

Seminararbeit

Convolutional Neural Networks

von

Bastian Bertholdt (319560),
Ajit Parikh (387730)

Dozent: Prof. Dr.-Ing Reinhold Orglmeister
Betreuer: M.Sc. Michael Klum

Fachbereich Elektronik und medizinische Signalverarbeitung
Institut für Energie- und Automatisierungstechnik

Berlin, Januar 2019

Inhaltsverzeichnis

Abbildungsverzeichnis	ii
1 Motivation	1
1.1 Berichtstruktur	1
2 Topologie	2
2.1 Convolutional Layer	3
2.2 Pooling Layer	7
2.3 Fully-connected Layer	8
2.4 Dropout	8
2.5 Transpose Convolution oder Deconvolution	9
3 Backpropagation in CNNs	10
4 Varianten	13
4.1 Inception	13
4.2 DensNet	14
5 Geschichte	15
6 Zusammenfassung und Ausblick	16
Literatur	17
Anhang	18

Abbildungsverzeichnis

2.1	MLP zu CNN (modifiziert von [5])	2
2.2	Simple CNN Architektur (By Aphex34 CC BY-SA 4.0)	3
2.3	Berechnung der Pixelwerte der Feature map (modifiziert von [1])	4
2.4	Effekte verschiedener Filterkernel	5
2.5	Convolutional Layer mit 10 Filter mit jeweils einer Tiefe von 3 (zwei davon: rot/grün) [2].	6
2.6	ReLU vs Parametric ReLU ('Leaky ReLU' für $a = 0.01$) [11]	7
2.7	Beispielhafte Darstellung von max pooling (modifiziert von [2])	8
2.8	Dropout visualisiert (modifiziert von [2])	9
3.1	Full Convolution [10]	12
4.1	Naive Inception, es werden 1x1, 3x3 und 5x5 Faltung sowie 3x3 max pooling in einem Ausgang zusammengefasst [12]	13
4.2	Inception mit Bottleneck Layer [12]	14
4.3	Feature Propagation in einem DensNet Block [7]	15
4.4	Struktur eines DensNet [7]	15

1 Motivation

Künstliche neuronale Netze haben in den vergangenen Jahren in vielen Anwendungsbereichen zu Erfolgen beim maschinellem Lernen geführt. Vorallem im Bereich der Bilderkennung und -klassifizierung wurden große Fortschritte gemacht. Zum einen, weil man heutzutage Zugang zu umfangreichen Datensätzen an beschrifteten Trainingsdaten¹ hat, zum anderen durch neue Verfahren und Arten von neuronalen Netzen ansich. Eine Klasse von neuronalen Netzen die sich besonders gut für Bildklassifizierung eignet, sind die sogenannten **Convolutional neural networks (CNN oder ConvNet)**. Sie zeichnen sich durch eine Deep Learning Architektur mit mehreren Schichten aus und sind inspiriert durch biologische Prozesse im visuellen Cortex von Tieren [8]. Eine wichtige Aufgabe von CNN ist es, die Verarbeitung von großen räumlich korrelierten Datensätzen, wie zum Beispiel Bilder, zu vereinfachen. Bilder können heutzutage aus Millionen von Pixeln bestehen, was zu einem erheblichen Rechenaufwand sowie Speicherplatzverbrauch für "normale" Neuronale Netze führen kann. Innerhalb einer Schicht mit m Inputs und n Outputs berechnet sich der daraus entstehende Aufwand zu $\mathcal{O}(m \times n)$. Um dieses Problem zu vermeiden werden nicht jedem Neuron einer Schicht die Ausgänge aller Neuronen der vorherigen Schicht zugewiesen, sondern eben nur eine bestimmte Anzahl k , die einige Größenordnungen kleiner als m sein kann. So reduziert sich der Aufwand zu $\mathcal{O}(k \times n)$ pro Schicht ohne wesentlich an Effizienz einzubüßen (O-Notation, Beispiel aus [9], S. 350). Diese Eigenschaft wird ausgenutzt um Rechenaufwand und Speicherplatz zu sparen [9].

1.1 Berichtstruktur

Im folgenden Kapitel wird der Aufbau eines Convolutional neural networks beschrieben. Im Einzelnen werden die verschiedenen Schichten und die zugrundeliegenden mathematischen Operationen erläutert und Techniken wie Dropout und Transpose convolution beschrieben. Im darauf folgenden Kapitel setzen wir uns damit auseinander, wie Backpropagation in CNNs funktioniert. Anschließend werden wir verschiedene Architekturvarianten vorstellen und kurz auf die geschichtlichen Ursprünge von CNNs eingehen. Letztlich stellen wir unsere, im Rahmen dieses Seminars erstellte, eigene Anwendung vor. Diese wurde in Form eines Ipython Notebooks erstellt und ist daher in diesem Dokument nur zum Lesen als Anhang verfügbar.

¹z.B. ImageNet: <http://image-net.org/index>

2 Topologie

Wie bei einem typischen Feed-Forward Netzwerk wird die Eingabe durch mehrere verschiedene aufeinanderfolgende Schichten (*Layer*) verarbeitet. Jedoch unterscheiden sich CNNs von typischen Netzen (wie zum Beispiel dem Multilayer Perceptron) im Wesentlichen durch drei Eigenschaften:

- **Lokale Vernetzung:** Neuronen sind nur teilweise mit Neuronen der vorangegangenen Schicht verknüpft (in einer lokalen Region).
- **Geteilte Gewichte:** Die Parametrisierung/Gewichtung dieser Verknüpfung ist für alle Neuronen innerhalb einer Schicht (pro Filter) gleich.
- **Pooling:** Der Zweck bestimmter Schichten ist es, lediglich die Neuronenanzahl zu verringern (*subsampling*), indem Regionen der vorangegangenen Schicht gebündelt werden.

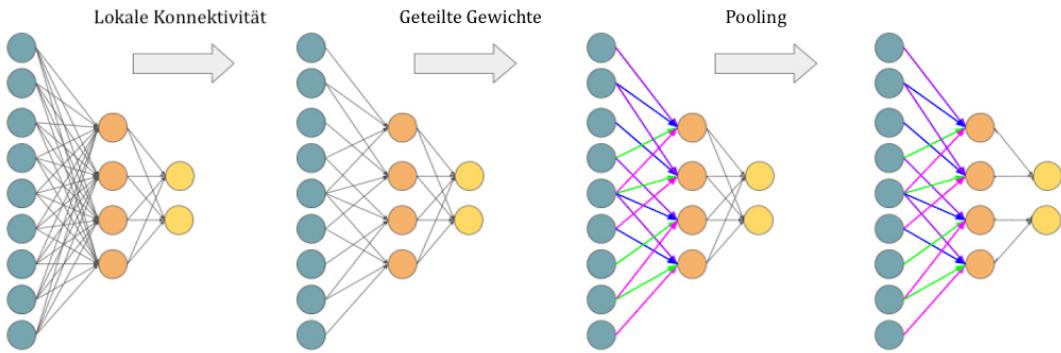


Abbildung 2.1: MLP zu CNN (modifiziert von [5])

Diese ersten beiden Eigenschaften werden effizient durch Faltung (*Convolution*) umgesetzt, welche den CNNs ihren Namen verleiht. Die Ausgabe jeder Schicht ist eine Sammlung sogenannter *feature maps*. Ein Eintrag in einer *feature map* ist als Neuron zu verstehen. Somit sind Neuronen in CNNs dreidimensional angeordnet. Bei den Schichten handelt es sich im Wesentlichen um Filterschichten (*Convolutional Layer*) und Bündelungsschichten (*Pooling/ Subsampling Layer*). Ein Convolutional Layer hat eine feste Anzahl an Filtern mit denen die Eingabe parallel gefaltet wird (siehe Abschnitt 2.1). Dadurch wird die gleiche Anzahl an *feature maps* generiert. Nach einer Transformation dieser *feature maps* durch eine nicht lineare Aktivierungsfunktion (z.B. *ReLU*), durchlaufen sie den Pooling Layer (siehe Abschnitt 2.2), der sie wie eine Art Sieb in ihrer Größe reduziert. Dieser Prozess kann theoretisch beliebig oft und in unterschiedlichen Varianten wiederholt werden bis am Ende die *feature maps* zu einem einzelnen Vektor verarbeitet werden (*flattening*), der dann die Eingabe eines vollständig vernetzten neuronalen Netzes (*Fully-connected Layer* - siehe Abschnitt 2.3) bildet.

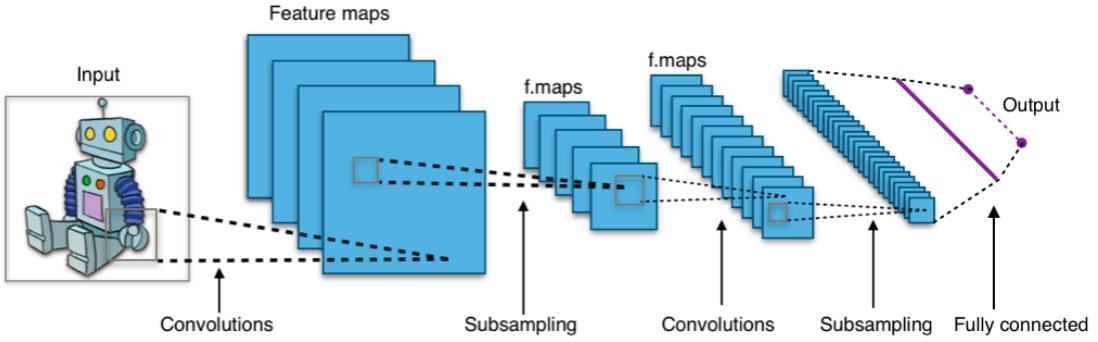


Abbildung 2.2: Simple CNN Architektur (By Aphex34 CC BY-SA 4.0)

2.1 Convolutional Layer

Der Convolutional Layer führt mehrere verschiedene Faltungen der Input-matrix durch. Eine Faltung (*Convolution*) ist ein mathematischer Operator der für zwei Funktionen $f, g : \mathbb{R}^n \rightarrow \mathbb{C}$ eine dritte Funktion ($f * g$) berechnet.

$$(f * g)(x) = \int_{\mathbb{R}^n} f(\tau)g(x - \tau)d\tau \quad (2.1)$$

Für den Fall, dass f und g nur auf einen beschränkten Bereich \mathbb{D} definiert sind, setzt man die Funktionen auf den ganzen Raum fort: z.B. durch $f|_{\mathbb{R} \setminus \mathbb{D}} \equiv 0$. Für Convolutional networks ist das erste Argument als Eingabematrix, und das zweite Argument als sogenannter *Kernel* oder Filter zu verstehen. Die Resultat der Faltung der Eingabe mit dem Kernel nennen wir *Feature map*. Sei I die Eingabe, K der Kernel und F die Feature map mit $I, K, F \in \mathbb{R}^2$. $I(m, n)$ ist zu verstehen als der Wert in I und der Position m, n . Die Faltung ist somit wie folgt definiert:

$$F(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n) \quad (2.2)$$

Faltung ist kommutativer Operator, was bedeutet dass:

$$F(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n) \quad (2.3)$$

Bildlich kann man sich das so vorstellen, dass der Kernel über jede Position der Eingabe gleitet. An jeder Position wird eine elementweise Multiplikation durchgeführt und alles zu einem Wert summiert. Genaugenommen muss die Kernelmatrix vorher um 180° gedreht werden (*kernel flip*). Tut man dies nicht, berechnet man keine Faltung, sondern eine Kreuzkorrelation (*cross-correlation*):

$$F(i, j) = (I \star K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n) \quad (2.4)$$

Der Unterschied ist, dass die Kreuzkorrelation im Gegensatz zur Faltung nicht kommutativ ist. Im Kontext von Machine Learning spielt es jedoch keine Rolle,

ob man den Kernel flippt, denn die Kernelwerte werden gelernt, was bedeutet, dass der Algorithmus mit Faltung genauso funktioniert wie mit Kreuzkorrelation. Lediglich die gelernten Kernel werden sich von der Orientierung her unterscheiden. In Beispielen und Veranschaulichungen wird von nun an davon ausgegangen, dass der Kernel entweder schon geflippt wurde oder es sich um eine Convolution-variante ohne flipping handelt.

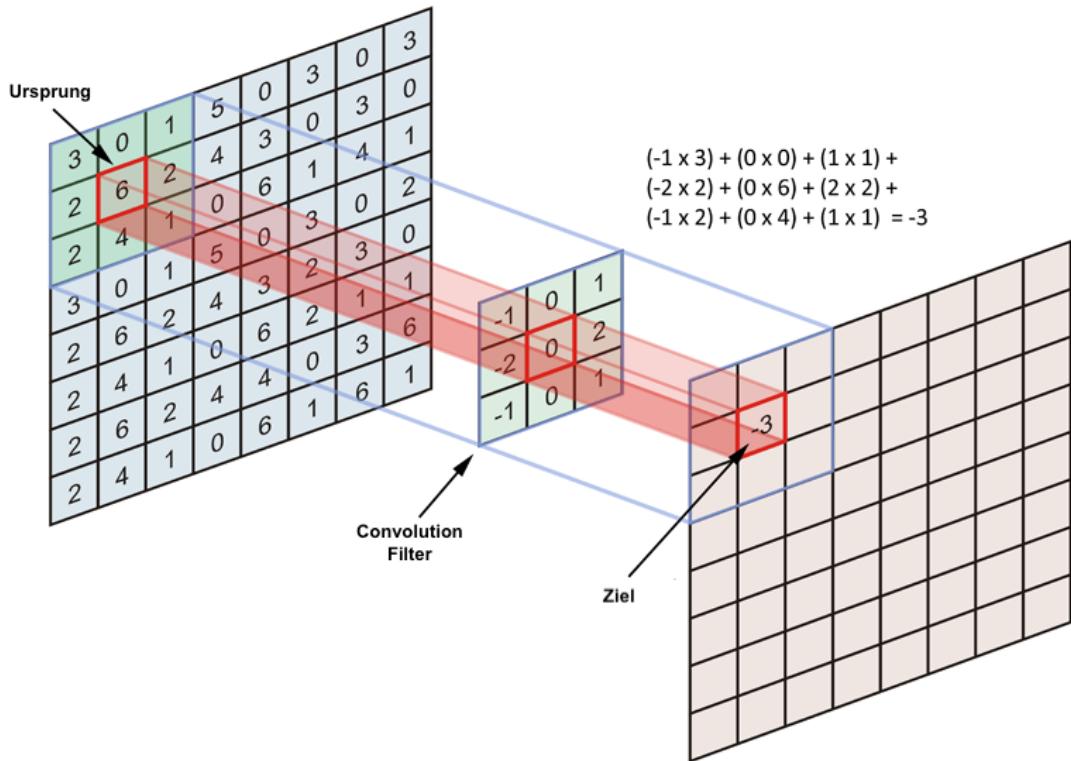


Abbildung 2.3: Berechnung der Pixelwerte der Feature map (modifiziert von [1])

Die aus der Faltung resultierende Feature map ist wiederum eine Matrix. Diese Matrix hat nicht zwingend dieselbe Größe wie die Eingabe, sondern hängt von der Größe des Kernels (*kernel-size*), der Schrittweite de Kernels (*stride*) und dem Verhalten an den Rändern der Matrix (z.B. *padding*) ab.

Verschiedene Kernel haben verschiedene Effekte (siehe Abbildung 2.4). In manchen Fällen kann das Resultat verstanden werden als eine Karte (*Feature map*), die an jeder Bildposition aufzeigt, zu welchem Grad eine bestimmte Eigenschaft an dieser Stelle vorhanden ist. Je nach Kernelgröße und den jeweiligen Kernelwerten, können diese Eigenschaften einfache Strukturen wie Linien mit einer bestimmten Orientierung, Kanten bis hin zu komplexeren Strukturen wie Teile von Gesichtern sein. In einem Convolutional neural networks sind die Kernelwerte die Gewichte die zusammengegerechnet werden mit den Werten eines Ausschnitts der Eingabe (rezeptives Feld) und somit die Aktivierung eines Neurons (korrespondierender Pixelwert der Ausgabe) bilden.

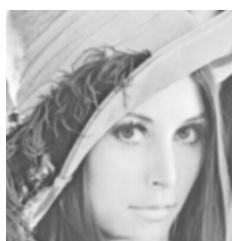
Name	Kernel	Resultat
Identität	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Sobel (links)	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	
Sobel (rechts)	$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$	
Kanten	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Verwischung	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	
Verschärfung	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	

Abbildung 2.4: Effekte verschiedener Filterkernel

Bisher haben wir uns Faltung im 2-dimensionalen Raum angesehen. Aber die Eingabe (und somit auch der Filterkernel) kann eine dritte Dimension (Tiefe) haben. Ein Farbbild hat bspw. eine Tiefe von drei (Pixelwerte für rot, grün und blau). Ein Filterkernel der eine Faltung mit diesem Bild vornimmt, hat demnach auch eine Tiefe von drei, bewegt sich jedoch nur in Höhe und Breite entlang des Bildes. Zur Berechnung des Ausgabewertes werden jedoch alle Werte (auch die, in Richtung der Tiefe) zusammengerechnet. Ein Convolutional Layer hat jedoch meist mehrere Filterkernle. Die Ergebnismatrizen der Faltung mit jedem Filter werden gestapelt. Das bedeutet, dass wenn ein Convolutional Layer n Filterkernel hat, so hat deren Ausgabe ebenfalls eine Tiefe von n .

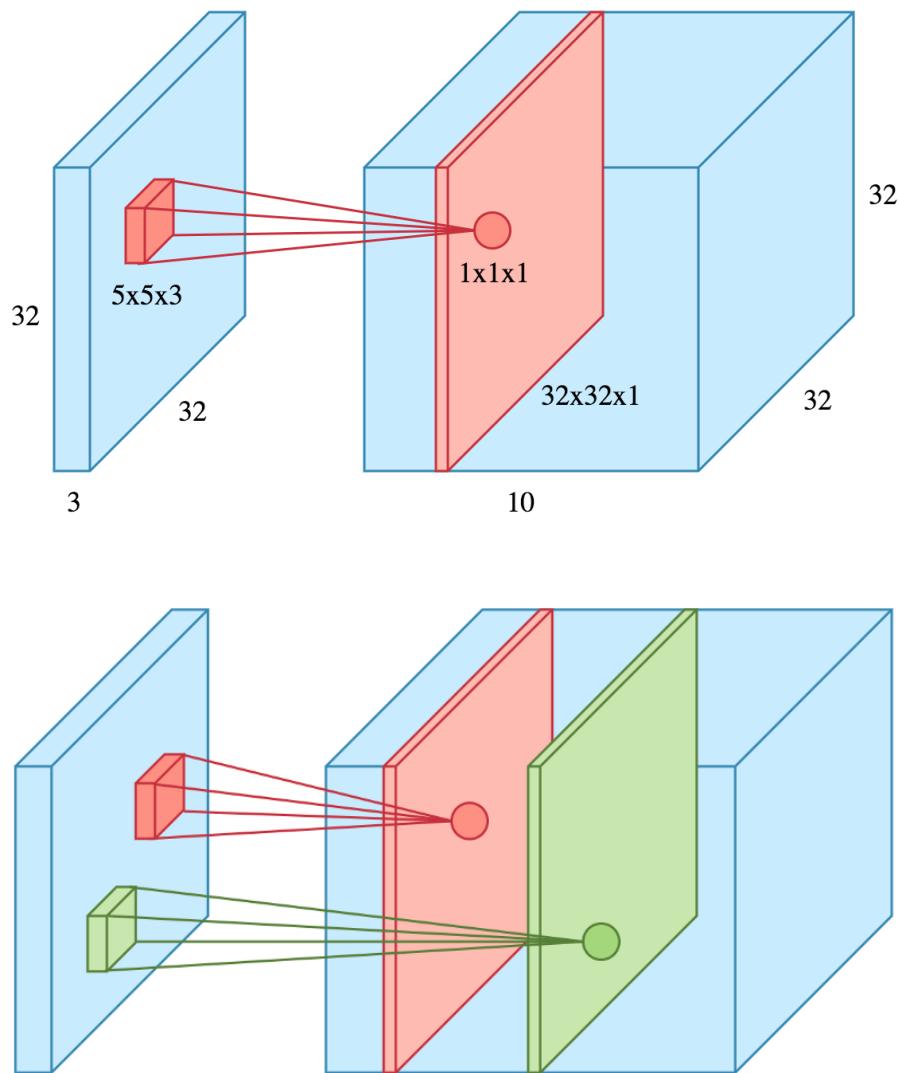


Abbildung 2.5: Convolutional Layer mit 10 Filter mit jeweils einer Tiefe von 3 (zwei davon: rot/grün) [2].

Die Ausgabe in Abbildung 2.5 könnte theoretisch die Eingabe eines weiteren Convolutional Layers sein. Allerdings haben die Filter der nächsten Schicht eine

Tiefe von 10. Das bedeutet die Tiefe der Filter ist bestimmt durch die Anzahl der Filter in der vorherigen Filterschicht.

Convolution ist eine lineare Operation, oder besser gesagt, eine Operation, die dieselbe lineare Operation an kleineren lokalen Regionen über die gesamte Eingabe ausführt. Wie bei jedem neuronalen Netz bedarf es jedoch an **Nicht-Linearität**. In typischen Netzen wird dies erreicht indem man die Ausgabe der Neuronen durch eine nicht lineare Aktivierungsfunktion transformiert. Bei CNNs ist das nicht anders. Eine häufig verwendete Funktion verwendet ist die *Rectified Linear Unit* (ReLU). Sie ist definiert als

$$f(x) = \begin{cases} 0, & \text{if } x < 0. \\ x, & \text{sonst} \end{cases} \quad (2.5)$$

Die Funktion wird häufig in Deep Learning Architekturen verwendet, vor allem weil sie im Gegensatz zu anderen gängigen Aktivierungsfunktionen wie z.B die Sigmoid-funktion leicht zu berechnen ist. Für das Training des Netzes ist es ebenfalls wichtig, dass die Funktion differenzierbar ist. ReLU's Ableitungen sind leicht zu berechnen (0 für $x < 0$ und 1, sonst). Lediglich bei $x = 0$ ist die Funktion nicht differenzierbar. Das wird jedoch meistens dadurch behoben, dass man dafür einen Wert (1 oder 0) festlegt. Bei einigen Anwendungsfällen kann *ReLU* problematisch werden, da negative Eingaben nicht miteinbezogen werden können. Da die Steigung dort auch 0 ist, ist es auch schwer aus diesem Zustand wieder herauszukommen. Das führt zu Neuronen deren Output immer negativ sein wird und somit quasi 'tot' (*dying ReLU*) sind. Um das zu vermeiden gibt es Varianten, die negative Eingaben nicht auf 0 mappen, sondern auf eine lineare Funktion mit geringer Steigung.

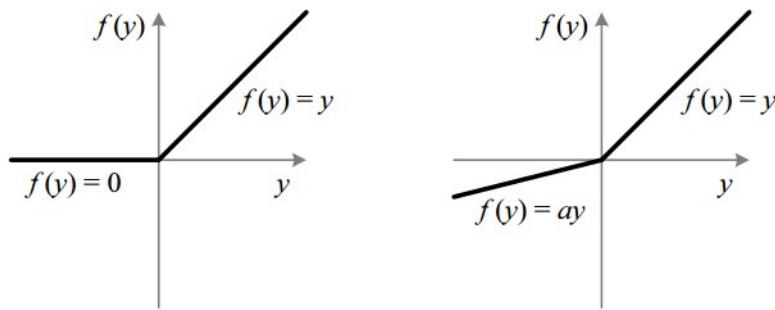


Abbildung 2.6: ReLU vs Parametric ReLU ('Leaky ReLU' für $a = 0.01$) [11]

Weitere nennenswerte Aktivierungsfunktionen sind *Softplus* ($f(x) = \ln(1+e^x)$), *Exponential Linear Unit* (ELU, $f(x) = x$ für $x > 0$, sonst: $f(x) = \ln(e^x - 1)$) und *Gaussian Error Linear Unit* (GELU [6])

2.2 Pooling Layer

Der Pooling Layer folgt einem Convolutional Layer und aggregiert dessen Ergebnisse, sodass mehrere Neuronen-ausgaben in einer lokal beschränkten Region zu einem Wert gebündelt werden. Diese Bündelung passiert für jede feature map einzeln, so dass der Output eine kleinere Höhe und Breite hat, die Tiefe jedoch gleich bleibt.

Der Zweck des Pooling Layers ist es die Anzahl der Parameter und somit die Dimensionalität zu reduzieren, was die Trainingszeit verkürzt und Overfitting vorbeugt. Wie zuvor auch, muss die Filtergröße (meistens 2x2 oder 3x3), die Schrittweite und die Art des Poolings bestimmt werden. Meistens wird *max pooling* verwendet, was nur den maximalen Wert eines bestimmten Bereichs benutzt. Andere Varianten benutzen unter anderem ein (gewichtetes) Mittel aller Werte in der Region.

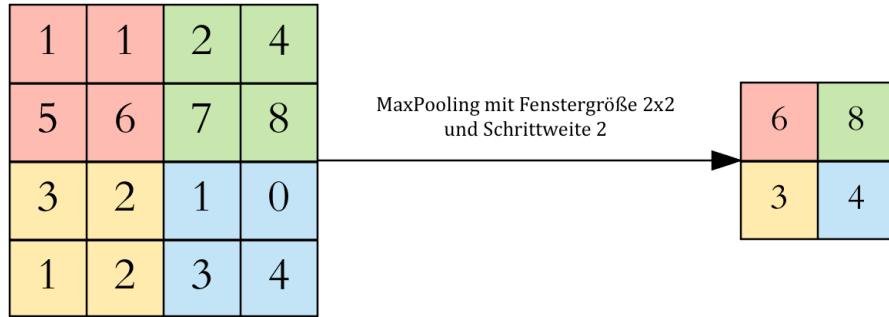


Abbildung 2.7: Beispielhafte Darstellung von max pooling (modifiziert von [2])

Pooling hilft dabei die Repräsentation invariant zu kleinen Translationen in der Eingabe zu machen. Das bedeutet, dass wenn ein Merkmal in der Eingabe um etwas verschoben wird, die Ausgabe des Pooling Layers fast gleich bleibt. Dies ist hilfreich, da man meist nur daran interessiert ist, dass ein Merkmal vorkommt und nicht, ob es genau an einer bestimmten Position vorkommt.

2.3 Fully-connected Layer

Der Fully-connected Layer stellt das Ende des Convolutional Networks dar. Zuvor werden die features zu einem langen Vektor aufgereiht (*flattening*). Dieser Vektor stellt dann die Eingabe eines herkömmlichen neuronalen Netzes mit vollständiger Verknüpfung dar. Man kann daher die vorherigen Schichten als eigenes Netzwerk ansehen, dass die hochdimensionale Eingabe vorverarbeitet und auf einzelne features hervunterbricht. Das vollständig verknüpfte Netz verarbeitet diese features zu der Ausgabe. Soll ein Netz beispielsweise Bilder in 10 verschiedenen Klassen klassifizieren, so hat die letzte Schicht 10 Output Neuronen, die jeweils die Wahrscheinlichkeit angeben, mit welcher das Bild zu einer bestimmten Klasse gehört. Als Aktivierungsfunktion nimmt man hierfür typischerweise *Softmax*.

2.4 Dropout

Dropout ist eine Regularisierungstechnik in Deep Learning Netzwerken. und wirkt gegen *Overfitting*. Bei Dropout werden bei jeder Trainingsiteration Neuronen zufällig mit einer bestimmten Wahrscheinlichkeit (*Dropout rate*) ausgeschaltet und somit auch deren Ein- und Ausgänge. Da soll verhindern, dass das Netz zu abhängig von bestimmten einzelnen Neuronen wird und zwingt jedes Neuron unabhängig mitzulernen. Dropout sollte jedoch nur bei den Input- und den Hidden Layern angewendet

werden. Die Anzahl der Trainingsiteration bis zur Konvergenz wird dadurch zwar erhöht, jedoch lernt das Netz robustere features.

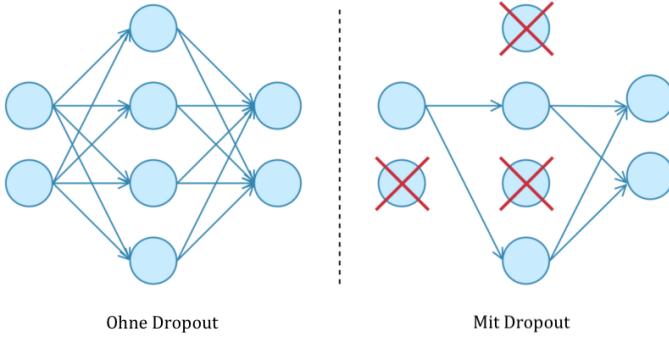


Abbildung 2.8: Dropout visualisiert (modifiziert von [2])

2.5 Transpose Convolution oder Deconvolution

Deconvolution oder auch Transpose Convolution ist selber wieder eine Faltung und nicht wie der Name Deconvolution suggeriert die Umkehrfunktion der Faltung. Die Transpose Convolution kann wie auch die Faltung mit einer Matrizenmultiplikation ausgedrückt werden. In den Gleichungen 2.6 und 2.7 ist ein Beispiel für einer Faltung mit einem eindimensionalen Eingang \vec{x} und einer Kernelgröße drei gezeigt. Hier wird der Kernelvektor \vec{k} in eine Matrix K umgeformt, sodass sich die Faltung auch als einfache Matrizenmultiplikation ausdrücken lässt.

$$\vec{k} * \vec{x} = K\vec{x} \quad (2.6)$$

$$\begin{bmatrix} k_1 & k_2 & k_3 & 0 & 0 & 0 \\ 0 & k_1 & k_2 & k_3 & 0 & 0 \\ 0 & 0 & k_1 & k_2 & k_3 & 0 \\ 0 & 0 & 0 & k_1 & k_2 & k_3 \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix} = \begin{bmatrix} k_2x_1 + k_3x_2 \\ k_1x_1 + k_2x_2 + k_3x_3 \\ k_1x_2 + k_2x_3 + k_3x_4 \\ k_1x_3 + k_2x_4 \end{bmatrix} \quad (2.7)$$

Wenn nun bestimmten Teile oder Pixel eines Bildes bestimmte Eigenschaften oder Objekte zurückgegeben werden sollen und diese am Ende von dem Netzwerk erkannt oder markiert werden sollen (Semantic Segmentation), ist es notwendig upsampling zu betreiben. Das heißt, dass die durch das Netzwerk in den vorherigen Schichten gewonnen Erkenntnisse wieder in ein Bild der gleichen Größe zur räumlichen Auflösung übertragen werden. Dafür gibt es weitere Schichten in dem Netzwerk, die für Deconvolution oder auch transpose Convolution designet sind. In der gleichen Art und Weise, wie das Downsampling von dem Netz gelernt wird, kann das Netz auch das Upsampling lernen wie die Gewichte eingestellt werden um bestimmte Objekte (zum Beispiel: Gesicht, Hund, Bäume, Himmel, Hintergrund etc.) in einem Bild zu klassifizieren. Der Input der Transpose Convolution Schicht gibt also die Gewichte für die Filter vor. Falls sich zwei Bereiche überlappen werden diese einfach addiert, wie auch in der Gleichung 2.9 mit dem stark vereinfachten Beispiel zu sehen

ist. Am Ende erhält man wieder einen Vektor, der die gleiche Dimension, wie der Ursprüngliche Inputvektor \vec{x} aus Gleichung 2.6 hat.

$$\vec{k} *^T \vec{x} = K^T \vec{x} \quad (2.8)$$

$$\begin{bmatrix} k_1 & 0 & 0 & 0 \\ k_2 & k_1 & 0 & 0 \\ k_3 & k_2 & k_1 & 0 \\ 0 & k_3 & k_2 & k_1 \\ 0 & 0 & k_3 & k_2 \\ 0 & 0 & 0 & k_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} k_1 x_1 \\ k_2 x_1 + k_1 x_2 \\ k_3 x_1 + k_2 x_2 + k_1 x_3 \\ k_3 x_2 + k_2 x_3 + k_1 x_4 \\ k_3 x_3 + k_2 x_4 \\ k_1 x_4 \end{bmatrix} \quad (2.9)$$

Hier muss eventuell beachtet werden, dass es sich nur bei einer Schrittweite von eins um eine Operation handelt, welche mit der normalen Faltung korrespondiert [3].

3 Backpropagation in CNNs

Lernen mit CNNs funktioniert ähnlich wie in herkömmlichen neuronalen Netzen mit *backpropagation* und *gradient descent*. Der größte Unterschied bei der Berechnung entsteht durch die lokale Vernetzung und dem Teilen der Gewichte. Wir werden zeigen, dass sich dadurch die Berechnung der Gradienten ebenfalls durch convolutions darstellen lässt. Im Folgenden betrachten wir die Berechnung der Fehlergradienten innerhalb eines Convolutional Layers.

Sei X der Input, F der Filterkernel und O der Output (das Ergebnis der Convolution $(*)$ von X mit F). Weiterhin sei mit X_{ij}^L der Eintrag an der Position i, j in X in der L -ten Schicht gemeint (analog: F_{ij}^L und O_{ij}^L). Des Weiteren sei f eine Aktivierungsfunktion mit $f(O^{L-1}) = X^L$. Da wir nur eine Schicht betrachten, sparen wir uns zunächst die Angabe von L (um welche Schicht es sich handelt). Zur Veranschaulichung nehmen wir einen 3x3 Input und einen 2x2 Filterkernel wobei Convolution ohne Flipping und ohne Padding ausgeführt wird.

$$\begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix} = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} * \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \quad (3.1)$$

Ausgeschrieben:

$$\begin{aligned} O_{11} &= F_{11}X_{11} + F_{12}X_{12} + F_{21}X_{21} + F_{22}X_{22} \\ O_{12} &= F_{11}X_{12} + F_{12}X_{13} + F_{21}X_{22} + F_{22}X_{23} \\ O_{21} &= F_{11}X_{21} + F_{12}X_{22} + F_{21}X_{31} + F_{22}X_{32} \\ O_{22} &= F_{11}X_{22} + F_{12}X_{23} + F_{21}X_{32} + F_{22}X_{33} \end{aligned} \quad (3.2)$$

Wir berechnen den Fehlergradienten bezüglich des Filters F . Durch Anwendung der Kettenregel ergibt sich dieser als Produkt vom gegebenen Fehlergradienten und dem lokalen Gradienten (E ist der Fehler):

$$\frac{\partial E}{\partial F} = \frac{\partial E}{\partial O} \cdot \frac{\partial O}{\partial F} \quad (3.3)$$

Der Fehlergradient bezüglich des ersten Filtereintrags sieht dann folgendermaßen aus:

$$\frac{\partial E}{\partial F_{11}} = \frac{\partial E}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial F_{11}} + \frac{\partial E}{\partial O_{12}} \cdot \frac{\partial O_{12}}{\partial F_{11}} + \frac{\partial E}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial F_{11}} + \frac{\partial E}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial F_{11}} \quad (3.4)$$

Die lokalen Gradienten lassen sich leicht aus (3.1) ableiten. Zur weiteren Vereinfachung der Notation definieren wir $\delta F := \frac{\partial E}{\partial F}$, $\delta O := \frac{\partial E}{\partial O}$ und $\delta X := \frac{\partial E}{\partial X}$. Dadurch ergibt sich:

$$\begin{aligned} \delta F_{11} &= \delta O_{11}X_{11} + \delta O_{12}X_{12} + \delta O_{21}X_{21} + \delta O_{22}X_{22} \\ \delta F_{12} &= \delta O_{11}X_{12} + \delta O_{12}X_{13} + \delta O_{21}X_{22} + \delta O_{22}X_{23} \\ \delta F_{11} &= \delta O_{11}X_{21} + \delta O_{12}X_{22} + \delta O_{21}X_{31} + \delta O_{22}X_{32} \\ \delta F_{11} &= \delta O_{11}X_{22} + \delta O_{12}X_{23} + \delta O_{21}X_{32} + \delta O_{22}X_{33} \end{aligned} \quad (3.5)$$

Bei genauer Betrachtung fällt auf, dass sich dies als Convolution (*) schreiben lässt:

$$\begin{bmatrix} \delta F_{11} & \delta F_{12} \\ \delta F_{21} & \delta F_{22} \end{bmatrix} = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} * \begin{bmatrix} \delta O_{11} & \delta O_{12} \\ \delta O_{21} & \delta O_{22} \end{bmatrix} \quad (3.6)$$

Auf ähnliche Weise lassen sich die Gradienten δX berechnen (vgl. 3.1):

$$\begin{aligned} \delta X_{11} &= \delta O_{11}F_{11} \\ \delta X_{12} &= \delta O_{11}F_{12} + \delta O_{12}F_{11} \\ \delta X_{13} &= \delta O_{12}F_{12} \\ \delta X_{21} &= \delta O_{11}F_{21} + \delta O_{21}F_{11} \\ \delta X_{22} &= \delta O_{11}F_{22} + \delta O_{12}F_{21} + \delta O_{21}F_{12}\delta O_{22}F_{11} \\ \delta X_{23} &= \delta O_{12}F_{22} + \delta O_{22}F_{12} \\ \delta X_{31} &= \delta O_{21}F_{21} \\ \delta X_{32} &= \delta O_{21}F_{22} + \delta O_{22}F_{21} \\ \delta X_{33} &= \delta O_{22}F_{22} \end{aligned} \quad (3.7)$$

Diese Berechnung entspricht einer *Full Convolution* von δX mit dem um 180° gedrehten Filterkernel (siehe folgende Abbildung).

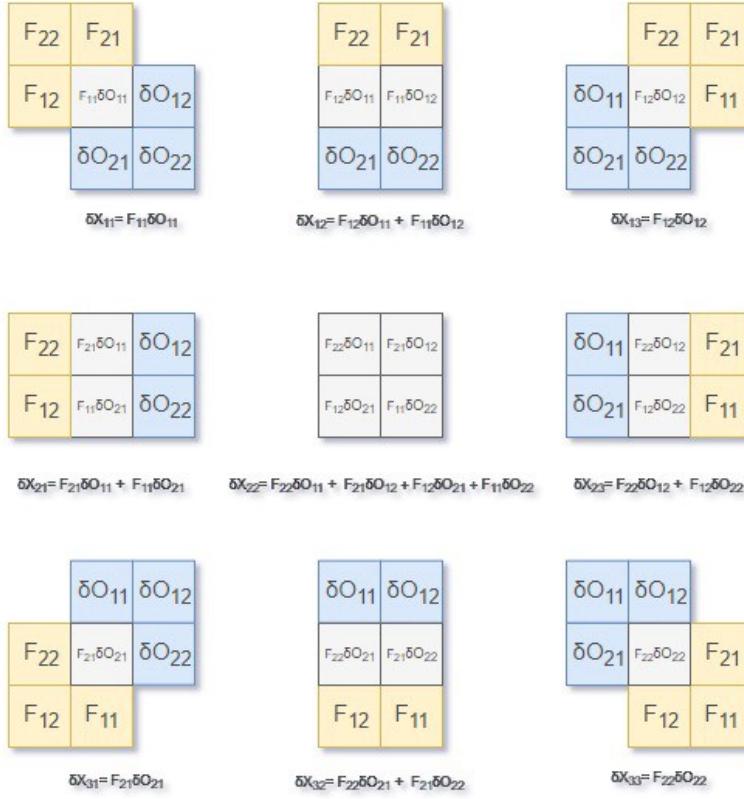


Abbildung 3.1: Full Convolution [10]

Das bedeutet:

$$\begin{bmatrix} \delta X_{11} & \delta X_{12} & \delta X_{13} \\ \delta X_{21} & \delta X_{22} & \delta X_{23} \\ \delta X_{31} & \delta X_{32} & \delta X_{33} \end{bmatrix} = \begin{bmatrix} \delta O_{11} & \delta O_{12} \\ \delta O_{21} & \delta O_{22} \end{bmatrix} \text{ Full } * \begin{bmatrix} \delta F_{22} & \delta F_{21} \\ \delta F_{12} & \delta F_{11} \end{bmatrix} \quad (3.8)$$

Nun wollen wir noch die Aktivierungsfunktion f miteinbeziehen:

$$f(O^{L-1}) = X^L$$

$$\begin{aligned} \Rightarrow \quad \frac{\partial E}{\partial f(O^{L-1})} &= \frac{\partial E}{\partial X^L} \\ \Rightarrow \quad \frac{\partial E}{\partial O^{L-1} \cdot f'(O^{L-1})} &= \frac{\partial E}{\partial X^L} \\ \Rightarrow \quad \delta O^{L-1} &= f'(O^{L-1}) \cdot \delta X^L \end{aligned} \quad (3.9)$$

Insgesamt ergibt sich dadurch :

$$\delta F^L = \text{Convolution}(X^L, \delta O^L) \quad (3.10)$$

$$\delta O^{L-1} = f'(O^{L-1}) \cdot \text{Full Convolution}(\delta O^L, \text{Flip}(F^L)) \quad (3.11)$$

Wie bereits erwähnt, können zwischen den Convolutional Layern, Pooling Layer auftreten. An diesen findet jedoch kein Machine Learning statt, da sie nur die Aktivierungen der vorangegangenen Schicht stichprobenartig wiedergeben. Ein NxN Inputblock wird zu einem Wert reduziert. Bei Backpropagation wird der Fehlergradient an die entsprechende (zuvor gemerkte) Stelle(n) weitergeleitet (*gradient routing*). Bei *max pooling* wird der Fehler nur an die Stelle des Maximums weitergegeben da die anderen Neuronen nicht zum Fehler beigetragen, im Gegensatz zu *average pooling*, wo der Fehler gleichmäßig auf die entsprechenden Input Neuronen verteilt wird.

4 Varianten

4.1 Inception

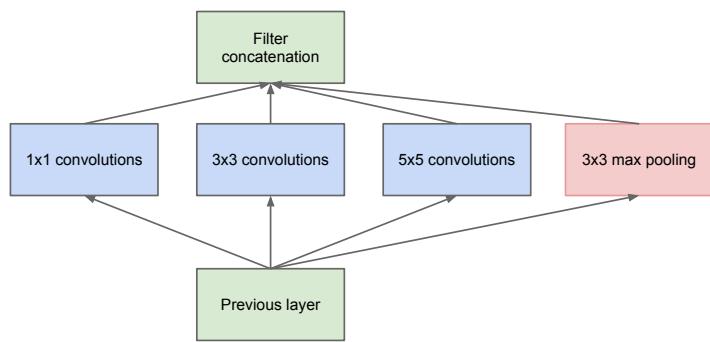


Abbildung 4.1: Naive Inception, es werden 1x1, 3x3 und 5x5 Faltung sowie 3x3 max pooling in einem Ausgang zusammengefasst [12]

ImageNet veranstalten jährlich eine Challenge in der bestimmte Aufgaben, wie zum Beispiel das Kennzeichnen von vorgegebener Objekten in Bildern möglichst effizient und mit einer geringen Fehlerrate erledigt werden sollen. Im Rahmen dieses Events² hat Google im Jahr 2014 ein Netzwerk mit dem Namen GoogLeNet vorgestellt, welches erstmalig Inception verwendet und in diesem Jahr den 1. Platz belegte. GoogLeNet ist ein CNN mit 22 Schichten und ist damit schon ein tiefes Neuronales Netz. Inception bedeutet in diesem Zusammenhang, dass der Ausgang mehrerer Filtergruppen einer Schicht zusammengefasst werden. In diesem Beispiel sollen die Informationen, wie in Abbildung 4.1 gezeigt, einer 1x1, 3x3 und 5x5 Faltung sowie einem 3x3 max pooling zusammengefasst werden. Dies hat abhängig von der Tiefe oder Anzahl der Feature Maps sehr rechenaufwändig. Um die Tiefe zu reduzieren werden sogenannte Bottleneck-Schichten eingeführt, wie in Abbildung 4.2 gezeigt, welche mittels einer 1x1 Convolution pro Schicht weniger Filter verwenden. Hier wird eine Art Feature Pooling durchgeführt, wo nur ausgewählte Features der Feature Maps weiter verwendet werden um die Tiefendimension der Feature Map, also der Eingabe, zu verringern. Hierbei gehen Daten verloren, was aber zum Teil für

²siehe <http://www.image-net.org/challenges/LSVRC/2014/>

weitere Berechnungen kaum relevant ist und durch Redundanzen und Näherungen abgedeckt ist. 1x1 Convolutions bieten sich hier an, da diese besonders wenig Rechenaufwand benötigen. Da die Tiefe der Eingangsdaten pro Schicht immer weiter zunimmt sind schon ab wenigen Schichten solche 1x1 Convolutions notwendig damit der Rechenaufwand beherrschbar bleibt. Am Ausgang werden alle Ausgänge dieser Schicht in einem einzelnen Vektor gebündelt, welcher gleichzeitig der Eingang der nächsten Schicht ist.

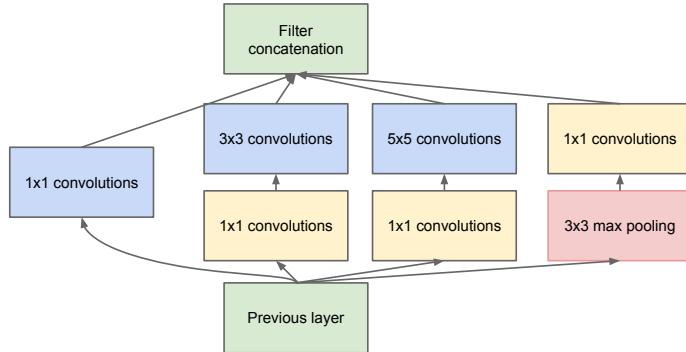


Abbildung 4.2: Inception mit Bottleneck Layer [12]

Je tiefer ein Netzwerk ist, desto mehr Details und Features kann es extrahieren. Ein Problem, was hier entsteht, ist dass die Anzahl der benötigten berechneten Parameter stark steigt. Dies kann schnell zu Overfitting führen, was das Resultat am Ende wieder verschlechtert. Ein weiterer Vorteil ist, dass ein Netzwerk wie GoogLeNet mit deutlich weniger Parametern auskommt und so den Vorteil von einem tiefen Netz hat ohne eines der Hauptnachteile zu haben. Das Ganze kommt mit dem Trade Off, dass Näherungen zur Reduzierung des Rechenaufwandes gemacht werden müssen [12].

4.2 DensNet

Densely Connected Convolutional Networks (DensNet) bieten eine weitere Möglichkeit die Anzahl der Parameter zu reduzieren sowie das Vanishing Gradient Problem zu verringern.

Beim Vanishing Gradient Problem ändern sich beim Training des Netzwerkes die Gewichte nicht mehr oder nur sehr wenig, sodass das Netzwerk nicht weiter effektiv trainiert werden kann. Das kann dadurch zustande kommen, da der Gradient nur noch sehr kleine Werte aus gibt, welche dann bei Backpropagation durch den schlechten Informationsfluss kaum noch Auswirkung auf die vorderen Schichten des Netzwerkes haben.

In einem DensNet hat jede Schicht als Eingang die Ausgänge aller vorherigen Schichten, was vorteilhaft für Feature Propagation ist. DensNets können für tiefe Architekturen benutzt werden. Die Parameterreduktion kommt dadurch zustande, da redundante Feature Maps nicht mehrfach wieder erlernt werden müssen. Ansätze, wie das randomisierte Ignorieren von Gewichten von Schichten, da diese offenbar nur einen sehr geringen Einfluss haben, wie es zum Beispiel in ResNets gemacht

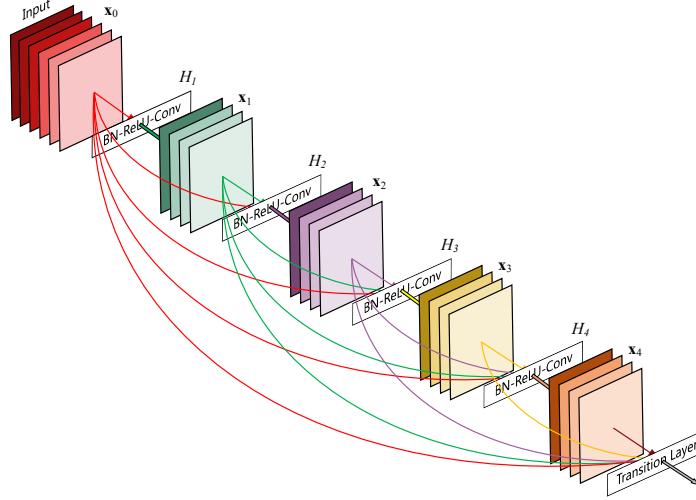


Abbildung 4.3: Feature Propagation in einem DensNet Block [7]

wird, bringt hier nur einen kleinen Vorteil, da die Anzahl der Parameter nicht geringer wird. Beim DensNet werden die Gewichte über mehrere Schichten erhalten und ermöglichen eine große Reduktion der benötigten Parameter. Der verbesserte Informationsfluss in solchen Netzen ermöglicht ein effizienteres Training der Netze, was eine Menge Zeit und Rechenleistung spart. Auch hier wird Overfitting aufgrund der kleineren Parameterzahl stark durch einen selbstregulierenden Effekt reduziert.

Wie in Abbildung 1 gezeigt, hat die n -te Schicht n Inputs. Damit hat die Gesamtzahl der Schichten in einem Netzwerk mit N Schichten $\frac{N(N + 1)}{2}$ Verbindungen, anstatt N Verbindungen, wie in herkömmlichen Netzwerken. Daher auch die Bezeichnung Densely Connected Convolutional Networks [7].

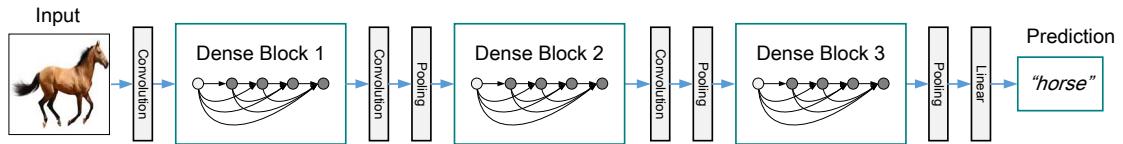


Abbildung 4.4: Struktur eines DensNet [7]

5 Geschichte

In einem Paper von Hubel und Wiesel aus dem Jahre 1967 wurde beschrieben, wie bestimmte Regionen in den Gehirnen von Affen und Katzen auf bestimmte visuelle Reize reagieren. So sind unterschiedliche kleinere Bereiche vom visuellen Kortex für unterschiedliche kleine Bereiche im Sichtfeld verantwortlich. Diese kleinen Bereiche sind für die Detektierung von hell-dunkel-Konturen sowie von deren Änderung in Richtung oder Form und werden von sogenannten "Simplen Zellen" im visuellen Kortex von sich überlappenden rezeptiven Feldern verarbeitet. Sogenannte "Komplexe Zellen" im visuellen Kortex haben größere rezeptive Felder, welche die Position

der Konturen im Sichtfeld bestimmen. Nach diesem Vorbild der Unterteilung des Sichtfeldes in bestimmte Strukturen wurden die CNN entwickelt [8].

Auf dieser Basis wurde in den 1980er Jahren das Neocognitron, ein hierarchisches, mehrschichtiges künstliches neuronales Netzwerk von Kunihiko Fukushima, entwickelt. Dieses Netz eignete sich besonders gut für die Erkennung von Handschrift und anderer Pattern und diente als Vorbild für spätere CNN. Es arbeitet komplett selbst organisiert und "ohne Lehrer". Ähnlich wie aus dem biologischem Vorbild gibt es zwei Zelltypen. Sogenannte S-Cells, welche für die Erkennung von Eigenschaften genutzt werden und sich in der ersten Schicht befinden und sogenannte C-Cells, welche für die Detektierung von Änderung, Verschiebung und Verformung dieser Eigenschaften bestimmt sind und sich in der zweiten Schicht befinden. Gerade die Idee der lokale Eigenschaften zu detektieren gilt hier als Vorbild für später folgende Modelle [4].

Der erste CNN war ein Time Delay Neural Network, welches zeitinvariante Berechnungen von Audiodateien zur Spracherkennung durchführt und später auch das erste mal unter dem Namen Convolutional Neural Network für Bilderkennung genutzt wurde.

6 Zusammenfassung und Ausblick

Convolutional Neural Networks sind eine besondere Art von künstlichen neuronalen Netzen, die heutzutage vor allem bei räumlich korrelierten Datensätzen, wie z.B Bild- und Audiodateien, Verwendung finden. Aufgebaut aus einer Vielzahl verschiedenartiger Schichten, zählen sie zu den Deep Learning Architekturen. Sie zeichnen sich dadurch aus, dass Neuronen zweier aufeinanderfolgenden Schichten nicht vollständig, sondern nur lokal begrenzt miteinander verknüpft sind. Zusätzlich dazu teilen sich die Neuronenverbindungen die Gewichte. Diese Eigenschaften werden durch Faltungen (Convolutions) umgesetzt, welche namensgebend für diese Art von Netzen sind. Neben den Faltungsschichten gibt es noch die Pooling Schichten, an denen die Neuronenanzahl verringert werden. Die Eigenschaften erlauben sehr tiefe und robuste Netzstrukturen. CNNs eignen sich besonders gut, um Merkmale (*features*) aus Daten zu extrahieren. Dies ermöglicht u.A Transferlernen - also Teile von bereits vortrainierten CNNs zu nehmen und für seine eigenen Zwecke anzupassen, statt das Netz von Grund auf selber zu trainieren. Wie bereits erwähnt, erzielen CNNs hervorragende Ergebnisse im Bereich der Bildklassifizierung. Das Anwendungsspektrum für CNNs ist daher sehr breit: Von Analysen bei bildgebenen Verfahren in der Medizin und anderen Naturwissenschaften bis hin zu Echtzeit-anwendungen im Straßenverkehr oder gar komplett autonomes Fahren. Allerdings könnten auch Überwachungsstaaten von dem Potential der Gesichtserkennung und -profilierung profitieren und ihre Überwachungsinfrastruktur dementsprechend ausbauen und erweitern. Zusammenfassend lässt sich sagen, dass CNN, welche nur aufgrund der heutigen Rechenleistungen nutzbar geworden sind und damit eine relativ junge Technologie ist, viel Potential für allerlei Anwendung birgt.

Literatur

- [1] D. Cornelisse. An intuitive guide to convolutional neural networks, 2018.
- [2] A. Dertat. Applied deep learning - part 4: Convolutional neural networks, 2017.
- [3] S. Y. Fei-Fei Li, Justin Johnson. Lecture 11: Detection and segmentation, 2017.
- [4] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr 1980.
- [5] G. Gwardys. Convolutional neural networks backpropagation: from intuition to derivation, 2016.
- [6] D. Hendrycks and K. Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.
- [7] G. Huang, Z. Liu, and K. Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.
- [8] D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology (London)*, 160:106–154, 1962.
- [9] Y. B. Ian Goodfellow and A. Courville. Deep learning. Book in preparation for MIT Press, 2016.
- [10] S. Rai. Forward and backpropagation in convolutional neural network, 2017.
- [11] S. Sharma. Activation functions: Neural networks, 2017.
- [12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

Anhang

Dieses Notebook (<https://colab.research.google.com/drive/1YA8WVQ2xnD8Pn0Nb1beSiLPtgsakwWW4>) stellt eine Beispielanwendung von Convolutional Neural Networks in Python mit Keras (<https://keras.io/>) und Tensorflow (<https://www.tensorflow.org/>) vor. Es wird ein CNN Architektur definiert und anschließend dazu trainiert, Bildklassifikation durchzuführen. Als Datensatz wird CIFAR-10 (<https://www.cs.toronto.edu/~kriz/cifar.html>) verwendet. Anschließend werden Anhand eines bewährten vortrainierten Models VGG16 (http://www.robots.ox.ac.uk/~vgg/research/very_deep/) die Feature maps zu einem bestimmten Bild visualisiert.

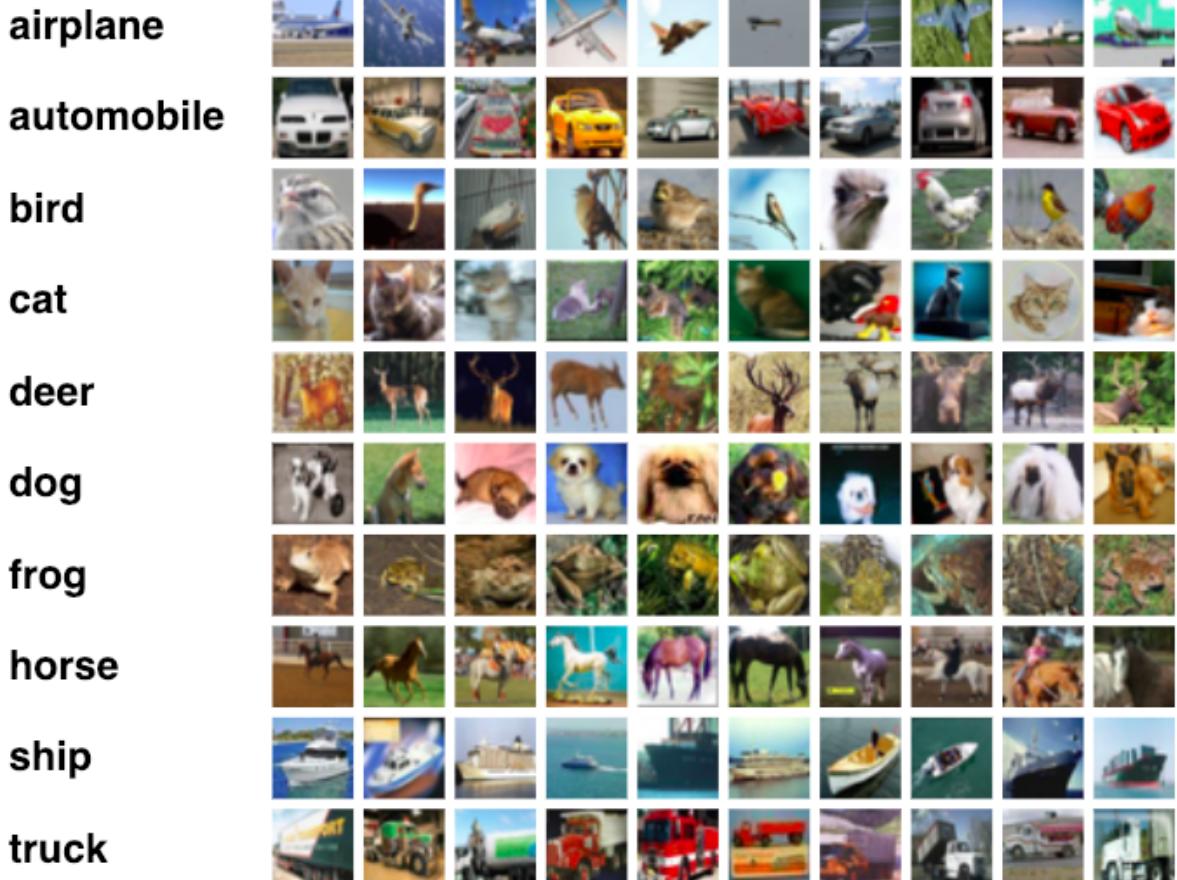
Convolutional Neural Network für Bildklassifizierung

Datensatz

CIFAR-10 besteht aus 60000 32x32 Farbbildern aus 10 Klassen:

```
In [0]: from IPython.display import Image  
Image('/content/cifar10.png')
```

Out[0]:



Imports und Hilfsfunktionen

```
In [0]: import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
import tensorflow as tf

def plot_images(data):
    plt.figure(figsize=(10,3))
    for i in range(30):
        plt.subplot(3,10,i+1)
        plt.xticks([])
        plt.yticks([])
        plt.grid(False)
        plt.imshow(data[i])

def plot_compare(history):

    acc = history.history['acc']
    val_acc = history.history['val_acc']
    loss = history.history['loss']
    val_loss = history.history['val_loss']

    plt.figure(figsize=(12, 4))
    plt.subplot(1,2,1)
    plt.plot(loss, c='#0c7cba', label='Train Loss')
    plt.plot(val_loss, c='#0f9d58', label='Val Loss')
    plt.xticks(range(0, len(loss), 5))
    plt.xlim(0, len(loss))
    plt.xlabel('Epoch')
    plt.title('Train Loss: %.3f, Val Loss: %.3f' % (loss[-1], val_loss[-1]), fontsize=12)
    plt.legend()

    plt.subplot(1,2,2)
    plt.plot(acc, c='#0c7cba', label='Train Acc')
    plt.plot(val_acc, c='#0f9d58', label='Val Acc')
    plt.xticks(range(0, len(acc), 5))
    plt.xlim(0, len(acc))
    plt.xlabel('Epoch')
    plt.title('Train Accuracy: %.3f, Val Accuracy: %.3f' % (acc[-1], val_acc[-1]), fontsize=12)
    plt.legend()

    plt.tight_layout()

from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions
from keras.applications.vgg16 import VGG16

def vgg_predict(path):

    model = VGG16()
    image = load_img(path, target_size=(224, 224))
    plt.imshow(image)
    plt.axis('off')
    plt.show()
    image = img_to_array(image)
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))

    image = preprocess_input(image)
    yhat = model.predict(image)
    label = decode_predictions(yhat)
    label = label[0][0]
```

```
print(' %s (%.2f%%)' % (label[1], label[2]*100))
```

Using TensorFlow backend.

Datensatz laden

Keras kommt mit einer Bibliothek [datasets \(<https://keras.io/datasets/>\)](https://keras.io/datasets/), welche man nutzen kann um Datensätze direkt vom Server zu laden (in unserem Fall: cifar10).

```
In [0]: from tensorflow.keras.datasets import cifar10  
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Analyse

Als nächstes werden wir uns den Datensatz genauer anschauen und uns vergewissern, dass es sich wirklich um 60000 32x32 Farbbilder mit 10 Klassen handelt.

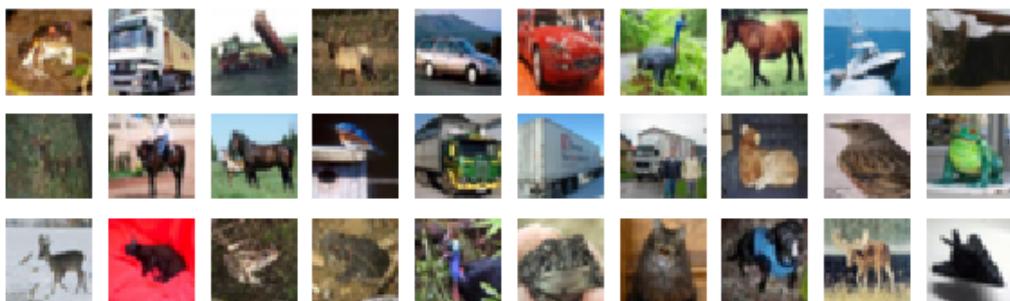
```
In [0]: print('Training data shape : ', x_train.shape, y_train.shape)
        print('Testing data shape : ', x_test.shape, y_test.shape)

Training data shape : (50000, 32, 32, 3) (50000, 1)
Testing data shape : (10000, 32, 32, 3) (10000, 1)
```

```
In [0]: classes = np.unique(y_train)
nClasses = len(classes)
print('number of classes : ', nClasses)

number of classes :  10
```

```
In [0]: # 30 erste Bilder  
plot_images(x_train)
```



Vorverarbeitung

Zunächst werden wir die Daten normalisieren.

```
In [0]: x_train = tf.keras.utils.normalize(x_train, axis=1)
        x_test = tf.keras.utils.normalize(x_test, axis=1)
```

Im Moment sind die Labels als Nummern von 0-9 gegeben. Da benutzen One Hot Encoding um die Label als 10 stelligen Vektor zu kodieren. Beispielsweise wird dann aus der Nummer 3 der Vektor [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

```
In [0]: # Onehot encode Labels
from keras.utils.np_utils import to_categorical
y_train_one_hot = to_categorical(y_train)
y_test_one_hot = to_categorical(y_test)

print('Original label:', y_train[0])
print('One hot encoded label:', y_train_one_hot[0])
```

```
Original label: [6]
One hot encoded label: [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

Damit unser Modell gut generalisiert, werden wir 20% des Trainingdatensatzes für Validierung nutzen. Das reduziert *Overfitting* und führt so zu besserer Test Performanz.

```
In [0]: from sklearn.model_selection import train_test_split
x_train,x_valid,y_train_one_hot,y_valid_one_hot = train_test_split(x_train, y_train_one_hot,
                                                               test_size=0.1, random_state=21)

# check shapes, just to make sure
x_train.shape,x_valid.shape,y_train_one_hot.shape,y_valid_one_hot.shape
```

```
Out[0]: ((45000, 32, 32, 3), (5000, 32, 32, 3), (45000, 10), (5000, 10))
```

Ein weitere Vorverarbeitungsschritt besteht darin, den Trainingsdatensatz zu augmentieren. Für Bilder bspw., werden weiter Bilder generiert, die z.B rotiert, geneigt, verschoben, etc sind. Wir nutzen hierfür die [ImageDataGenerator](#) (<https://keras.io/preprocessing/image/>) Klasse von Keras. Der generator muss definiert und anschließend

```
In [0]: from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(width_shift_range=0.1,
                             height_shift_range=0.1,
                             zoom_range=0.1,
                             rotation_range=30,
                             horizontal_flip=True)

# berechnet benötigte Quantitäten wie std, mean, etc.
datagen.fit(x_train)

train_generator = datagen.flow(x_train, y_train_one_hot, shuffle=True)
```

Modellierung

Für das Convolutional Neural Network, anhand wir unser Modell trainieren wollen, legen wir eine Architektur fest. Zunächst werden alle nötigen Module importiert.

```
In [0]: from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Activation, MaxPooling2D
from tensorflow.keras.layers import Dense, Dropout, Flatten, LeakyReLU
```

Nach dem man ein Modell instantiiert hat, lassen sich in Keras lassen die Netwerkschichten (Layer) stapeln in dem man sie nacheinander einfügt (`add()`). Wir werden mehrere Convolutional Layer mit LeakyReLU als Aktivierungsfunktion benutzen. Auf jeden Convolutional Layer folgt eine MaxPooling Schicht. *Flatten* transformiert die featuremaps in einen Vektor, der von vollständig vernetzten Schichten mit SoftMax zum Output verarbeitet wird.

```
In [0]: # Feed Forward network
my_model = Sequential()

# Der erste Layer muss wissen was für eine Form der Input hat
my_model.add(Conv2D(32,(3,3), padding="same", activation='relu',
                   input_shape = x_train.shape[1:]))
my_model.add(Conv2D(32,(3,3), padding="same", activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2, 2)))

my_model.add(Conv2D(64,(3,3), padding="same", activation='relu'))
my_model.add(Conv2D(64,(3,3), padding="same", activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2, 2)))

my_model.add(Conv2D(128,(3,3), padding ="same", activation='relu'))
my_model.add(Conv2D(128,(3,3), padding ="same", activation='relu'))
my_model.add(MaxPooling2D(pool_size=(2, 2)))

# Vectorization + DropOut
my_model.add(Flatten())
my_model.add(Dropout(0.5))

# Fully Connected Layer
my_model.add(Dense(128, activation='relu'))
my_model.add(Dense(64, activation='relu'))
my_model.add(Dense(10, activation='softmax'))
```

Das Modell lässt sich mit `summary()` visualisieren (Output-shape, Parameteranzahl)

```
In [0]: my_model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 32, 32, 32)	896
conv2d_13 (Conv2D)	(None, 32, 32, 32)	9248
max_pooling2d_12 (MaxPooling)	(None, 16, 16, 32)	0
conv2d_14 (Conv2D)	(None, 16, 16, 64)	18496
conv2d_15 (Conv2D)	(None, 16, 16, 64)	36928
max_pooling2d_13 (MaxPooling)	(None, 8, 8, 64)	0
conv2d_16 (Conv2D)	(None, 8, 8, 128)	73856
conv2d_17 (Conv2D)	(None, 8, 8, 128)	147584
max_pooling2d_14 (MaxPooling)	(None, 4, 4, 128)	0
flatten_4 (Flatten)	(None, 2048)	0
dropout_5 (Dropout)	(None, 2048)	0
dense_11 (Dense)	(None, 128)	262272
dense_12 (Dense)	(None, 64)	8256
dense_13 (Dense)	(None, 10)	650
<hr/>		
Total params: 558,186		
Trainable params: 558,186		
Non-trainable params: 0		

Nachdem wir das Modell kreiert haben, müssen wir es kompilieren. Hierbei werden u.A. die [Verlustfunktion](https://keras.io/losses/) (<https://keras.io/losses/>) und der [Optimizer](https://keras.io/optimizers/) (<https://keras.io/optimizers/>) (beinhaltet Lernrate, etc.) festgelegt.

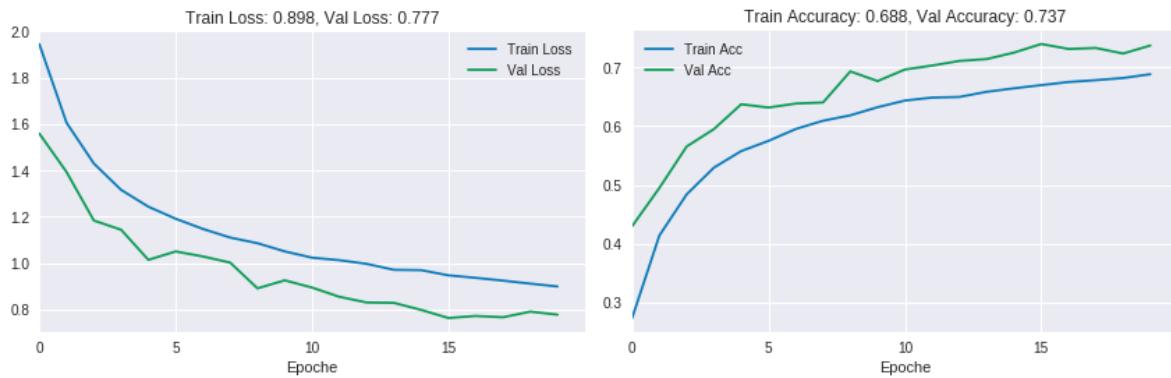
```
In [0]: my_model.compile(optimizer='adam', loss='categorical_crossentropy',
                      metrics=['accuracy'])
```

Training

Nun werden wir unser Modell trainieren. Das geschieht normalerweise durch *fit()*. Da wir jedoch einen Generator für die Datenaugmentierung genutzt haben, verwenden wir *fit_generator()*. Wir trainieren für 20 Epochen. Der Output der Funktion speichern wir in einer *history* variable um die Metriken plotten zu können.

```
In [0]: history = my_model.fit_generator(  
        train_generator,  
        epochs=20,  
        verbose=1,  
        validation_data=(x_valid, y_valid_one_hot))  
  
Epoch 1/20  
1407/1407 [=====] - 43s 30ms/step - loss: 1.9443 - ac  
c: 0.2740 - val_loss: 1.5585 - val_acc: 0.4298  
Epoch 2/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.6049 - ac  
c: 0.4141 - val_loss: 1.3923 - val_acc: 0.4948  
Epoch 3/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.4292 - ac  
c: 0.4837 - val_loss: 1.1830 - val_acc: 0.5652  
Epoch 4/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.3152 - ac  
c: 0.5294 - val_loss: 1.1433 - val_acc: 0.5948  
Epoch 5/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.2427 - ac  
c: 0.5574 - val_loss: 1.0134 - val_acc: 0.6370  
Epoch 6/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.1907 - ac  
c: 0.5748 - val_loss: 1.0496 - val_acc: 0.6314  
Epoch 7/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.1466 - ac  
c: 0.5950 - val_loss: 1.0279 - val_acc: 0.6382  
Epoch 8/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.1094 - ac  
c: 0.6090 - val_loss: 1.0014 - val_acc: 0.6400  
Epoch 9/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.0852 - ac  
c: 0.6181 - val_loss: 0.8904 - val_acc: 0.6928  
Epoch 10/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.0496 - ac  
c: 0.6321 - val_loss: 0.9249 - val_acc: 0.6764  
Epoch 11/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.0226 - ac  
c: 0.6433 - val_loss: 0.8938 - val_acc: 0.6960  
Epoch 12/20  
1407/1407 [=====] - 41s 29ms/step - loss: 1.0120 - ac  
c: 0.6483 - val_loss: 0.8540 - val_acc: 0.7028  
Epoch 13/20  
1407/1407 [=====] - 41s 29ms/step - loss: 0.9961 - ac  
c: 0.6493 - val_loss: 0.8294 - val_acc: 0.7106  
Epoch 14/20  
1407/1407 [=====] - 41s 29ms/step - loss: 0.9702 - ac  
c: 0.6582 - val_loss: 0.8279 - val_acc: 0.7138  
Epoch 15/20  
1407/1407 [=====] - 41s 29ms/step - loss: 0.9689 - ac  
c: 0.6641 - val_loss: 0.7974 - val_acc: 0.7248  
Epoch 16/20  
1407/1407 [=====] - 41s 29ms/step - loss: 0.9462 - ac  
c: 0.6697 - val_loss: 0.7624 - val_acc: 0.7394  
Epoch 17/20  
1407/1407 [=====] - 41s 29ms/step - loss: 0.9355 - ac  
c: 0.6748 - val_loss: 0.7713 - val_acc: 0.7310  
Epoch 18/20  
1407/1407 [=====] - 41s 29ms/step - loss: 0.9236 - ac  
c: 0.6780 - val_loss: 0.7657 - val_acc: 0.7326  
Epoch 19/20  
1407/1407 [=====] - 42s 30ms/step - loss: 0.9109 - ac  
c: 0.6815 - val_loss: 0.7898 - val_acc: 0.7232  
Epoch 20/20  
1407/1407 [=====] - 42s 30ms/step - loss: 0.8987 - ac  
c: 0.6880 - val_loss: 0.7769 - val_acc: 0.7368
```

```
In [0]: plot_compare(history)
```



Evaluierung

Nachdem wir das Modell trainiert haben, wollen wir wissen wie akkurat wir den Testdatensatz klassifizieren können. Wie man sieht, konnten wir selbst mit dem verhältnismäßig einfachen Modell 72% des Testdatensatzes richtig klassifizieren.

```
In [0]: test_eval = my_model.evaluate(x_test, y_test_one_hot, verbose=1)

print('Test loss:', test_eval[0])
print('Test accuracy:', test_eval[1])

10000/10000 [=====] - 2s 208us/step
Test loss: 0.8087519835472107
Test accuracy: 0.723
```

Es macht jedoch Sinn, nicht nur die Performance allgemein anzuschauen, sondern innerhalb der einzelnen Klassen. 'Accuracy' als Maß kann jedoch irreführend sein (siehe [Link \(https://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/\)](https://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/)). Daher nutzen wir 'Precision' und 'Recall'. Generell ist gewollt, alle Werte hoch sind.

```
In [0]: predicted_classes = my_model.predict(x_test)
predicted_classes = np.argmax(np.round(predicted_classes),axis=1)

from sklearn.metrics import classification_report
target_names = ['Airplane','Automobile','Bird','Cat',
                'Deer','Dog','Frog','Horse','Ship','Truck']
print(classification_report(y_test, predicted_classes, target_names=target_name
s))
```

	precision	recall	f1-score	support
Airplane	0.31	0.85	0.45	1000
Automobile	0.90	0.83	0.86	1000
Bird	0.82	0.43	0.56	1000
Cat	0.80	0.30	0.43	1000
Deer	0.81	0.53	0.64	1000
Dog	0.81	0.43	0.56	1000
Frog	0.66	0.86	0.75	1000
Horse	0.85	0.71	0.77	1000
Ship	0.87	0.83	0.85	1000
Truck	0.78	0.88	0.82	1000
micro avg	0.66	0.66	0.66	10000
macro avg	0.76	0.66	0.67	10000
weighted avg	0.76	0.66	0.67	10000

Um das Modell mit zu speichern kann man `save()` aufrufen. Anhand der gespeicherten Datei kann man das Modell später wieder laden.

```
In [0]: my_model.save("/content/cifar10_cnn.h5py")
```

Bildklassifizierung mit VGG16

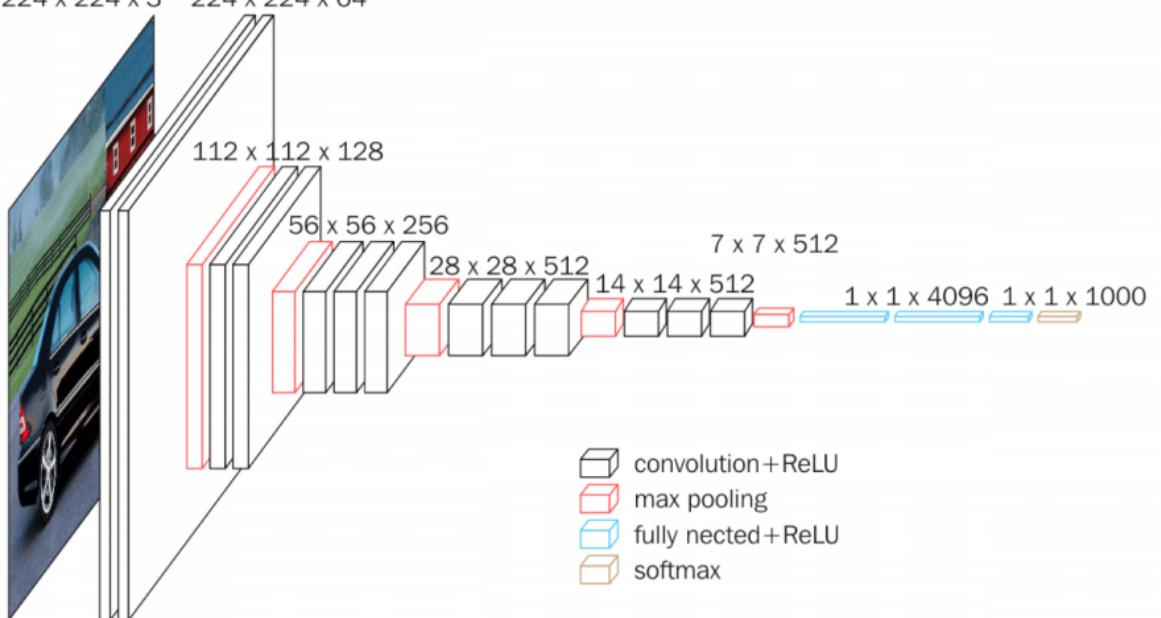
Für den nächsten Abschnitt benutzen wir ein bestehendes Modell. Die [VGG Modelle](https://arxiv.org/pdf/1409.1556.pdf) (<https://arxiv.org/pdf/1409.1556.pdf>) haben 2014 bei der [ImageNet](http://www.image-net.org/) (<http://www.image-net.org/>) Challenge die ersten Plätze belegt und eigne sich daher besonders gut um Bilder jeglicher Art zu klassifizieren. In Keras lässt sich das Modell samt Gewichte (trainiert auf ImageNet) wie folgt laden:

```
In [0]: from keras.applications import VGG16  
vgg = VGG16(weights='imagenet')
```

```
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg16_weights_tf_dim_ordering_tf_kernels.h5  
553467904/553467906 [=====] - 20s 0us/step
```

```
In [0]: Image('/content/vgg16-1-e1542731207177.png')
```

```
Out[0]: 224 x 224 x 3 224 x 224 x 64
```



```
In [0]: vgg.summary()
```

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000
<hr/>		
Total params: 138,357,544		
Trainable params: 138,357,544		
Non-trainable params: 0		

Klassifizierung nach VGG16

Wir haben eine Funktion (`vgg_predict()`) geschrieben anhand der man jegliche Bilder mit VGG16 klassifizieren kann.

```
In [0]: vgg_predict("/content/tiger.png")
```



tiger (85.78%)

```
In [0]: vgg_predict("/content/car.jpg")
```



sports_car (75.89%)

```
In [0]: vgg_predict("/content/beach.jpg")
```



seashore (84.35%)

```
In [0]: vgg_predict("/content/darth-vader.jpg")
```



electric_fan (41.20%)

Visualisierung der Feature maps

Man kann die Aktivierungen an den einzelnen Schichten visualisieren. In dem man ein Bild durch das Netz schickt und zufällig feauture maps auswählt. Da die Schichten für das Keras-VGG16 model benannt sind kann man gezielt sich nur die feature maps der convolutional layer anschauen. In diesem Beispiel stehe helle Farben für eine hohe Aktivierung. (Der Code für diesen Teil wurde entnommen aus einem [Blog Beitrag](https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2). (<https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2>) und leicht angepasst).

```
In [0]: from keras.models import Model

img_tensor = image_tensor('/content/tiger.png')
layer_names = ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1',
'block5_conv1']

layer_outputs = [layer.output for layer in vgg.layers if layer.name in layer_names]
activation_model = Model(inputs=vgg.input, outputs=layer_outputs)
intermediate_activations = activation_model.predict(img_tensor)

images_per_row = 8
max_images = 24
# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, intermediate_activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]
    n_features = min(n_features, max_images)

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

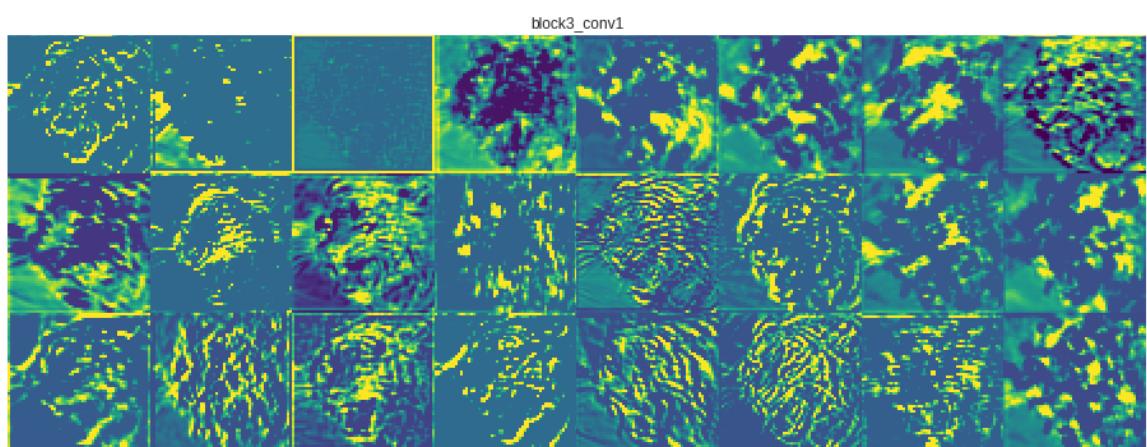
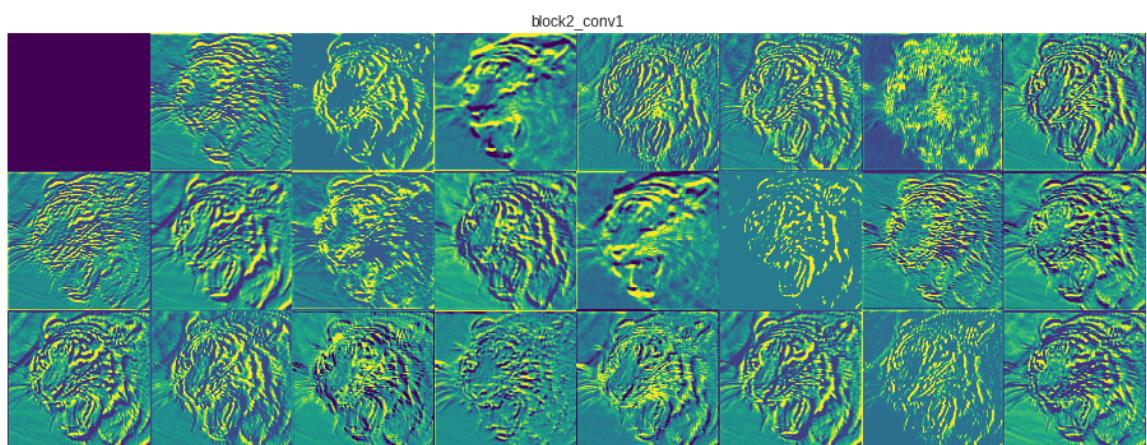
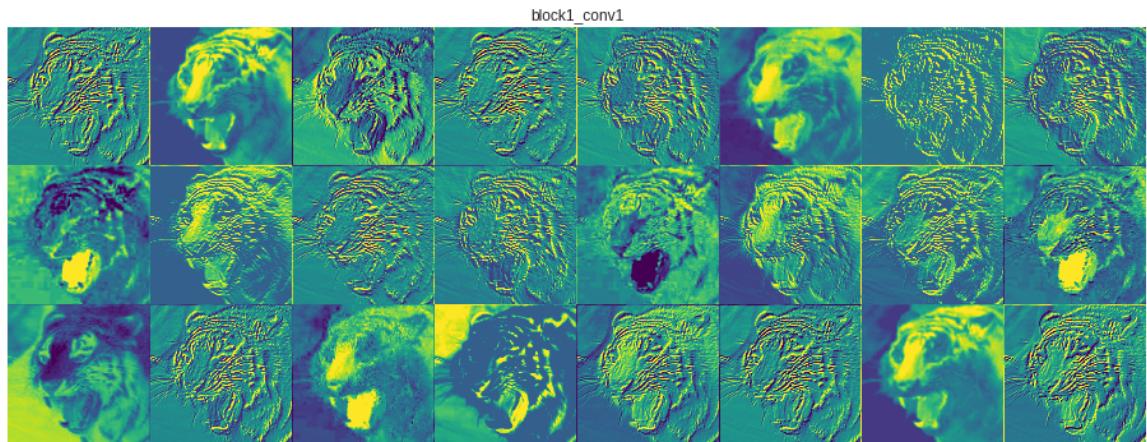
    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 2. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.axis('off')
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

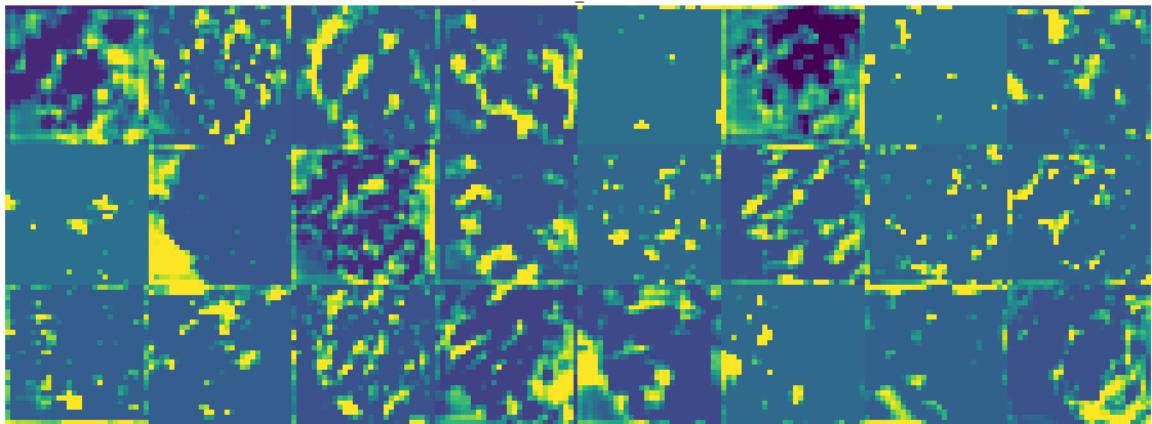
plt.show()
```

(1, 224, 224, 3)

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:33: RuntimeWarning: invalid value encountered in true_divide



block4_conv1



block5_conv1

