somit für alle $x \leq 0$ 0 und für alle $x > 0$ x.

```python
class Net(nn.Module):
def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28 * 28, 64)
        self.fc2 = nn.Linear(64, 120)
        self.fc3 = nn.Linear(120, 120)
        self.fc4 = nn.Linear(120, 64)
        self.fc5 = nn.Linear(64, 10)

def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = self.fc5(x)
        return F.log_softmax(x, dim=1)
```

Abbildung 30: Der Code um das in diesem Projekt genutzte Klassifizierungsnetz zu definieren.

reduiziert. Schlussendlich wird der Ausgabevektor mit den zehn Klassenwahrscheinlichkeiten berechnet.
in der `forward()` Funktion wird jedem Layer außer dem output Layer noch eine ReLu Aktivierungsfunktion zugewiesen und der SoftMax sowie der Logarythmus werden angewandt.

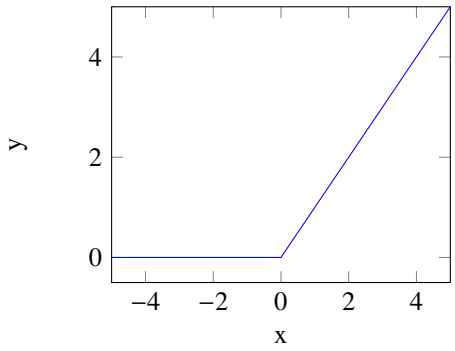

Abbildung 31: Der Graph der ReLu Aktivierungsfunktion

5.4 Ergebnis

Das Netz wurde innerhalb von 2 Stunden 200 Epochen lang trainiert und erreicht am Ende eine Genauigkeit von 98, 13%. Das verwendete System bestand aus einer Nvidia GeForce GTX 960, sowie einer CPU mit 4 Kernen, die hier jedoch nicht weiter relevant sind, da das Training auf der GPU stattfand, unter Ubuntu 18.04. Die Batchsize betrug 200. Der Kreuzentropiefehler wurde jedes mal, nachdem er berechnet wurde mit Hilfe von weights and biases geloggt. Nach jeder Epoche wurde das Netz außerdem auf dem Testdatensatz getestet und die Trefferquote wurde ebenfalls geloggt. Wie sich diese im Laufe der Zeit entwickelt ist in Abbildung 32 dargestellt.

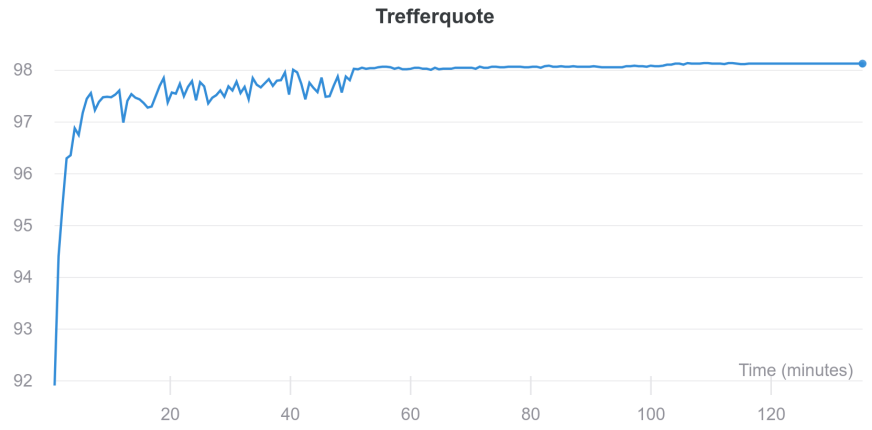

Abbildung 32: Ein Plot der Trefferquote aufgetragen gegen die Trainingszeit

Aus den Daten geht hervor, dass der Anstieg der Trefferquote in den ersten 10 Epochen extrem groß ist. In den nächsten rund 65 Epochen schwankt sie zwischen etwa 97% und 98% und stagniert danach knapp über 98% wo sie nur noch eine geringe Verbesserung aufweist. Das Netz hat also nach 75 Epochen seinen Höhepunkt erreicht und konnte sich nicht weiter verbessern. Dies korrespondiert auch mit dem in Abbildung 33 dargestellten Plot, der die Entwicklung des Kreuzentropiefehlers im Laufe des Trainings darstellt.
Auch hier lässt sich ein extremes Abnehmen des Fehlers in den ersten 10 Epochen ablesen, das von einem starken Schwanken bis etwa zur 75. Epoche gefolgt ist. Von da an stagniert der Fehler bei 0. Das Ziel von 97% Genauigkeit wurde also um einen Prozent überschritten und somit lässt sich feststellen, dass das Netz sehr gut in der Lage ist handgeschriebene Ziffern zu klassifizieren.

6 Schlusswort

Maschinelles Lernen ist ein extrem komplexes Forschungsgebiet, das ein enormes Potential aufweist. Die daraus hervorgehenden Technologien können das Leben revolutionieren und haben dies bereits in vielen Bereichen getan. Neuronale Netze

Abbildung 33: Ein Plot des Kreuzentropiefehlers aufgetragen gegen die Trainingszeit

stellen hier eine häufig verwendete Methode maschinellen Lernens dar. Sie sind an das menschliche Lernen angelehnt und können klassifizierungs, regressions und viele weitere Probleme lösen. Ihnen liegen algebraische Prozesse zu grunde, die aus dem Bereich der Statistik stammen. Um die Netze zu trainieren müssen große Datensätze vorhanden sein. Dies kann ein großes datenschutztechnisches Problem darstellen. Die Ausgabe neuronaler Netze ist außerdem nie zu 100% verlässlich. Trotz des großen Potentiales ist maschinelles Lernen jedoch nicht dass Allheilmittel und kann zwar viele aber bei weitem nicht alle Probleme lösen und ist bei einem Großteil der Problemen schlichtweg nicht effizient und verlässlich genug. maschinelles Lernen hat dennoch einen Einzug in unser Alltagsleben gefunden und wir begegnen ihm am Flughafen, im Supermarkt und am Smartphone (Die Gesichtserkennungssoftware zum Entsperren vieler Geräte nutzt maschinelles Lernen um eine höhere Genauigkeit zu erzielen). Von einer Welt, die von selbstbewussten und intelligenten Maschinen beherrscht wird, sind wir allerdings noch weit entfernt.

Literatur

[1] Hands-On Machine Learning with Scikit-Learn and TensorFlow
von Aurélien Géron
Veröffentlicht: March 2017 O'Reilly Media, Inc
ISBN: 9781491962282

[2] Die Logistik des Lernens eine Studie der LMU München
Quelle: www.uni-muenchen.de/forschung/news/2013/
f-71-13_kiebler_nervenzellen.html –abgerufen am 16.11.2019

[3] Common Loss functions in machine learning
Von Ravindra Parmar
Veröffentlicht am 02.09.2018, abgerufen am 07.01.2020
Quelle: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23

[4] Facial Recognition Is Everywhere at China's New Mega Airport
Bloomberg, 11. Dezember 2019
https://www.bloomberg.com/news/articles/2019-12-11/face-recognition-tech-is-everywhere-at-china-s-new-mega-airport
Abgerufen am 23.01.2020

[5] A US government study confirms most face recognition systems are racist
20.12.2019 MIT technology review
https://www.technologyreview.com/f/614986/ai-face-recognition-racist-us-government-nist-study/
Abgerufen am 23.01.2020

[6] Offizielle Dokumentation des PyTorch Frameworks
https://pytorch.org/docs/stable/nn.functional.html
Abgerufen am 30.01.2020

[7] Adam: A Method for Stochastic Optimization
Diederik P. Kingma und Jimmy Ba
arXiv:1412.6980 [cs.LG] (https://arxiv.org/abs/1412.6980)
Abgerufen am 31.01.2020

Abbildungsverzeichnis