Universität Leipzig

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Einführung in Autoencoder und Convolutional Neural Networks

Seminararbeit im Seminar: Deep Learning

vorgelegt von:
Elias Saalmann

Matrikelnummer:
3731902

Betreuer:
Victor Christen

Inhaltsverzeichnis

1 Einleitung

Heutzutage sind Stichworte wie *Maschinelles Lernen* oder *Künstliche Intelligenz* nicht mehr aus der Informatik wegzudenken. Formal sind damit Computerprogramme gemeint, deren Ergebnis durch die Verarbeitung von Erfahrungen (Training) verbessert werden kann. In den letzten Jahren gerieten gehäuft *Künstliche neuronale Netze* (KNN) und *Deep Learning* in den Fokus von Wirtschaft und Wissenschaft. In dieser Arbeit werden zwei spezielle Arten künstlicher neuronaler Netze erläutert: *Autoencoder* und *Convolutional Neural Networks*. Dazu werden zunächst die Grundlagen des maschinellen Lernens sowie künstlicher neuronaler Netze besprochen. In den nachfolgenden Kapiteln werden dann Autoencoder und Convolutional Neural Networks behandelt. Diese Arbeit beruht grundlegend auf [L$^+$15a] und [L$^+$15b].

Maschinelles Lernen Jedes Problem des maschinellen Lernens lässt sich in den Bereich des *überwachten* oder *unüberwachten* Lernens einordnen. Beim überwachten Lernen liegt ein Trainingsdatensatz bestehend aus Tupeln $(x^{(i)}, y^{(i)})$ vor, wobei die $x^{(i)}$ aus einem Eingaberaum X und die $y^{(i)}$ aus einem Ausgaberaum Y stammen. Dabei sind die Elemente des Eingaberaums sogenannte *Feature-Vektoren*, deren Einträge bestimmten Merkmalen eines Datums entsprechen. Die Elemente des Ausgaberaums sind die den Eingabedaten zugeordneten Werte, z.B. der Wertebereich einer Funktion (Regression) oder eine Menge von Klassen (Klassifikation). Die Tupel stellen bereits vorhandenes Wissen über die Eingabedaten dar, i.e. welchen Wert einer Funktion ein bestimmtes Eingabedatum annimmt oder welche Klasse ein Eingabedatum hat. Ziel des überwachten Lernens ist es nun, eine möglichst generalisierende Funktion $h : X \to Y$ bzw. deren Paramater θ zu lernen, die den Zusammenhang von Eingaberaum und Ausgaberaum möglichst gut beschreiben. Zumeist wird dazu eine Fehlerfunktion J aufgestellt, die für eine konkrete Parameter-Wahl θ einen Fehler auf dem Trainingsdatensatz berechnet. Betrachtet man das Minimierungsproblem über die Fehlerfunktion J, liefert eine Lösung davon zugleich die gesuchte Funktion h. Die Minimierungsprobleme werden in der Praxis häufig über Verfahren aus der Differentialrechnung, i.e. über Gradienten, gelöst. Beim unüberwachten Lernen liegen im Trainingsdatensatz nur Feature-Vektoren $x^{(i)}$ vor, d.h. es liegt kein bereits vorhandenes Wissen über die Daten vor. Beim unüberwachten Lernen wird versucht, Strukturen innerhalb des Eingaberaums zu finden, z.B. mittels Clustering.

Künstliche neuronale Netze Die Wahl der Struktur von h und die damit einhergehende Definition der zu lernenden Parameter ist ein nicht-triviales Problem. Für lineare funktionale Zusammenhänge bietet sich eine einfache lineare Funktion der Form

$$h(x) = \theta x + b$$

an. Dies entspricht der *linearen Regression*.

Für ein binäres linear trennbares Klassifizierungs-Problem hat sich die *logistische Regression* bewährt. Hierbei wird der lineare Ausdruck zusätzlich über die Sigmoid-Funktion g in das Intervall $[0, 1]$ transformiert:

$$h(x) = g(\theta x + b) \text{ mit } g(z) = \frac{1}{1 + \exp(-z)}$$

Wie bereits erwähnt, lassen sich die optimalen Parameter θ über Differentialrechnung bestimmen - entweder über eine explizite Lösung oder über numerische annähernde Verfahren, z.B. das *Gradientenverfahren*. Für unbekannte nicht-lineare Zusammenhänge liefern lineare bzw. logistische Regression allerdings keine akzeptablen Ergebnisse mehr. Neben weiteren Verfahren zur Klassifikation von Daten mit nicht-linearen Zusammenhängen, z.B. spezielleren Regressionsverfahren oder Support Vector Machines, sind *Künstliche neuronale Netze* eine vielversprechende Lösung für derartige Probleme. Dieses Verfahren ist im Gegensatz zu bestehenden statistischen Verfahren an einem biologischen Vorbild ausgerichtet: Ein neuronales Netz besteht aus *Neuronen*, die miteinander verbunden sein können. Dabei verarbeitet jedes Neuron die Signale der eingehenden Neuronen und berechnet eine Ausgabe auf Grundlage einer gewichteten Summe. Das Ergebnis dieses linearen Ausdrucks wird ähnlich der logistischen Regression mittels einer sogenannten Aktivierungsfunktion g weiterverarbeitet und entsprechend der Bedürfnisse skaliert. Die konkrete Gestalt der Aktivierungsfunktion ist nicht fest. Die Hintereinanderausführung solcher Neuronen lässt sich mathematisch als Funktion von Funktionen formalisieren. Betrachte z.B. die folgende Struktur von h:

$$h(x) = \underbrace{g(\omega_1 \cdot \underbrace{g(\underbrace{\theta_1 x + b_1}_{(a)})}_{(b)} + \omega_2 \cdot \underbrace{g(\underbrace{\theta_2 x + b_2}_{(b)})}_{(b)} + c)}_{(c)} \tag{1.1}$$

Dann sind $(a), (b)$ und (c) offensichtlich einfache lineare Ausdrücke bzw. gewichtete Summen (hier in Vektorschreibweise), wobei (a) und (b) Neuronen sind, die dem Neuron (c) als Eingabe dienen.

Offensichtlich eignet sich die mathematische Schreibweise zur Definition bzw. Konstruktion solcher Netzwerke aufgrund mangelnder Übersichtlichkeit schlecht. Alternativ kann der Zusammenhang als gerichteter Graph dargestellt werden: Neuronen können als Knoten und neuronale Verbindungen als Kanten gezeichnet werde. Kanten können mit dem zugehörigen Parameter (Gewicht) bezeichnet werden. Abbildung 1.1 zeigt ein derart dargestelltes neuronales Netz. Dabei symbolisiert die Art der Knoten-Zeichnung die Wahl der Aktivierungsfunktion.

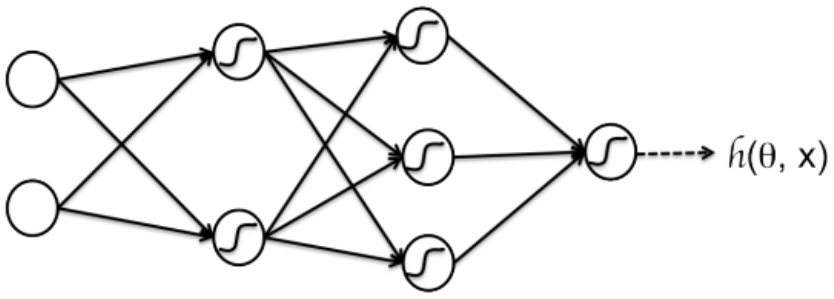


Abbildung 1.1: Beispielhafter Aufbau eines künstlichen neuronalen Netzes [L+15a]

Die Schicht der Eingabeneuronen (ohne eingehende Neuronen) wird auch Eingabeschicht (engl. Input Layer) genannt - die Schicht der Ausgabeneuronen (ohne ausgehende Neuronen) wird als Ausgabeschicht (engl. Output Layer) bezeichnet. Innere Schichten werden versteckte Schicht (engl. Hidden Layer) genannt. Ein künstliches neuronales Netz mit vielen versteckten Schichten wird als *tief* bezeichnet - dies begründet den Begriff *Deep Learning*.

Ein neuronales Netz, welches in seiner Graph-Repräsentation gerichtet und azyklisch ist, also keine Kreise enthält, wird als *Feedforward*-Netz bezeichnet. Sind für eine Schicht alle Neuronen mit allen möglichen Neuronen der darauf folgenden Schicht verbunden, spricht man von schichtweise vollständig verbundenen Netzen.

Backpropagation An der beispielhaften mathematischen Beschreibung (1.1) eines sehr einfachen Feedforward-Netzes wird ersichtlich, welche Parameter trainiert werden müssen: Es sind diejenigen Belegungen für $\omega_1, \omega_2, \theta_1, \theta_2, b_1, b_2, c$ gesucht, welche die Fehlerfunktion J minimieren. Aufgrund der Konstruktion über verkettete Funktionen verkompliziert sich das Finden einer solchen Lösung bzw. entsprechender Gradienten.

Ein mathematisch korrektes Verfahren zur Berechnung aller Gradienten zum Training eines Feedforward-Netzes liefert der *Backpropagation*-Algorithmus. Im Hinblick auf Fokus und Umfang dieser Arbeit sei dieser nachfolgend nur grob umrissen. Grundlegend basiert das Verfahren auf der sogenannten δ-Regel, welche wiederum auf die aus der Differentialrechnung bekannten Kettenregel aufbaut. Die Aktualisierung der Gewichte eines neuronalen Netzes erfolgt grob in den folgenden drei Schritten:

1. Im sogenannten *Vorwärtsdurchlauf* wird das Ergebnis für ein angelegtes Eingabedatum unter Verwendung der aktuellen Gewichte durch einfaches Abarbeiten der Schichten berechnet. Auf Basis des Fehlers des durch den Trainingsdatensatz vorgegebenen Tupels zum tatsächlich berechneten Ergebnis müssen nun neue Parameter bzw. Parameterveränderungen Δw_i bestimmt werden.
2. Im *Rückwärtsdurchlauf* werden für alle w_i die Gradienten $\frac{\partial J}{\partial w_i}$ unter Verwendung bestimmter Rechenvorschriften und des im Vorwärtsdurchlauf bestimmten Fehlers berechnet.
3. Abschließend werden die Parameter gemäß $\Delta w_i = \alpha \frac{\partial J}{\partial w_i}$ aktualisiert, wobei α eine vom Nutzer definierte Lernrate ist.

Der Backpropagation-Algorithmus ist i.A. nur auf einfache Feedforward-Netze anwendbar - im Rahmen dieser Arbeit werden nötige Erweiterungen des Algorithmus an entsprechender Stelle behandelt. Neben der tatsächlichen Berechnung der Gewichte mit Backpropagation stellt sich die Frage, wie diese initialisiert werden sollen. Eine Initialisierung mit 0en hat sich als wenig vielversprechend erwiesen - eine zufällige Initialisierung scheint i.A. sinnvoller zu sein. Nichtsdestotrotz kann auch eine zufällige initiale Belegung zu einem schlechten Klassifizierungsergebnis führen.

2 Autoencoder

Als Motivation zu Autoencodern sei zunächst das Interesse an der niedrig-dimensionalen Kodierung von hoch-dimensionalen Daten genannt. Auf einem Datensatz mit mehreren Hundert Merkmalen kann eine Reduzierung auf einige wenige (miteinander kombinierte) Merkmale von Interesse sein, insbesondere zur Kompression oder Visualisierung.

In diesem Kapitel werden *Autoencoder* als ein spezielle Form von künstlichen neuronalen Netzen eingeführt, die eine solche niedrig-dimensionale Kodierung berechnen. In Abschnitt 2.1 wird diese zunächst mathematisch definiert und in Abschnitt 2.2 in den Kontext von neuronalen Netzen gebracht. In Abschnitt 2.3 werden daraufhin verschiedene Anwendungen der Autoencoder besprochen.

2.1 Definition

Nachfolgend sei ein Autoencoder ein Funktionenpaar (f, g) über einem d-dimensionalen Eingaberaum $X \subseteq \mathbb{R}^d$ definiert. Die Elemente des Eingaberaums seien hier, wie im maschinellen Lernen allgemein üblich, Vektoren, deren Elemente i.A. Merkmalen (engl. *features*) entsprechen. Gesucht sind nun Funktionen

- $f : X \to F$ (Encoder-Funktion)
- $g : F \to X$ (Decoder-Funktion)

wobei $F \subseteq \mathbb{R}^p$ ein p-dimensionaler Zwischenraum mit $p \ll d$ ist und für alle $x \in X$ gilt:

$$|x - (g \circ f)(x)| \to \min \tag{2.1}$$

Anschaulich bedeutet die Forderung (2.1), dass man bei Anwendung der Funktion g auf das Ergebnis der Anwendung von f auf ein beliebiges Element x des Eingaberaums, optimalerweise genau das ursprüngliche Element $\hat{x} = x$ erhält. Da dies je nach Komplexität von f und g nicht immer möglich ist, wird in der Definition eines Autoencoders die Minimierung des Fehlers zwischen x und $\hat{x}$ gefordert [Bal12]. Da p deutlich kleiner als d sein soll, berechnet die Encoder-Funktion offensichtlich eine niedrig-dimensionale Kodierung, wie sie zu Beginn des Kapitels beschrieben wurde. Der Autoencoder liefert zudem die Decoder-Funktion, über die aus einem bereits codierten Datensatz das Ursprungsdatum berechnet werden kann - die Kodierung ist also umkehrbar.

Eine solche Definition eines Autoencoders ist sehr verallgemeinernd, da keine Forderungen an die konkrete Gestalt der Funktionen enthalten ist. Für einen endlichen Eingaberaum lassen sich die gesuchten Funktionen formal in trivialerweise definieren.

Im Folgenden betrachten wir ein Beispiel unter der Annahme, dass f und g lineare Funktionen bzw. Polynome ersten Grades sind, um anschließend auf die Realisierung mittels künstlicher neuronaler Netze einzugehen.

Beispiel Für einen einen Eingaberaum X mit Zwischenraum Y sollen die beiden Funktionen f und g eines Autoencoders in diesem Beispiel folgende Gestalt haben:

- $f(x^{(i)}) = W_1 x^{(i)} + b_1$
- $g(x^{(i)}) = W_2 x^{(i)} + b_2$

Gesucht sind nun also die Parameter W_1, b_1, W_2, b_2, die für jedes Eingabedatum den Fehler (2.1) minimieren.

Unter Zuhilfenahme der Methode der kleinsten Quadrate [Gau77] erhält man für den Eingaberaum $X = \{x^{(1)}, \dots, x^{(m)}\} \subset \mathbb{R}^d$ folgendes Optimierungsziel J:

$$J(W_1, b_1, W_2, b_2) = \sum_{i=1}^{m} (x^{(i)} - (g \circ f)(x^{(i)}))^2 \tag{2.2}$$

$$= \sum_{i=1}^{m} (x^{(i)} - (W_2(W_1 x^{(i)} + b_1) + b_2))^2 \tag{2.3}$$

$$\rightarrow min$$

Dabei ergibt sich (2.3) durch einfaches Einsetzen der Hintereinanderausführung von f und g in (2.2). Eine Lösung für W_1, b_1, W_2, b_2, die diesen Ausdruck minimiert, ist ein Autoencoder. Nachfolgend wird die Verwendung von künstlichen neuronalen Netzen für das Lernen dieser Parameter erläutert.

2.2 Autoencoder als künstliche neuronale Netze

Wie bereits in der Einführung erwähnt, ist es möglich, mit einem künstlichen neuronalen Netz praktisch jede Funktion zu berechnen [MP43]. Entsprechend lassen sich auch ein Autoencoder bzw. die Encoder- und Decoder-Funktionen über neuronale Netze berechnen. Offensichtlich benötigt der Autoencoder über einem Eingaberaum $X \subseteq \mathbb{R}^d$ jeweils genau d Eingabe- und Ausgabe-Neuronen.

Betrachtet man Funktionen der im vorangegangen Beispiel definierten Gestalt, ergibt sich als einfachster Autoencoder ein schichtweise vollständig verbundenes neuronales Netz mit d Eingabe- und Ausgabe-Neuronen sowie einem Hidden Layer mit p Neuronen, wenn p die Dimension des Zwischenraums ist. Ein solches Netz ist in Abbildung 2.1 dargestellt.

In dem dargestellten Netz wurde beispielhaft eine lineare Aktivierungsfunktion gewählt - die konkrete Wahl der Hyperparameter spielt i.A. keine Rolle hinsichtlich der Definition eines Autoencoders. Ab nun bezeichnen wir mit Autoencoder ein künstliches neuronales Netz, welches eine Hintereinanderausführung (zweier oder mehrerer) linearer Funktionen gemäß der mathematischen Definition berechnet bzw. gelernt hat.

Um diese Parameter zu lernen, muss das neuronale Netz trainiert werden. Dazu kann der Backpropagation-Algorithmus verwendet werden, da ein Autoencoder keine Unterschiede zu einem schichtweise vollständig-verbundenen Feedforward-Netz aufweist. Der Trainingsdatensatz

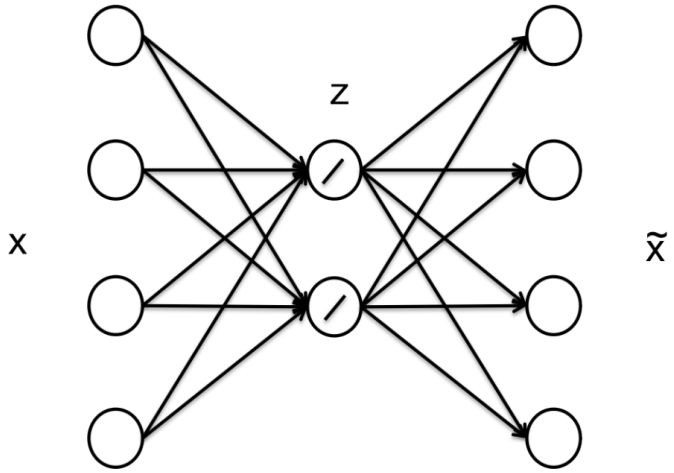


Abbildung 2.1: Beispiel-Autoencoder KNN [L+15b]

ist dabei entweder der Eingaberaum selbst oder eine (möglichst repräsentative) Teilmenge des Eingaberaums. Das Training hat so zu erfolgen, dass für jedes Eingabedatum x genau dieses Datum x auch an die Ausgabeschicht gelegt wird. Dies entspricht genau der intuitiven Vorstellung der Funktion, die der Autoencoder lernen soll. Da die Eingabedaten nicht klassifiziert vorliegen müssen, sondern ohne zusätzliches Wissen verarbeitet werden, ist ein solcher Autoencoder bzw. das Training und Lernen der Parameter in den Bereich des unüberwachten Lernens einzuordnen.

Liegt ein als Autoencoder trainiertes neuronale Netz vor, kann die niedrig-dimensionale Codierung eines Eingabedatums x berechnet werden, indem das Datum an die Eingabeschicht angelegt wird. Die Codierung entspricht dann den p berechneten Werten der Aktivierungsfunktionen für jedes Neuron des Hidden Layers (in der Abbildung mit z bezeichnet). Es kann selbstverständlich auch ein Eingabedatum angelegt werden, das nicht Teil des Trainingsdatensatzes war - optimalerweise ist die Kodierung jedoch so gut gelernt, dass auch für ein solches Datum die Ausgabe der ursprünglichen Eingabe (nahezu) entspricht.

Die Anzahl der Hidden Layer eines Autoencoders ist nicht per Definition beschränkt. So könnte die Dimension von Schicht zu Schicht schrittweise verringert werden, sodass die Zieldimension p erst nach einigen Schichten erreicht wird. Bildlich lässt sich so ein Autoencoder als Trichter darstellen. Analog kann sich die Erhöhung der Dimension über mehrere Schichten hinweg ziehen.

2.3 Anwendungen

In den vorangegangenen Abschnitten wurde bereits verdeutlicht, dass ein Autoencoder zur niedrig-dimensionalen Kodierung der Daten eines Eingaberaums verwendet wird. Diese Kodierung aller Elemente des Eingaberaums kann auch als Dimensionreduzierung des Raums bzw. der Elemente im Raum bezeichnet werden. Ein bekanntes und weit verbreitetes statistisches

Verfahren zur Dimensionsreduzierung ist die Hauptkomponentenanalyse (PCA, engl. *Principal Component Analysis*) [Pea01]. Dabei werden, ähnlich zu Autoencodern, Elemente aus dem Eingaberaum in einen niedrig-dimensionalen Raum projeziert. In [HS06] werden verschiedene Anwendungen von Dimensionsreduzierung aufgezeigt und die Ergebnisse von Autoencodern mit denen einer PCA verglichen. Diese sollen hier als Veranschaulichung einiger Anwendungen von Dimensionsreduzierung dienen.

Dimensionsreduzierung zur Visualisierung Ist die gelernte Codierung 2- oder 3-dimensional, können hochdimensionale Daten durch die Codierung und damit gegebene Einbettung in den $\mathbb{R}^2$ oder $\mathbb{R}^3$ visualisiert werden. Abbildung 2.2 zeigt die 2-dimensionale Einbettung des MNIST-Datensatzes, wobei die Kodierungen `A` mittels einer PCA und `B` mithilfe eines Autoencoders bestimmt wurden. Der MNIST-Datensatz ist ein weit verbreiteter Datensatz, der aus klassifizierten handgeschriebenen Ziffern (0 bis 9) besteht [LCB10]. Eine handgeschriebenen Zahl entspricht dabei einem Graustufen-Bild. Das Graustufen-Bild wird als Vektor dargestellt, in dem jeder Eintrag einem Farbwert eines bestimmten Pixels entspricht. Bei einer Bildgröße von 28×28 Pixeln ist der Vektor also 784-dimensional. Jeder solche Vektor wurde in einen 2-dimensionalen Vektor transformiert und entsprechend visualisiert. Die zehn verschiedenen Ziffern haben jeweils eine eindeutige Kombination aus Symbol und Farbe, sodass sich in Abbildung 2.2 eindeutige Unterschiede hinsichtlich der Separation der Klassen erkennen lassen. Offensichtlich haben die 2-dimensionalen Repräsentationen, die mittels eines Autoencoder gelernt wurden (`B`), geringere Überlappung und deutlichere Abgrenzungen - die Repräsentation ist daher i.A. von höherer Qualität als diejenige, die mittels PCA berechnet wurde.

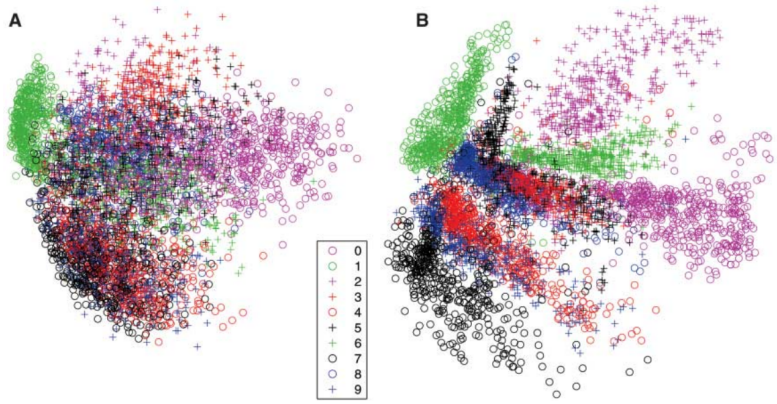


Abbildung 2.2: 2D-Codierung von handgeschriebenen Zahlen [HS06]

Dimensionsreduzierung im Kontext der Gesichtserkennung Im vorangegangen Beispiel wurde die Qualität der niedrig-dimensionalen Kodierung betrachtet. Von Interesse ist zudem die Qualität der Rückkodierung, i.e. der Decoder-Funktion. In [HS06] wurde dazu die Dimension von Bildern menschlicher Gesichter reduziert. Dem zugrunde liegt die Tatsache, dass einige wenige Merkmale zur Charakterisierung bzw. Erkennung von Gesichtern genügen. Diese Theorie wurde auf Basis von PCA entwickelt - die Rekonstruktionen der niedrig-dimensionalen Repräsentationen sind als *Eigengesichter* bekannt geworden [SK87]. Abbildung 2.3 zeigt in der obersten Zeile

die Originalgesichter, in der mittleren Zeile die mittels Autoencoder kodierten und anschließend rekonstruierten Bilder und in der untersten Zeile die mittels PCA zurückgerechneten Gesichter. Als Datensatz diente *The Database of Faces* von den AT&T Laboratories Cambridge [SH94]. Offensichtlich liefert der Autoencoder auch hinsichtlich der Rekonstruktion der Ursprungsdaten bessere Ergebnisse als eine PCA: Die Gesichter in der mittleren Zeile sind deutlich *schärfer* und es fällt einem Menschen deutlich leichter, eine Person zu identifizieren. Die mittels PCA gewonnenen Rekonstruktionen lassen nur wenig erkennen.

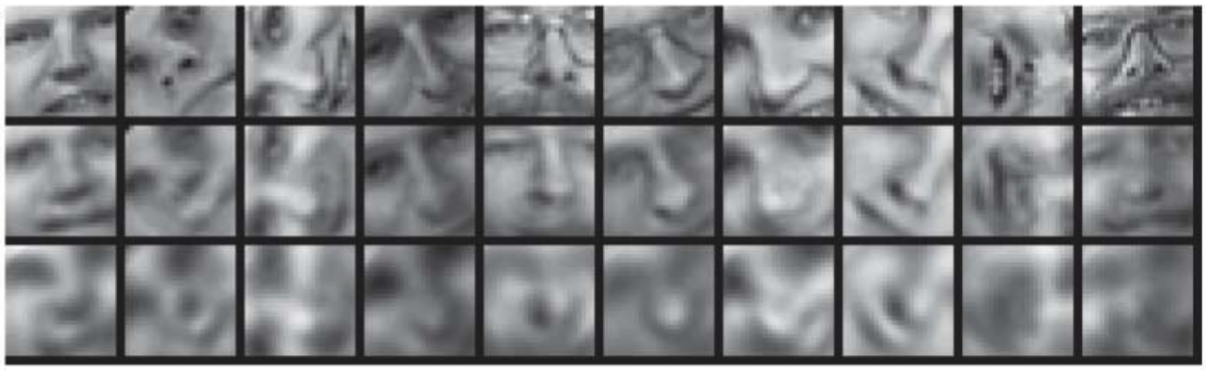


Abbildung 2.3: Rekonstruktion von codierten (30-dimensionalen) Bildern [HS06]

Dimensionsreduzierung zur Kompression Eine weitere einfache Anwendung von Dimensionsreduzierung ist die Kompression: Wird ein d-dimensionales Datum in ein p-dimensionales Datum transformiert, benötigt die komprimierte Variante nur einen Anteil von $\frac{p}{d}$ des ursprünglichen Speicherbedarfs (unter der Annahme, dass der Datentyp einer Dimension gleich bleibt). Optimalerweise ist die Komprimierung zusätzlich verlustfrei - dies würde nach Definition des Autoencoders einem Fehler von 0 für jedes Datum entsprechen und ist daher i.A. nicht zu erreichen. Für verlustbehaftete Komprimierung finden sich in der Praxis zahlreiche Anwendung. Betrachte beispielsweise folgendes einfaches Beispiel:

Angenommen, eine Smartphone-App möchte das Datum $\{x_1, x_2, ..., x_d\}$ an einen Server senden, allerdings ohne unnötig viel Traffic zu erzeugen. Dazu sollte zunächst ein Autoencoder für die entsprechende Art von Daten trainiert werden. Angenommen ein solcher Autoencoder liegt hinreichend gut trainiert auf Client und Server vor, bietet sich folgende Prozedur zur Traffic-Reduzierung an:

1. Kodiere das Datum $\{x_1, x_2, ..., x_d\}$ auf dem Client mittels der gelernten Encoder-Funktion zu $\{z_1, z_2, ..., z_p\}$ mit $p \ll d$.
2. Sende $\{z_1, z_2, ..., z_p\}$ an den Server
3. Dekodiere $\{z_1, z_2, ..., z_p\}$ auf dem Server mittels der vorab gelernten Dekoder-Funktion zu $\{\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_d\}$

Optimalerweise ähneln sich dann die am Server berechneten $\tilde{x}_i$ so sehr den ursprünglichen x_i, dass die App ihren Zweck voll erfüllen kann. Ein etwaiger Fehler ist insbesondere bei audiovisuellen Rohdaten in gewissem Maße hinnehmbar.

Autoencoder zur KNN-Initialisierung In der Einführung wurde bereits angesprochen, dass die Initialisierung der Gewichte in neuronalen Netzen zu Beginn des Trainings von hoher Bedeutung und nicht trivial ist. Insbesondere ist eine zufällige Initialisierung nicht immer optimal. Eine Möglichkeit, die sich als vielversprechend erwiesen hat, beruht auf einer Initialisierung mit Autoencodern [EBC+10].

Das Verfahren besteht aus folgenden drei Phasen:

1. **Pre-Training:** In der ersten Phase werden nacheinander alle Schichten (exklusive der Ausgabe-Schicht) bearbeitet und jeweils ein Autoencoder mit einem Hidden Layer konstruiert. Abbildung 2.4 zeigt im oberen Bereich ein kleines Feed-Forward-Netz, dessen Gewichts-Matrizen W_1 und W_2 in dieser Phase initialisiert werden sollen. Dazu wird zunächst ein Autoencoder konstruiert, der aus den beiden Eingabe-Neuronen, dem ersten Hidden Layer und einem temporären Ausgabe-Layer mit zwei Neuronen besteht (in der Abb. unten grün umrahmt). Dort werden nun alle Trainingsdaten an Eingabe- und Ausgabe-Neuronen (unüberwacht) angelegt und somit W_1 und W_1' gelernt. W_1' wird verworfen und W_1 für die kommenden Iterationen fest gesetzt. Anschließend wird der nächste Autoencoder konstruiert, der die Gewichte W_2 der zweiten Schicht unüberwacht lernt (in der Abb. unten rot umrahmt). Die Anzahl an Eingabe- und Ausgabeneuronen muss in einem solchen unüberwachten Autoencoder selbstverständlich übereinstimmen.

2. **Fine-Tuning 1:** In der zweiten Phase werden die Gewichte des letzten Layers trainiert, allerdings ohne einen unüberwachten Autoencoder. Die Gewichte werden (mit den fest gesetzten bereits gelernten initialen Gewichten der vorherigen Schichten) überwacht auf Basis das Trainingsdatensatzes gelernt.

3. **Fine-Tuning 2:** In der finalen Phase werden das eigentliche Training des Netzes mithilfe von Backpropagation durchgeführt und die initialen Gewichte schrittweise angepasst.

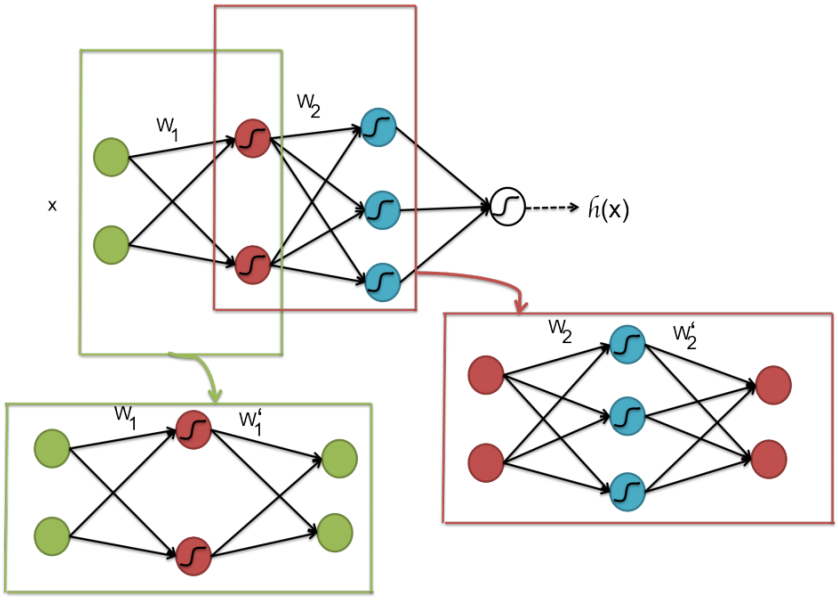


Abbildung 2.4: Autoencoder als Initialisierungsmethode [L+15b]

3 Convolutional Neural Networks

Die Verarbeitung von hochdimensionalen natürlichen Daten, wie z.B. Fotos oder Sprachen, erfreut sich heutzutage in Wirtschaft und Wissenschaft besonders hoher Relevanz. Insbesondere in Anwendungen der künstlichen Intelligenz müssen solche Daten z.B. zur automatischen Spracherkennung oder als Grundlage autonomen Fahrens verarbeitet werden. Ein vielversprechender Ansatz für die Verarbeitung solcher Daten sind die *Convolutional Neural Networks* (CNN) - eine spezielle Form von künstlichen neuronale Netzen, deren zugrundeliegende Ideen in diesem Kapitel besprochen werden.

Nachdem im vorangegangen Kapitel ein spezieller Typ von schichtweise vollständig verbundenen Netzen eingeführt wurde, werden in diesem Kapitel zunächst lokal verbundene Netze eingeführt. Weiterhin wird in Abschnitt 3.2 der mathematische Convolution-Operator eingeführt, der die Grundlage für CNNs bildet. Daraufhin werden Aufbau und Funktion der verschiedenen Schichten eines CNN beleuchtet. Im abschließenden Abschnitt werden Verweise auf populäre CNN-Anwendungen aufgeführt.

3.1 Lokal verbundene neuronale Netze

Sollen hochdimensionale natürliche Daten mit einem künstlichen neuronalen Netz verarbeitet werden, ergibt sich bei schichtweise vollständig verbundenen Netzen eine sehr hohe Anzahl an Kanten bzw. Parametern: Angenommen ein Bild habe 1 Million Pixel, also 1 Million Dimensionen. Besteht weiterhin der erste Hidden Layer aus 100.000 Neuronen, ergeben sich bei vollständiger Verbundenheit $10^6 \cdot 10^5 = 10^{11} = 100$ Milliarden Kanten bzw. Gewichte, die trainiert werden müssen. Im Kontext von Deep Learning sind weitere Vielfache durch nachfolgende Hidden Layer hinzuzurechnen. Gewichts-Matrizen in solchen Dimensionen erhöhen also i.A. den Trainingsaufwand. Weiterhin besteht die Gefahr der Überanpassung (engl. *Overfitting*) des neuronalen Netzes an den Trainingsdatensatz - ist dieser hinreichend klein, kann man sich dieses Overfitting anschaulich als „auswendig lernen“ vorstellen. Eine triviale Art, die Anzahl zu lernender Parameter zu reduzieren, ist das einfache Weglassen von Verbindungen zwischen Neuronen. Solche Netze werden als lokal verbundene neuronale Netze bezeichnet [L$^+$15b]. Abbildung 3.1 zeigt ein solches Netz. Ein lokal verbundenes Netz ist weiterhin ein Feedforward-Netz, das mittels Backpropagation trainiert werden kann: Nicht vorhandene Verbindungen werden in der Gewichtsmatrix dauerhaft mit 0 belegt.

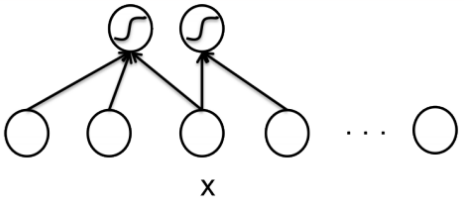


Abbildung 3.1: Lokal verbundenes KNN [L$^+$15b]

3.2 Der Convolution-Operator

Convolutional Neural Networks tragen bereits in ihrem Namen die Bezeichnung *Convolution*. Damit ist der sogenannte diskrete Faltungs-Operator (engl. Convolution) gemeint, welcher häufig in der Bild- und Signalverarbeitung verwendet wird [Her05]. Da dieser Operator die Grundlage für CNNs bildet, wird er nachfolgend jeweils für den 1-dimensionalen und 2-dimensionalen Fall definiert und veranschaulicht.

1D-Convolution (Diskrete Faltung) Sei $A = (a_1, \ldots, a_n)$ ein Eingabevektor. Sei weiterhin F ein sogenannter Filtervektor (auch Kernelvektor genannt) mit $F = (f_1, \ldots, f_k)$ und $k < n$. Dann ist der 1-dimensionale Convolution-Operator $*$ wiefolgt definiert:

$$A * F := (a * f)_x = \sum_{i=1}^{k} a_{x+i-1} \cdot f_i \text{ mit } x \in \{1, \ldots, n-k+1\}$$

Betrachte zur Veranschaulichung folgende Beispielvektoren für A und F:

$$A = \begin{bmatrix} 10 \\ 50 \\ 60 \\ 10 \\ 20 \\ 40 \\ 30 \end{bmatrix} \text{ und } F = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix}$$

Offensichtlich ist $n = 7$ und $k = 3$. Dann hat der Ergebnis-Vektor nach Anwendung des Convolution-Operators auf A und F genau $n - k + 1 = 7 - 3 + 1 = 5$ Einträge. Gemäß der Definition ergibt sich:

$$A * F = \begin{bmatrix} (a*f)_1 \\ (a*f)_2 \\ (a*f)_3 \\ (a*f)_4 \\ (a*f)_5 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{3} a_{1+i-1} \cdot f_i \\ \sum_{i=1}^{3} a_{2+i-1} \cdot f_i \\ \sum_{i=1}^{3} a_{3+i-1} \cdot f_i \\ \sum_{i=1}^{3} a_{4+i-1} \cdot f_i \\ \sum_{i=1}^{3} a_{5+i-1} \cdot f_i \end{bmatrix} = \begin{bmatrix} 10 \cdot \frac{1}{3} + 50 \cdot \frac{1}{3} + 60 \cdot \frac{1}{3} \\ 50 \cdot \frac{1}{3} + 60 \cdot \frac{1}{3} + 10 \cdot \frac{1}{3} \\ 60 \cdot \frac{1}{3} + 10 \cdot \frac{1}{3} + 20 \cdot \frac{1}{3} \\ 10 \cdot \frac{1}{3} + 20 \cdot \frac{1}{3} + 40 \cdot \frac{1}{3} \\ 20 \cdot \frac{1}{3} + 40 \cdot \frac{1}{3} + 30 \cdot \frac{1}{3} \end{bmatrix} = \begin{bmatrix} 40 \\ 40 \\ 30 \\ 20 \\ 30 \end{bmatrix}$$

Auf den ersten Blick erschließt sich möglicherweise nicht der Sinn der vorangegangen veranschaulichten Convolution-Operation. Offensichtlich überlappt der Filter-Vektor stets einen Teilbereich der Eingabe. Stellt man sich den Eingabevektor A hingegen z.B. als Messkurve eines elektrischen Signales vor, könnte das Ergebnis der Faltung bei dieser Wahl von F z.B. eine Glättung des Signals verursachen. Extreme Ausreißer nach oben (60) oder unten (10) sind im Ergebnis nicht mehr enthalten. Eine andere Wahl von F hätte möglicherweise andere Informationen „heraus gefiltert“. Die Anwendung des Faltungs-Operators auf ein Eingabesignal unter Verwendung eines bestimmten Filters ist eine häufige Vorgehensweise in der Signalverarbeitung.

Im 2-dimensionalen Fall, der nachfolgend definiert und erläutert wird, erschließt sich diese Art von Verarbeitung noch anschaulicher.

2D-Convolution (Diskrete Faltung) Sei $A = (a_{ij}) \in K^{n \times n}$ eine Eingabematrix. Sei weiterhin $F = (f_{ij}) \in K^{k \times k}$ mit $k < n$ eine sogenannte Filtermatrix (auch Kernelmatrix genannt). Dann ist der 2-dimensionale Convolution-Operator $*$ wiefolgt definiert:

$$A * F := (a * f)_{xy} = \sum_{i=1}^{k} \sum_{j=1}^{k} a_{(x+i-1)(y+j-1)} \cdot f_{ij}$$

$$\text{mit } x, y \in \{1, \ldots, n-k+1\}$$

Betrachte zur Veranschaulichung folgende Beispielmatrizen für A und F:

$$A = \begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \text{ und } F = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Offensichtlich ist $n = 6$ und $k = 3$. Dann gilt für das Ergebnis:

$$A * F \in K^{(n-k+1) \times (n+k+1)} = K^{4 \times 4}$$

Aus Gründen der Übersichtlichkeit seien nachfolgend nur der erste der insgesamt 16 Einträge der Ergebnismatrix ausführlich berechnet:

$$\begin{aligned} (a * f)_{11} &= \sum_{i=1}^{k} \sum_{j=1}^{k} a_{ij} \cdot f_{ij} \\ &= 10 \cdot 1 + 10 \cdot 0 + 10 \cdot -1 + 10 \cdot 1 + 10 \cdot 0 + 10 \cdot -1 + 10 \cdot 1 + 10 \cdot 0 + 10 \cdot -1 \\ &= 0 \end{aligned}$$

Analog werden die anderen Einträge berechnet - daraus ergibt sich das Gesamtergebnis:

$$A * F = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

Anschaulich wurde die Filtermatrix zur Berechnung eines Teilergebnisses ähnlich der 1-dimensionalen Convolution über Teile der Eingabematrix gelegt. Im vorangegangenen Beispiel wurde F so gewählt, dass die Operation mit F Kanten in A erkennt. Dabei sollte A als eine Matrix bestehend aus Pixelfarbwerten interpretiert werden, wobei 10 z.B. einen sehr dunklen und 0 einen sehr hellen Farbton repräsentiert. Offensichtlich ist in der Ergebnismatrix die Kante in der

Mitte des „Bilder“ erkannt und entsprechen dargestellt worden (betrachte die Einträge mit Wert 30). Dies verdeutlicht das Grundprinzip von Bildverarbeitung mittels 2-dimensionaler Faltung. Je nach Wahl der Filtermatrix können neben der Kantenerkennung eine Vielzahl weiterer Effekte erzielt werden, z.B. Glättung, Schärfung oder Weichzeichnung [Her05].

Im vorangegangenen Beispiel wurde ein 3×3-Filter gewählt. Aufgrund der Konstruktion des Convolution-Operators ist das Ergebnis der Anwendung eine Filters solcher Größe auf eine $n \times n$-Eingabematrix stets kleiner - es entsteht eine $(n-2) \times (n-2)$-Matrix. Unter Umständen kann jedoch ein Beibehalten der ursprünglichen Größe gewünscht sein. Um dies zu erreichen, kann ein Rand (engl. *Padding*) um die Eingabematrix hinzugefügt, also die tatsächliche Eingabegröße um 1 in jeder Dimension erhöht werden. Das Ergebnis hat dann wiederum die Ursprungsabmessungen. Welche Werte die Matrixeinträge am Rand einnehmen soll, hängt vom Filter ab - diverse Lösungsansätze wurden z.B. in der Bildverarbeitung entwickelt.

Entgegen der Anforderung, ein Ergebnis mit gleichbleibender Größe zu erhalten, kann auch ein deutlich kleineres Ergebnis gewünscht sein. Dies kann beispielsweise der Kompression oder Reduzierung der Anzahl zu lernender Parameter dienen. Eine einfache Art, diese Anforderung zu erfüllen, ist die Schrittweite in der Definition der Faltung zu erhöhen, i.e. den Summenindex in jedem Schritt um mehr als 1 zu erhöhen. Eine solche Art der Faltung wird auch *Strided Convolution* genannt.

3.3 Aufbau von Convolutional Neural Networks

Nachdem im vorangegangenen Abschnitt der Convolution-Operator ausführlich eingeführt wurde, wird nun der Aufbau von Convolutional Neural Networks u.a. unter Verwendung von lokal verbundenen neuronalen Netzen und der Idee des Convolution-Operators erläutert.

Ein Convolutional Neural Network ist ein teilweise lokal verbundenes neuronales Feedforward-Netz, welches zu allermeist aus folgenden Schichten zusammengesetzt ist:

1. Convolutional Layer
2. Pooling Layer
3. Vollständig-verbundener Layer

Dabei folgt ein Pooling Layer i.A. auf einen Convolution Layer - diese Paarungen sind wiederum mehrfach hintereinander geschaltet. Sowohl Pooling Layer als auch Convolution Layer sind lokal verbundene Teilnetze, d.h. die Anzahl an zu lernenden Gewichten hält sich auch bei sehr großen Eingabedaten (z.B. Bildern) in Grenzen. Üblicherweise folgt abschließend ein vollständig-verbundener Layer, dessen Anzahl an eingehenden Neuronen-Verbindungen durch entsprechende Konstruktion der vorangegangenen Layer jedoch deutlich geringer als die Anzahl an Eingabe-Neuronen in der Eingabeschicht ist. Nachfolgend wird detaillierter auf die einzelnen Schichten eingegangen.

Convolutional Layer Wie der Name bereits sagt, ist diese Schicht an die mathematische Faltung (Convolution) von Eingabesignalen angelehnt. Der Convolutional Layer bildet dabei den Convolution-Operator ab, d.h. jedes Neuron in einem Convolutional Layer berechnet genau einen Eintrag der Ergebnis-Matrix. Dies ist möglich, da die Berechnung der Faltung analog zur Konzeption von neuronalen Netzen auf gewichteten Summen basiert. Soll die Filter-Matrix $3 \times 3 = 9$ Einträge habe, muss folglich jedes der Neuronen im Convolutional Layer mit genau 9 Eingabeneuronen verbunden sein. Daraus folgt, dass die Schicht nicht vollständig zur Eingabe, sondern lokal verbunden ist. Angenommen die Berechnung an einem Neuron soll auf Grundlage einer fest definierten Filter-Matrix erfolgen - dann entsprechen die Gewichte der Verbindungen zu der Eingabe genau den Einträgen der Filter-Matrix. Die Hauptidee von Convolutional Neural Networks ist es nun, dass *die Einträge der Filter-Matrix F nicht vorab feststehen, sondern überwacht unter Verwendung eines Trainingsdatensatzes für den gewünschten Zweck gelernt werden.* Die trainierten Filter bzw. Convolutional Neural Networks können anschließend zur Klassifizierung der Eingabesignale genutzt werden, z.B. um für ein beliebiges Bild die Aussage treffen zu können, ob ein bestimmtes Objekt zu erkennen ist.

Betrachten wir zur Veranschaulichung einen Filter-Vektor F mit 3 zunächst unbekannten Gewichten, der für 1D-Convolution verwendet werden soll:

$$F = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

Angenommen es soll Strided Convolution mit einer Schrittweite von 2 durchgeführt werden. Dann lässt sich die Convolution mittels des in Abbildung 3.2 dargestellten Convolutional Layers berechnen.

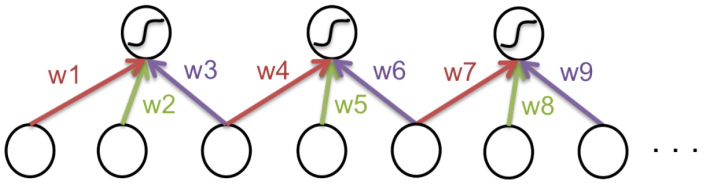


Abbildung 3.2: 1D-Faltung als KNN [L+15b]

Zur weiteren Verarbeitung wird das Ergebnis der Convolution an jedem Neuron zusätzlich von einer Aktivierungsfunktion verarbeitet. Offensichtlich hat das in der Abbildung dargestellte lokal verbundene Netz insgesamt 9 neuronale Verbindungen. Gemäß der Definition der Convolution wird für die Berechnung des Ergebnisses ein fester Filter verwendet, d.h. für die Gewichte in diesem Beispiel muss gelten:

$$w_1 = w_4 = w_7 \text{ und } w_2 = w_5 = w_8 \text{ und } w_3 = w_6 = w_9$$

Die Gleichheit der Gewichte ist in der Abbildung farblich gekennzeichnet. Offensichtlich ergibt sich eine zusätzliche Nebenbedingung für das Training von Convolutional Neural Networks: die Gleichheit bestimmter Gewichte. Desweiteren ist der Speicherbedarf i.A. geringer, da weniger (unterschiedliche) Gewichte gespeichert werden müssen.

Um die Nebenbedingung gleicher Gewichte zu erfüllen, genügt schon eine minimale Änderung am Backpropagation-Algorithmus:

1. Im Vorwärtsdurchlauf können alle w_i mit ihren konkreten Werten betrachtet werden, unabhängig davon, ob manche von ihnen gleiche Werte haben, oder nicht.
2. Im Rückwärtsdurchlauf können weiterhin für alle w_i die Gradienten $\frac{\partial J}{\partial w_i}$ berechnet werden.
3. Beim Aktualisieren der Gewichte werden diejenigen, die laut Nebenbedingung gleich bleiben müssen (z.B. $w_1 = w_4 = w_7$), identisch angepasst. Um den Einfluss jedes der Gewichte zu berücksichtigen, wird dazu der Durchschnitt aller betroffenen Gradienten gebildet. Es ergibt sich dadurch z.B. für w_1:

$$w_1 = w_1 - \alpha(\frac{\partial J}{\partial w_1} + \frac{\partial J}{\partial w_4} + \frac{\partial J}{\partial w_7})$$

Offensichtlich muss für das Training eines Convolutional Layers nur der letzte Schritt des Backpropagation-Algorithmus angepasst werden.

Pooling Layer Der Pooling-Layer (auch Subsampling-Layer genannt) besteht aus Neuronen, die zumeist den Maximum-Operator auf der Menge ihrer Eingabewerte ausführen. Der Operator in dieser Schicht wird i.A. auf das Ergebnis (die Ergebnis-Matrix) eines Convolutional Layers angewendet. Das nachfolgende Beispiel verdeutlicht die Funktionsweise dieses Operators bzw. dieser Schicht: Offensichtlich ist der linke obere Eintrag des Ergebnisses (9) das Maximum der vier linken oberen Werte ($\{1, 3, 2, 9\}$) der Eingabe-Matrix.

$$\begin{bmatrix} 1 & 3 & 2 & 1 \\ 2 & 9 & 1 & 1 \\ 1 & 3 & 2 & 3 \\ 5 & 6 & 1 & 2 \end{bmatrix} \rightarrow_{MAX} \begin{bmatrix} 9 & 2 \\ 6 & 3 \end{bmatrix}$$

Das Beispiel aus dem Abschnitt über den Convolutional Layer fortführend, zeigt Abbildung 3.3 einen hinter den Convolutional Layer geschalteten Pooling Layer. Dabei werden aus dem 1-dimensionalen Ergebnis der Faltung jeweils zwei Werte im Sinne ihres Maximums zusammengefasst.

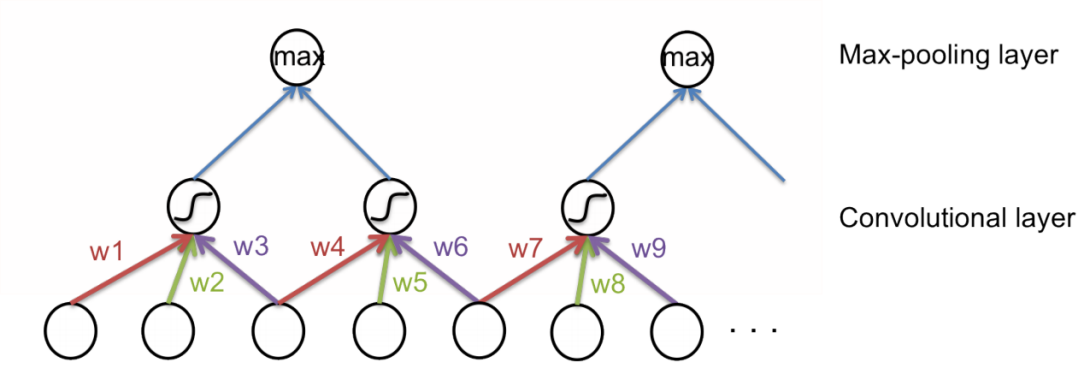
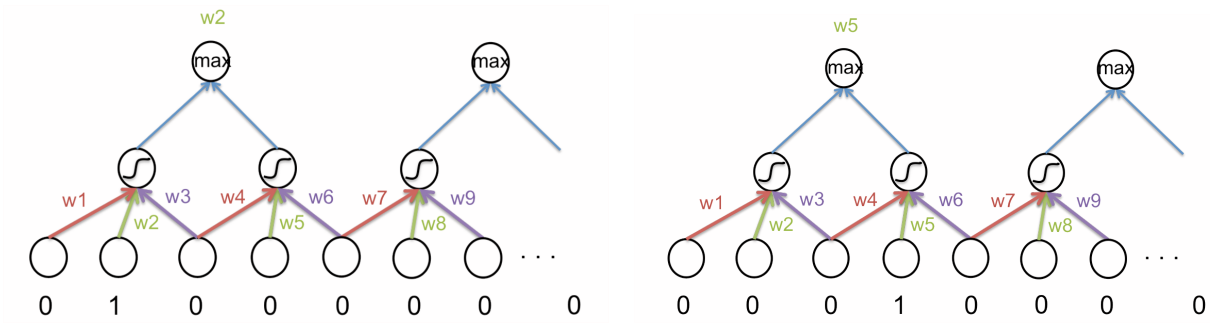


Abbildung 3.3: Beispiel Pooling-Layer [L+15b]

An diesem Beispiel wird bereits eine der intendierten Funktionen des Pooling-Layers deutlich: Das Ergebnis einer Convolution wird vergröbert und überflüssige Informationen werden optimalerweise verworfen. So genügt bei hochdimensionalen Daten oftmals die grobe Information, wo genau sich in den Daten ein Feature befindet. Vielmehr ist häufig eine Erkenntnis der Form „Das eingegebene Bild enthält links oben eine rote Ampel“ von Interesse. Die Verwendung des Maximum-Operators hat sich in der Praxis als äußert vielversprechend für solche Klassifizierungs-Aufgaben erwiesen. Statt des Maximums kann auch der Durchschnitt gebildet werden, diese Variante erreicht i.A. allerdings etwas schlechtere Ergebnisse [SMB10].

Ein weiterer Nutzen des Pooling-Layers ist die Toleranz gegenüber gewissen Verschiebungen in den Eingabedaten - die sogenannte *Translationsinvarianz*. Die Translationsinvarianz bedeutet, dass bei einer Verschiebung derselben Eingabedaten möglichst die gleiche Ausgabe (im Rahmen einer Klassifikation) erreicht wird. Diese Verschiebung ist im Bereich von Rohdaten wie Ton oder Bild durchaus vorhanden - man stelle sich verschiedene Positionen beim Fotografieren oder leichten zeitlichen Versatz bei Tonaufnahmen vor. Ein sehr einfaches Beispiel zur Verschiebung von Eingabedaten ist in Abbildung 3.4 gegeben: Die einzige 1 in den Eingabedaten (a) wird um zwei Positionen nach rechts verschoben (b) - die Ausgabe (in diesem Fall die 1) bleibt aufgrund der Verwendung des Max-Operators gleich. Die Gleichheit der Werte ist in der Abbildung farblich (grün) gekennzeichnet.



(a) Originale Eingabe (b) Verschobene Eingabe

Abbildung 3.4: Beispiel Translationsinvarianz im Pooling-Layer [L+15b]

Eine offensichtliche Eigenschaft des Pooling Layers ist, dass keine Gewichte für die neuronalen Verbindungen gelernt werden müssen. Das Neuron führt nur eine einfache statische Funktion aus, die sich leicht implementieren lässt. Während des Trainings ist zu beachten, dass im Vorwärtslauf gespeichert werden muss, welche Pfade im Netz die Maximums-Operationen „gewonnen“ haben. Nur die Gewichte neuronaler Verbindungen, die auf solchen Pfaden liegen, müssen in der Ausführung des Backpropagation-Algorithmus aktualisiert werden.

Vollständig-verbundener Layer Auf den letzten Pooling-Layer in einem Convolutional Neural Network folgen häufig einige schichtweise vollständig-verbundene Layer. Zumeist nimmt dann die Anzahl an Neuronen schrittweise bis zur benötigten Ausgabedimension ab. Diese Schichten sind als klassische Klassifizierung-Schicht nicht-linearer Daten zu verstehen - nicht zuletzt da die letzte solche Schicht die Ausgabeschicht des gesamten Netzes ist. An diese werden während des überwachten Trainings die Klassen des Trainingsdatensatzes angelegt.

3.4 Erweiterungen

Der Vollständigkeit halber seien zum Abschluss der Einführung in Convolutional Neural Networks zwei Erweiterungen der bisher vorgestellten Architektur genannt. Wie bereits im Abschnitt über die mathematische Definition von Convolution ersichtlich wurde, ist es möglich, sowohl auf 1-dimensionalen als auch auf 2-dimensionalen Eingabedaten eine Faltung durchzuführen. Darüberhinaus können auch weitere Dimensionen hinzukommen - z.B. im Bereich der Signal- oder Bildverarbeitung verschiedene *Kanäle*. In der Bildverarbeitung können dies z.B. die verschiedenen Farbkanäle sein, i.e. jeweils ein Kanal für den Rot-, Gelb- und Blauwert. Abbildung 3.5 zeigt ein kleines Convolutional Neural Network, welches 1-dimensionale Strided Convolution auf zwei Eingabekanälen ausführt. Offensichtlich stellt dies keine nennenswerte Herausforderung bei der Konstruktion eines CNN dar.

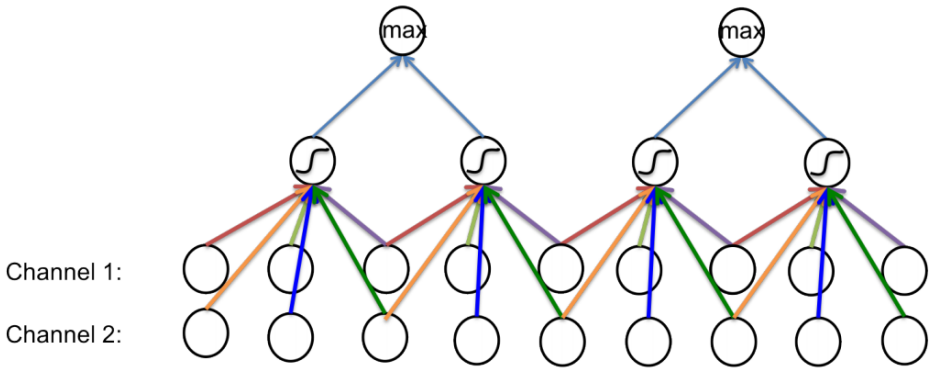


Abbildung 3.5: Beispiel 2-dimensionales CNN [L+15b]

Neben der Verarbeitung verschiedener Eingabekanäle kann ein Convolutional Neural Network auch Abbildungen mehrerer Convolution-Operationen innerhalb einer Schicht beinhalten. So könnte z.B. eine Operation auf Kantenerkennung und eine andere auf Weichzeichnen trainiert werden. Die Ergebnisse einer solchen Operation werden häufig als *Map* bezeichnet. Abbildung 3.6 zeigt ein CNN, welches im Convolutional Layer zwei Faltungs-Operationen enthält. Es müssen also zwei Filter-Vektoren mit jeweils 3 Gewichten gelernt werden.

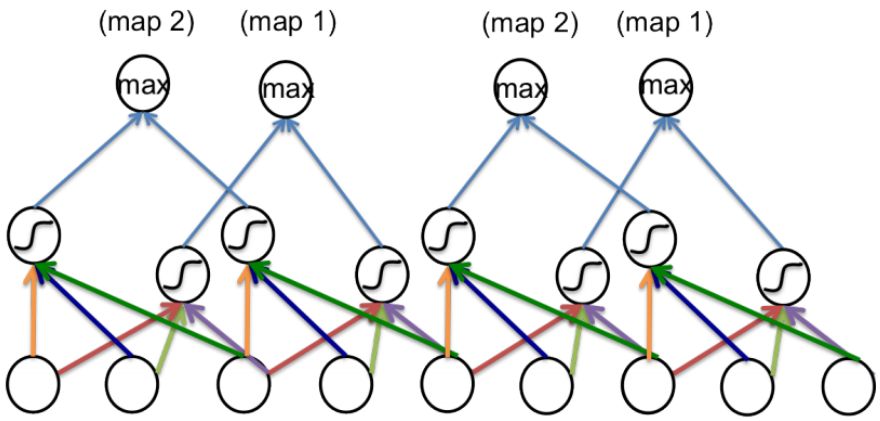


Abbildung 3.6: Beispiel CNN mit zwei Faltungs-Operatoren (Maps) [L+15b]

3.5 Anwendungen

Convolutional Neural Networks der vorangehend erläuterten Gestalt liefern in verschiedensten Bereichen hervorragende Klassifizierungs-Ergebnisse. Inbesondere zur Erkennung von handgeschriebenen Zeichen konnten LeCun et. al. in [LBBH98] mit einer Architektur namens LeNet-5 sehr gute Ergebnisse liefern. Dies hat sämtliche CNN-Architekturen bzw. deren Erfolg begründet. Abbildung 3.7 zeigt diese Architektur. Offensichtlich tauchen alle in dieser Arbeit besprochenen Schichten (Convolutional, Pooling, vollständig verbunden) sowie die Verwendung mehrerer Maps auf.

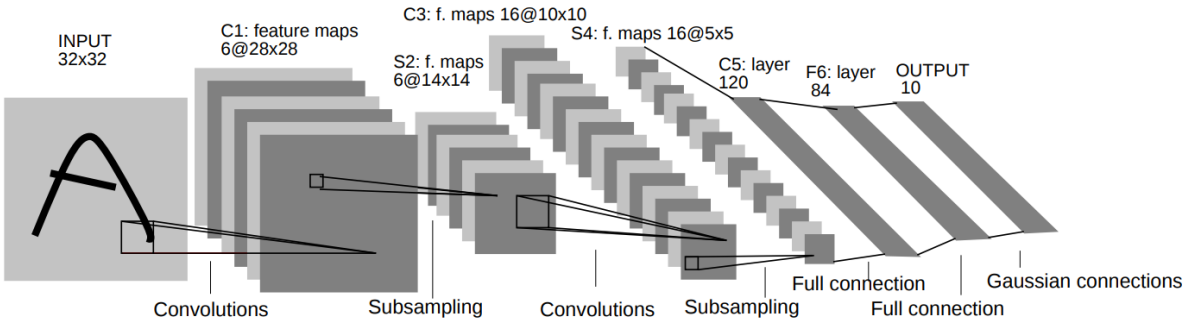


Abbildung 3.7: Architektur von LeNet-5 [LBBH98]

Großen Erfolg bei der Klassifikation von hoch aufgelösten Bildern mit Hilfe eines CNN konnten Krizhevsky et. al. in [KSH12] erzielen. Die Klassifikation zielte dabei auf die Erkennung bestimmter Objekte auf den Bildern ab (Objekterkennung). Die Architektur dieses CNN zeigt Abbildung 3.8 - abermals lassen sich alle kennengelernten Komponenten und Konzepte wiederfinden.

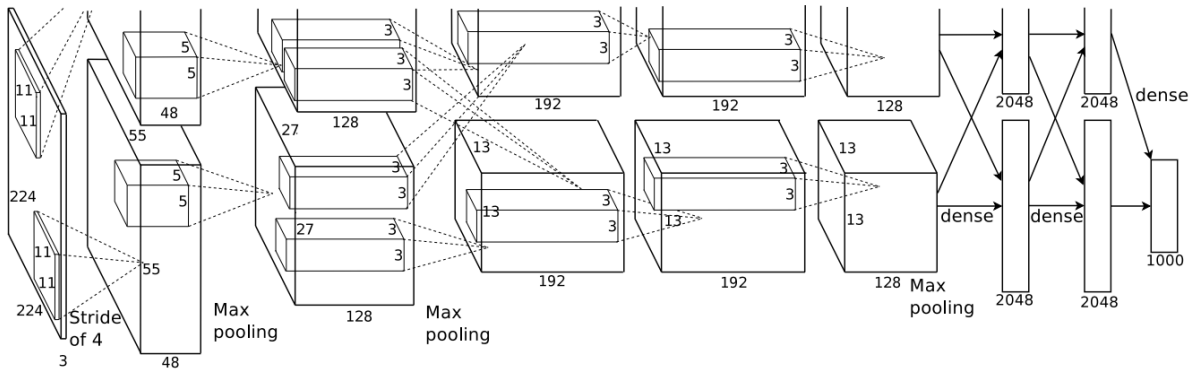


Abbildung 3.8: Architektur Objekterkennung in Bildern [KSH12]

Neben der Verarbeitung von Bilddaten erfreuen sich CNN auch im Bereich der Spracherkennung (Natural Language Processing) hoher Beliebtheit. So werden beispielsweise in [Kim14] natürlichsprachliche Sätze klassifiziert. Populäre praktische Anwendung fand Google Inc. für CNNs unter anderem bei der Implementierung der Smartphone Sprachsteuerung [SP15], bei der Sprach-Roh-Daten in maschinenfreundliche Daten umgewandelt werden müssen. Zudem wurden CNNs in der Implementierung von AlphaGo [SHM+16] verwendet.

4 Zusammenfassung

In dieser Arbeit wurden zwei spezielle Formen künstlicher neuronaler Netze eingeführt.

Autoencoder dienen der Dimensionsreduzierung und werden darauf trainiert, eine umkehrbare niedrig-dimensionale Kodierung eines Eingabedatums zu lernen. Die berechneten Einbettungen finden vielerlei praktische Anwendung, z.B. Kompression oder Visualisierung. Im Vergleich zur PCA haben Autoencoder auf Basis von KNN bessere Ergebnisse erzielt. Darüberhinaus werden Autoencoder erfolgreich als Initialisierungsmethode für Gewichte von allgemeinen künstlichen neuronalen Netzen verwendet.

Convolutional Neural Networks bestehen grundsätzlich aus Convolutional-, Pooling- und vollständig verbundenen Schichten. Die Convolutional-Schicht bildet dabei den mathematischen Faltungs-Operator aus der Bild- und Signalverarbeitung ab. Während des Trainings werden die der Faltungs-Operation zugrunde liegenden Filter gelernt. CNNs sind heutzutage *state of the art* im Bereich der Klassifizierung von hoch-dimensionalen natürlich Daten, z.B. der Objekterkennung in Bildern.

Literaturverzeichnis

[Bal12] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, pages 37–49, 2012.

[EBC+10] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

[Gau77] Carl Friedrich Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*, volume 7. FA Perthes, 1877.

[Her05] Thorsten Hermes. Digitale bildverarbeitung. *Digitale Bildverarbeitung—Eine praktische Einführung*, pages 134–135, 2005.

[HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[Kim14] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[L+15a] Quoc V Le et al. A tutorial on deep learning part 1: Nonlinear classifiers and the backpropagation algorithm, 2015.

[L+15b] Quoc V Le et al. A tutorial on deep learning part 2: Autoencoders, convolutional neural networks and recurrent neural networks. *Google Brain*, 2015.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LCB10] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[Pea01] K Pearson. On lines and planes of closest fit to systems of points in space, philosophical magazine2 (6): 559–572, 1901.

[SH94] Ferdinando S Samaria and Andy C Harter. Parameterisation of a stochastic model for human face identification. In *Applications of Computer Vision, 1994., Proceedings of the Second IEEE Workshop on*, pages 138–142. IEEE, 1994.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[SK87] Lawrence Sirovich and Michael Kirby. Low-dimensional procedure for the characterization of human faces. *Josa a*, 4(3):519–524, 1987.

[SMB10] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pages 92–101. Springer, 2010.

[SP15] Tara N Sainath and Carolina Parada. Convolutional neural networks for small-footprint keyword spotting. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.