Why deep neuronal networks are difficult to train

Mashine Learning Seminar

Christian Gommeringer

betreut durch Prof. Schreiber und Prof. Hinderhofer

Tübingen, den 21. Juni 2023

Motivation

In unserem heutigen Vortrag werden wir die ersten Schritte ausgehend von den bisher betrachteten flacheren Netzwerken mit nur einem Hidden Layer hin zu tiefen neuronalen Netzen mit vielen Layern, die in der Anwendung oft benutzt werden, unternehmen. Mit den bisherigen Netzwerken haben wir ja bereits gute Erebnisse erziehlt. Es lassen sich allerdings Gründe finden, wieso die Leistung von Deep Learning höher sein kann. Zum einen lässt sich das Konzept meherer auf einander aufbauender Level gut mit der natürlichen Herangehensweise identifizieren, mit der man ausgehend von grundlegenden Bausteinen sein Modell mit jedem Abstraktionsniveau weiter verfeinert.[1]
Es lässt sich auch zeigen, dass durch Benutzung mehrer Layer von stückweise linearen Neuronen sich eine Outpunktfunktion generieren lässt, bei der die Anzahl an linearen Teilen, in die man die Funktion aufteilen kann, größer ist als bei flachen Netzwerken. Dadurch kann die Outpunktfunktion kompliziertere Funktionen annähern, was im Ende ja das Ziel von Mashine Learning ist (je komplizierter die Funktion ist, desto feinteiliger kann man unterscheiden). Der Beweis kann mit Hilfe der Geometrie von Schnitten von Räumen durch lineare Manigfaltigkeiten (oder so ähnlich :) geführt werden. Diesen Beweis werde ich nicht vorstellen. Ich möchte aber eine Vorstellung dafür vermitteln, in dem ich den Fall für die rectified Aktivierungfunktion, die im nächsten Kapitel auch des öfteren zur Anwendung kommt, untersuche. Die Aktivierungsfunktion rectified ist einfach die im Negativen abgeschnittene Identität.

$$id_{\text{rect}}(x) = \begin{cases} x & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases}$$

Ich werde id_{rect} auch komponentenweise auf einen Vektor angewandt verwenden. Ich wähle hier die Größe von input, 1. hiddden und 2. hidden Layer gleich und bezeichne sie mit n. W_1, W_2, b_1 und b_2 sind die entsprechenden Gewichtsmatrizen und Vektoren. Untersuchen wir zunächst das Output a_1 des ersten hidden Layers, das sich folgendermaßen darstellt

$$a_1 = id_{\text{rect}}\ (W_1\ + b_1)$$

Wenn W_1 eine Invertierbare Matrix ist wird jeder Punkt im N-dimensionalen Vektorraum erreicht. Es ergeben sich verschiedene lineare Teilbereiche. Ein

[1] Michael Nielsen. *Neural Networks and Deep Leaning*. Dec 2019.

linearer Teilbereich des Inputraums ist der, bei welchem die Output Vektoren nur positive Komponenten besitzt. Die nächsten n Teilgebiete sind die bei denen jeweils nur eine Komponente negativ ist. Dann gibt es noch die Teilgebiete, bei denen 2 Komponenten negativ sind und so weiter. Es gibt also insgesamt

$$N_1 = \sum_{k=0}^{n} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} 1^k \, 1^{n-k} = 2^n$$

lineare Teilgebiete. für diese linearen Teilmengen gilt die Linearitätsbedingung

$$a_1 \left(\lambda \, x + \mu \, x'\right) = \lambda \, a_1(x) + \mu \, a_1(x')$$

für

$$\lambda, \mu \geqslant 0$$

, da so die Eigenschaft der Komponenten, ob sie positiv oder negativ sind nicht verändert werden. Diese Mengen sind zusammenhängend, wessen man sich auf folgende Weise vergewissern kann. Die Funktion

$$g : \mathbb{R}_{>0} \times \mathbb{R}_{>0} \to \mathbb{R}^n : (q, p) \mapsto \frac{1}{q+p} \, \left(q \, x_1 + p \, x_2\right)$$

für zwei Punkte $x_1,\ x_2$ im selben Linearitätsgebiet ist eine stetige Funktion, bei der das Output die obige Linearitätsbedingung erfüllt und für die gilt

$$g(1, 0) = x_1$$
$$g(0, 1) = x_2$$

Es gibt also einen stetigen Weg von x_1 nach x_2, der ganz im Linearitätsgebiet liegt, wodurch diese Menge wegzusammenhängend und somit auch zusammenhängend ist. Außerdem liegt für ein x auch die gesamte Strecke vom Ursprung ins Unendliche im Linearitätsgebiet (wieder aufgrund der obigen Linearitätsbedingung). Trennflächen zwischen den Gebieten können somit nur Ebenen sein, die den Ursprung schneiden. Mit dieser Vorstellung können wir weiter mit der Betrachtung des zweiten Layers fortfahren. Die Bildmenge des ersten hidden Layers ist die Teilmenge des $\mathbb{R}^n$, in der alle Komponenten größer gleich Null sind, wobei der Bereich, in dem alle Komponenten strikt größer als Null sind, ein einziges Linearitätsgebiet ist, und die die anderen Linearitätsgebiete Teile der Seitenflächen sind. Schauen wir uns als Beispiel

die Konfiguration

$$\begin{aligned} W_1 &= id \\ b_1 &= 0 \\ W_2 &= Rot_z(\alpha) \\ b_2 &= 0 \end{aligned}$$

für $n = 3$ an. Hier sind die Linearitätsgebiete die kanonischen achtel des $\mathbb{R}^3$. Das Output von Luyer 1 ist dann, wie zuvor im $\mathbb{R}^3_{\geqslant 0}$. Wenn wir im 2. Layer nun vor der rectified Aktivierung um einen kleinen Winkel um die z-Achse drehen, und dann mit rectify aktivieren, führen wir in der Nähe der yz-Ebene ein neues Linearitätsgebiet ein, während wir die bisherigen Linearitätsgebiete erhalten. Wir können innerhalb des zweiten Layers noch zusätzlich um weitere Achsen drehen, oder mögliche andere lineare Operationen durchführen, um weitere Linearitätsgebiete zu erzeugen. Auf so eine Art und weise könnte man in den nächsten Layer fortfahren und eine immer komplexere Funktion generieren. Wir betrachten uns zur Veranschaulichung die Vektoren

$$\begin{aligned} x_1 &= \begin{pmatrix} a \\ -\varepsilon \end{pmatrix} \\ x_2 &= \begin{pmatrix} b \\ c \end{pmatrix} \end{aligned}$$

mit a, b, c, $\varepsilon > 0$. Nach dem ersten Layer, das die Identität in der Aktivierungsfunktion enthält, werden die Vektoren folgendermaßen umgeformt.

$$\begin{aligned} a_1(x_1) &= (a, 0) \\ a_1(x_2) &= (b, c) \end{aligned}$$

und nach dem zweiten Layer

$$\begin{aligned} a_2(x_1) &= id_{\text{rect}}\left(\begin{pmatrix} \cos(\alpha)\ a \\ \sin(\alpha)\ a \end{pmatrix}\right) \\ a_2(x_2) &= id_{\text{rect}}\left(\begin{pmatrix} \cos(\alpha)\ b - \sin(\alpha)\ c \\ \sin(\alpha)\ b + \cos(\alpha)\ c \end{pmatrix}\right) \\ a_2(x_1) + a_2(x_2) &= id_{\text{rect}}\left(\begin{pmatrix} \cos(\alpha)\ a \\ \sin(\alpha)\ a \end{pmatrix}\right) + id_{\text{rect}}\left(\begin{pmatrix} \cos(\alpha)\ b - \sin(\alpha)\ c \\ \sin(\alpha)\ b + \cos(\alpha)\ c \end{pmatrix}\right) \end{aligned}$$

während

$$a_1(x_1 + x_2) = (a + b, c - \varepsilon)$$

für ε klein genung, und

$$a_2(x_1 + x_2) = id_{\text{rect}} \left(\begin{pmatrix} \cos(\alpha)\ (a + b) - \sin(\alpha)\ (c - \varepsilon) \\ \sin(\alpha)\ (a + b) + \cos(\alpha)\ (c - \varepsilon) \end{pmatrix} \right)$$

Man erkennt hier zum einen, dass bestehende Linearitätsgebiete nicht wieder zusammen geführt wurden, da sich $a_2(x_1 + x_2)$ von $a_2(x_1) + a_2(x_2)$ durch den ε-Term unterscheidet. Zum andern ist zu sehen, dass sich ein neues Linearitätsgebiet ergibt. Für eine groß genuge Drehung α gibt es nähmlich Punkte

$$x_1 = (a, b)$$
$$x_2 = (a', b')$$

mit a, b, a', $b' > 0$, für die gilt

$$Rot_\alpha(a_1(x_1))_0 > 0$$
$$Rot_\alpha(a_1(x_2))_0 < 0$$

Es unterscheiden sich also die Vorzeichen der 0-Komponenten vor der Anwendung der Aktivierungsfunktion des 2. Layers, wodurch diese beiden Vektoren nach dem 2. Layer in unterschiedliche Linearitätsgebiete fallen.

Verschidene Verfahren zur Verbesserung des Lernverhaltens tiefer neuronaler Netze

In diesem Abschnitt stelle ich zunächst den Vorteil zweier verbesserterter Optimierungsverfahren vor, der Hessian-free Optimierung und Nesterov's Accelerated Gradient (NAG). Danach stelle ich kurz eine leicht verbesserte Inintialisierungsmethode vor und zum Ende gehe ich noch ein wenig auf unsupervised pre-training ein.

Optimierungsverfahren

Beginnen wir mit dem Ablauf der Hessian-free Methode[2] und mit einer Betrachtung zu dessen Vorteil bei allgemeinen Optimierungsproblemen. Die

[2] J. Martens. „Deep learning via Hessian-free optimization“. In: (2010).

Kernidee dieses Verfahrens ist es die Funktion bis zur zweiten Ordnung zu approximieren

$$f(x) \approx T_2(x) = f(p) + \nabla f(p) \cdot x + \frac{1}{2} x^T H x$$

und dann die Funktion T_2 zu minimieren, was die Lösung einer Gleichung der Form

$$-\nabla f(p) = H x$$

erfordert. Es sei noch erwähnt, dass in der Praxis H oft noch etwas modifiziert wird, um die Leistung noch zu steigern. Die Schwierigkeit zur Lösung dieser Gleichung, besteht auch aber nicht haupsächlich in der Bestimmung von H, für die jedoch relativ gute Algorithmen vorhanden sind. Sie besteht zu einem großen Teil darin, dass gewöhnliche Algorithmen oft zur Lösung dieses Problems die Matrix H zu invertieren, was viel kostet. Der Hessian-free Algorithmus umgeht diese Probleme, indem er zunächst

$$H x = \frac{\nabla f(p + \varepsilon x) - \nabla f(p)}{\varepsilon} \tag{1}$$

für ein kleines ε approximiert und dann die Lösung der quadratischen Gleichung der Minimierung von T_2 mittels des konjugierten Gradientenabstiegsverfahrens mit einigen wenigen Iterationen annähert. Die Lösung der Gleichung ist somit nicht exakt, genügt jedoch um die Vorteile eines Hesseverfahrens gegenüber eines einfachen Gradientenabstiegs zu erhalten. Diese liegen nämlich in der Analyse des Krümmungsverhaltens durch den Algorithmus. Nehmen wir an die Lösung von Gleichung (1) liegt in Richtung eines Eigenvektors d. Dann lässt sich die die Lösung schreiben als

$$x = \frac{\nabla f(p)}{|\frac{\partial^2 f(p)}{\partial d^2}|}$$

Im Update wird der Gradient also mit der Krümmung in diese Richtung gewichtet.

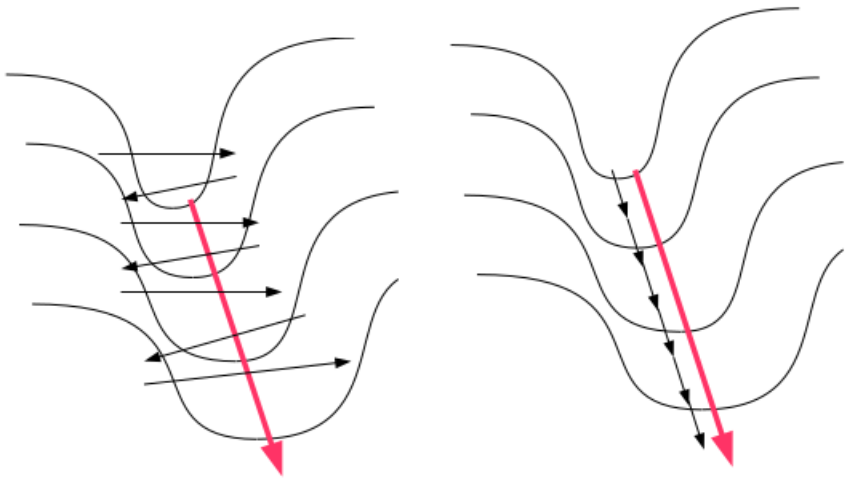

Abbildung 1: Beispiel für eine Optimierungslandschaft (J. Martens. „Deep learning via Hessian-free optimization“. In: (2010))

Wenn wir obiges Bild betrachten, sehen wir eine Situation, in der ein Hesse Verfahren von Vorteil ist. Es handelt sich hierbei um ein steil eingeschnittenes Tal, das auf seiner Sohle flach aber auf einer großen Entfernung abfällt. Die Steigung vertikal zum Verlauf des Tals ist hier größer, weshalb der gewöhnliche Gradientenabstieg einen größeren Schritt in diese Richtung macht als in die parallele und deshalb oft von den Talseiten hin und herspringt. Der Hessian-free Algorithmus wird zusätzlich mit der Krümmung gewichtet und macht daher größere Schritte entlang des Tals, wodurch er in weniger Schritten zum Minimum findet. Wir können uns plausibel machen, dass solche Situationen nicht selten sind, da es sinnvoll erscheint, dass eine weniger gektrümmte Steigung einen nachhaltigeren Abfall bewirkt. Die Hessian-free Optimierung ist für das Trainieren tiefer neuronaler Netze im besonderen von Vorteil, weil er auch beim Kampf gegen das unstable Gradien Problem hilft. Wir erinnern uns, dass der Grund darür die Rekursion zur berechnung der Gradienten verschiedener Layer ist, bei der man viele potentiell kleine Zahlen mit einander multipliziert.
Mit der Notation aus dem Buch für die Aktivierung nach dem l-ten Layer a^l und

$$z^l = W^l\, a^{l-1} + b^l$$

lässt sich die Ableitung der Kostenfunktion schreiben als

$$\frac{\partial C}{\partial z^i} = \frac{\partial C}{\partial z^L} \prod_{k=L-1}^{i} \frac{\partial z^{k+1}}{\partial z^k}$$

für die zweite Ableitung ergibt sich daraus

$$\begin{aligned}\frac{\partial^2 C}{\partial z^j \partial z^i} =& \frac{\partial^2 C}{\partial z^j \partial z^L} \prod_{k=L-1}^{i} \frac{\partial z^{k+1}}{\partial z^k} \\ &+ \frac{\partial C}{\partial z^L} \sum_{\max\{j,i\}}^{L-1} \left[\left(\prod_{k=L-1,\, k\neq l}^{i} \frac{\partial z^{k+1}}{\partial z^k} \right) \frac{\partial^2 z^{l+1}}{\partial z^j \partial z^l} \right]\end{aligned}$$

Diese Ausdrücke schätze ich ab in dem alle Gewichte ungefähr von der gleichen Größenordnung sind und schreibe.

$$\frac{\partial C}{\partial z^i} = O(w^{L-i})$$

die zweite Ableitung ist in diesem Fall auch in einer ähnlichen Größenordnung.

$$\frac{\partial^2 C}{\partial z^j \partial z^i} = \sum_{k=i}^{L} O(w^{L-i+\max\{0,k-j\}}) = O(w^{L-j})$$

Hier ist zu sehen, dass erste und zweite Ableitung z.b. nach dem ersten hidden Layer von gleicher Größenordnung bezüglich der Gewichte sind (und beinahe gleicher Größenordnung mit berücksichtigung nicht stückweise linearer Aktivierungsfunktionen). Da beim Hessian-free Algorithmus noch durch die zweite Ableitung geteilt wird, wirkt das dem unstable Gradient Problem entgegen.

Als zweiten vorteilhaften Algorithmus möchte Nesterov's Accelerated Gradient Algorithmn vorstellen. Es handelt sich hierbei um eine leichte veränderung des Gradientenabstiegs mit Momentum. Das Update Schema lautet[3]

$$\begin{aligned} v_{t+1} &= \mu\, v_t + \epsilon\, \nabla f(x_t + \mu\, v_t) \\ x_{t+1} &= x_t + v_{t+1} \end{aligned}$$

[3]G. Dahl G. Hinton I. Sutskever J. Martens. „On the importance of initialization and momentum in deep learning“. In: (2013).

Man kann sich an dem Talbeispiel von vorhin veranschaulichen, dass dieses Schema auch die Krümmung der Landschaft berücksichtigen kann. Das Update Schema erlaubt es dem Algorithmus nämlich ein wening voraus zu schauen und der Gradient in Richtung senkrecht zum Talverlauf würde durch Eingbeziehung der anderen Seite des Tal, auf der er im nächsten Schritt landen würde, Korriegiert. Oszillationen können sich auf diese Weise nicht konstruktiv aufsummieren. Auch dem vorherschenden vanishing Gradient Problem kann dieser Algorithmus entgegenwirken, da er erlaubt, dass sich die Gradienten über die Zeit aufsummieren. Auch hier erlangen nachhaltige Abstiege ein größeres Gewicht und unnötige Oszillationen summieren destruktiv. Ein großer Vorteil dieses Algorithmus ist, dass er weniger aufwändig ist, da er mit viel weniger Gradienten Auswertungen auskommt. In der Praxis hat dieser Algorithmus dennoch gute Ergebnisse erbracht.

Initialisierung

Wie schon im Vortrag von Peter und David, besprochen ist es wichtig für das anfängliche Lernen des Systems, wie die Gewichte am Anfang verteilt sind. Für tiefe neuronale Netzwerke muss die Varianz der Anfangsverteilung leicht angepasst werden im Vergleich zu flachen Netzen. Wie Bengio und Glorot zeigten lässt sich die fortgepflanzte Varianz der Gradienten der Gewichte -als Kennzeichen deren Größe- schreiben als

$$Var[\frac{\partial C}{\partial W^i}] = (n\,Var[W])^L\,Var[x]\,Var[\frac{\partial C}{\partial z^L}]$$

wobei die n die Zahl der Neuronen in einem Layer ist, die hier als Gleich für alle Layer angenommen wird. Um einen verschwindenden oder explodierenden Gradient für großes L zu verhindern, ist zu wünschen, dass

$$n\,Var[W] = 1$$

ist. Um dies zu erreichen, kann W^i aus einer gleicförmigen Verteilung aus dem Intervall

$$\left[-\sqrt{\frac{6}{n_i + n_{i+1}}}, \sqrt{\frac{6}{n_i + n_{i+1}}}\right]$$

gezogen werden.[4]

[4]Xavier Glorot und Yoshua Bengio. „Understanding the difficulty of training deep feedforward neural networks“. In: (2010).

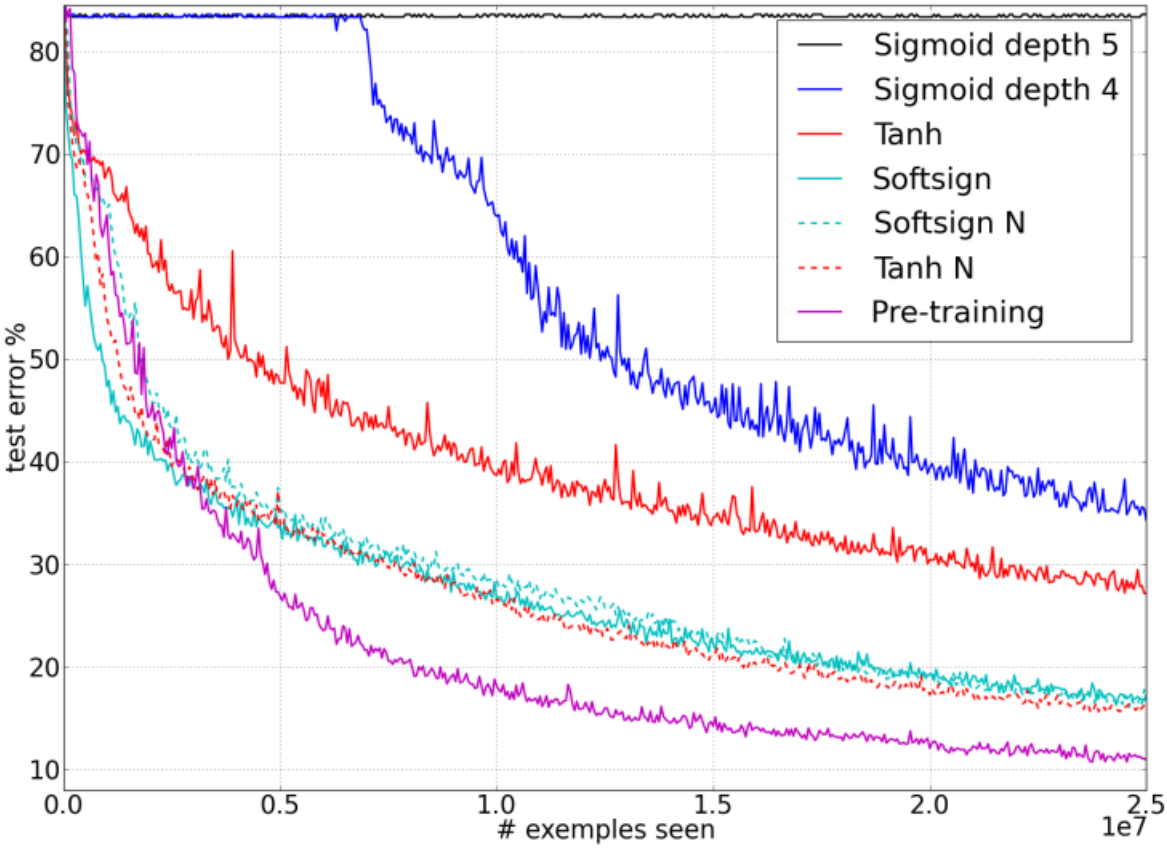

Abbildung 2: Trainingsverlauf für verschiedene Aktivierungsfunktionen. Das label N steht hier für eine Initialisierung nach obiger Methode. (Xavier Glorot und Yoshua Bengio. „Understanding the difficulty of training deep feedforward neural networks“. In: (2010))

In Obigem Beispiel sieht man, dass diese Initialisierung, die als normalized Initialization bezeichnet wird, sehr gut abschneidet und nur von unsupervised pre-Learning übertroffen wird. Die Staffelung der Leistung für die verschiedenen Aktivierungsfunktionen bei gewöhnlicher Initialisierung wird von Glorot & Bengio (cite) so erklärt, dass man beobachtet, dass beim Start des Lernens hauptsächlich von den Biases im Outputlayer gelernt wird, und von den Input daten nur ein Random Output ohne Information ankommt. Daher ist das System bestrebt, den Effekt des Input zu verringern und daher die Gewichte klein (gegen 0) zu machen. Da die Sigmoid Funkion für

zero-Output komplett gesätigt ist, beginnt das System nur sehr langsam zu lernen. Da tanh und softsign ($= x/(|x|+1)$) symetrisch um 0 als Output verteilt ist, und diese Funktionen sich dort im linearen Regime befinden, findet bei diesen Aktivierungen ein schnelleres Wachstum statt. Der Unterschied zwischen softsign und tanh besteht darin, dass tanh exponentiell gegen seine Asymptote geht und softsign quadratisch. Softsign sättigt dadurch weniger schnell und besitzt deshalb ein besseres Lernverhalten.

Unsupervised pre-Training

Die besten Ergebnisse für das oben betrachtete Problem lieferte, wie angesprochen, eine Initialisierung mit unsupervised pre-training. Unsupervised learning ist eine Art von Mashine Learning, in der anders als beim supervised Learning nicht ein Paar von Input und erwünschten korrekten Output Combinationen übergeben wird, sondern nur Input Daten übergeben werden. Ziel ist es hier eine gewisse Struktur in den Input Daten zu erkennen, wie beispielsweise Clusster oder Features. Allgenein kann das Ziel von unsupervised learning möglicherweise so formuliert werden, dass versucht wird eine von gewissen Prarmetern ω abhähngige Wahrscheinlichkeitsverteilung $P(\mathbf{x}, \omega)$ der Input Daten zu finden, die die Wahrscheinlichkeit für das Auftreten der spezifischen Input Daten maximiert.
Da die Ziele und Methoden im unsupervised learning Bereich jedoch so verschieden sind, ist es schwer eine allgemeine Herangehensweise zu formulieren. Ich stelle deshalb nur das spezifische Verfahren dar, das in obigem Fall verwendet wurde und das sich dem Anschein nach besonders gut eignet, um tiefe neuronale Netze zu initialisieren. Es handelt sich hierbei um einen Denoising Autoencoder.[5] Ein gewöhnlicher Autoencoder hat zwei hidden Layers. Im ersten Layer wird der Input comprimiert. Das erste Layer hat in diesem Fall eine niedrigere Dimension als das Input Layer, es handelt sich hierbei um ein gewöhnliches Neuronales Layer, dessen Output

$$f(x) := \sigma(W\,x + b)$$

ist. Im zweiten und letzten Layer, welches wiederum ein gewöhnliches neuronales Layer ist, wird versucht die verlorene Information zu rekonstruieren.

[5] P. Vincent H. Larochelle Y. Bengio P.-A. Manzagol. „Extracting and Composing Robust Features with Denoising Autoencoders“. In: (2008).

Ich schreibe das Output des zweiten Layers als

$$g(x) := \sigma(W' x + b').$$

Das System wird trainiert, in dem der Abstand von Output und Input minimiert wird.

$$\begin{aligned} &\frac{1}{n} \sum_i ||x_i - g(f(x^i))||^2 \\ &= \mathbf{E_{p(x)}}[||\mathbf{x} - \mathbf{g(f(x))}||^{\mathbf{2}}] \end{aligned}$$

wobei x^i ein Datenpunkt aus den Trainingsdaten und p die intinsische Wahrscheinlichkeitsverteilung der Datenpunkte ist.

$$\mathbf{E_{p(x)}}[\mathbf{F(x)}]$$

beschreibe hier den Erwarungswert von F bezüglich der Wahrscheinlichkeitsverteilung p. In dem Paper, auf das ich mich beziehe, wurde eine andere Kostenfunktion verwendet, die in der Praxis oft verwendet wird. Zunächst führe ich dafür die Bernoulli Verteilung ein

$$\mathfrak{B}_\alpha(x) = \begin{cases} \alpha & \text{wenn } x = 1 \\ 1 - \alpha & \text{wenn } x = 0 \end{cases}$$

Im folgenden ist die die Variable x in dieser Definition verdeckt und von keiner Bedeutung. Die verwendete Kostenfunktion ist

$$\begin{aligned} &\mathbf{E_{p(x)}}\left[\mathbf{L_H}(\mathbf{x}, \mathbf{g(f(x))})\right] \\ &:= \mathbf{E}_{\mathfrak{B}_\mathbf{x}}\left[-\log(\mathfrak{B}_{\mathbf{g(f(x))}})\right] \\ &= -\frac{1}{n} \sum_i \sum_{k=0}^{d} \left[x_k^i \log(g(f(x_k^i))) + (1 - x_k^i) \log(g(f(x_k^i)))\right] \end{aligned}$$

d ist die Dimension der Input Daten. Für den vollen von Vincent et al. (P. Vincent H. Larochelle Y. Bengio P.-A. Manzagol. „Extracting and Composing Robust Features with Denoising Autoencoders“. In: (2008)) verwendeten denoising autoencoder, wurde noch ein zufälliges Rauschen eingebaut. Dafür wurde je nach Stärke des Rauschens eine Bestimmte Zahl an Komponenten

eines Inputvektors zufällig ausgewählt und auf 0 gesetzt. Diese Operation bezeichne ich mit $\tilde{x}(x)$ oder oft auch einfach mit $\tilde{x}$. Minimiert wurde

$$\mathbf{E}_{\mathbf{p(x)}}\left[\mathbf{L}_{\mathbf{H}}(\mathbf{x}, \mathbf{g}(\mathbf{f}(\tilde{\mathbf{x}})))\right]$$

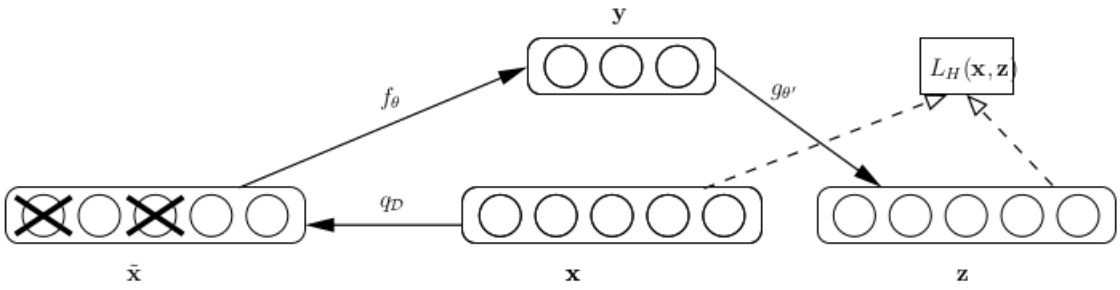

Abbildung 3: Aufbau eines denoising Autoencoders (P. Vincent H. Larochelle Y. Bengio P.-A. Manzagol. „Extracting and Composing Robust Features with Denoising Autoencoders“. In: (2008))

Für den Fall des gewöhnlichen Autoencoders ist erforderlich, dass die Dimension des ersten Hidden Layers kleiner als die Dimension der Input Punkte ist, damit das Netz nicht einfach die Identität lernen kann. Für den Fall der des denoising Autoencoders ist dies aufgrund der stochastischen Informationsreduktion ausgeschlossen. Daher kann die Größer der Layer beliebig gewählt werden. Dies ist vorteilhaft für die Anwendung als Initialisator neuronaler Netze. Zur Initialisierung wird startend vom ersten nach dem Input jedes Layer nach einander durchgegangen und zusammen mit dem vorherigen nach dem denoising autoencoder Verfahren trainiert, wobei als Input immer das zufällig verähnderte output des vorherigen Layers verwendet wird.
Was durch das denoising autoencoding Training heuristisch betrachtet erreicht wird, ist, dass das Netzwerk robuster gegen kleine Störungen des Inputs ist und besser die wirklich wichtigen Charackteristika der Daten erfasst. Man trainiert ja das mittlere Layer (bei der Initialisierung) und erwartet, dass es das Signal so umwandelt, dass es die (wichtigen) Informationen erhält, um danach eine gute Entschlüsselung zu ermöglichen. Und tatschächlich erreicht man, dass das Endresultat robuster gegen die (ja immer noch for

dem Pre-training notwendige) Initialisiertung wird. Systeme die diese pre-Training Phase in ihrem Training durchlaufen, zeigen auch andere positive Eigenschaften. Es scheint, als ob das pre-Training das System in einen Bereich des Parameterraums bringt, in dem bessere Optimierung möglich ist, wie in folgender Darstellung zu erkennen ist.

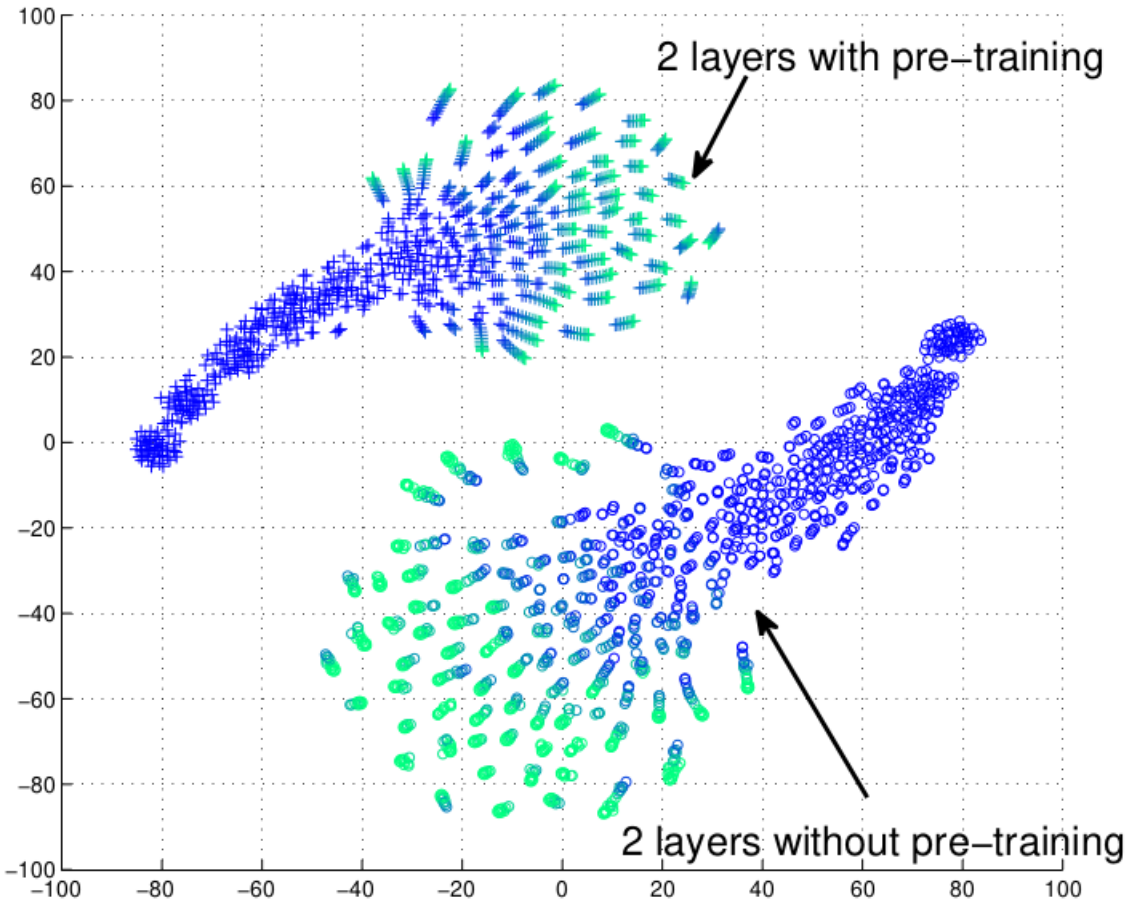

Abbildung 4: Trainingsverlauf im dimensionsreduzierten Output Raum (D. Erhan Y. Bengio A. Courville P.-A. Manzagol P. Vincent S. Bengio. „Why Does Unsupervised Pre-training Help Deep Learning?“ In: (2010))

Es kann auch so gesehen werden, dass da pre-Training den zugänglichen Parameterbereich auf ein Gebiet mit besserer Optimierungmöglichkeit beschränkt und dadurch ein Stück weit wie eine Regularisierung wirkt.

Außerdem ist es nicht der Fall, dass, wie man vermuten könnte, eine pre-Training Phase einer zufälligen Inintialisierung mit optimierter Wahrscheinlichkeitverteilung entspricht. Wenn man nämlich aus den Gewichten und Biases nach der pre-Training Phase eine Wahrscheinlichkeits Verteilung für die Verteilung der Gewichte und Biases gewinnt, und diese entsprechend dieser Wahrscheinlichkeitsverteilung noch einmal zufällig initialisiert, schneidet das Netzwerk schlechter ab. Daraus lässt sich vermuten, dass das System im pre-Training wirklich interessante/representierende Merkmale der Daten lernt.

Quellen

Literatur

Bengio, D. Erhan Y. Bengio A. Courville P.-A. Manzagol P. Vincent S. „Why Does Unsupervised Pre-training Help Deep Learning?“ In: (2010).

Glorot, Xavier und Yoshua Bengio. „Understanding the difficulty of training deep feedforward neural networks“. In: (2010).

I. Sutskever J. Martens, G. Dahl G. Hinton. „On the importance of initialization and momentum in deep learning“. In: (2013).

Manzagol, P. Vincent H. Larochelle Y. Bengio P.-A. „Extracting and Composing Robust Features with Denoising Autoencoders“. In: (2008).

Martens, J. „Deep learning via Hessian-free optimization“. In: (2010).

Nielsen, Michael. *Neural Networks and Deep Leaning*. Dec 2019.