FH-Wedel

Besondere Methoden zum Trainieren und Auswerten tiefer neuronaler Netze

Autor:
Dennis Maas

Betreuer:
Prof. Dr. Sebastian Iwanowski

4. Juni 2019

Inhaltsverzeichnis

1 Einführung **2**

2 Motivation **2**

3 Besondere Methoden **3**

4 Tiefe neuronale Netze **3**

5 Vanishing Gradient Problem **4**
5.1 Definition 4
5.2 Entstehung 4
5.3 Exploding Gradient Problem 6
5.4 Aktivierungsfunktion 6

6 Internal Covariant Shift **8**
6.1 Entstehung 8
6.2 Batch Normalization 10
6.3 Momentane Forschungsergebnisse 12

7 Overfitting **12**
7.1 Regularisierung 14
7.2 Validation 14
7.3 Dropout 15
7.4 Jumpout 16
7.5 Batch Normalization 17

8 Degradation of training accuracy **17**
8.1 Entstehung 17
8.2 Residual Node 17

9 TI-Pooling **19**

10 Self-Normalizing Networks **21**

11 Versuchsdaten **22**

12 Schlusswort **23**

1 Einführung

Mit steigender Leistungsfähigkeit der Hardware und nutzen dedizierter Recheneinheiten, wie der Grafikkarte, für komplexere Rechnungen, wie Matrixoperationen, sind die Möglichkeiten für die Entwicklung immer tieferer neuronaler Netze gewachsen. ResNet, YOLO und HMM-Basierte Modelle lösen bereits algorithmisch schwer bis nicht lösbare Probleme, um nur ein paar Beispiele tiefer neuronaler Architekturen zu nennen. Mit steigender Komplexität der zu lösenden Probleme und Zwecks dessen entwickelter KI-Modelle, werden neue Verfahren zum Trainieren und Entwickeln dieser notwendig.

2 Motivation

Der Trend zum Nutzen tiefer neuronaler Netze steigt stetig. Über die letzte Jahrzehnte habe sich immer mehr Forschungsarbeiten damit beschäftigt, wie diese effizienter trainierbar und auch in ressourcenärmeren Umgebungen anwendbar werden. Die Anwendung der daraus entstandenen „Best Practices“, zumindest zum Teil, sollen hier nicht nur erklärt, sondern auch der Grund ihres Funktionierens über intuitive Beispiele vermittelt werden. Dies soll ermöglichen über eine Große Menge an verschiedenster „Best Practices“, eine abstrakte, anwendungsfallorientierte Bewertung machen zu können und den zukünftigen Einstieg in neue Techniken einfach zu gestalten. Zunächst werden sehr grundlegende Probleme adressiert, um mit dem Verständnis zu komplexeren Methoden über gehen zu können.

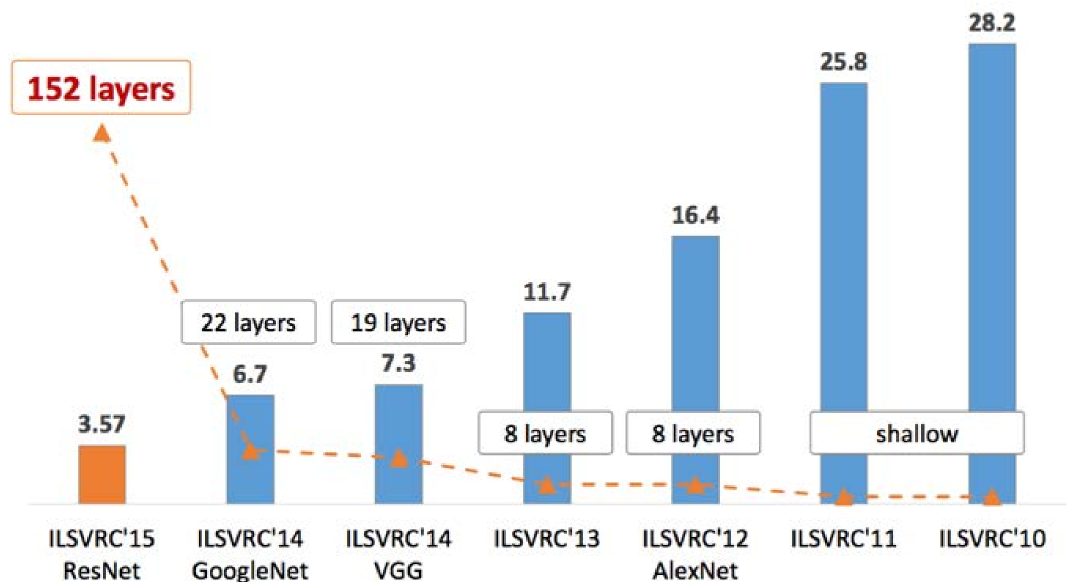

Abbildung 1: Darstellung der Netzwerktiefen verschiedener Modelle im Rahmen der ILSVRC von 2010-2015 im Bezug auf den Anteil fehlerhafter Aussage in Prozent. Den Trend zu tieferen Modellen kann man an dem Beispiel gut erkennen. [Tre]

3 Besondere Methoden

Eine besondere Methode ist innerhalb dieser Ausarbeitung folgendermaßen definiert: Eine Methode qualifiziert sich durch Adressieren eines Problems, dass bei tiefen neuronalen Netzen im speziellen oder schwerwiegendem Maße auftritt. Im folgenden werden auch Techniken wie Residual Nodes in Kapitel 8.2 erläutert, die nur im Kontext tiefer neuronaler Netze Sinn ergeben.

4 Tiefe neuronale Netze

Ein neuronales Netzwerk ist tief, wenn es neben der Eingabe- und Ausgabeschicht weitere Schichten, genannt Hidden Layers, enthält. Verallgemeinernd gilt also: Ein neuronales Netz ist Tief ab zwei oder mehr Hidden Layers. Diese Definition schließt sich dem Konsens im Rahmen der Recherche an, wobei nicht streng wissenschaftlich geklärt ist, wann ein Netz in Abhängigkeit seiner Tiefe zu einem tiefen neuronale Netz klassifiziert werden kann.

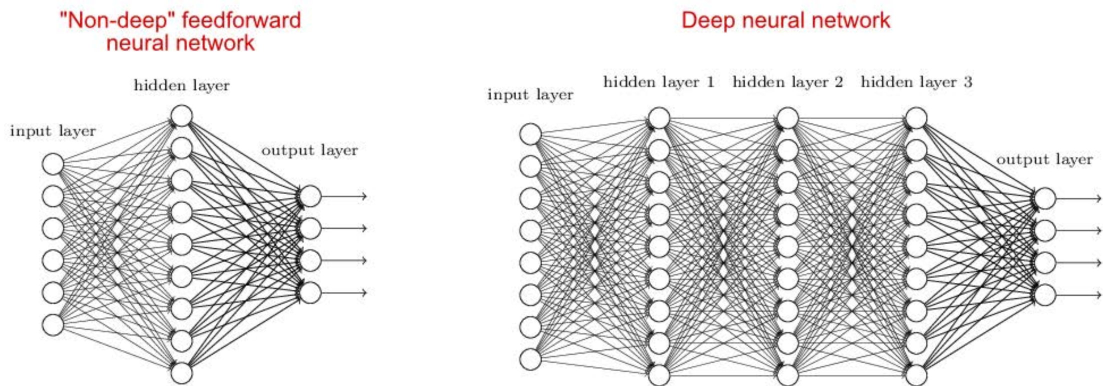

Abbildung 2: Gegenüberstellung eines flachen und tiefen Netzes: [Sha]

Interessanter wird es, wenn die Definition von der erwarteten Funktionalität abgeleitet wird. Die Idee hinter steigender Netzwerktiefe ist, dass beispielsweise die erste Schicht sehr grundlegende Zusammenhänge erkennt. Diese werden dann in der zweiten Schicht weiter zu neuen Zusammenhängen verarbeitet. Mit steigender Tiefe profitieren also folgende Schichten von höherer Komplexität der erkannten Zusammenhänge und können Informationen allgemeingültiger und weniger fein granuliert generalisieren, um ähnliche Probleme zu erkennen, ohne das Problem bereits direkt durch nur eine Abbildung lösen zu müssen. [GB10]

Ein Beispiel stellt die Zahlenerkennung dar:

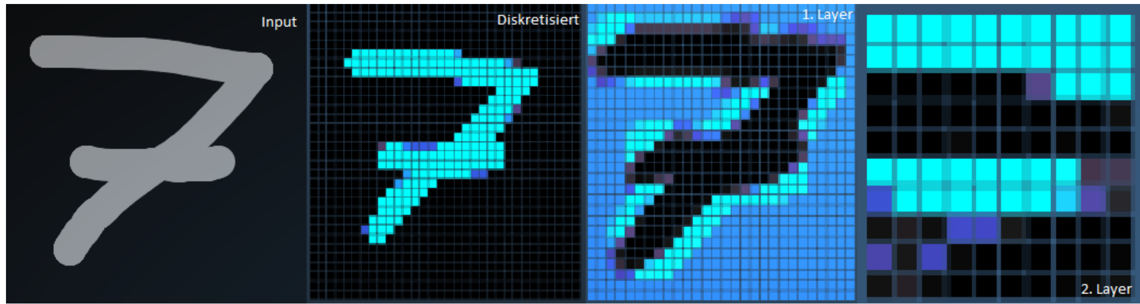

Abbildung 3: Aufnahme aus dem Simulator: [Ima]

Zunächst wird von einer handgeschriebenen 7 ausgegangen, die von der KI ausgewertet werden soll. (Abb. 3 Input) Diese wird dann diskretisiert und auf die Eingangsneuronen des gegebenen neuronalen Netzes abgebildet. (Abb. 3 Diskretisiert) Die erste Schicht extrahiert Kanten, in hellblau dargestellt (Abb. 3 1. Layer), die zweite Schicht kombiniert wiederum die Kanten zu einem abstrakteren Gebilde (Abb. 3 2.Layer), um anschließend mit der letzten Schicht auf die verschiedenen Zahlen abzubilden. Die erkannten Zusammenhänge werden Features genannt. Der Begriff Feature beschreibt demnach eine Charakteristik der ausgewerteten Eingangsdaten. In diesem Kontext wäre die Kante einer Zahl ein erkanntes Feature.

5 Vanishing Gradient Problem

5.1 Definition

Um das Problem darzustellen, sollen die Ergebnisse einer Zahlenerkennung, analog zum Beispiel in der Definition tiefer neuronaler Netze, auf Basis des bekannten MNIST-Datensatzes analysiert werden. Die genauere Beschreibung des Versuchs und der Ergebnisse ist aus Kapitel 11 zu entnehmen.

Neuronen	Schichten	Fehler	Klassifizierungsfehler	Validierung
30	1	1,78	0.00	100%
60	2	2,27	0,70	80%
90	3	2,31	0,93	20%
120	4	2,31	0,93	20%

Abbildung 4: Ergebnisse des Trainings mit der Sigmoid-Aktivierungsfunktion und verschiedener Anzahl an Schichten.

Erkennbar ist die sinkende Erfolgsquote trotz beziehungsweise durch Vergrößerung des Netzes im Vergleich zu den vorherigen Erfolgsquoten. Ein ähnlicher Test ist in der Quelle [Nie18] zu finden.

5.2 Entstehung

Um die Ursache des Problems zu erkennen, muss ein Teil des Backpropagation-Lernalgorithmusses analysiert werden. Bei der Backpropagation wird eine Fehlerfunktion zwischen den erwarteten Ergebnissen und den tatsächlichen Ergebnissen einer Auswertung gebildet. Für jedes Neuron wird dieser Fehler beginnend mit der

Ausgabeschicht bis zur Eingabeschicht durchpropagiert und mit Hilfe des Gradienten der Fehlerfunktion im Verhältnis zum jeweiligen Kantengewicht die Werte des Neurons angepasst, sodass der Fehler des Netzes für die Auswertung sinkt.

Der Gradient ist in diesem Kontext, ohne weiter auf die Backpropagation einzugehen, ein Optimierungsfaktor für die Parameter des betrachteten Neurons, der auf Basis vorhergehenden Fehler im Verhältnis zu den Kantengewichten errechnet wird. Während in den hinteren Schichten nahe der Ausgabeschicht noch wenige, vorhergehende Fehler miteinbezogen werden, steigt diese Anzahl je weiter nach vorne iteriert wird. Um dies Exemplarisch darzustellen, dient das minimalistische Netz aus Abbildung 5.

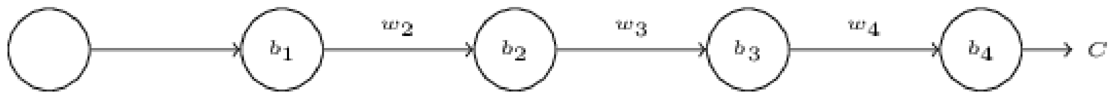

Abbildung 5: Vereinfachte Form eines tiefen Neuronalen Netzes mit nur einem Neuron pro Schicht. [Nie]

Formelglossar für folgende Rechnung

a$_x$	ungewichteter Eingangswert des Neurons
z$_x$	gewichteter Eingangswert des Neurons
sig(x)	Sigmoid Funktion
sig'(x)	Ableitung der Sigmoid Funktion
$\frac{\partial C}{\partial a_4}$	Fehlerfunktion bei der Ausgabeschicht.
$\frac{\partial C}{\partial b_x}$	Gradient bei Neuron mit dem Bias von b_x.

Für die Rechnung wird eine gängige Aktivierungsfunktion, genannt Sigmoid, betrachtet, die auch bei der Auswertung 4 genutzt wurde. Den Verlauf der Ableitung kann man aus Abbildung 6 entnehmen, wobei 0.25 das Maximum darstellt.

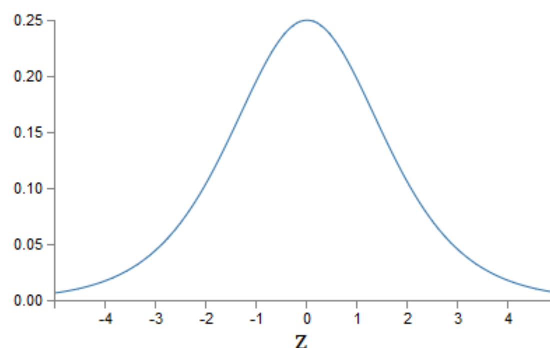

Abbildung 6: Ableitung der Sigmoid-Funktion: [Nie18]

Nun wird zunächst die stark vereinfachte Gleichung für den Gradienten vom Neuron mit dem Biaswert b_4 errechnet. Für die komplette Beschreibung der Backpropagation sei auf die Quelle [Nie18] Kapitel 2 verwiesen.

$$\frac{\partial C}{\partial b_4} = sig'(z_4) * \frac{\partial C}{\partial a_4} \quad (1)$$

Werden nun alle Werte für den Gradienten von Neuron b_1 einbezogen, resultiert das in der Formel 2.

$$\frac{\partial C}{\partial b_1} = sig'(z_1) * sig'(z_2) * sig'(z_3) * sig'(z_4) * \frac{\partial C}{\partial a_4} \quad (2)$$

Jeder der Faktoren kann, wie bei Formel 1, auch hier maximal einen Wert von 0.25 annehmen, also insgesamt $0.25^4 * \frac{\partial C}{\partial a_4}$.

Da nun mehr Faktoren für den Gradienten weiter vorne im Netz liegend einbezogen werden, wird der Gradient somit stetig kleiner.

Verallgemeinernd kann man also sagen: Bilden die Faktoren aus den hinteren Schichten auf ein Wert < 1.0 ab, so verringert sich der betrachtete Gradient exponentiell. Dies bezeichnet man als Vanishing Gradient Problem. Das Prinzip der Rechnung kann aus der Quelle [Nie18] Kapitel 5 entnommen werden.

Der Gradient gibt die Schrittweite und Richtung an, in die sich die Neuronenparameter anpassen. Aus einem kleinen Gradienten folgt also eine langsame Anpassungs- und damit Trainingsrate. Dies geschieht nicht homogen über das gesamte Netz, sondern verschlechtert sich nach vorne gehend immer weiter, sodass hintere Schichten des Netzes mehr und vordere Schichten weniger trainiert werden. Würde dies homogen über alle Schichten geschehen, so kann einfach die Learning-Rate auf einen hohen Faktor in Abhängigkeit der Tiefe des Netzes angepasst werden.

5.3 Exploding Gradient Problem

Da sich die Exploding Gradients analog zu den Vanishing Gradients erklären lassen, sollen sie hier nicht unerwähnt bleiben. Statt der Sigmoid-Funktion wird nun eine allgemeine Aktivierungsfunktion f(x) genutzt, dessen Ableitung auch über den Wert 1.0 abbildet.

Sollten nun bei folgender Funktion in der Gleichung von $\frac{\partial C}{\partial b_1}$ die Gradienten $f'(z_x)$ 1.0 übersteigen, so z.b. 1.25, verhält sich das Wachstum exponentiell und die Anpassung des Neurons übersteigt den Zielwert, potentiell sogar so stark, dass bei der nächsten Backpropagation der Gradient noch deutlich größer wird. Der umgekehrte Effekt des Vanishing Gradient. Dies bezeichnet man als Exploding Gradient Problem.

5.4 Aktivierungsfunktion

Bei Analyse der Backpropagation in Kapitel 5.2 mit der Sigmoid-Aktivierungsfunktion wurde festgestellt, dass die Ableitung Einfluss auf den Vanishing oder Exploding Gradient nimmt. Der naive Ansatz bedeutet in dem Fall die Aktivierungsfunktion auszutauschen.

Nun wird eine Aktivierungsfunktion betrachtet, dessen Steigungsverhalten und somit die Ableitung sich anders verhält.

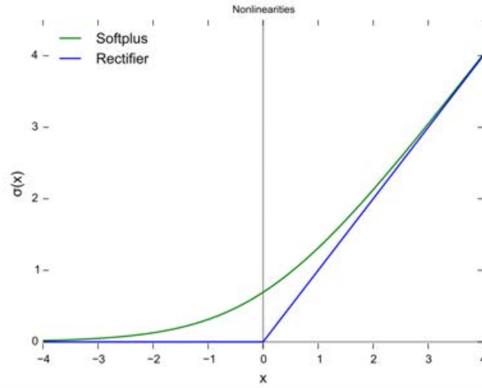

Abbildung 7: Die ReLU- (blau) und Softmax-Funktion (grün) als Graph: [Ali]

Zunächst die gängigste Aktivierungsfunktion neben Sigmoid, Rectified Linear Unit, kurz ReLU. Zu sehen in Abbildung 7. Im Gegensatz zu Sigmoid bildet die Ableitung von ReLU auf 1 ab, womit die Vanishing Gradients an der Stelle vermieden werden.

Neuronen	Schichten	Fehler	Klassifizierungsfehler	Validierung
30	1	0,18	0.00	100%
60	2	0,29	0,00	100%
90	3	0,74	0,14	100%
120	4	0,96	0,45	100%

Abbildung 8: Ergebnisse des Trainings mit der ReLU-Aktivierungsfunktion und verschiedener Anzahl an Schichten.

Aus Abbildung 8 kann man eine deutliche Verbesserung entnehmen, jedoch ist die Verschlechterung der Ergebnisse nach wie vor erkenntlich. Es bestehen also noch weitere Probleme mit zunehmender Tiefe von Netzen.

Eines davon ist die Ableitung der ReLU bei $x < 0$. Bildet der Gradient auf 0 ab, so werden, unabhängig von der Lernrate, keine Veränderungen am Neuron vorgenommen, analog zur Rechnung 2. Das Neuron kann also nicht mehr lernen. Es entstehen „tote Neuronen“. Dies kann zur Folge haben, dass das Netz effektiv schrumpft, wobei die „toten Neuronen“ trotzdem vorgehalten werden.

Um diesem Problem entgegen zu wirken, setzt man Derivate der Funktion ein, beispielsweise Leaky ReLUs [MHN13]. Die Funktion der ReLU verändert sich von $r_x = max(0, x)$ zu

$$f(x) = \begin{cases} x & \text{wenn } x > 0 \\ 0.01x & \text{sonst } x \end{cases} \tag{3}$$

Dies erlaubt bei negativen Werten einen kleinen Gradienten.

Eine weitere wichtige Aktivierungsfunktion in diesem Kontext stellt die ELU dar, Abbildung 9.

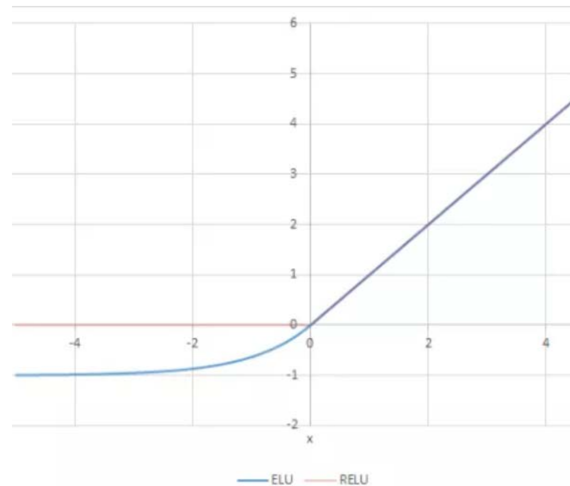

Abbildung 9: Graph der ELU-Funktion: [ELU]

Auch hier ist der Gradient grundsätzlich immer ungleich 0 beziehungsweise nähert sich 0 im Negativen an. Bei der Aktivierungsfunktion wird die Ausgangsverteilung eines Neurons indirekt, ähnlich zur Batch Normalization, normalisiert, dies wird jedoch weiter in Kapitel 6.2 ausgeführt. [CUH15]

Neuronen	Schichten	Fehler	Klassifizierungsfehler	Validierung
30	1	0,10	0.00	100%
60	2	0,10	0,00	100%
90	3	0,10	0,00	100%
120	4	0,09	0,00	100%

Abbildung 10: Ergebnisse des Trainings mit der ELU-Aktivierungsfunktion und verschiedener Anzahl an Schichten.

Die zuvor erkenntliche Verschlechterung bei der Auswertung 8 ist bei der Auswertung 10 nicht mehr sichtbar und mit steigender Tiefe sinkt der Fehler weiter. Auch im Rahmen des Papers [CUH15] wurden 2 tiefe Netze verglichen, bei denen jeweils die Aktivierungsfunktionen aus ReLUs und ELUs bestanden und besseres Konvergenzverhalten seitens der ELUs festgestellt wurde.

Es soll aber auch nicht unerwähnt bleiben, dass ReLU basierte Verfahren anfällig gegenüber den Exploding Gradients sind, da sie sich keinem Wert im positiven annähern, sondern gegen ∞^+ konvergieren. Die Gradienten sind neben den Ableitungen ihrer Aktivierungsfunktionen zusätzlich abhängig vom Kantengewicht. Sind die Kantengewichte demnach sehr hoch oder niedrig, so entsteht das Instabel Gradient Problem, bloß durch einen anderen Faktor.

6 Internal Covariant Shift

6.1 Entstehung

Zunächst wird von zwei gleichen Graubilder, A und B, ausgegangen. Während A intern für jeden Pixel mit einem Wert von [0.0, 1.0] repräsentiert wird, ist der Wertebereich von B [0.2, 1.4]. Trainieren man eine beliebige KI nur mit Bildern von Typ

A, kann diese natürlich Bilder des Typs B nicht auswerten, da sich die Verteilung ihres Wertebereichs unterscheidet und die Kantengewicht mit Biaswerten nur für den uhrsprünglichen Wertebereich optimiert wurden. Der Wertebereich kann dabei verschoben, [0.0, 1.0] -> [0.2, 1.2] oder skaliert, [0.0, 1.0] -> [0.0, 1.5] werden.

Die gleiche Situation wird nun auf die Hidden Layer während der Backpropagation bezogen, Abbildung 11.

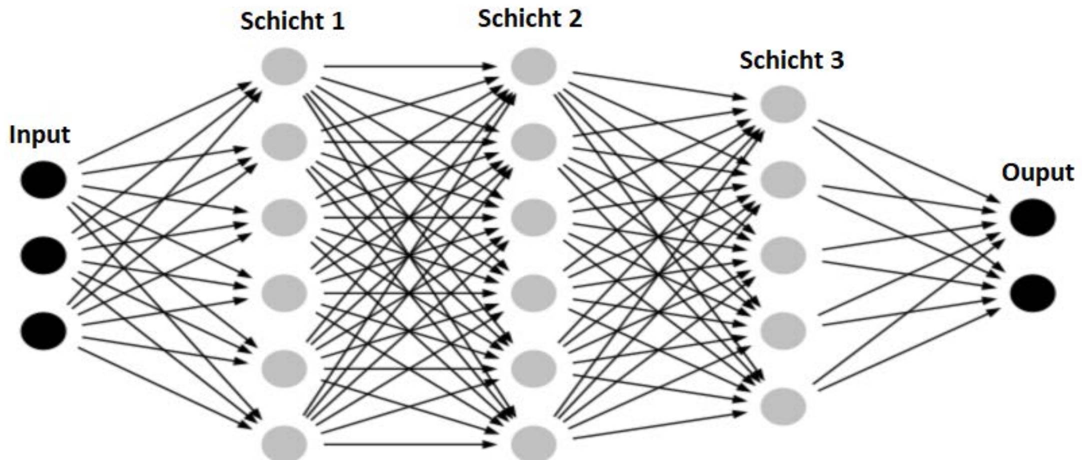

Abbildung 11: Darstellung eines kleineren tiefen neuronalen Netzes: [Sma]

Exemplarisch wird davon ausgegangen, dass Schicht eins nur auf Werte von [0.2, 0.8], Schicht zwei von [0.2, 0.8] auf [x, y] abbildet. Nun wird die Backpropagation für Schicht zwei ausgeführt. Der Definitionsbereich bleibt gleich, der Wertebereich verändert sich, um den errechneten Fehler über den Gradient zu reduzieren. Nun wird das Gleiche für Schicht eins angewendet. Durch Verändern des Biaswertes wird beispielsweise eine Verschiebung auf Y-Achse und über das Kantengewicht eine Skalierung auf der X-Achse des Wertebereichs angewendet. (Exemplarisch für die ReLU: $f(x) = (max(x, 0)) - b$, für $x = w_0 * ...w_n$, w_n relevante Kantengewichte, sowie b als Biaswert des Neurons). Die Abbildung verändert sich beispielsweise zu [0.0, 1.0] -> [0.3, 0.6]. Für Schicht zwei wäre es nun jedoch ideal gewesen die Verteilung des Werte- und Definitionsbereichs anzupassen. Da der Wertebereich von Schicht eins und der Definitionsbereich von Schicht drei nicht im Bezug auf Schicht zwei angepasst werden kann, ist die Verteilung von Schicht 2 zum reduzieren des Fehlers auf die momentanen Bereich beschränkt. Da tiefe neuronale Netze viele Schichten beinhalten, wird die potentiell entstehende Abweichung von den Idealverteilungen einzelner Schichten durch zunehmender Entfernung der Schichten potentiell stärker und nicht deterministisch. Im genannten Beispiel würden die Änderungen der Schicht eins $Zyklus_t$ betrachtet, die änderungen an Schicht zwei noch auf den Eingangsneuronen von $Zyklus_{t-1}$ basieren und bei Schicht drei die Eingangsneuronen von $Zyklus_{t-2}$ bewertet werden, wobei t der diskrete Zeitindex des Trainingszyklus, genannt Epoche, ist. Anders gesagt steigt damit die Komplexität und der Anpassungsbedarf zum Anstreben der Idealverteilung für eine Schicht in Abhängigkeit der Auswirkungen aller vorhergehenden und nachgehenden inhomogenen Verteilungen der Schichten. Die Auswirkung der Veränderung der Verteilungen zwischen Schichten bezeichnet man als Internal Covariant Shift.

Dies führt zu erschwertem Training, damit deutlich erhöhte Trainingsdauer und größeren Mengen an notwendigen Trainingsdaten. Das Ideal wäre aber das Training unabhängig der Verteilungen durchführen zu können.

6.2 Batch Normalization

Um diesem Problem entgegenzuwirken, normalisiert man die Eingabe- oder Ausgabeverteilungen jeder Schicht eines Netzes, damit die Auswirkungen der Verteilungen zwischen Schichten unabhängig werden. Die einzelnen Schichten können demnach autonom im Bezug auf die Verteilungen trainiert werden.

Zunächst betrachtet für die Eingabeschicht. An diesem Punkt können die Daten noch manuell aufgearbeitet werden. Exemplarisch sollen Datensätze bestehen aus Preis und Alter von Häusern verarbeitet werden. Der resultierende Graph der Verteilungen besitzt einen stark abweichenden Definitions- und Wertebereich, Abbildung 12 links.

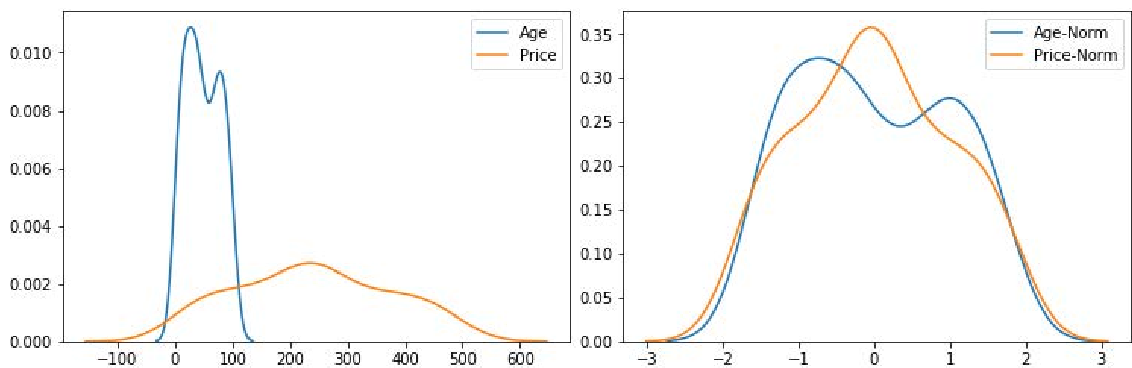

Abbildung 12: Roh- und normalisierte Daten: [Dis]

Wird durch lineare Interpolation der Wertebereich auf [0.0, 0.35], sowie der Definitionsbereich per Histogramm-Transformation auf [-3.0, 3.0] angeglichen, werden die stark unterschiedlichen Datensätze deutlich homogener, dadurch können die Gewichte und Biaswerte durch ähnliche große Veränderungsschritte innerhalb einer Schicht beim Lernen trainiert werden, Abbildung 12 rechts. Die Gradienten werden also unter anderem auch stabiler hinsichtlich des Exploding-/ Vanishing Gradient Poblem. Es wird also eine Invarianz gegenüber der Werte- beziehungsweise Definitionsbereichsverschiebung und -zerrung gewonnen.

Im Gegensatz zur Eingabeschicht, sind die Hidden Layer nicht mehr transparent genug für solche manuellen Anpassungen. Für die Hidden Layer wird also eine allgemeinere Form der Normalisierung notwendig. Die nötige Verteilungsanpassung ist in Abbildung 13 zusehen.

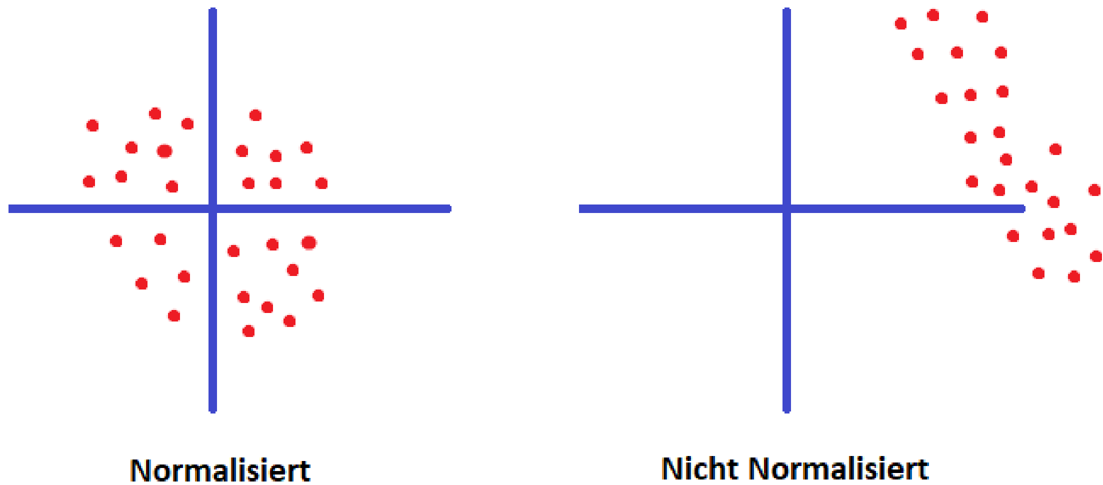

Abbildung 13: Darstellung der Veränderung der Verteilung von rechts, nicht normalisiert zu links, normalisiert

Die genutzte Funktionen sind erkenntlich in Abbildung 6.2.

x$_i$		zu normalisierender Wert des i-ten Batch-Teils
m		Anzahl der Trainingsdaten pro Batch
ϵ		geringfügiges Rauschen für numerische Stabilität
μ	$\frac{1}{m}\sum_{i=1}^{m}(x_i)$	Durchschnitt von x_m
σ^2	$\frac{1}{m}\sum_{i=1}^{m}(x_i-\mu)$	Varianz
$\hat{x_i}$	$\frac{x_i-\mu}{\sqrt{\sigma^2+\epsilon}}$	normalisierter Wert von x_i

Im Grunde wird dort nur mit Hilfe des Durchschnitts und Varianz über einen Batch ein Wert x auf x_{norm} abgebildet, jedoch fehlt dort noch das Wissen über Batches beziehungsweise Minibatches. Während einer Epoche trainiert man meist nicht nur mit einem Tupel aus Eingangs- und erwarteten Werten, sondern mit mehreren. Die Gradienten zur Optimierung streben dabei für alle Trainingsdaten des Batches ein Minimum an, somit wird die resultierende Veränderung des Netzes auch allgemeingültiger auf die gesamte Menge aller Daten bezogen. Im Kontext der Regularisierung wird der Nutzen weiter erklärt. Die zuvor erwähnte Werteverteilung eines Neurons wird also über einen Batch gebildet.

Der Effekt der Normalisierung ist einfach zu beschreiben: Die Verteilung des Wertebereichs jedes einzelnen Neurons einer Schicht wird um den Koordinatenursprung gleichmäßig verteilt, idealer Weise angewandt auf alle Schichten des Netzes. Damit wird erzwungen, dass die resultierende Werte über einen Batch pro Neuron einer normalverteilt sind. So wird Autonomie der Verteilungen einzelner Schicht gegenüber den potentiell unterschiedlichen Verteilungen anderer Schichten während des Trainings ermöglicht.

Betrachtet man jedoch nochmal die ReLU-Aktivierungsfunktion, so wird einem auffallen, dass alle Werte < 0 auf 0 abgebildet werden. Damit werden effektiv ein Teil aller Wert ihres Einflusses beraubt und es entstehen latent tote Neuronen . Die

Normalisierung muss also in Anbetracht der Aktivierungsfunktion weiter angepasst werden.

Formelglossar

x	nicht normalisierter Wert des Neurons
BN(x)	Normalisierungsfunktion
z	normalisierter Wert des Neurons
a	Skalierungsfaktor für die Ergebnisverteilung
b	Verschiebung für die Ergebnisverteilung
n	transformierter und normalisierter Wert des Neurons

$$n = az + b | z = BN(x) \tag{4}$$

Typischerweise werden zwei Parameter zum Anpassen der Normalisierung verwendet. Für die ReLU ist z.b. $b = 1.0$ eine Verschiebung der Verteilung in den Bereich, bei der die ReLU nicht mehr auf null abbildet. Ist die Verteilung [-0.5, 0.5] gegeben, so wäre b im Bezug auf die ReLU Aktivierungsfunktion als 0.5 zu wählen. a dient zum Stauchen beziehungsweise Strecken der Verteilung. Beide Parameter werden während des Trainings automatisch angepasst, um eine möglichst hohe Trainingsrate zu erzielen und den Wertebereich nicht nur auf eine Normalverteilung zu beschränken, müssen jedoch Initialisiert werden. Es erfolgt beim Training eines Neurons eine Adaption der Verteilung für die Folgeneuronen bzw. Schichten.

Zum Zeitpunkt der Auswertung, außerhalb des Trainings, werden die Parameter der Normalisierung konstant. Diese werden über den Durchschnitt der letzten verarbeiteten X Epochen gebildet werden, um möglichst allgemein gültig zu sein. Diese Art der beschriebenen Normalisierung für die Hidden Layers nennt man Batch Normalization.

6.3 Momentane Forschungsergebnisse

Aktuelle Forschungen zeigen, dass die Batch Normalization den Internal Covariant Shift nicht zwingend reduzieren muss. Dies ist jedoch die verbreitetste Ansicht, warum Batch Normalization bessere Ergebnisse erzielt und hat in diesem Rahmen dem Verständnis des Aufbaues und Anwendung der Batch Normalization bezüglich tiefen neuronalen Netzen gedient. Weitere Forschungen zeigen, dass die Fehlerfunktion und die Gradienten beim Training geglättet werden, bewertet mit der Lipschitz-Stetigkeit. Das letzte Beispiel stellt das „Length-Direction Decoupling“ dar. Anschaulich in der Abbildung 12 dargestellt, trägt die Richtung der Vektoren vom Ursprung zu den Punkten in der nicht normalisierten Form im Vergleich zur normalisierten Form weniger Informationen, denn die Richtungen der Vektoren der nicht normalisierten Werte unterscheidet sich weniger stark. [KDL+19]

7 Overfitting

Um dieses Problem zu verstehen, soll an die Motivation von neuronalen Netzen erinnert werden. Es soll eine Funktion approximiert werden und diese sollte allgemein gültig auch außerhalb das Trainingskontextes sein.

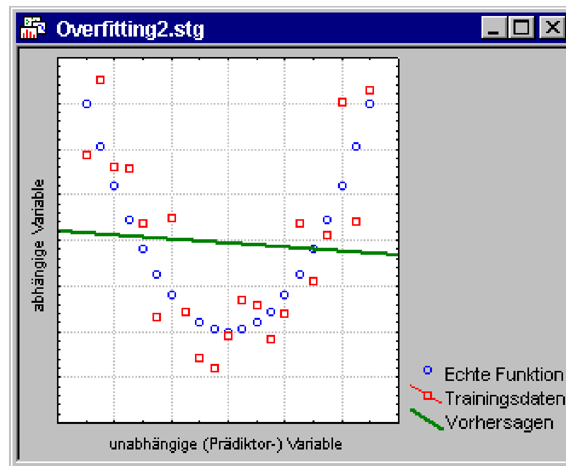

Abbildung 14: Untrainierte Vorhersagen. [Ovec]

Gegeben in Abbildung 14 sind drei Funktionen. Die Funktion (grün), die die KI (momentan) abbildet, die Ideal-Funktion (rot), die die Trainingsdaten abdeckt und die Ideal-Funktion (blau), die Lösung für die Realsituation. Nun wird die KI-Abbildung durch das Training immer mehr Richtung Ideal-Abbildung (rot) verändert und das Training wird erst gestoppt, bis der errechnete Fehler, der Unterschied zwischen den erwarteten und errechneten Ergebnissen, null beträgt, zusehen in der Abbildung 15.

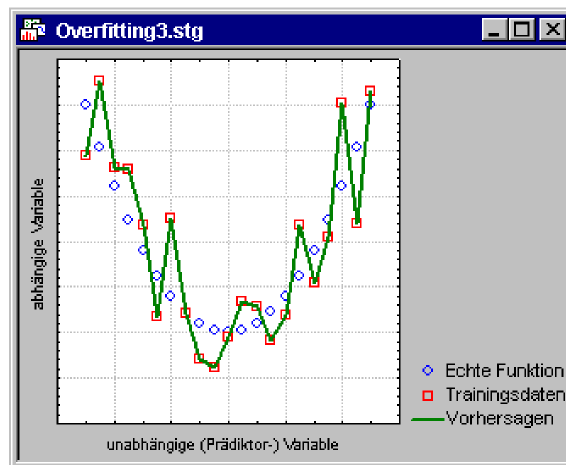

Abbildung 15: Beispiel des Overfittings. [Oveb]

Zwar ist der Fehler im Trainingsfall immer null und die KI-Abbildung befindet sich dichter an der Ideal-Funktion, hat sich aber schlussendlich den Trainingsdaten zu sehr genähert und sich damit wieder von der realen Situation entfernt. Die KI ist übertrainiert, genannt Overfitted.
Das Ideal, welches die KI jedoch erreichen soll, stellt diese Abbildung 16 dar.

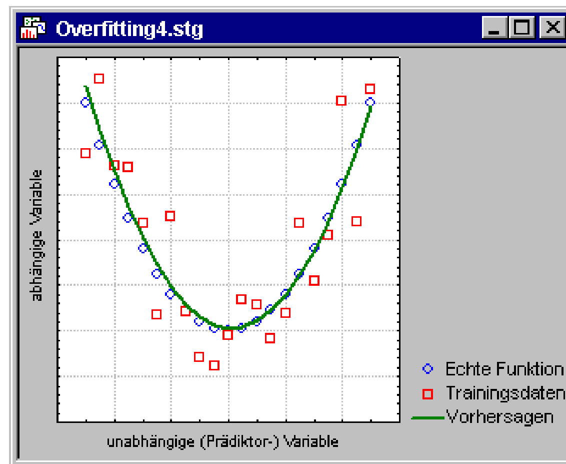

Abbildung 16: Idealfunktion der Vorhersagen bezüglich der Realsituation. [Ovea]

Um zu erreichen, sich den Testdaten ausreichend zu nähern, dabei aber nicht die Trainingsszenarien zu konkret abzubilden, werden Verfahren zur Regularisierung genutzt.

Dieses Problem ist allgemeinerer Natur. Es beschränkt sich nicht nur auf tiefe neuronale Netze. Es wurden aber gesonderte Methoden zur Regularisierung speziell für tiefere Netze entwickelt. Um diese soll es im folgenden hauptsächlich gehen, jedoch wird für das Verständnis die Grundtechnik benötigt. Auch wird häufig argumentiert, dass sich neuronale Netze mit steigender tiefe, komplexeren Problemen besser annähern können und deswegen anfällig gegenüber der Adaption des Hintergrundrauschens, neben der approximierten Idealfunktion bezüglich der realen Anwendung, sind. Eine genauere Ausführung kann man im Rahmen des Artikels [Wen] finden. Beim Dropout-Verfahren Kapitel 7.3 wird dies genauer ausgeführt.

7.1 Regularisierung

Unter Regularisierung versteht man im allgemeinen eine Klasse von Methoden zur Prävention des Overfittings.

7.2 Validation

Eine zentrale Methode, um unter anderem das Overfitting einer trainierten KI einschätzen zu können, ist die Validation.

Abbildung 17: Aufteilung der Trainingsdaten. [Tra]

Die Testdaten werden in Trainings- und Validation-Satz geteilt, wie in Abbildung 17 zusehen, wobei die Testszenarien in dem Validation-Satz nicht im Trainingssatz genutzt werden sollten, da sonst die Aussagekraft der Validation im Bezug auf das

Overfitting verloren geht. Durch das Auswerten der Validation-Daten nach dem Training, wird geprüft, ob die KI das genannte Problem korrekt generalisieren konnte, oder sich zu sehr auf die Trainingsdaten eingestellt hat.

7.3 Dropout

Bei tiefen neuronalen Netzen besteht die Möglichkeit, dass sie das sogenannte Hintergrundrauschen adaptieren, analog zur „roten“ Funktion aus Kapitel 6.1. Hintergrundrauschen beschreibt also die Funktion, die durch den Abstand der beiden Idealfunktionen bezüglich der Real- und Traningssituation definiert wird. Tiefe neuronale Netzwerke stellen eine multiple nicht-lineare Abbildung dar, die sehr komplexe Zusammenhänge darstellen kann, so auch das besagte Hintergrundrauschen. [SHK+14] Neben der möglichen Komplexität der Abbildung ist die Überparametrisierung der Modelle eine weitere Erklärung zur Affinität des Overfittings von tiefen neuronalen Netzen. [Wen]

Auf Basis dieser Erklärungen sind zur Prävention des Overfitting zwei Faktoren zu optimieren. Die Komplexität der möglichen Abbildung erklärt durch die sogenannte Co-Dependeny , bei der Neuronen gegenseitige Abhängigkeiten ausprägen. Im Bezug auf die Überparametrisierung muss die Netzgröße angepasst werden.

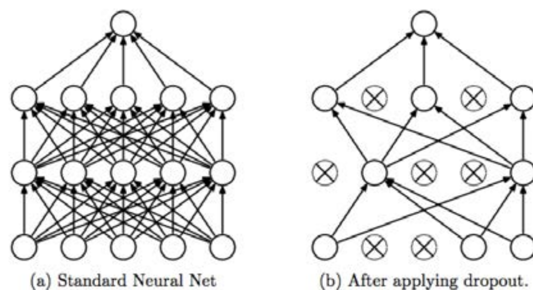

Abbildung 18: Beispiel der Anwendung des Dropouts auf ein neuronales Netz. [SHK+14]

Um die Autonomie eines Neurons zu erzwingen, werden Teile des Netzwerks mit ihren Verbindungen für die einzelnen Trainingsintervalle einer Epoche zufällig aus bezogen, zu sehen in Abbildung 18. Richtwert dafür ist etwa 20 - 50 Prozent, die Größe variiert jedoch von Fall zu Fall, darauf wird aber beim Jumpout-Verfahren in Kapitel 7.4 genauer eingegangen. Dieses Verfahren nennt man Dropout. Neben der Reduzierung der Co-Dependeny wird das Modell zuzüglich beim Training durch das Ausbeziehen von Neuronen verkleinert.

Ein visuell eingängiges Beispiel für das Minimieren des Hintergrundrauschens stellt Abbildung 19 dar.

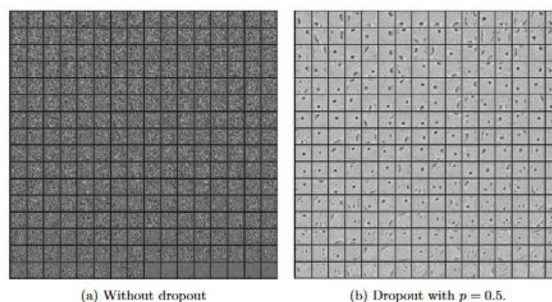

Abbildung 19: Darstellung der gelernten Features eines Klassifizierers mit und ohne Dropout. [SHK+14]

7.4 Jumpout

Bei Untersuchung des Verfahrens im Kontext von tiefen neuronalen Netzen mit ReLU-Aktivierungsfunktion verbessern sich die Resultate der Validation. Es wurden jedoch Verbesserungen zum Verfahren vorgeschlagen.

Die Dropout-Rate für eine Schicht wird zufällig von einer homogen sinkenden Verteilung gewählt, anstatt sie als Konstante zu behandeln. Das damit zu behandelnde Problem ist sehr aufwändig zu Erklären, aber es sei auf die Erklärung an der Stelle verwiesen: [WZB18], Kapitel 3.1 Modification I: Monotone Dropoutrate for local Smoothness.

Während des Trainings entstehen Neuronen die bereits „tot“ sind, also keinen Einfluss mehr auf den Gradienten haben können, da gilt $ReLU'(x) = 0 | x < 0$. Der Einfluss der Ableitung der Aktivierungsfunktion wurde im Kapitel 5.2 bereits geklärt. Werden diese Neuronen durch das Dropout aus bezogen, hat dies keinen Einfluss auf den Gradienten. Damit wird die Dropout-Rate indirekt reduziert. Diese ruhenden Neuronen werden nun nicht mehr berücksichtigt. Außerdem wird die Dropout-Rate wird im Verhältnis des Einflusses der Neuronen einer Schicht auf den Gradienten angepasst. Dies kann man sich wie eine Normalisierung der Dropout-Rate für Schichten über jede Epoche vorstellen. Damit stellt die Dropout-Rate für jede Schicht den gleichen Einfluss bereit.

Möchte man Batch Normalization mit Dropout verwenden, um die Vorteile beider Methoden zu kombinieren, fluktuiert durch den Ausschluss einiger Neuronen die errechnete Varianz zufällig. Durch Reskalieren der Neuronen Ausgabeverteilung auf die vor dem Anwenden des Dropouts errechnete Varianz, wird die Varianz und damit die Normalisierung konsistent gehalten, um bessere Kompatibilität zwischen den beiden Methoden herzustellen.

Das so entstehende Verfahren, basierend auf Dropout, wird Jumpout genannt. Beim Test im Direktvergleich zu Dropout erzielt das Jumpout-Verfahren bessere Ergebnisse, Abbildung 20.

Dataset	CIFAR10(s)	CIFAR10	CIFAR100	Fashion	STL10	SVHN	ImageNet
Dropout	86.50	95.23	79.41	96.09	81.37	98.22	71.15
Jumpout	90.24	96.82	82.48	97.17	84.02	98.51	71.48

Abbildung 20: Vergleich zwischen der Vorhersagegenauigkeit in Prozent von Dropout und Jumpout. [WZB18]

Es soll aber nicht unerwähnt bleiben, dass durch das Dropout-/Jumpout-Verfahren mehr Epochen für das Training nötig werden, da Teile des Netzes nicht bei jeder Epoche mit trainiert werden. Als Faustregel sagt man: Die Anzahl der nötigen Epochen verdoppelt sich.

7.5 Batch Normalization

Auch die Batch Normalization soll die Eigenschaft haben zu regularisieren. [IS15] (Kapitel 3.4)

8 Degradation of training accuracy

8.1 Entstehung

> Deeper neural networks are more difficult to train. [HZRS15]

Wie bereits bei dem Vanishing Gradient Problem erklärt, kann sich das Ergebnis durch das Hinzufügen von weiteren Schichten verschlechtern, obwohl intuitiv das Gegenteil eintreten sollte. Tatsächlich wurde das Problem noch nicht komplett adressiert, denn selbst mit ReLUs und Batch Normalization wurde eine Verschlechterung, genannt degradation of training accuracy, festgestellt, Abbildung 21.

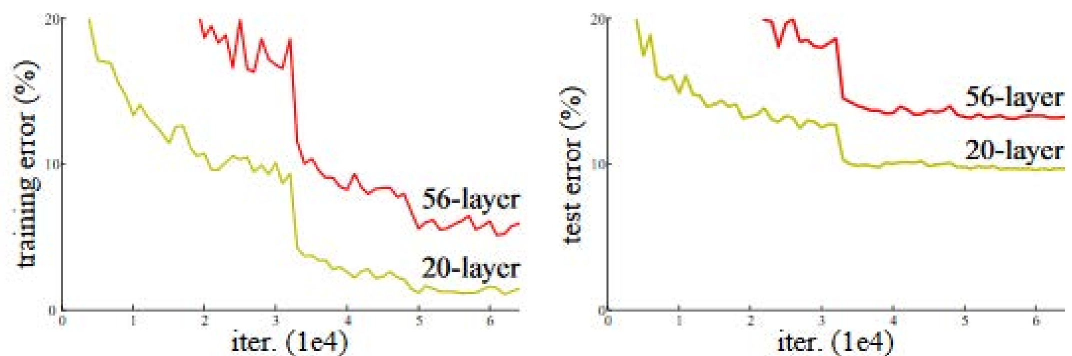

Abbildung 21: Folgen von tieferen Netzen trotz Normalisierung. [HZRS15]

Das Potential der Ausprägung noch generalisierterer Features nimmt jedoch mit der Tiefe zu;- besonders bei der Bild-Klassifizierung haben sich "Deep Convoltutional Neural Networks" als zuverlässig erwiesen, waren jedoch limitiert in ihrer Tiefe.

Um zu erahnen, warum dieses Phänomen auftritt, wird folgende Situation konstruiert. Ein flacheres, teilweise trainiertes Netz wird als Ausgangspunkt verwendet. Nun wird an dieses Netz Schichten angefügt, die eine identitäre Abbildung darstellen. Nun werden beide Netze weiter trainiert. Beide sollten im folgenden während des Trainings ähnliche Fehler produzieren, jedoch findet die tiefere Variante keine passende Lösung.

8.2 Residual Node

Um dem entgegen zu wirken wurden nun neben den regulären Verbindungen zwischen zwei Schichten auf eine Ausgabe eine weitere Verbindung von der ersten

Schicht additiv direkt auf die Ausgabe der zweiten Schicht erstellt, in Abbildung 22 zusehen.

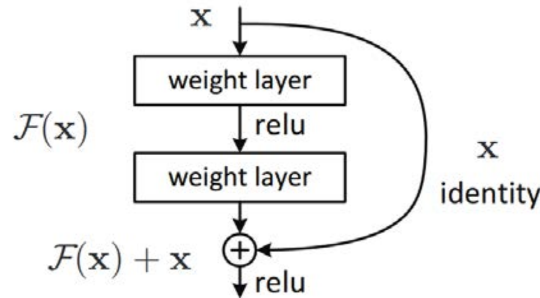

Abbildung 22: Beispiel des Aufbaus eines Residual Nodes. [HZRS15]

Der Gedanke hinter diesem Aufbau ist, dass es einfacher ist $F(x) = 0$ zu ermittelt, damit nur, beziehungsweise auch, die identitäre Abbildung weiterverarbeitet wird, anstatt die Abbildung von $F(x)$ zu erwarten. Somit kann, sobald es optimaler ist, auch eine identitäre Abbildung verwendet werden, sprich eine Schicht kann übersprungen werden. Dies stellt ein Residual Node dar. Ein Netzwerk aus diesen Bausteinen nennt man Residual Network. (Residual kann man in dem Kontext als verschleppen oder zurückbleibend verstehen.) In Residual Nodes kann man die additive Abbildung auch erst nach zwei beziehungsweise mehreren aufeinander folgende Schichten einfügen. Damit die Abbildung jedoch wirklich identitär ist, muss die Addition erst nach den Gewichten der Folgeschicht erfolgen.

Werden nun zwei Netze konstruiert, das eine ohne Residual Nodes (,genannt plain-XX), das andere mit Residual Nodes (, genannt ResNet-XX), wobei XX für die Anzahl an entsprechenden Schichten steht, kann man folgendes Verhalten, zusehen in Abbildung 23,beim Training beobachten.

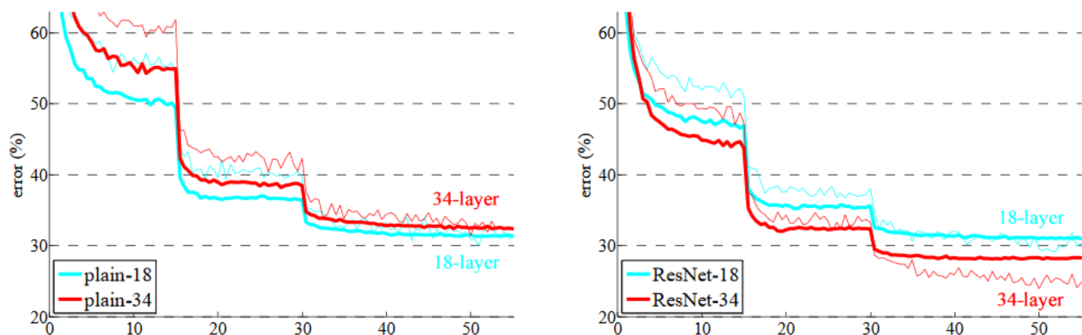

Abbildung 23: Direktvergleich der Konvergenz bei einem Netz mit und einem ohne Residual Nodes: [HZRS15]

Mit steigender Anzahl an Schichten, sinkt der erzeugte Fehler des Trainings bei den Residual Networks weiter. (Abbildung 23) Es tritt also weniger Degradation auf.

9 TI-Pooling

Um dieser Methodik näher zu kommen, kann man das Beispiel der Zahlenerkennung erneut verwenden. Das Ausgangsproblem bleibt gleich, es sollen Zahlen erkannt werden, diese können jedoch nun in jeder Form von Skalierung und Rotation, also geometrisch transformiert, vorliegen. Als Mensch kann man beispielsweise eine fünf rotiert, gestreckt oder gezerrt erkennen, bei KIs hingegen ist dies nicht selbstverständlich und abhängig von den Eingangsdaten.

Methoden wie RIFT und SWIFT erlauben als vorverarbeitenden Schritt tranformationinvariante Merkmale eines Bildes zu extrahieren, nimmt der KI jedoch die größte Stärke: die Fähigkeit der aufgabengebundenen Adaption an das Problem. Das neuronale Netz soll sich idealerweise die Tranformationsinvarianz selbst herleiten können.

Zwecks dessen, wurden mehrere Verfahren zum sogenannten TI-Pooling kombiniert.

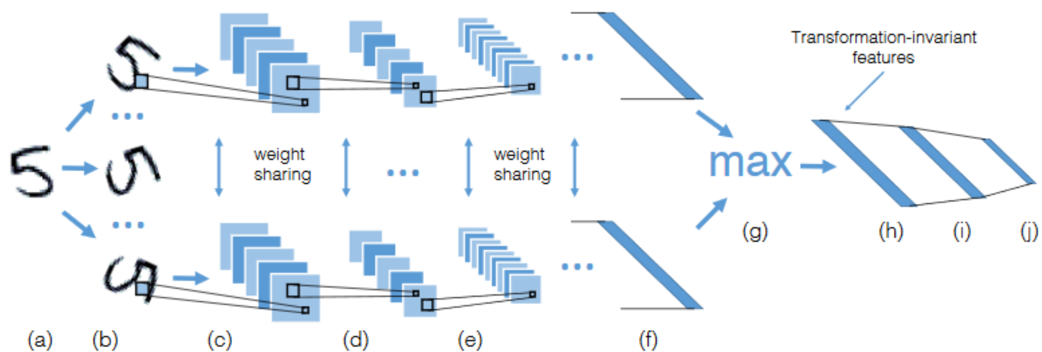

Figure 1. Network topology and pipeline description. First, input image x (a) is transformed according to the considered set of transformations Φ to obtain a set of new image instances $\phi(x), \phi \in \Phi$ (b). For every transformed image, a parallel instance of partial siamese network is initialized, consisting only of convolutional and subsampling layers (two copies are shown in the top and in the bottom of the figure). Every instance is then passed through a sequence of convolutional (c, e) and subsampling layers (d), until the vector of scalars is not achieved (e). This vector of scalars is composed of image features $f_k(\phi(x))$ learned by the network. Then TI-POOLING (element-wise maximum) (g) is applied on the feature vectors to obtain a vector of transformation-invariant features $g_k(x)$ (h). This vector then serves as an input to a fully-connected layer (i), possibly with dropout, and further propagates to the network output (j). Because of the weight-sharing between parallel siamese layers, the actual model requires the same amount of memory as just one convolutional neural network. TI-POOLING ensures that the actual training of each features parameters is performed on the most representative instance $\phi(x)$.

Abbildung 24: Abbildung aus dem uhrsprünglichen Paper [LSBP16]. Die gesamte Netztopologie wird zudem auf Englisch erklärt mit knapper Beschreibung des TI-Poolings.

Inspiriert am MIL (Multi Instance Learning) wird ein Eingangswert, ein Bild der fünf (Abbildung 9 Schicht a) beispielsweise, in allen, oder zumindest ausreichend fein gegliederten, Transformationen (Abbildung 9 Schicht b), bei denen Invarianz ausgeprägt werden soll, parallel in ein Netz parametriert, (also in Abbildung 9 Schicht c zugewiesen), wobei pro einer spezifischen Transformation, beispielsweise eine Rotation um fünf Grad, ein bestimmter Eingang des Netzwerks genutzt wird. Wird das Netz mit Fünfen, Rotiert um null Grad, fünf grad und minus fünf grad, trainiert, so wird an einen spezifischen Eingang immer das Bild mit einer Rotation von fünf grad angelegt, sodass sich für jede Art der Transformation, wie anfangs gesagt, ein Eingang ergibt. Bei der Auswertung werden die Eingangsdaten an jeden Eingang gleich

angelegt. Jeder Eingang besitzt, zunächst autonom, Folgeschichten, die für eine Art der Transformation entsprechende Merkmale erkennen können. Der Einfachheit halber wird eine Folge an Schichten für einen Eingang bis zu Schicht f Abfolge genannt, wie in Abbildung 25 zusehen.

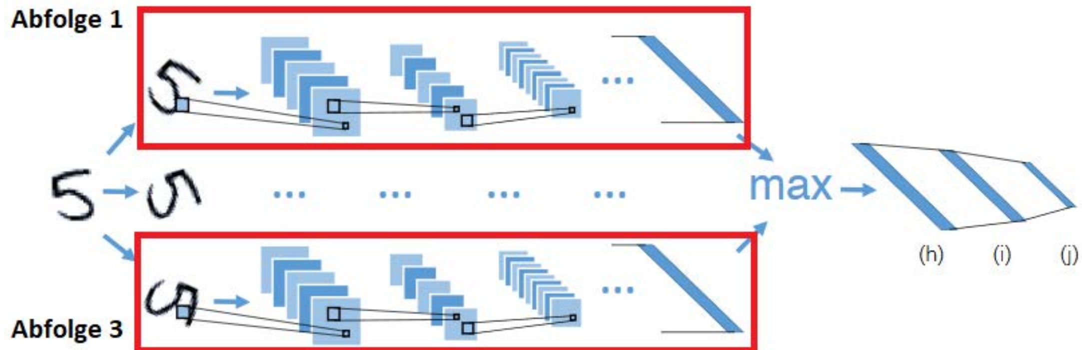

Abbildung 25: Darstellung zweier Abfolgen in rot markiert.

Diese Merkmale werden am Ende der Abfolgen, für jedes Merkmal an gleicher Position der Ausgangswerte jeder Abfolge, zu einem Vektor zusammengefasst und über den Vektor wird eine Max-Pooling-Funktion, als das Maximum $(x_1, ..., x_N)$, wobei N die Anzahl der Schichten darstellt, gebildet, sodass das zutreffendste Merkmal, also das Merkmal mit dem höchsten Wert, aus allen Abfolgen übernommen wird, wie in Abbildung 26 zusehen ist. Daraus folgt, dass jede Abfolge nur mit einer Art von Rotation trainiert wird und umgehen kann.

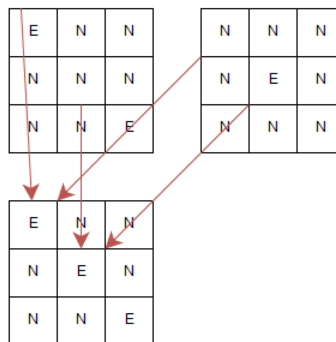

Abbildung 26: Darstellung des genutzten Poolings beim TI-Pooling über 2 Output-Matrizen zu einer Output-Matrix.

Die erkannten Merkmale können nun nach dem Max-Pooling als tranformationsinvariant betrachtet und weiter verarbeitet werden.

Für jede Tranformationsart ein eigenes Teilnetz aufzubauen ist für Training und Auswertung natürlich ein gewaltiger Datenaufwand.

Nun kommt ein Zusatz hinzu: Anfangs wurde angenommen, die Abfolgen sind autonom. Dies ist aber nicht der Fall. Jede Abfolge nutzt das sogenannte Weight-Sharing, mit dem sich Teil-Siamesische Netzwerke modellieren lassen, sodass die Kantengewichte über jede Schicht der Abfolgen geteilt werden. Jedes Kantengewicht

einer Abfolge, wird also für jede andere Abfolge auch genutzt. Veränderungen der Gewichte wirken sich also auf alle Abfolgen gleichermaßen aus. Dadurch benötigen alle Abfolgen insgesamt etwa die gleiche Datenmenge, wie nur eine Abfolge. [LSBP16]

Das beschriebene Vorgehen nennt sich Transformation Invariant - Pooling oder kurz TI-Pooling.

10 Self-Normalizing Networks

Orientiert an der Batch Normalization wurde die Idee der Normalisierung als autonome Methode, die erst integriert werden muss, erweitert. Ein neuronales Netz soll nun immer normalisiert sein.

Auf Basis der bereits erwähnte ELU wurde eine neue Art der Aktivierungsfunktion eingeführt, die sogenannte scaled exponential linear units, kurz SELU. Ihre Kerneigenschaft ist es, dass eine Funktion $g(x)$ existiert, die die Verteilung der Ausgaben der Aktivierungsfunktion mit dem Durchschnitt 0 und der Varianz 1 normalverteilt, in Abbildung 27 dargestellt.

$$\begin{pmatrix}\mu\\ \nu\end{pmatrix} \mapsto \begin{pmatrix}\tilde{\mu}\\ \tilde{\nu}\end{pmatrix} : \quad \begin{pmatrix}\tilde{\mu}\\ \tilde{\nu}\end{pmatrix} = g\begin{pmatrix}\mu\\ \nu\end{pmatrix}$$

Abbildung 27: 1. Theorem der SELUs. [KUMH17]

Alle Theoreme im Rahmen von SELUs die hier erwähnt werden, wurden mathematisch im Paper [KUMH17] bewiesen. Somit ist die SELU-Aktivierungsfunktion selbstnormalisierend. Verarbeitet die SELU bereits normalisierte Features, so sind die ausgehenden Features auch stets normalisiert.

Die Vorteile der Normalisierung wurden bereits im Rahmen der Batch Normalization Kapitel 6.2 geklärt. Features werden durch angleichen der Ausgabeverteilungen von Neuronen vergleichbar. Die Gradienten werden stabiler, demnach sind die Gradienten durch eine nicht ideale Learning-Rate weniger von Exploding Gradients betroffen.

$$SELU(x) = \delta \begin{cases} x & \text{wenn } x > 0 \\ \alpha e^x - \alpha & \text{sonst } x \end{cases} \tag{5}$$

Auch hier sind wieder zwei Parameter, analog zur Batch Normalization in Kapitel 6.2, α und δ, vorhanden, die die Verteilung manipulieren können.

Es wird wieder Dropout zur Regularisierung angewendet, da die Normalisierung nun ein latenter Bestandteil des Netzes ist und nicht mehr zur Regularisierung genutzt werden kann, jedoch in veränderter Form, dem sogenannten α-Dropout, bei dem zufällig Kantengewichte auf α, anstatt 0, gesetzt werden. Außerdem wird auch hier wieder die Varianz durch eine affine Tranformation der vor dem Dropout angeglichen. Dadurch umgeht man Probleme, bereits genannt beim Jumpout-Verfahren in Kapitel 7.4, bezüglich der Varianz und dem Durchschnitt für die Normalisierung.

Weiterer Vorteil ist, dass Exploding- und Vanishing Gradients hier nicht auftreten können, definiert durch das 2. und 3. Theorem der SELUs, vorausgesetzt die Initialisierung des Netzes und die Learning-Rate sind sinnvoll gewählt. Für die Initialisierung wurde im Paper vorgeschlagen das Netz direkt normalisiert zu initialisieren.

Vielversprechende Ergebnisse liefert ein Direktvergleich zur Batch Normalization, in Abbildung 28 dargestellt.

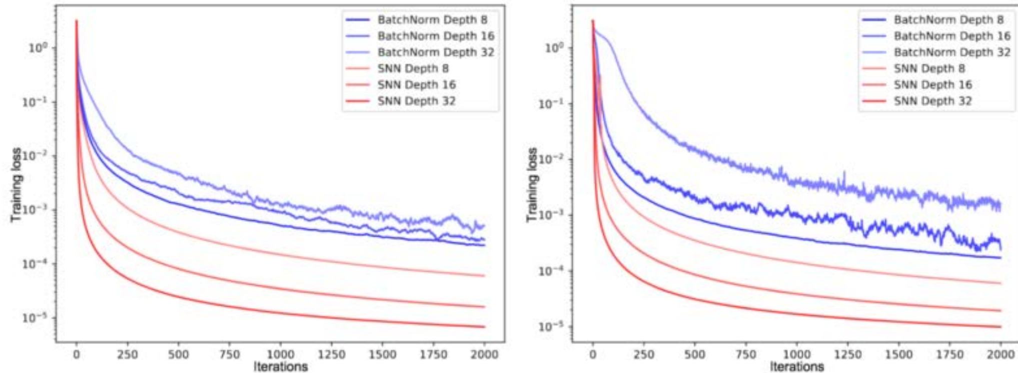

Figure 1: The left panel and the right panel show the training error (y-axis) for feed-forward neural networks (FNNs) with batch normalization (BatchNorm) and self-normalizing networks (SNN) across update steps (x-axis) on the MNIST dataset the CIFAR10 dataset, respectively. We tested networks with 8, 16, and 32 layers and learning rate 1e-5. FNNs with batch normalization exhibit high variance due to perturbations. In contrast, SNNs do not suffer from high variance as they are more robust to perturbations and learn faster.

Abbildung 28: Vergleich verschiedener Modelle mit Batch Normalization und Self-Normalization: [KUMH17]

Die Varianten mit Self-Normalization übertreffen ihre Batch Normalization-Pendants und skalieren besser mit der Tiefe der Netze, aber interessanter sind die Auswirkung auf den Kurvenverlauf. Der Fehler fluktuiert nicht, wie in Abbildung 28 zu sehen ist, sonder entspricht einem weichen, stetigen Abstieg des Fehlers. [KUMH17]

Neuronen	Schichten	Fehler	Klassifizierungsfehler	Validierung
30	1	0,08	0.00	100%
60	2	0,07	0,00	100%
90	3	0,07	0,00	100%
120	4	0,05	0,00	100%

Abbildung 29: Ergebnisse des Trainings mit der SELU-Aktivierungsfunktion und verschiedener Anzahl an Schichten.

Die Ergebnisse in der Auswertung 29 des Eigenversuchs sind erwartend weiter verbessert worden.

11 Versuchsdaten

Angelehnt an den Versuch von Michael Nielsen ([Nie18]. Kaptitel 5) wurde auf Basis des MNIST-Datensatzes der Versuch minimalistisch nachgestellt. Es sollen Bilder der

Größe 28x28 mit handgeschriebenen Zahlen von null bis neun erkannt werden. Es stehen 1000 Trainingstupel dem Training und 100 der Validierung zur Verfügung. Ein Trainingstupel besteht aus einem Bild mit zugehörigem Erwartungswert.

Der Aufbauf des Versuchsnetzes besteht aus sogenannten Fully Connected- oder Dense-Layern. Jede Schicht beinhaltet 30 Neuronen und eine Aktivierungsfunktion. Die Ausgabeschicht bildet auf 10 Klassen ab, die die Zahlen von null bis neun repräsentieren.

Es wird zum Training Stochastic Gradient Descent genutzt. Da es sich um ein Klassifizierungsproblem handelt kann als Loss-Funktion der Cross Entropy Loss mit dem Classification Error als begleitenden Optimierungswert genutzt werden.

Die Learning-Rate beträgt 0.0003125, die Größe der Minibatches 64 über 20 Epochen trainiert. Minibatches werden im Zuge der Batch Normalization in Kapitel 6.2 erklärt. Die Auszüge der beiden Fehler beschreiben den Zustand nach Minibatch 120.

Der Loss beziehungsweise Fehler beschreibt wie dicht die Asuwertung am erwarteten Ergebnis liegt. Die Klassifizierungsfehler beschreibt, wie groß der Anteil korrekter Vorhersagen im Verhältnis zu allen Vorhersagen über ein Minibatch ist. Hat die KI also knapp alle Ergebnisse richtig ausgewertet ist der Klassifizierungsfehler null, der Fehler kann jedoch größer als null sein. Der Idealwert beider Werte entspricht 0. Bei der Klassifizierung bedeutet 1.0, dass alle vorhersagen inkorrekt sind.

Die Validierung prüft wie viele der 100 Trainingstupel korrekt ausgewertet wurden.

Als Basis für die Tests wurde das CNTK Framework von Microsoft mit der C#-Schnittstelle genutzt. Die Tests aus dem offiziellen Repository wurden für die Versuche angepasst beziehungsweise durch eigene Modelle ersetzt und erweitert.

12 Schlusswort

Es wurden viele Probleme von tiefen neuronalen Netzen mit ihren Auswirkungen erklärt und dazu ältere wie neuere Lösungsansätze erläutert. Erkenntnisse im Gebiet Machine-Learning sind jedoch sehr volatil. Entsprechend utopisch ist natürlich der Ansatz alle Möglichkeiten zusammenzufassen, insbesondere im zeitlichen Rahmen eines Seminars.

Mit neuen Fortschritten könnten angesprochene Techniken schnell veralten oder ersetzt werden und einige Aussagen, basierend auf aktuellen Ergebnissen, widerlegt werden. Es ist aber idealerweise ein Überblick und tieferes Verständnis für tiefen neuronalen Netzen und sich daraus ergebende Probleme und Methoden vermittelt worden.

Literatur

[Aa16] Alias:ducha-aiki. Batchnorm after relu. https://github.com/gcr/torch-residual-networks/issues/5, 2016. Zuletzt eingesehen am 9.5.2019. Klärt einen intressanten Ansatz zur Batch Normalization.

[Ali] Alias:Pccoronado. Plot of the rectifier (blue) and softplus (green) functions near x = 0. https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#/media/File:Rectifier_and_softplus_functions.svg. Zuletzt eingesehen am 9.5.2019.

[Ali17] Alias:amoeba. What is the difference between a neural network and a deep neural network, and why do the deep ones work better? https://stats.stackexchange.com/questions/182734/what-is-the-difference-between-a-neural-network-and-a-deep-neural-network-and-w, 2017. Zuletzt eingesehen am 9.5.2019.

[Ali18] Alias:DeepLizard. Vanishing & exploding gradient explained | a problem resulting from backpropagation (series). https://www.youtube.com/watch?v=qO_NLVjD6zE&list=PLZbbT5o_s2xq7LwI2y8_QtvuXZedL6tQU&index=32, 2018. Zuletzt eingesehen am 9.5.2019. Dies ist eine Video-Serie über DeepLearning im Allgemeinen, jedoch wurde die Backpropargation komplett erklärt und durchgerechnet.

[BF13] Jimmy Ba and Brendan Frey. Adaptive dropout for training deep neural networks. In *Advances in Neural Information Processing Systems*, pages 3084–3092, 2013. Dies kann man Online auch in Form eines Parpers finden.

[Che18] Chengwei. One simple trick to train keras model faster with batch normalization. https://www.dlology.com/blog/one-simple-trick-to-train-keras-model-faster-with-batch-normalization/, 2018. Zuletzt eingesehen am 9.5.2019.

[Civ] Damon Civin. An intro to self-normalising neural networks (snn). https://medium.com/@damoncivin/self-normalising-neural-networks-snn-2a972c1d421. Zuletzt eingesehen am 9.5.2019. Bezieht sich auf das Paper von Klambauer Unterthiner, Mayr und Hochreiter.

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[Dis] Roh- und normalisierte-daten. https://cdn-images-1.medium.com/max/873/1*4T4y3kIOR9A1k_2pe6B4Pg.png. Zuletzt eingesehen am 9.5.2019.

[Eco] Adrien Lucas Ecoffet. Paper repro: “self-normalizing neural networks”. https://becominghuman.ai/paper-repro-self-normalizin

g-neural-networks-84d7df676902. Zuletzt eingesehen am 9.5.2019. Bezieht sich auf das Paper von Klambauer Unterthiner, Mayr und Hochreiter.

[ELU] Elu-graph. https://ml-cheatsheet.readthedocs.io/en/latest/_images/elu.png. Zuletzt eingesehen am 29.5.2019.

[Gal08] Mark Gales. *The application of hidden Markov models in speech recognition*. Hanover, Mass. : Now Publishers, 2008. In dem Buch wurde nur die Architektur einer HMM-basierten Erkennung genutzt. Mehr als ein einführendes Beispiel wurde daraus nicht.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[Har15] Adam W. Harley. An interactive node-link visualization of convolutional neural networks. In *Advances in Visual Computing*, pages 867–877. Springer International Publishing, 2015.

[HG18] Mohamed Hajaj and Duncan Gillies. Batch normalization and the impact of batch structure on the behavior of deep convolution networks. *http://arxiv.org/abs/1802.07590v1*, 2018.

[Huy17] Denhanh Huynh. Applying dropout to prevent shallow neural networks from overtraining. *http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8899895&fileOId=8899896*, 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *http://arxiv.org/abs/1512.03385v1*, 2015.

[Ima] Teilbeispiel einer zahlenerkennung. http://scs.ryerson.ca/~aharley/vis/conv/flat.html. Wurde mit Hilfe des Tools angefertig. Tool zuletzt eingesehen am 29.5.2019.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *http://arxiv.org/abs/1502.03167v3*, 2015.

[KDL+19] Jonas Kohler, Hadi Daneshmand, Aurelien Lucchi, Thomas Hofmann, Ming Zhou, and Klaus Neymeyr. Exponential convergence rates for batch normalization: The power of length-direction decoupling in non-convex optimization. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 806–815, 2019.

[Kra16] Frederik Kratzert. Understanding the backward pass through batch normalization layer, 2016. Zuletzt eingesehen am 29.5.2019.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *http://arxiv.org/abs/1706.02515v5*, 2017.

[LCHY18] Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. Understanding the disharmony between dropout and batch normalization by variance shift. *arXiv preprint arXiv:1801.05134*, 2018.

[LFL+16] Yang Li, Chunxiao Fan, Yong Li, Qiong Wu, and Yue Ming. Improving deep neural network with multiple parametric exponential linear units. *http://arxiv.org/abs/1606.00305v3*, 2016.

[LS16] Shiyu Liang and R. Srikant. Why deep neural networks for function approximation? *http://arxiv.org/abs/1610.04161v2*, 2016.

[LSBP16] Dmitry Laptev, Nikolay Savinov, Joachim M Buhmann, and Marc Pollefeys. Ti-pooling: transformation-invariant pooling for feature learning in convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 289–297, 2016.

[MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[Nie] Michael A. Nielsen. 1 dimensionales tiefes neuronales netzwerk. `http://neuralnetworksanddeeplearning.com/images/tikz37.png`. Zuletzt eingesehen am 9.5.2019.

[Nie18] Michael A. Nielsen. Neural networks and deep learning, kapitel 5: Why are deep neural networks hard to train? `http://neuralnetworksanddeeplearning.com/chap5.html`, 2018. Kapitel 5, eingesehen zuletzt am 9.5.2019, dies ist ein Buch in Form einer Webseite. Kapitel 2 eklärt die Backpropagation detailiert.

[Ovea] Overfitted good trained. `https://www.statsoft.de/glossary/Images/OverfittingFigure4.gif`. Zuletzt eingesehen am 9.5.2019.

[Oveb] Overfitting overtrained. `https://www.statsoft.de/glossary/Images/OverfittingFigure2.gif`. Zuletzt eingesehen am 9.5.2019.

[Ovec] Overfitting untrained. `https://www.statsoft.de/glossary/Images/OverfittingFigure1.gif`. Zuletzt eingesehen am 9.5.2019.

[QXC17] Suo Qiu, Xiangmin Xu, and Bolun Cai. Frelu: Flexible rectified linear units for improving convolutional neural networks. *http://arxiv.org/abs/1706.08098v2*, 2017.

[RDGF15] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *http://arxiv.org/abs/1506.02640v5*, 2015. Nur für ein einführendes Beispiel genutzt.

[Rec19] Rectifier (neural networks). `https://en.wikipedia.org/wiki/Rectifier_(neural_networks)`, 2019. Zuletzt eingesehen am 9.5.2019. Bietet nur eine Übersicht der aktuellen ReLU-basierten Verfahren.

[RF16] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. *http://arxiv.org/abs/1612.08242v1*, 2016. Nur für ein einführendes Beispiel genutzt.

[Sat18] Vijay Sathish. How does the number of hidden layer affect the accuracy in deep learning? `https://www.quora.com/How-does-the-number-of-hidden-layer-affect-the-accuracy-in-deep-learning`, 2018. Zuletzt eingesehen am 9.5.2019. In Form eines Forenbeitrags.

[Sha] Gegenüberstellung eines flachen und tiefen neuronalen netzes. `https://i.stack.imgur.com/OH3gI.png`. Zuletzt eingesehen am 9.5.2019.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[Sin17] Vikas Sindhwani. shallow vs deep:the great watershed in learning. `http://www.princeton.edu/~amirali/Public/Teaching/ORF523/S17/ORF523_S17_Lec17_guest.pdf`, 2017. Zuletzt eingesehen am 29.5.2019.

[SKS+16] Anish Shah, Eashan Kadam, Hena Shah, Sameer Shinde, and Sandip Shingade. Deep residual networks with exponential linear unit. *http://arxiv.org/pdf/1604.04112v4*, 2016.

[Sma] Darstellung eines mehrschichtigen neuronalen netzes. `https://cdn-images-1.medium.com/max/873/1*-QyZahd2pGwQv2T_Wo11VA.png`. Zuletzt eingesehen am 9.5.2019. Die Grafik wurde leicht angepasst.

[Til16] Ottokar Tilk. What is the vanishing gradient problem? `https://www.quora.com/What-is-the-vanishing-gradient-problem`, 2016.

[TIP] Tipoolingtensorflow. `https://github.com/dlaptev/TI-pooling`. Tensorflow-Implementation und Zusammenfassung. Zuletzt eingesehen am 9.5.2019.

[Tra] Train-test-split-diagram. `https://elitedatascience.com/wp-content/uploads/2017/06/Train-Test-Split-Diagram.jpg`. Zuletzt eingesehen am 9.5.2019.

[Tre] Vergleich tiefer neuronaler netz-modelle der ilsvrc. `https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-`

vgg-googlenet-resnet-and-more-666091488df5. Zuletzt eingesehen am 27.5.2019.

[Unk17a] Unkown. Gradients, batch normalization and layer normalization. https://tomaxent.com/2017/05/09/GRADIENTS-BATCH-NORMALIZATION-AND-LAYER-NORMALIZATION/, 2017. Zuletzt eingesehen am 9.5.2019.

[Unk17b] Unkown. Overfitting in machine learning: What it is and how to prevent it. https://elitedatascience.com/overfitting-in-machine-learning, 2017. Zuletzt eingesehen am 9.5.2019.

[Wan19] Chi-Feng Wang. The vanishing gradient problem the problem, its causes, its significance, and its solutions. https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484, 2019. Zuletzt eingesehen am 9.5.2019. Prägnante Zusammenfassung des Vanishing Gradient Problems und Zusammenhang mit der Batch Normalization, sowie Residual Nodes.

[Wen] Lilian Weng. Are deep neural networks dramatically overfitted? https://lilianweng.github.io/lil-log/2019/03/14/are-deep-neural-networks-dramatically-overfitted.html#classic-theorems-on-compression-and-model-selection. Zuletzt eingesehen am 28.05.2019.

[WZB18] Shengjie Wang, Tianyi Zhou, and Jeff Bilmes. Jumpout: Improved dropout for deep neural networks with rectified linear units. *https://openreview.net/forum?id=r1gRCiA5Ym*, 2018.

[YYR18] Xin Yu, Zhiding Yu, and Srikumar Ramalingam. Resnet sparsifier: Learning strict identity mappings in deep residual networks. *http://arxiv.org/abs/1804.01661v4*, 2018.