Deep Neural Networks

Architekturen - Möglichkeiten - Grenzen

Julian Lorenz

Seminar: Neuste Trends in Big Data Analytics
Leitung: Dr. Julian Kunkel
Betreuer: Tobias Finn

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Scientific Computing
Universität Hamburg
Deutschland
31.03.2018

Contents

Abstract

Viele Arbeiten, die sich mit dem Thema der neuronalen Netze auseinandersetzen, gehen nur auf spezielle Architekturen ein oder beleuchten ein spezielles Problem. Es ist von elementarer Bedeutung, die abstrakten Konzepte und Mathematik dahinter zu verstehen, um selbst erfolgreich Netze zu programmieren, die lernen. Das Ziel dieser Arbeit ist es, einen Einblick auf das Perzeptron, das Multi-Layer Perzeptron und das Convolutional Network als große Vertreter des Forschungsgebietes zu gewähren.

1. Einführung

Im Jahr 1943 haben der Neurowissenschaftler Warren McCulloch und der Logiker Walter Pitts gemeinsam eine Arbeit mit dem Titel "A logical calculus of the ideas immanent in nervous activity"[2] im Journal "The bulletin of mathematical biophysics" veröffentlicht. Da Nervenzellen im Gehirn nur dann ein Signal senden, wenn ein bestimmter Schwellenwert überschritten wurde und sonst nicht - man spricht auch vom "alles-oder-nichts" Prinzip -, haben die beiden versucht diese Aktivitäten durch logische Modelle zu beschreiben. Damit wurde die Basis für ein ganz neues Forschungsgebiet geschaffen. Sechs Jahre später hat der Psychologe Donald Hebb die erste Lernregel für künstliche neuronale Netze niedergeschrieben.

$\Delta w_{i,j} = \eta * a_i * o_j$[3] ist die älteste und einfachste Form zur Darstellung eines Lernvorgangs in neuronalen Netzen. Dabei stellt $\Delta w_{i,j}$ die Änderung des Gewichts von Neuron i nach Neuron j dar. a_i beschreibt die Aktivierung vor dem Anwenden der sog. Aktivierungsfunktion. Im menschlichen Gehirn ist das die "alles-oder-nichts" Funktion. o_j ist die Ausgabe - nach dem Anwenden der Aktivierungsfunktion - an Neuron j. η ist die Lernrate mit der die Gewichte angepasst werden. Für gewöhnlich eine kleine Zahl zwischen 0 und 1.

Ein Einschnitt in die Forschung fand durch eine Arbeit aus dem Jahr 1969 statt. In einer genaueren mathematischen Analyse kamen Minsky und Papert[4] zu dem Schluss, das das XOR-Problem nicht mit einem einzelnen Perzeptron zu lösen sei und größere Netze ähnliche Limitationen hätten. Daraufhin wurde es ruhiger um das Forschungsgebiet, da viele Forschungsgelder gestrichen wurden. Heutzutage erleben neuronale Netze ihre (zweite) Blütezeit. Mit der exponentiell zunehmenden Bedeutung von Daten und deren Analyse sind neuronale Netze nicht mehr wegzudenken. Der Vorteil eines Neuronalen Netzes gegenüber eines klassischen Algorithmus ist, dass ein Netz das Problem nur möglichst gut approximieren will, ohne das es das Problem in seiner ganzen Fülle verstehen muss. Um einen Algorithmus zu entwerfen, muss das Problem analytisch verstanden worden sein. Ein Neuronales Netz geht nicht nach einem festgeschriebenem Schema vor wie etwa ein Algorithmus, sondern berechnet aus den Eingaben mithilfe einer sehr komplexen Funktion eine Ausgabe. Diese Funktion wiederum ist nicht gegeben, sondern wird im Verlauf des Lernprozesses gelernt. Anstatt einem Computer eine Vorschrift zu geben mit diesen Daten jenes zu machen und mit solchen Daten anderes zu berechnen, wird bei neuronalen Netzen lediglich eine Berechnungsvorschrift gegeben, wobei die Werte dieser Berechnung bei jeder Iteration angepasst werden. Da sich ein neuronales Netz an die Daten anpasst und "lernt" ein Problem möglichst gut zu approximieren wird oft auch der Begriff der künstlichen Intelligenz verwendet. Die kleinste Berechnungseinheit eines Neuronalen Netzes ist Das Perzeptron. Die verbreitetste Methode ein Netz lernen zu lassen ist die sog. Backpropagation. Ein Deep Neural Network bezeichnet ein neuronales Netz, das aus vielen aufeinanderfolgenden Schichten besteht. Was genau diese Schichten sind und woraus sie aufgebaut sind hängt von der Architektur ab und wird im Laufe der Arbeit erläutert. Es gibt keine offizielle Definition ab wie vielen Schichten ein Netz als "Deep" gilt, wichtig ist nur die Tatsache, dass neben der Eingabe- und der Ausgabeschicht noch andere, verdeckte Schichten existieren. Diese Schichten werden im Folgenden input output und hidden Layer genannt.

2. Verwandte Arbeiten

Wie Eingangs bereits erwähnt ist es schwer eine Arbeit zu finden, die einen guten Überblick über das Thema gibt. Das ist auch nicht einfach bei so einem großen Forschungsgebiet. Die Arbeit von Jason Yosinski et al. befasst sich z.B. nur mit der Frage inwiefern es möglich ist bestimmte Gruppen von Gewichten von einem Netz auf ein anders zu übertragen[18]. Dabei wird unter anderem untersucht, wie sehr diese Features von den zugrundeliegenden Daten abstrahieren[18, p. 3]. Andere Arbeiten befassen sich mit einer einzigen Variante der Optimierung des Lernens[13]. In Ref.[13] soll verhindert werden, dass sich das Netz zu nah an den Daten bewegt und nicht hinreichend abstrahiert. Während des Trainings wird jedes Neuron mit einer bestimmten Wahrscheinlichkeit ausgeschaltet, seine Ausgabe also unabhängig von der Eingabe auf Null gesetzt. Die Arbeit visualisiert eindrucksvoll, wie zu geringe Abstraktion auf Ebene der Gewichte aussieht und wie das höhere Abstraktion-

sniveau durch den sog. Dropout erzielt wird[13, p. 15]. Eine viel abstraktere Arbeit, die jedoch auch auf nur ein spezielles Problem abzielt, beschäftigt sich nur mit der Erkennung von Buchstaben der Gujarati Sprache, einer offiziellen Sprache in Indien, die aber auch in Pakistan von ca. 3 Mio. Menschen gesprochen wird[6].

3. Perzeptron

Ein Perzeptron ist die kleinste lernende Einheit eines Neuronalen Netzes und ist vergleichbar mit einer Nervenzelle im Gehirn. Es besteht aus einem einzelnen Neuron mit beliebig vielen Eingaben, die auf eine Ausgabe abgebildet werden. Dabei stellt das Perzeptron nur eine mathematische Funktion dar die im Laufe des Lernprozesses optimiert werden soll. Der Prozess des Lernens soll im späteren Verlauf genauer untersucht werden, an dieser Stelle soll lediglich die Berechnung eines Ergebnisses betrachtet werden.

3.1. Berechnung

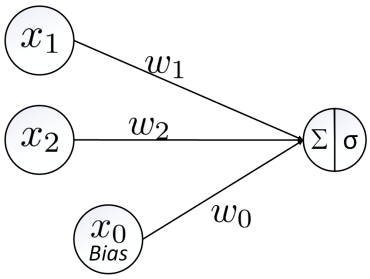

Figure 1. Perzeptron

Das Perzeptron summiert zuerst die gewichteten Eingaben auf.

$$a = x_1 * w_1 + x_2 * w_2 + x_0 * w_0$$

Der Bias spielt in Neuronalen Netzen eine wichtige Rolle. Er hat die Besonderheit, dass seine Eingabe immer konstant Eins ist $x_0 = 1$. Er kontrolliert den Schwellwert des Neurons. Ist der Bias - genauer das Gewicht des Bias - negativ, wird eine größere Aktivierung benötigt, damit das Neuron ein Signal ausgibt. Ist der Bias hingegen positiv, reichen kleinere Werte aus, damit das Neuron feuert. Das passiert im vorderen Bereich des Neurons. Als zweites wird im hinteren Bereich auf diese Summe eine Aktivierungsfunktion angewendet. Es gibt zahlreiche Aktivierungsfunktionen von denen hier nur wenige angesprochen werden sollen. Sie übersetzen die Aktivierung des Neurons in seine Ausgabe. Im Gehirn wird nach dem Alles-oder-Nichts Prinzip aktiviert. Ist ein bestimmter Schwellwert

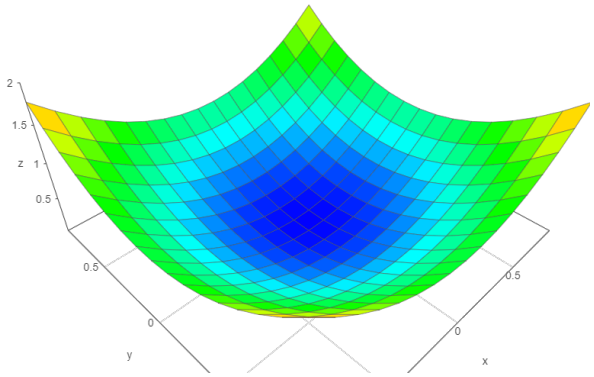

Figure 2. Beispielhafte Fehlerfunktion

bei der Aktivierung erreicht, feuert das Neuron. In der Informatik werden meistens andere Funktionen benutzt, wie z.B. die Sigmoide Logistikfunktion.

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \forall x \in \mathbb{R} : \sigma(x) \in (0,1)$$

Der Vorteil dieser Funktion ist ihre Eigenschaft, jede Eingabe auf einen Wert zwischen Null und Eins abzubilden, sowie ihre stetige Differenzierbarkeit. Der Tangens hyperbolicus ist eine weitere beliebte Funktion. Er bildet alle Eingaben auf das Intervall $I = (-1,1)$ ab. Des Weiteren lässt sich seine Ableitung relativ gut berechnen, doch dazu später mehr. Sofern nicht anders angegeben wird im Laufe der Arbeit $\sigma(x)$ als Synonym für eine beliebige Aktivierungsfunktion verwendet. Wenn man anwendet, dass der Bias x_0 immer Eins ist lautet die Ausgabe des Perzeptrons:

$$y = \sigma(x_1 * w_1 + x_2 * w_2 + w_0)$$

Bisher wurde angenommen, die Gewichte seien einfach gegeben. Das diese Annahme in der Realität nicht gültig ist, liegt auf der Hand. Es soll im folgenden betrachtet werden, wie man Gewichte findet, die ein bestimmtes Problem lösen. Das Lernen erfolgt durch Optimierung der Gewichte auf für ein bestimmtes Problem und ist keine Magie sondern nur Mathematik. Ähnlich wie der Mensch, lernt auch ein neuronales Netz aus seinen Fehlern und versucht jedes Mal etwas besser zu werden. Das verbreitetste Verfahren zur Optimierung der Gewichte heißt Backpropagation und soll hier näher untersucht werden.

3.2. Backpropagation

Das Ziel der Backpropagation ist es, für jedes Gewicht zu berechnen, wie sehr eine Änderung des

Gewichtes die Ausgabe in welcher Form beeinflusst. Abb. 2 zeigt eine solche Beispielhafte Fehlerfunktion. Die Achsen x und y repräsentieren den Wert des jeweiligen Gewichtes und die z-Achse enthählt den dazugeörigen Fehler. Gesucht wird der Punkt mit dem geringsten Fehler. Das Minimum der Fehlerfunktion. Zu Beginn befindet sich das Netz an einem zufälligen Punkt auf der Fehlerfunktion. Nach Berechnung der Ausgabe lassen sich die Gradienten der Fehlerfunktion nach den Gewichten bestimmen, um so in Richtung des Optimums zu "wandern", indem die Gewichte entsprechend angepasst werden. Das soll im Detail anhand eines Beispiels nachvollzogen werden.

Das klassische UND Gatter besitzt zwei Eingaben und eine Ausgabe, die beide Eingaben mit dem logischen UND verknüpft. Nur wenn beide Eingaben Eins sind, soll auch die Ausgabe Eins sein. Das verwendete Perzeptron entspricht dem Schema von Abb. 1. Das Ergebnis y wird analog zu den bereits genannten Formeln berechnet. Als erwartete Ausgabe wird hier und im Laufe der Arbeit $\hat{y}$ verwendet. Bevor das Netz seine Gewichte anpassen kann, muss mithilfe einer Fehlerfunktion berechnet werden, wie falsch die Ausgabe des Netzes war. In der Literatur findet man unter anderem die Synonyme Cost-function C oder Loss-function L, hier wird das Synonym Fehlerfunktion (Error-function) E. Um den Fehler eines Netzes zu berechnen, können viele Funktionen verwendet werden, hier soll erstmal nur der mittlere quadratische Fehler (Mean squared error - MSE) betrachtet werden, da er einfach anzuwenden ist, und evtl. bereits aus der Statistik bekannt ist. Der MSE berechnet sich über die Formel

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

Der Faktor $1/2$ wird lediglich zur Vereinfachung der Berechnungen verwendet, da er sich bei der Ableitung mit dem Exponenten zu Eins kürzen lässt. Da die Summe über jedes Ausgabeneuron geht, wir aber nur eines haben, lässt sich die Formel zu $E = \frac{1}{2} * (\hat{y}-y)^2$ vereinfachen. Ziel ist es, die Fehlerfunktion E für jedes Tupel von Eingabe und erwarteter Ausgabe $(x,\hat{y})$ zu minimieren. Da wir die Gewichte anpassen wollen um E zu minimieren, muss die Fehlerfunktion nach jedem Gewicht abgeleitet werden.

$$\frac{\partial E}{\partial w_{ij}}$$

Die Fehlerfunktion enthält bis jetzt nur die erwartete Ausgabe und die tatsächliche Ausgabe des Netzes. Die tatsächliche Ausgabe lässt sich jedoch durch $\sigma(a)$ ersetzen, wobei a die Aktivierung des Neurons beschreibt. Durch Anwenden der Kettenregel lässt sich die obige Gleichung erweitern.

$$\frac{\partial E}{\partial \sigma(a)}\frac{\partial \sigma(a)}{\partial w_{ij}}$$

Die Aktivierung eines Neurons wiederum lässt sich über die Summierung der gewichteten Eingaben berechnen und ersetzt in der Gleichung a. Ein weiteres Anwenden der Kettenregel liefert die Ableitung der Fehlerfunktion nach den Gewichten.

$$\frac{\partial E}{\partial \sigma(a)}\frac{\partial \sigma(a)}{\partial a}\frac{\partial a}{\partial w_{ij}}$$

Als nächstes werden die partiellen Ableitungen durch die Formeln ersetzt.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \sigma(a)}\frac{\partial \sigma(a)}{\partial a}\frac{\partial a}{\partial w_{ij}} \tag{1}$$

$$\frac{\partial E}{\partial w_{ij}} = -(\hat{y}-\sigma(a))\frac{\partial \sigma(a)}{\partial a}\frac{\partial a}{\partial w_{ij}} \tag{2}$$

$$\frac{\partial E}{\partial w_{ij}} = -(\hat{y}-\sigma(a))\sigma'(a)\frac{\partial a}{\partial w_{ij}} \tag{3}$$

$$\frac{\partial E}{\partial w_{ij}} = -(\hat{y}-\sigma(a))\sigma'(a)x_j \tag{4}$$

x_j entspricht in diesem Beispiel der Eingabe an Neuron j. In Gleichung (2) wurde die Ableitung der Fehlerfunktion nach der Aktivierungsfunktion ersetzt. In Gleichung (3) wurde die Ableitung der Aktivierungsfunktion ersetzt. Praktisch ist es offensichtlich von Vorteil, wenn nicht nur die Aktivierungsfunktion schnell und einfach zu berechnen ist, sondern auch ihre Ableitung nicht allzu komplizierte Berechnungen erfordert. Dabei ist jedoch aus praktischer Sicht nicht unbedingt auf die mathematische Einfachheit zu achten, sondern viel mehr auf die Effizienz und den Berechnungsaufwand für einen Computer. In Gleichung (4) wurde

$$a = x_1 * w_1 + x_2 * w_2 + x_0 * w_0$$

nach jedem $w_j : j \in \{0,1,2\}$ abgeleitet.

$$\forall w_j : \frac{\partial a}{\partial w_j} = x_j$$

Die Ableitung nach jedem Gewicht entspricht der zugehörigen Eingabe. Definieren wir weiter

$$\delta = (\hat{y}-\sigma(a))\sigma'(a)$$

lässt sich (4) auch schreiben als:

$$\frac{\partial E}{\partial w_{ij}} = -\delta x_j$$

Die berechneten Fehler werden mit einer Lernrate η multipliziert um ein langsames, aber kontinuierliches Konvergieren der Fehlerfunktion zu erlauben. Der Beweis, dass das Perzeptron konvergiert, wird hier aus Gründen der Komplexität nicht aufgeführt, kann jedoch bei Interesse in "Learning, Linear Separability and Linear Programming"[25, p. 5] im Detail nachvollzogen werden. η ist normalerweise ein kleiner Wert im Intervall $I = (0,1]$ Als Änderung für die Gewichte ergibt sich:

$$\Delta w_{ij} = -\eta \delta x_j$$

Das Netz minimiert bei jeder Iteration die Fehlerfunktion und wird mit jedem Mal besser. Im Optimalfall sind Iterationen und Erfolgsquote positiv korreliert.

Das logische UND Gatter zählt zu den linear seperablen Problemen. Bei dieser Klasse von Problemen lassen sich die Eingaben in Form von Vektoren im $\mathbb{R}^n$ Raum durch eine Hyperebene in diesem Raum in zwei Bereiche trennen, sodass jeder Punkt eines solchen Bereiches das selbe Soll-Ergebnis hat (Vgl. Abb. 3). Diese Probleme lassen sich durch ein Perzeptron wie es oben beschrieben ist lösen. Das lässt sich durch einige Umformungen schnell zeigen. Betrachtet wird die Funktion zur Berechnung der Aktivierung a eines Neurons. Dieser Wert ist hier beliebig wählbar, da er lediglich eine Additive Konstante ist, die auf das Konzept keinen Einfluss hat. Der Einfachheit halber soll als Aktivierung 0 verwendet werden. Diese Gleichung lässt sich nach x_2 umstellen, sodass (mit $x_0 = 1$) gilt

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{w_0}{w_2}$$

Da die Gewichte explizite Zahlen sind, gibt $-\frac{w_1}{w_2}$ die Steigung der Geraden und $-\frac{w_0}{w_2}$ die Verschiebung auf der x_2 Achse an. Diese Gerade kann im $\mathbb{R}^2$ Raum eine Punktewolke in zwei Bereichte trennen. Ist ein Problem linear seperabel, so existieren Parameter für diese Gerade (allg. im $\mathbb{R}^n$ eine Hyperebene), sodass beide Klassen getrennt sind und auf jeder Seite der Hyperebene nur Elemente der gleichen Klasse vorhanden sind. Ein berühmtes Problem wenn es um nicht linear seperable Probleme geht ist das XOR Gatter.

4. Multi-Layer Perzeptron

Um das XOR Problem zu lösen bedarf es einer neuen Architektur. Wie der Name schon andeutet, besteht das Multi-Layer Perzeptron (MLP) aus mehreren Schichten von Perzeptrons. Die erste Schicht besteht aus einem Perzeptron mit zwei Eingaben und zwei Ausgaben. Diese dienen als Eingabe für die nächste Schicht (Layer), welche aus einem Perzeptron

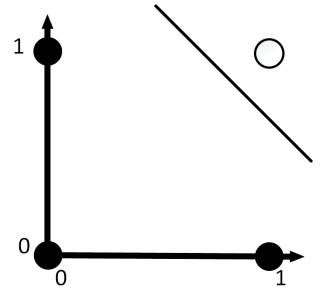

Figure 3. UND Gatter

mit zwei Eingaben und einer Ausgabe besteht. Das Ausgabeneuron dient hier als binärer Indikator für das XOR Gatter.

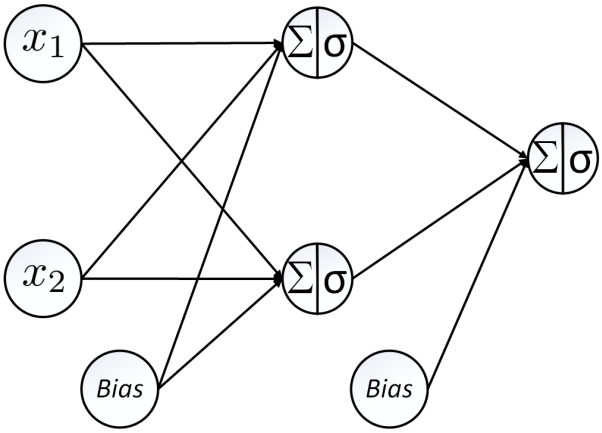

Figure 4. MLP

Die Berechnungen erfolgen Analog zum Perzeptron, mit dem Unterschied, dass in der ersten Schicht zwei Ausgaben existieren. So wird aus der Formel

$$a = x_1 * w_1 + x_2 * w_2 + x_0 * w_0$$

$$a_j = \sum_{i=0}^{n} (x_i w_{ij})$$

Dabei errechnet sich die Aktivierung des Neurons j über die Summe aller n gewichteten Eingaben $x_i w_{ij}$. Als Ergebnis erhält man einen Vektor $\vec{a}$ der die Aktivierung jedes Neurons enthält. Auf jedes Element dieses Vektors wird die Aktivierungsfunktion σ angewendet, sodass man $o_j = \sigma(a_j)$ erhählt. Diese Ausgaben werden als neue Eingaben x_i verwendet. Die anschließende Berechnung entspricht dem vorherigen Beispiel des Perzeptrons, da hier nur eine Ausgabe zu berücksichtigen ist.

$$a_j^{L+1} = \sum_{i=0}^{n_L} (x_i^L w_{ij}) \quad (1)$$

$$o_j^{L+1} = \sigma(a_j^{L+1}) \tag{2}$$

$$x_i^{L+2} = o_j^{L+1} \tag{3}$$

Die Aktivierung für den nächsten Layer $L+1$ ergibt sich wie gewohnt aus (1). In zwei wird jedes Element aus $\vec{a}$ aktiviert und somit eine Ausgabe. Diese Ausgabe ist gleichzeitig die Eingabe für den Layer $L+2$. Man beachte den Indexwechsel von o_j nach x_i, sodass die Formeln wieder angewendet werden können.

4.1. Backpropagation

Das Verfahren der Backpropagation muss erweitert werden, um auch auf die nicht-Ausgabe Schicht angewendet werden zu können. Der erste Teil bleibt identisch mit den bisher erarbeiteten Ergebnissen. Die Neuronen im hidden Layer bekommen jetzt allerdings zwei Werte aus der Ausgabeschicht um ihren Fehler zu berechnen. Der Grund ist die Kapselung der Berechnungen. Die Ausgabe des hidden Layer dient als Eingabe für den output Layer. Somit kann die Eingabe x_i im output Layer durch den output des hidden Layer o^{L-1} ersetzt werden. Dieser lässt sich wiederum durch die Aktivierungsfunktion und die Summe der gewichteten Eingaben ersetzen.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial \sigma(a_1)} \frac{\partial \sigma(a_1)}{\partial a_1} \frac{\partial a_1}{\partial \sigma(a_0)} \frac{\partial \sigma(a_0)}{a_0} \frac{\partial a_0}{w_{jk}} \tag{4}$$

$$\delta_1 = \frac{\partial E}{\partial \sigma(a_1)} \frac{\partial \sigma(a_1)}{\partial a_1} = (\hat{y} - \sigma(a))\sigma'(a) \tag{5}$$

$$\frac{\partial E}{\partial w_{jk}} = \delta_1 \frac{\partial a_1}{\partial \sigma(a_0)} \frac{\partial \sigma(a_0)}{a_0} \frac{\partial a_0}{w_{jk}} \tag{6}$$

$$\frac{\partial E}{\partial w_{jk}} = \delta_1 \delta_0 \frac{\partial a_0}{w_{jk}} \tag{7}$$

Für die Berechnung des Fehlers müssen jedoch noch die Gewichte aus dem output Layer hinzugezogen werden, da die Kette beim output Layer beginnt und bei einem Gewicht im input Layer aufhört. So muss die Aktivierung am Ausgabeneuron a_1 nicht mehr nach dem Gewicht, sondern nach der Ausgabe des Neurons abgeleitet werden, welches mit dem Gewicht - nach dem Abgeleitet werden soll - verknüpft ist. Die Ableitung der Aktivierung nach der Ausgabe dieses Neurons ist das Gewicht, welches die eben jenes Neuron mit dem Ausgabeneuron verknüpft $\frac{\partial a_1}{\partial \sigma(a_0)} = w$. Anschließend muss - um das gesuchte Gewicht zu erreichen - mithilfe der Kettenregel $\sigma(a_0)$ nach a_0 abgeleitet werden, was genau $\sigma'(a_0)$ entspricht, um als letztes a_0 nach dem Gewicht ableiten zu können. Die Formel $\Delta w_{ij} = -\eta \delta x_j$ aus dem Perzeptron wird verallgemeinert zu

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} x_j$$

wobei x_j die Eingabe in den Layer ist, in dem sich das Gewicht befindet, nach dem Abgeleitet wird. Das kann die Eingabe in ein Netz sein, aber auch die Ausgabe eines tiefer liegenden Layers. Um diese Theorie praktisch zu veranschaulichen, zeigen Algorithmus 1 die Berechnung des Ergebnisses und Algorithmus 2 die Berechnung des Fehlers sowie der Gewichtsänderungen anhand eines möglichen Konzepts der Implementation.

Algorithm 1: Berechnung der Ausgabe im MLP

```
   Data: Input x⃗, Matrix W_0, Matrix W_1
   Result: y⃗
 1 for i ← 0 to numHiddenOutput do
 2 |   a_0[i]=0;
 3 |   for j ← 0 to numInputs do
 4 |   |   a_0[i]+=x[j]*W_0[i][j];
 5 |   end
 6 |   o[i]=σ(a_0[i]);
 7 end
 8 for i ← 0 to numOutputs do
 9 |   a_1[i]=0;
10 |   for j ← 0 to numHiddenOutput do
11 |   |   a_1[i]+=o[j]*W_1[i][j];
12 |   end
13 |   y[i]=σ(a_1[i]);
14 end
```

Zur Berechnung des Ergebnisses mithilfe von Algorithmus 1 wird in der ersten for-Schleife der Aktivierungsvektor $\vec{a_0}$ des hidden Layer berechnet. Für jedes Element dieses Vektors werden die Eingaben mit den zugehörigen Gewichten multipliziert und aufsummiert. Das Anwenden der Aktivierungsfunktion auf jedes dieser Elemente liefert in Zeile 6 den Ausgabevektor des hidden Layer $\vec{o}$. Dieser wird in der Schleife in Zeile 8 als Eingabe verwendet, um die Aktivierung des output Layers $\vec{a_1}$ zu berechnen. Auch hier wird die Aktivierungsfunktion σ auf jedes Element dieses Vektors angewendet, um das Resultat des Netzes, den Vektor $\vec{y}$ zu berechnen.

Zur Berechnung des Fehlers wird in Algorithmus 2 im Gegensatz zu Algorithmus 1 nicht vorwärts, sonder rückwärts durch das Netz gegangen. Die erste Schleife betrachtet den output Layer, in der zunächst der bereits definierte δ-Wert für diesen Layer berechnet wird, da dieser an die darunter liegenden Layer weitergegeben wird und für die eigenen Berechnun-

Algorithm 2: Fehlerberechnung durch Backpropagation im MLP

Data: Target $\hat{\vec{y}}$, Output $\vec{y}$, Matrix W_0, W_1
Result: Error e, Gewichtänderungen ΔW

for $i \leftarrow 0$ **to** *numOutputs* **do**
  δ_1[i] = ($\hat{y}$[i] - y[i])*$\sigma'(a_1$[i]);
  e+=$0.5 * (\hat{y}[i] - y[i])^2$;
  for $j \leftarrow 0$ **to** *numHiddenOutput* **do**
    ΔW_1[i][j] = δ_1[i]*o[j];
  end
end
for $i \leftarrow 0$ **to** *numHiddenOutput* **do**
  δ_0[i] = 0;
  for $j \leftarrow 0$ **to** *numOutputs* **do**
    δ_0[i]+= W_1[j][i]*δ_1[j];
  end
  for $j \leftarrow 0$ **to** *numInputs* **do**
    ΔW_0[i][j] = δ_0[i]*x[j];
  end
end

gen nötig ist. Als Fehlerfunktion wird der MSE verwendet. In der inneren Schleife werden die eigenen Gewichtsänderungen berechnet in dem der δ-Wert mit der Eingabe des output Layers (der Ausgabe des hidden Layer) multipliziert wird. Nach Abschluss dieser Berechnungen kann der hidden Layer seinen Fehler berechnen indem er die Ableitungskette aus Gleichung (6) fortführt. Hierfür berechnet er seinen eigenen δ-Wert, der weitergegeben werden würde, falls noch mehr Layer existieren würden. Anschließend berechnet auch dieser Layer seine Gewichtsänderungen.

Algorithmus 3 zeigt wie die Gewichte aktualisiert werden, nachdem jeder Layer seine ΔW Matrix berechnet hat. Δw steht für die Änderung des aktuell betrachteten Gewichts w und wird der jeweiligen Matrix ΔW entnommen.

Algorithm 3: Aktualisierung der Gewichte

Data: Matrix $W_0, W_1, \Delta W_0, \Delta W_1$
Result: Matrix W_0, W_1

foreach $w \in W_0$ **do**
  w -= $\eta \Delta w$
end
foreach $w \in W_1$ **do**
  w -= $\eta \Delta w$
end

5. Anwendungen und Grenzen des MLP

Mithilfe von MLPs lassen sich ohne großen Aufwand Ziffern auf einem 28x28 Feld erkennen. Alles was dafür benötigt wird, sind genügend Layer und genügend Neuronen pro Layer[19]. Dabei reicht schon ein hidden Layer mit 800 Neuronen aus, um eine Genauigkeit von 99.3% zu erzielen. Ein MLP mit sechs Layern und jeweils 500 bis 1000 Neuronen konnte eine Genauigkeit von 99.65% erzielen[19] und ist damit das präziseste MLP auf diesen Daten[28]. Ein Beispiel aus den frühen Jahren der neuronalen Netze ist NETtalk, ein MLP mit drei Layern (203 Eingabeneuronen, 80 verdeckten Neuronen und 26 Ausgabeneuronen). Dieses Netz versucht menschliche Sprache nachzubilden, indem es eine Funktion von geschriebenem Englisch auf Phonetische Laute abbildet. Die Ergebnisse bei einem so übersichtlichem Netz sind erstaunlich, auch wenn es von flüssiger menschlicher Sprache weit entfernt ist. Zwar gab es schon vorher Programme, die Text in Sprache umgewandelt haben, allerdings hatten diese Programme eine deutlich längere Entwicklungszeit als NETtalk. Die 203 Eingabeneuronen repräsentierten sieben Buchstaben auf einmal, da jeweils 29 Neuronen einen Buchstaben darstellen. Der Text wird nach jeder Iteration mit ein wenig Überlappung verschoben, sodass am Ende eine Sequenz von phonetischen Steuerbefehlen ausgegeben wird, die zu dem Buchstaben in der Mitte gehört. Da diese Phoneme nur konkateniert werden und kein weicher Übergang zwischen den Lauten stattfindet, klingt das Resultat sehr abgehackt und unnatürlich. Ein neueres Netz, das Text in gesprochene Inhalte übersetzt wurde von Google entwickelt und trägt den Namen WaveNet. Dieses soll an späterer Stelle aufgegriffen werden, da es auf einer anderen Architektur beruht. Da das MLP die erste wirklich anwendbare Netzwerkarchitektur war, wurde viel mit ihr experimentiert um Möglichkeiten und Grenzen zu testen. Mit stetig steigender Rechenleistung und Speicherkapazität konnten immer mehr Personen eigene Netze programmieren und testen, ohne in teure Hardware investieren zu müssen. So wurde diese Architektur angewendet, um vom einfachen Spiel Tic-Tac-Toe bis zum komplexen Spiel Backgammon viele beliebte Spiele zu lernen. Aber nicht nur dort überzeugten diese Netze. Auch in anderen Bereichen wie z.B. der Überbuchung von flügen. Als Eingabe wurden diverse Faktoren verwendet, die Einfluss auf die Quote der Leute haben könnte, die einen Flug verpassen, wie etwa das Datum des Fluges, die Uhrzeit oder das Wetter. Mit der Ausgabe konnten Fluggesellschaften den Grad der Überbuchung auf bestimmten Flügen anpassen[21]. Ein Weiteres be-

liebtes Problemgebiet für diese Architektur sind Zeitreihen. Gegeben seien Zustände aus der Vergangenheit, wie sieht der nächste Zustand aus? Was erwartet das Netz, was als nächstes passiert? Das m.E. nach ein interessantes Gebiet, da das menschliche Gehirn sich sehr stark auf seine Erwartungen an die Zukunft verlässt. Ein einfaches Beispiel für eine Zeitreihe ist eine Wettervorhersage. Das Netz bekommt als Eingabe bestimmt Daten (Temperatur, Luftfeuchtigkeit, etc.) der Letzten sieben Tage und soll das Wetter von morgen vorhersagen[20]. Überträgt man das auf größere Zeitabschnitte würde sich eine Klimasimulation ergeben. Andere Netze sollten Kursentwicklungen an der Börse vorhersagen, basierend auf vergangenen Kursen oder anderen Daten, allerdings waren diese Netze die Kurse nicht so präzise vorhersagen wie die Forscher gehofft hatten.

Doch die Präzision ist bei bestimmten Anwendungen ist nicht das einzige Problem. Im Bereich der Bilderkennung wusste man lange Zeit nicht wie man gute Netze entwerfen sollte. Zwar können MLPs schon Ziffern auf einem 28x28 Feld mit sehr hoher Genauigkeit erkennen, allerdings sind das nur 784 Eingabeneuronen und die Netze hatten zwischen 3 und 6 Layern. Wollte man größere Bilder klassifizieren, standen hauptsächlich zwei Probleme im Weg. Das erste Problem hat unter dem Begriff "Vanishing Gradient" Bekanntheit erlangt. Vergleicht man Gleichung (1) mit Gleichung (4)

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \sigma(a)} \frac{\partial \sigma(a)}{\partial a} \frac{\partial a}{\partial w_{ij}} \tag{1}$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial \sigma(a_1)} \frac{\partial \sigma(a_1)}{\partial a_1} \frac{\partial a_1}{\partial \sigma(a_0)} \frac{\partial \sigma(a_0)}{a_0} \frac{\partial a_0}{w_{jk}} \tag{4}$$

wird eines deutlich. Je mehr Layer ein Netz hat, desto weiter muss der Fehler nach unten durchgereicht werden. Beim Perzeptron ist die Kette der Ableitungen nur drei Elemente Lang, bei einem hidden Layer ist sie schon fünf Elemente lang. Fügt man die Definition für δ in (4) ein erhält man:

$$\frac{\partial E}{\partial w_{jk}} = \delta_1 \delta_0 \frac{\partial a_0}{w_{jk}}$$

Diese Reihe lässt nicht auf n Layer erweitern:

$$\frac{\partial E}{\partial w_{jk}} = \delta_n \delta_{n-1} * \cdots * \delta_0 \frac{\partial a_0}{w_{jk}}$$

Mit jedem mal wird der durchgereichte Gradient kleiner. Das hat zur Folge, das die Anpassung der Gewichte auf den untersten Schichten bei tiefen Netzen sehr langsam vorangeht. Um dem Problem entgegenzutreten wurde mit diversen Methoden gearbeitet, zum Beispiel mit unterschiedlichen Lernraten für die einzelnen Layer. So wäre es denkbar, sogar eine Lernrate $\eta > 1$ in Betracht zu ziehen um das Problem zu lösen. Das Zweite Problem lässt sich nicht einfach lösen. Es ist die begrenzte Rechenkapazität von heutigen Rechnern. Möchte man ein Bild klassifizieren, das nicht nur aus 784 Pixeln und Graustufen besteht, sondern eine Auflösung von 1920x1080 Pixelen hat und in Farbe ist. Aus diesen Daten würden $1920 * 1080 * 3 = 6.22 * 10^6$ Eingabeneuronen resultieren. Angenommen man würde diese Eingabe direkt auf die $2.2 * 10^4$ verschiedenen Kategorien der ImageNet Bilddatenbank gewichten. Das würde $6.22 * 10^6 * 2.2 * 10^4 = 1.37 * 10^{11}$ Gewichte ergeben. Es lässt sich leicht errechnen wie viele Operationen Algorithmus 1 benötigt, um einen Layer zu berechnen. Die Zuweisung erfolgt einmal pro Ausgabe. Pro Gewicht werden zwei Operationen benötigt, die Multiplikation mit der Eingabe und die Addition. Jede Ausgabe wird auch aktiviert. Hier soll als sehr optimistische Schätzung angenommen werden, dass die Sigmoide Logistikfunktion sechs Operationen benötigt um berechnet zu werden. Daraus resultieren $22000 + 1.37 * 10^1 1 * 2 + 22000 * 6 = 2.74^{11}$ Operationen die für die Berechnung benötigt werden. Laut Dr. Donald Kinghorn[26] liefert der Intel Core i7 5960X eine theoretische Leistung von 384 GFlop/s. Er kann theoretisch $3.84 * 10^{11}$ Gleitkommaoperationen pro Sekunde durchführen. Dieser Prozessor würde ca. 0.7 Sekunden benötigen, um ein Ergebnis zu berechnen. Selbst wenn nur 10^6 Bilder betrachtet werden, würde ein Durchlauf etwa acht Tage dauern. Ganz zu schweigen von den Immensen Speicherkosten bei solch einer Zahl von Gewichten. Zwar lässt sich durch Parallelisierung und unter Zuhilfenahme von Beschleunigerkarten wie z.B. einer GPU die Berechnungszeiten verringern, sind aber bei weitem nicht effizient. Eine andere Architektur muss gefunden werden.

6. Convolutional Neural Net

Diese Architektur (im Verlauf mit CNN abgekürtzt) hat ihren Ursprung in einer Annahme, die gerade bei Bildern nicht von ungefähr kommt. Anstatt jedes Eingabeneuron mit jedem Ausgabeneuron zu verknüpfen und damit eine Vollverknüpfung zu erreichen wird angenommen, dass näher beieinander liegende Neuronen größeren Einfluss aufeinander haben als weit entfernte. Die Korrelation der Eingaben steigt mit kleinerem Abstand der Eingaben. Aus diesem Konzept wurde die Idee entwickelt, nicht die gesamte Eingabe zu betrachten, sondern nur einen Teil. Dieser Teil beinhaltet Gewichte und wird über die Eingabe geschoben, um so an jeder möglichen

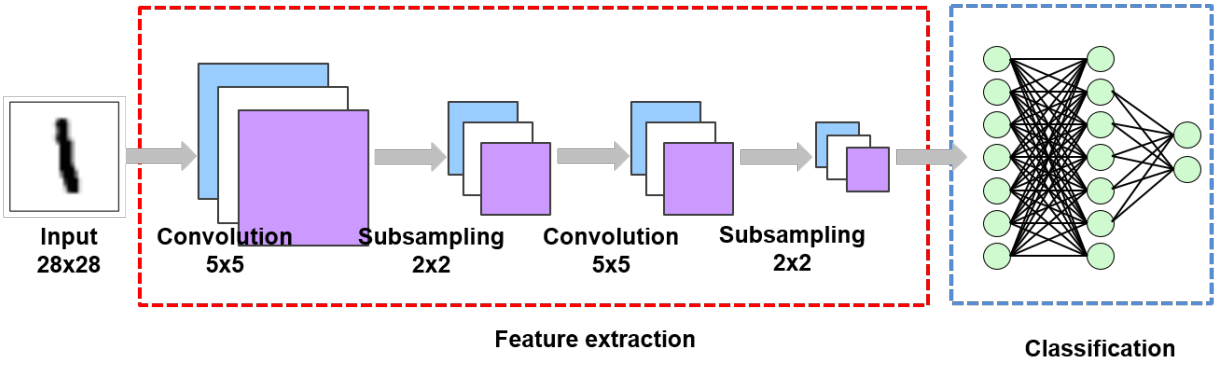

Figure 5. CNN
Source: Prof. Seungchul Lee [27]

Stelle ein Ergebnis zu bekommen. Die Eingaben haben jetzt nicht mehr eigene Gewichte die auf ein Neuron zeigen, sondern teilen sich die Gewichte des Teils mit allen anderen Eingaben. Dieser Teil wird Feature genannt. Die Ausgabe eines Features wird am Ende mit einer Aktivierungsfunktion σ aktiviert und. Mehrere Features die auf derselben Eingabe arbeiten werden zusammengefasst im Convolution Layer. Die Ergebnisse des Convolution Layer werden in Feature Maps gespeichert. Der Faltungsoperator $\otimes$ - der die Convolution durchführt - wird definiert als:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \otimes \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} x_{11}w_{11}+x_{12}w_{12}+ & x_{12}w_{11}+x_{13}w_{12}+ \\ x_{21}w_{21}+x_{22}w_{22} & x_{22}w_{21}+x_{23}w_{22} \\ x_{21}w_{11}+x_{22}w_{12}+ & x_{22}w_{11}+x_{23}w_{12}+ \\ x_{31}w_{21}+x_{32}w_{22} & x_{32}w_{21}+x_{33}w_{22} \end{bmatrix} \quad (1)$$

x_{ij} ist die Eingabe und w_{ck} ein Gewicht des Features. Das Feature wir auf die Eingabematrix gelegt und jedes Element mit dem darunterliegenden Multipliziert. Anschließend werden die Werte aufsummiert und in die Ergebnismatrix geschrieben. Das Feature wird an eine andere Position geschoben und das Verfahren wiederholt sich. Da es in dem Beispiel nur vier mögliche Positionen gibt, wurde aus einer 3x3 Matrix über die Faltung eine 2x2 Matrix. Ist die Eingabe sehr groß, kann durch viele aufeinanderfolgende Features die Dimension deutlich verkleinert werden.

Eine weitere Überlegung kommt von der Idee, dass es nicht von elementarer Bedeutung ist, wo ein bestimmtes Feature oder Merkmal auftritt, sondern ob es überhaupt auftritt. Wenn das Feature einen Hund in einem Bild erkennen kann, so ist es von sekundärer Bedeutung wo der Hund auf dem Bild ist. Viel wichtiger ist zunächst, ob überhaupt ein Hund auf dem Bild ist. Um das zu erreichen wird im allg. nach jedem Feature ein Pooling Layer eingebaut. In Abb. 5 wird der Name Subsampling verwendet. Beim Pooling legt man eine Matrixgröße fest und berechnet aus allen Eingaben eine Funktion. Die bekanntesten Funktionen sind das Max und das Mean Pooling. Beim Max Pooling wird nur der größte Wert übernommen, beim Mean Pooling wird ein Arithmetisches Mittel gebildet. Normalerweise ist die Pooling-Funktion eine, die auf eine Matrix angewendet nur einen Wert zurückgibt. In Abb. 5 wird ein kleines CNN beispielhaft dargestellt. Als Eingabe dient ein Bild mit 28x28 Pixeln. In dem Beispiel existieren im ersten Convolution Layer drei Features mit jeweils 5x5 Gewichten. Aus der Eingabe entstehen 3 Matrizen mit jeweils 23x23 Werten (28-5), da jedes Feature horizontal und vertikal an 23 Positionen platziert werden kann. Anschließend folgt ein Pooling Layer der größe 2x2. Hier ist das Ergebnis drei 21x21 Matrizen, da auf jeder Matrix davor das Pooling angewendet wird. Darauf folgt wieder ein Convolution Layer, welcher aus 3 Features besteht. Jedes dieser Features hat die Dimension 3x5x5, da das Feature alle Knäle der Eingabe betrachtet. Da das Bild nur aus Graustufen bestand (einem Kanal), hatten die Features eine Dimension von 1x5x5. Die resultierende Feature Map aus der Faltung mit drei 3x5x5 Features sind drei Matrizen der Dimension 16x16. Im Anschluss wird ein Pooling Layer angewendet, der die Eingabe auf drei Matrizen der größe 14x14 verkleinert. Aus einer ursprünglichen Eingabe von $28 * 28 = 784$ wurden $14 * 14 * 3 = 588$ Ausgabeneuronen. Diese verkleinerte Ausgabe kann vektorisiert werden, indem alle Elemente der Reihe nach in einem Vektor zusammengefasst werden. Dieser Vektor lässt sich gut als Eingabe für ein MLP verwenden, welches die abscließende Klassifikation vornimmt. Das CNN kann also als Vorstufe für ein MLP verstanden werden, um die Dimension der Eingabe zu verringern und die Korrelation naheliegender Bildpunkte

auszunutzen. Das MLP gewichtet die Relevanz eines Features für eine bestimmte Ausgabeklasse. Falls ein Feature einen Autoreifen "erkennen" könnte, und die Frage ist, ob das Bild ein Mensch oder ein Auto ist, so wäre die Gewichtung des Autoreifens zu Auto deutlich höher als zu Mensch.

Die Backpropagation des abschließenden MLPs erfolgt wie bereits gewohnt, und auch in den darunterliegenden Layern wird wird ähnlich vorgegangen um die partiellen Ableitungen nach allen Gewichten zu errechnen. Allerdings soll hier aus Gründen der Komplexität auf die gesamte Mathematik verzichtet werden. Bei Interesse können detaillierte Informationen von Intel's "Computational Intelligence"[30] oder Mayank Agarwal [29] erlangt werden.

7. Anwendungen und Grenzen des CNN

Das CNN erfreut sich immer größerer Beliebtheit, da es ein breites Spektrum an Problemen lösen kann, die auch praktischen Nutzen haben. So gibt es CNNs, die Fotos von Benutzern automatisch klassifizieren, und so das durchsuchen nach bestimmten Stichworten wie z.B. "Strand" erlauben. Ein anderes CNN errechnet die Aussagekraft eines Frames innerhalb eines Videos, um so entweder ein Thumbnail herauszusuchen, oder ein aus den wichtigsten Frames bestehendes Preview zu erstellen[8]. Aber nicht nur Bilder können mithilfe von CNNs erkannt werden, auch die Erkennung von Gesten ist möglich[9]. So können Systeme durch einfache Handbewegungen gesteuert werden und diese Handbewegungen individuell trainiert werden. Das Paper "Network in Network"[15] benutzt anstatt der Features in CNNs selbst kleine MLPs[15, p. 2-4] und versucht so noch komplexere Features zu modellieren, sowie eine höhere Präzision. Diese Anpassung der Architektur lässt sich auf viele Problemkategorien anwenden. Die Arbeit von Andrej Karpathy und Li Fei-Fei[16] benutzt eine Kombination aus einem CNN und einem Rekurrenten neuronalen Netz (Eine variante des MLP in der auch reflexive Gewichte und Gewichte zurück erlaubt sind), um zu einem gegebenem Bild und Bildbeschreibung die Orte zu den jeweiligen Objekten die in der Beschreibung genannt werden zu zeigen. Diese sog. Segmentation von Objekten erlaubt es eine Box um das Betreffende Objekt zu ziehen. Diese für einen Menschen vermeintlich einfache Aufgabe ist nicht trivial und erfordert durchaus Aufwand. Cha Zhang und Zhengyou Zhang aus der Microsoft Research Abteilung haben in einer Arbeit über multitaskingfähige CNNs zur Erkennung mehrerer Gesichter geschrieben[5].

Das wohl berümteste Beispiel bleibt jedoch AlphaGo von Google's Deep Mind[10]. Die erste Version dieses Netzes war eine Kombination aus zwei Netzen. Ein Policy und ein Value Netzwerk. Das Policy Netz wurde mithilfe einer Datenbank von Go-Spielen trainiert. Das Value Netz sollte zu einer bestimmten Spielposition die Gewinnwahrscheinlichkeit bestimmen. Auch dieses Netz wurde mithilfe der Datenbank gefüttert. Dieses Programm war in der Lage, den bis dahin mehrfachen Weltmeister Lee Sedol zu schlagen. Die Schwierigkeit von Go im Gegensatz zu anderen Brettspielen wie z.B. Backgammon oder Schach liegt in seiner viel größeren Komplexität, da der Zustandsraum der möglichen Züge gigantisch ist. Doch die Entwickler von AlphaGo waren noch nicht zufrieden und haben AlphaGo Zero entwickelt, eine Verbesserung des eben angesprochenen Vorgängers. Im Gegensatz zu AlphaGo wurden AlphaGo Zero nur die Spielregeln beigebracht, aber keine anderen Daten. Diese Art des Trainings wird auch Tabularaza Training genannt. Während AlphaGo mehrere Monate trainiert werden musste um das Niveau des Weltmeisters Lee Sedol zu erreichen, dauerte es bei AlphaGo Zero nur ein paar Tage. Nach einer Woche konnte die vorherige Version kein Spiel mehr gegen AlphaGo Zero gewinnen[11].

Das WaveNet[12] wurde bereits angesprochen und ist ein CNN, das erstaunlich gute Resultate im Bereich der Simmsynthese liefert[12, p. 6-7]. Der entscheidende Unterschied zu NETtalk ist neben der Architektur seine Ausgabe. Anstatt ein Phonem auszugeben ist die Ausgabe von WaveNet eine Audiowelle. Diese kann anschließend wiedergegeben werden. Das CNN versucht mithilfe von Daten aus der Vergangenheit den nächsten Zeitabschnitt vorherzusagen.

Allerdings sind auch CNNs grenzen Gesetzt. Diese Grenzen ergeben sich leicht, wenn die Annahme noch einmal betrachtet wird. Falls nahe beieinander liegende Eingaben nicht - wie in der Annahme - korreliert sind, wird auch ein CNN keine guten Ergebnisse liefern. Ob nahe beieinander liegende Eingaben korreliert sind lässt sich überprüfen indem man sich folgende Frage stellt. Wenn ich viele Zeilen oder Spalten meiner Eingabe vertausche, ändern sich die Daten dadurch Grundlegend? Vertauscht man genug Zeilen oder Spalten bei Bildern ist das Ergebnis völlig anders. hat man allerdings eine Tabelle von Schülern mit bestimmten Daten und möchte die Wahrscheinlichkeit, mit der ein Schüler Durchfällt, so ist es nicht relevant, in welcher Reihenfolge die Schüler angeordnet sind oder in welcher Reihenfolge die Attribute sind. Ein CNN wird hier nicht so gute Ergebnisse liefern.

8. Zusammenfassung

Auch neuronale Netze sind keine magischen Programme die alles von alleine machen. Nicht nur die Mathematik ist am Anfang nicht einfach zu verstehen - gerade die Berechnung des Fehlers -, sondern auch die Konzepte hinter komplizierteren Architekturen wie den CNNs können am Anfang Probleme bereiten. Doch es ist von elementarer Bedeutung das Grundgerüst des Perzeptrons und des MLP zu verstehen, da auf diesen Komzepten alle Architekturen aufbauen und diese Konzepte erweitern bzw. abändern. Aus diesem Grund ist es wichtig gerade den Vorgang des Lernens, die Backpropagation in seiner Mathematik verstanden zu haben, um eigene Netze programmieren zu können. Zwar ist das nicht nötigt dank diverser Bibliotheken, allerdings hilft einem die beste Bibliothek nichts, wenn man eine ganz neue Architektur entwerfen will. Da ist es von Vorteil die Konzepte bereits zu kennen.

Das MLP kann gut zur Klassifikation verwendet werden, hat aber bei zu vielen Layern das Problem, dass die Gradienten auf den tieferen Layern immer kleiner werden und dadurch die Gewichte noch weniger angepasst werden. Außerdem steigt die Berechnungszeit bei großen Eingaben deutlich, da die Zahl der zu verändernden Gewichte immer größer wird.

CNNs versuchen dieses Problem zu lösen indem sie die Anzahl der zu optimierenden Gewichte reduzieren. Das wird mithilfe von Features erreicht, einer Gruppierung von Gewichten, welche sich jede Eingabe teilt. Das klappt allerdings nur, wenn nahe beieinander liegende Eingaben auch wirklich eine höhere Korrelation haben als weiter entfernte. Diese Annahme ist bei Bild- oder Tondaten durchaus sinnvoll, aber auch nicht allgemein gültig. Die Schwierigkeit bleibt die geeignete für ein Problem zu finden, sowie geeignete Parameter wie z.B. die Lernrate η oder die zahl der hidden Layer, um nur zwei zu nennen.

References

[1] W. S. Mcculloch. “The brain computing machine”. In: *Electrical Engineering* 68.6 (1949), pp. 492–497. ISSN: 0095-9197. DOI: 10.1109/EE.1949.6444817.

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

[3] Donald O. Hebb. “The Organization of Behaviour”. In: (1949).

[4] Marvin Minsky and Seymour Papert. “Perceptrons. An Introduction to Computational Geometry.” In: *Science* (1969). URL: http://science.sciencemag.org/content/165/3895/780.

[5] C. Zhang and Z. Zhang. “Improving multiview face detection with multi-task deep convolutional neural networks”. In: *IEEE Winter Conference on Applications of Computer Vision*. 2014, pp. 1036–1041. DOI: 10.1109/WACV.2014.6835990.

[6] V. A. Naik and A. A. Desai. “Online handwritten Gujarati character recognition using SVM, MLP, and K-NN”. In: *2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2017, pp. 1–6. DOI: 10.1109/ICCCNT.2017.8203926.

[7] C. Huang, S. Ni, and G. Chen. “A layer-based structured design of CNN on FPGA”. In: *2017 IEEE 12th International Conference on ASIC (ASICON)*. 2017, pp. 1037–1040. DOI: 10.1109/ASICON.2017.8252656.

[8] M. A. Nahian et al. “CNN-based Prediction of Frame-Level Shot Importance for Video Summarization”. In: *2017 International Conference on New Trends in Computing Sciences (ICTCS)*. 2017, pp. 24–29. DOI: 10.1109/ICTCS.2017.13.

[9] C. J. Tsai et al. “Synthetic Training of Deep CNN for 3D Hand Gesture Identification”. In: *2017 International Conference on Control, Artificial Intelligence, Robotics Optimization (ICCAIRO)*. 2017, pp. 165–170. DOI: 10.1109/ICCAIRO.2017.40.

[10] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (2016). Article, 484 EP –. URL: http://dx.doi.org/10.1038/nature16961.

[11] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (2017). Article, 354 EP –. URL: http://dx.doi.org/10.1038/nature24270.

[12] Aron van den Oord et al. “WaveNet: A Generative Model for Raw Audio”. In: *Arxiv*. 2016. URL: https://arxiv.org/abs/1609.03499.

[13] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[14] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. “A Convolutional Neural Network for Modelling Sentences”. In: (2014). Article. URL: http://www.aclweb.org/anthology/P14-1062.

[15] Min Lin, Qiang Chen, and Shuicheng Yan. “Network in Network”. In: (2014). Article.

[16] Andrej Karpathy and Li Fei-Fei. “Deep Visual-Semantic Alignments for Generating Image Descriptions”. In: (2015). Article.

[17] Ronan Collobert et al. “Natural Language Processing (almost) from Scratch”. In: (2011). Article. URL: https://arxiv.org/pdf/1103.0398v1.pdf.

[18] Jason Yosinski et al. “How transferable are features in deep neural networks?” In: (2014). Article. URL: https://arxiv.org/pdf/1411.1792v1.pdf.

[19] Dan Claudio Ciresan et al. “Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition”. In: (2010). URL: https://arxiv.org/pdf/1003.0358.pdf.

[20] C. Voyant et al. “Meteorological time series forecasting based on MLP modelling using heterogeneous transfer functions”. In: (2015). URL: http://iopscience.iop.org/article/10.1088/1742-6596/574/1/012064/pdf.

[21] Russel Beale and Brandon Jackson. *Neural Computing - An Introduction*. Taylor & Francis Group, 1990.

[22] *Autoencoders*. URL: http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/ (visited on 03/26/2018).

[23] *Convolutional Neural Network*. URL: http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/ (visited on 03/26/2018).

[24] *Multi-Layer Neural Network*. URL: http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/ (visited on 03/26/2018).

[25] *Learning, Linear Separability and Linear Programming*. URL: https://courses.engr.illinois.edu/cs573/fa2013/lec/slides/25_notes.pdf (visited on 03/26/2018).

[26] Dr. Donald Kinghorn. *Linpack performance Haswell E (Core i7 5960X and 5930K)*. URL: https://www.pugetsystems.com/labs/hpc/Linpack-performance-Haswell-E-Core-i7-5960X-and-5930K-594/ (visited on 03/26/2018).

[27] Prof. Seungchul Lee. *CNN Example*. URL: http://i-systems.github.io/HSE545/machine\%20learning\%20all/Workshop/image_files/cnn_example.png.

[28] *The MNIST Database of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/ (visited on 03/26/2018).

[29] Mayank Agarwal. *Back Propagation in Convolutional Neural Networks Intuition and Code*. URL: https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199 (visited on 03/26/2018).

[30] Boris Ginzburg. *CNN: Back-propagation*. URL: http://courses.cs.tau.ac.il/Caffe_workshop/Bootcamp/pdf_lectures/Lecture\%203\%20CNN\%20-\%20backpropagation.pdf (visited on 03/26/2018).