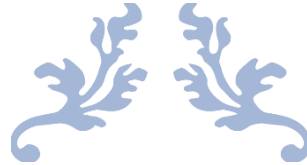


**Städtisches Gymnasium Nepomucenum
Coesfeld**



CONVOLUTIONAL NEURAL NETWORKS

Entwurf und Implementation eines künstlichen Neuronalen Netzes am
Beispiel des CIFAR 10 Datensatzes



Vorgelegt von Jan Bessler
Qualifikationsphase 1
im Fach Mathematik

Betreuender Fachlehrer
Michael Weiermann

Inhaltsverzeichnis

Inhaltsverzeichnis.....	1
Abbildungsverzeichnis	3
Vorwort.....	4
Einleitung	5
Neuronale Netze	5
a. Was ist ein Neuronales Netzwerk?	5
b. Was ist ein künstliches Neuron?	5
c. Das XOR-Problem	9
d. Warum der Bias wichtig ist	11
e. Das Trainieren von Neuronalen Netzen.....	12
Convolutional Neural Networks.....	16
a. Was ist ein Convolutional Neural Network?	16
b. Die Convolutional Layer	16
c. Die Pooling Layer	21
d. Die Flattening Layer	22
e. Das Trainieren von Convolutional Neural Networks.....	22
f. Implementation eines CNN in Python mit tensorflow und keras.....	27
Fazit.....	32
Fußnote	33
Quellen.....	34

Abbildungsverzeichnis

Fig. 1 [8].....	6
Fig. 2 [9].....	8
Fig. 3 [9].....	8
Fig. 4 [9].....	8
Fig. 5 [9].....	9
Fig. 7 [5].....	9
Fig. 8.....	10
Fig. 9.....	10
Fig. 10.....	11
Fig. 11 [9]	12
Fig. 12 [9]	12
Fig. 13.....	13
Fig. 14 [11]	15
Fig. 15.....	17
Fig. 16.....	17
Fig. 17.....	18
Fig. 18.....	19
Fig. 19.....	19
Fig. 20 [18]	20
Fig. 21 [19]	21
Fig. 22.....	22
Fig. 23.....	23
Fig. 24.....	24
Fig. 25.....	25
Fig. 26.....	27
Fig. 27.....	27
Fig. 28.....	28
Fig. 29.....	29
Fig. 30.....	29
Fig. 31.....	30
Fig. 32.....	30
Fig. 33.....	31

Vorwort

Von jetzt an werden folgende Abkürzungen/Alternativen für bestimmte Begriffe verwendet:

- Faltendes neuronales Netz (eng. Convolutional Neural Network) -> CNN
- Neuronales Netz (eng. Neural Network) -> NN
- Künstliche Intelligenz (eng. Artificial Intelligence) -> AI
- Faltung -> Convolution

Quellen und Fußnoten sind folgendermaßen gekennzeichnet:

- Fußnoten: Wort^[XX]
- Quellen: Absatz. [XX]

Quellen sind im Quellenverzeichnis wie folgt formatiert:

- Autor | Titel des Artikels | Link | Abrufdatum
- Ist eine Information nicht vorhanden wird sie durch ein „-„ ersetzt.

Bevor Sie sich mit meinen Programmen beschäftigen, lesen Sie sich bitte erst README.txt durch.

Einleitung

Artificial Intelligence. Diese Maschinen, die angeblich bald die Welt übernehmen sollen. Die Algorithmen, die Autos selber fahren lassen, und dir in Social Media Apps nur für dich relevante Posts anzeigen. Die Technik, die GesichtsfILTER auf z.B. Snapchat oder Instagram erst möglich macht. Was ist das genau, und wie funktioniert es? In dieser Facharbeit geht es um Neural Networks und Convolutional Neural Networks und wie sie verwendet werden um Bilder zu klassifizieren (am Beispiel des CIFAR 10 Datensatzes). Dazu wird ein CNN in Python mit keras und tensorflow implementiert, trainiert und getestet.

Neuronale Netze

a. Was ist ein Neuronales Netzwerk?

Als Neuronales Netzwerk bezeichnet man ein Netz aus künstlichen Neuronen, das auf Basis eines Inputs einen Output geben kann. Es kann durch verschiedene Algorithmen trainiert werden damit das Ergebnis richtiger wird bzw. der sog. Verlust reduziert wird. Es gibt verschiedene Topologien (Strukturen) bei Neuronalen Netzwerken von denen einige sehr kompliziert sind (siehe „A mostly complete chart of Neural Networks“ im Anhang). Daher wird sich in dieser Facharbeit nur auf Neural Networks (oder „Deep Feed Forward“, wie es in dem Bild genannt wird) und deren Erweiterung Convolutional Neural Networks konzentriert. [2][3][13]

b. Was ist ein künstliches Neuron?

Ein künstliches Neuron ist eine mathematisch/informatisch implementierte und vereinfachte Version eines realen biologischen Neurons. Um zu verstehen was, warum und wie implementiert wurde, wird zuerst das Prinzip von biologischen Neuronen erläutert und dann ihre Implementierung.

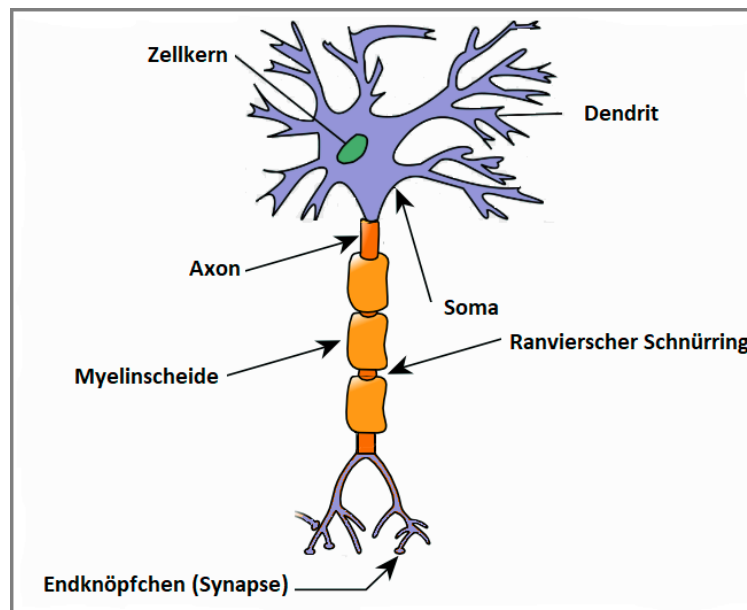


Fig. 1

Ein Neuron besteht aus 3 wesentlichen (für die Modellierung wichtigen) Bestandteilen:

I. Die Synapsen (siehe Fig.1). Die Synapsen sind die Schnittstellen zwischen Neuronen. Sie leiten Signale weiter und wirken dabei entweder hemmend oder erregend auf das Signal. Dieses Einwirken auf das Signal nennt man Gewicht und ist einer der trainierbaren Parameter eines NN. Der Prozess wird so modelliert, dass auf die Signale, die das künstliche Neuron erreichen sollen, vorher noch Gewichte einwirken. [4][6]

$$input = x * w$$

Hierbei steht x für das empfangene Signal und w für das Gewicht, das auf dieses Signal angewandt wird. [4][6]

II. Das Soma (siehe Fig.1). Das Soma sammelt alle Signale, die es von seinen umliegenden Dendriten bekommt. Je mehr und je stärker die empfangenen Signale, desto stärker auch die insgesamt Eingabe in den Zellkern. Dies kann in der Mathematik/Informatik so modelliert werden, dass alle Inputs addiert werden. Zu der Summe der Inputs wird dann noch ein Bias addiert. Was genau ein Bias ist wird später erklärt. Die letztendliche Summe aller Inputs, plus den Bias, bildet dann die Nettoeingabe des Neurons:

$$Nettoeingabe = \sum_{i=1}^n x_i * w_i + x_b * w_b$$

Normalerweise aber so geschrieben

$$z = \sum_{i=1}^n x_i * w_i + b$$

Wobei x an der Stelle i den i -ten Input, w an der Stelle i die i -te Gewichtung und n die gesamte Anzahl an Inputs darstellt. Den Bias kann man statt $x_b * w_b$ als w_b schreiben, da die Ausgabe (x_b) immer 1.0 ist. Der Verständlichkeit halber wird w_b aber einfach als b für Bias geschrieben. [4][6]

III. Der Axonhügel (siehe Fig.1 : der Teil, der Axon und Soma verbindet). Der Axonhügel in einem biologischen Neuron empfängt alle Signale, die das Neuron erhält. Dadurch baut sich ein Erregungspotential auf. Wird das Schwellenpotential erreicht, wird das Aktionspotential freigesetzt und durch das Axon ausgegeben. Dieser Prozess wird mit sogenannten Aktivierungsfunktionen umgesetzt, indem die vorher „gesammelte“ *Nettoeingabe* durch eine dieser Funktionen geleitet und der Output der Funktion als *Aktivierung* ausgegeben wird.

$$\text{Aktivierung} = \text{Aktivierungsfunktion}(\text{Nettoeingabe})$$

Einer Aktivierungsfunktion kann man bestimmte Eigenschaften zuordnen. Sie kann differenzierbar sein. D.h., dass sie zu jedem Zeitpunkt ableitbar ist, und sich dadurch gut für Neuronen in den Hidden Layers eignet, da man für dessen Training die Ableitungen der Aktivierungsfunktionen benötigt (mehr dazu später). Eine Aktivierungsfunktion kann zudem noch dabei helfen die Werte innerhalb des Netzwerkes zu stabilisieren, indem sie eine klare Unter- und/oder Obergrenze festlegt. Außerdem kann sie noch nicht-linear bzw. linear sein. Die Nicht-linearität der Aktivierungsfunktionen ermöglicht es dem NN auch komplexe nicht-lineare Zusammenhänge zu erkennen und zu lernen. [4][6]

Einige Aktivierungsfunktionen sehen zum Beispiel so aus:

Die Schwellenwertfunktion (siehe Fig. 2) folgt, genau wie der Axonhügel in einem biologischen Neuron, dem „Alles-oder-nichts“ Prinzip. Das heißt, die Funktion gibt 0.0, aus solange der Input negativ, bzw. unter dem Schwellenwert, und 1.0 solange er positiv, bzw. über oder gleich dem Schwellenwert, ist. Sie ist teilweise linear und nicht durchgehend differenzierbar.

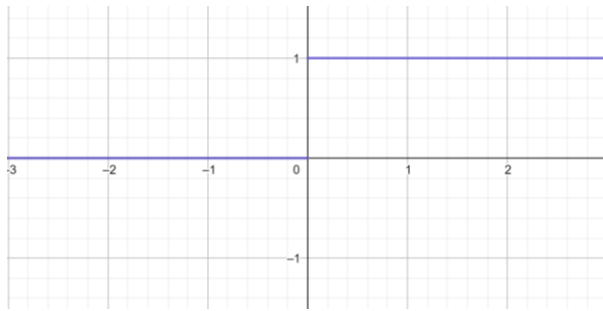


Fig. 2

Die Sigmoidfunktion (siehe Fig. 3) war für lange Zeit die Standard Aktivierungsfunktion. Sie ist nicht-linear, durchgehend differenzierbar und begrenzt den Output auf den Zahlenraum zwischen 0.0 und 1.0. Aufgrund der Begrenzung ihres Outputs ist die Sigmoidfunktion (oder die etwas generalisiertere Form: die Softmaxfunktion) oft in Klassifizierungsanwendungen zu finden, da ihr Output direkt als Wahrscheinlichkeit interpretiert werden kann. Sie wurde oft in Hiddenlayers in NNs eingesetzt, jedoch später von der sog. Tanh-Aktivierungsfunktion abgelöst. [5]

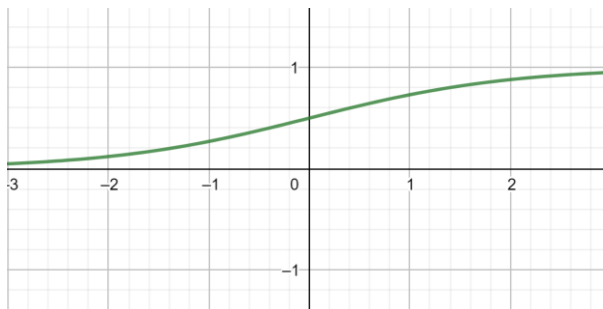


Fig. 3

Die Tanh-Funktion (siehe Fig. 4) löste die Sigmoidfunktion als Standardaktivierungsfunktion für Hiddenlayers in NNs ab. Sie ist nicht-linear, durchgehend differenzierbar und begrenzt den Output auf -1.0 bis 1.0 . Die Tanh-Funktion wird zur Klassifizierung zwischen zwei Klassen genutzt, wurde aber später mit der ReLU-Funktion ersetzt.

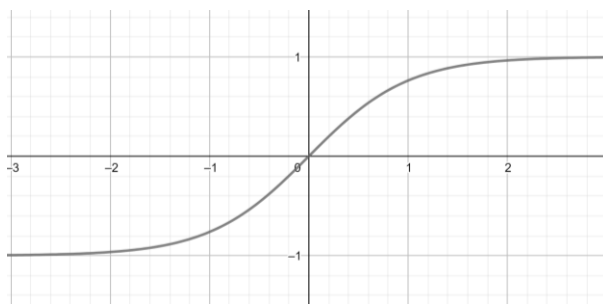


Fig. 4

Die Rectified Linear Unit-Funktion, kurz ReLU, (siehe Fig. 5) gibt 0.0 aus, wenn der Input negativ, und den Input selber, wenn dieser positiv ist. Sie wird in den meisten modernen NN als Aktivierungsfunktion in den Hiddenlayers genutzt, da sie zwar nicht-linear ist, aber nah genug an der Linearität um eine Verbesserung der Lernfähigkeit von NNs zu erreichen. Außerdem umgeht sie das Problem des verschwindenden Gradienten^[6].

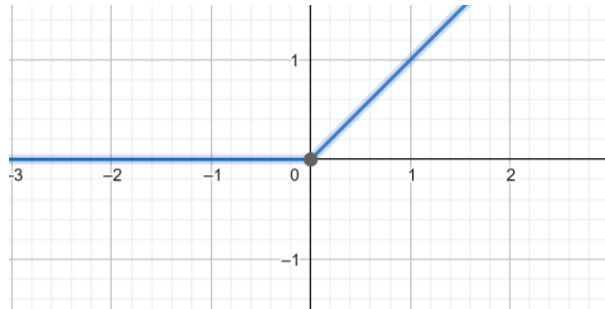


Fig. 5

Mit diesen Informationen sieht die Gleichung für die Aktivierung nun wie folgt aus:

$$a(x) = f(x)$$

Wobei a für die Aktivierung und f für eine beliebige Aktivierungsfunktion steht. [14]

c. Das XOR-Problem

NNs sahen in ihrer Anfangszeit noch aus wie in Fig.6. Sie bestanden ausschließlich aus einer Outputschicht. Diese Form eines NNs ermöglichte die Auflösung einfacher Operatoren wie OR (siehe Fig.7).

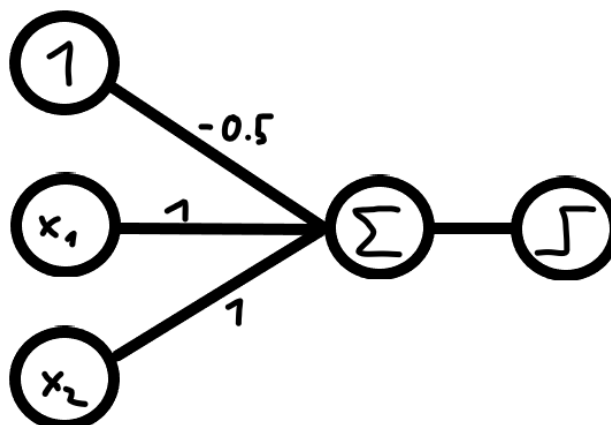


Fig. 6

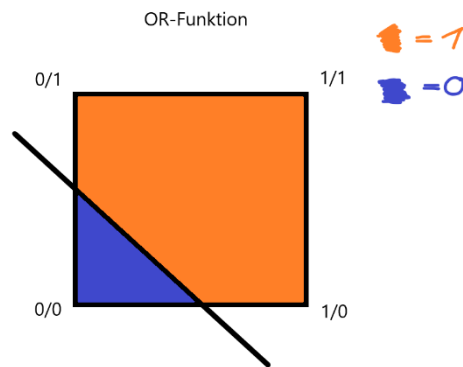


Fig. 7

Befehle für dieses Muster:
 >neuralNetworkFromConsole.py -e 2 -h [] -o 1
 Training inputs: [[1,0],[0,1],[1,1],[0,0]]
 Training outputs: [[1],[1],[1],[0]]
 >>>visualize [[1,0],[0,1],[1,1],[0,0]] false

Wie in der Fig. 8 zu sehen ist kann die Datenmenge linear separiert, d.h. man kann mit einer einzigen Gerade die verschiedenen Outputs (hier Blau/0 und Orange/1) voneinander trennen, und somit von einem einlagigen Perzeptron (siehe Fig. 7) gelöst werden. Schwieriger wird es, wenn dies nicht mehr der Fall ist, wie bei dem XOR-Operator.

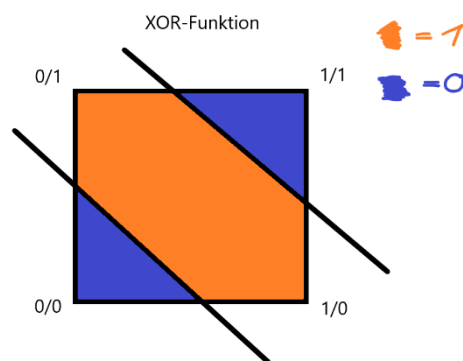


Fig. 8

Befehle für dieses Muster:
 >neuralNetworkFromConsole.py -e 2 -h [2] -o 1
 Training inputs: [[1,0],[0,1],[1,1],[0,0]]
 Training outputs: [[1],[1],[0],[0]]
 >>>visualize [[1,0],[0,1],[1,1],[0,0]] false

Wie in Fig.9 zu sehen ist kann die Datenmenge nicht mit einer linearen Funktion in Blau/0 und Orange/1 separiert werden. Die Lösung für dieses Problem waren die Hiddenlayers. Eine Hiddenlayer ist eine Schicht an Neuronen, dessen Input entweder direkt der Input des NNs oder der Output einer weiteren Hiddenlayer ist. Für ein

Hiddenlayerneuron ist es typisch mit jedem Neuron der vorherigen und der folgenden Schicht verbunden zu sein. Nach der letzten Hiddenlayer folgt die Outputlayer. [8]

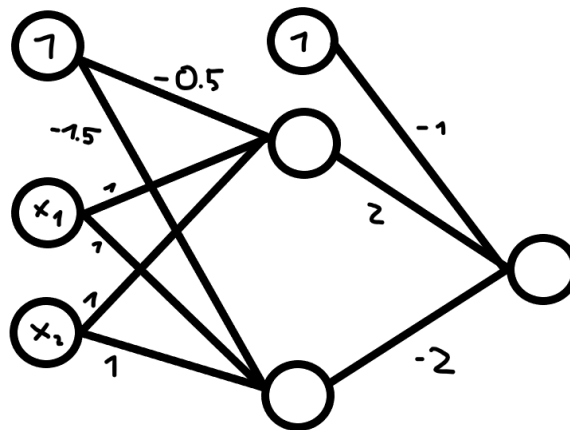


Fig. 9

Hier (Fig.10) sieht man ein mehrschichtiges Perzeptron, welches in der Lage ist den XOR-Operator aufzulösen. Das obere Neuron in der Hiddenlayer ist, von den Gewichten her, genau wie das Outputneuron aus Fig.7 konfiguriert und ist deswegen für OR zuständig. Das untere Neuron berechnet AND und das Outputneuron damit XOR.

d. Warum der Bias wichtig ist

Der Bias kann als ein spezielles Neuron gesehen werden. Er ist deshalb nicht wie die anderen, weil er immer eine Aktivierung von 1.0 und immer nur auf eine Schicht direkten Einfluss hat. Das heißt pro Schicht, außer der Inputschicht, gibt es ein Biasneuron mit der Aktivierung 1.0 und gewichteten Verbindungen zu jedem Neuron der jeweiligen Schicht. Der Bias ist deshalb wichtig, weil es z.B. unmöglich ist für ein einlagiges Perzeptron/NN ohne Bias bei einem Input von 0.0 und der Schwellenwertfunktion als Aktivierungsfunktion einen Output von 0.0 zu haben, da:

$$\sum_{i=1}^n 0 * w_i = 0$$

Und:

$$\text{Schwellenwertfunktion}(0) = 1$$

Der Bias bewirkt eine Verschiebung der Aktivierungsfunktion auf der x-Achse.

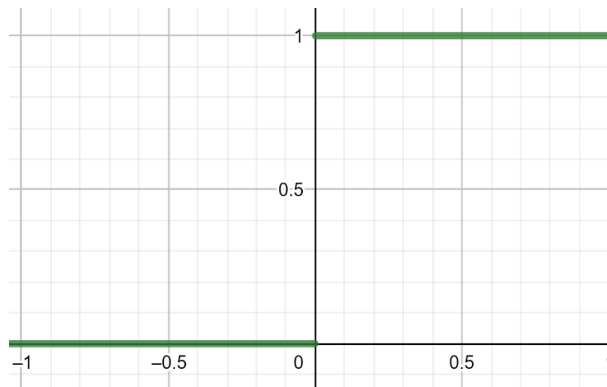


Fig. 10

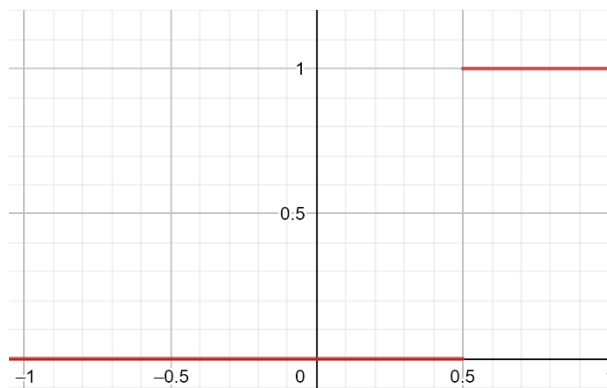


Fig. 11

In Fig. 11 ist die normale und in Fig. 12 die verschobene Schwellenwertfunktion mit einem Bias von -0.5 zu sehen. Die Funktion aus Fig. 12 ist die Aktivierungsfunktion des OR-Perzeptrons (siehe Fig.7). Weil durch ihre Verschiebung die Summe der beiden Inputs mindestens 0.5 erreichen muss, ergibt sich das Muster aus Fig.9 und der OR-Operator ist aufgelöst.

e. Das Trainieren von Neuronalen Netzen

Für das Trainieren von NN wird u.a. der Backpropagation-Algorithmus genutzt. Dabei wird das Gradientenabstiegsverfahren auf die Verlustfunktion^[3] des NN angewendet um die Parameter, die Gewichtungen, des Netzes so zu optimieren, dass der Verlust möglichst gering wird (ein lokales Minimum in der Verlustfunktion erreicht wird). Dazu wird hier ein Verfahren namens „supervised Learning“ (zu Deutsch „überwachtes Lernen“) verwendet. In diesem werden dem NN Trainingsbeispiele, von denen man den erwarteten Output kennt, als Input gegeben werden und dann berechnet wird, wie weit der gegebene Output von dem erwarteten Output abweicht. Neben überwachtem Lernen gibt es noch einige andere Methoden, wie „Reinforcement Learning“ oder „Inductive Learning“, die hier jedoch den Rahmen sprengen würden. [11]

Für dieses Beispiel wird ein NN mit zwei Inputs, zwei Hiddenlayerneuronen und einem Outputneuron verwendet:

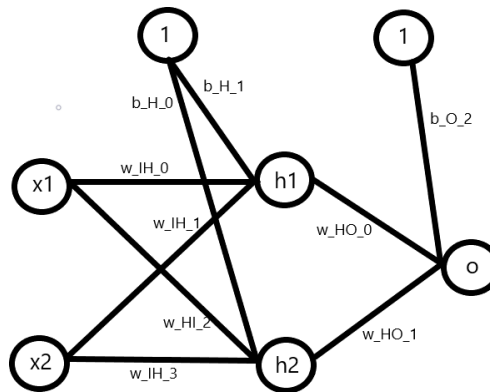


Fig. 12

Angenommen das NN hat soeben einen Trainingsinput erhalten und einen Output gegeben. Jetzt wird der Verlust mithilfe der sog. „Mean squared Error“-Funktion berechnet:

$$mse = \sum_{i=0}^N (\hat{y}_i - a^o_i)^2$$

Alternative Schreibweise ohne Summenzeichen:

$$mse = (\hat{y}_0 - a^o_0)^2 + \dots + (\hat{y}_N - a^o_N)^2$$

Von jetzt an werden hochgestellte Buchstaben für Schichten stehen (O=Outputlayer, H=Hiddenlayer, I=Inputlayer, IH=Gewicht von der Inputlayer in eine Hiddenlayer, ...) und tiefgestellte Zahlen/Buchstaben für die Position innerhalb einer Schicht (von oben nach unten).

Wobei N die Anzahl an Neuronen in der Outputlayer, \hat{y}_i der richtige Output für das i -te Outputneuron und a^o_i der gegebene Output (Aktivierung) des i -ten Outputneurons ist. Die partielle Ableitung dieser Funktion zu einem der Gewichte gibt an in welche Richtung dieses justiert werden muss (+ oder -) um den größten Zuwachs zu erhalten. Da man aber die größte Abnahme als Ziel hat, muss logischerweise in die entgegengesetzte Richtung justiert werden. Das heißt:

$$\Delta w = \text{Verlust}'(w)$$

$$w_{neu} = w - \Delta w$$

Anfangen mit den Gewichten, die die Hiddenlayer mit der Outputlayer verbinden, wird die partielle Ableitung zu w^{HO} , mit Hilfe der Verkettungsregel, folgendermaßen gebildet:

$$Verlust = mse \circ a \circ z$$

$$\frac{\partial Verlust}{\partial w^{HO}} = \frac{\partial Verlust}{\partial a^O} * \frac{\partial a^O}{\partial z^O} * \frac{\partial z^O}{\partial w^{HO}}$$

$$\frac{\partial Verlust}{\partial w^{HO}_0} = -2(\hat{y}_0 - a^O_0) * sigmoid'(z^O_0) * a^H_0$$

Wobei a^H die Aktivierung des Hiddenlayerneurons ist, das mit w^{HO} verbunden ist. Ähnlich sieht die Gleichung für den Bias aus, mit dem einzigen Unterschied, dass nun z in Abhängigkeit von dem Bias benötigt wird:

$$\frac{\partial Verlust}{\partial b^O} = \frac{\partial Verlust}{\partial a^O} * \frac{\partial a^O}{\partial z^O} * \frac{\partial z^O}{\partial b^O}$$

$$\frac{\partial Verlust}{\partial b^O_0} = -2(\hat{y}_0 - a^O_0) * sigmoid'(z^O_0) * 1$$

Um die Änderung der Gewichte in den hinteren Schichten (w^{IH}) zu berechnen muss nur die Verkettungsregel erweitert werden, indem nun z^O in Abhängigkeit von der Aktivierung der vorherigen Schicht a^H gebildet wird. Da jedoch die Hiddenlayerneuronen, zu denen die Gewichte der hinteren Schichten verbunden sind, (oft) selber durch mehrere Gewichte zu mehreren Outputneuronen verbunden sind, beeinflusst eine Änderung eines Gewichts in einer hinteren Schicht auf mehreren Wegen den Gesamtoutput des NN. Dann reicht es nicht den Gesamtverlust in Abhängigkeit von nur einer Aktivierung der Outputschicht zu berechnen, sondern man muss den Verlust in Abhängigkeit von allen, von dem zu ändernden Gewicht beeinflussten, Outputneuronen berechnen. Dazu wird der Verlust in Abhängigkeit von a^H , durch die Summe der Abhängigkeiten des Verlustes von a^H , über alle von w^{IH} beeinflussten Outputneuronen berechnet:

$$Verlust = mse \circ a^O \circ z^O \circ a^H \circ z^H$$

$$\frac{\partial Verlust}{\partial w^{IH}} = \left(\sum_{j=0}^N \frac{\partial Verlust}{\partial a^O_j} * \frac{\partial a^O_j}{\partial z^O_j} * \frac{\partial z^O_j}{\partial a^H} \right) * \frac{\partial a^H}{\partial z^H} * \frac{\partial z^H}{\partial w^{IH}}$$

N steht hier für die Anzahl an Neuronen in der Outputlayer. Das kann jedoch in diesem konkreten Beispiel vernachlässigt werden, da nur ein Outputneuron existiert.

$$\frac{\partial \text{Verlust}}{\partial w^{IH}_0} = \left(\sum_{j=0}^N -2(\hat{y}_j - a^O_j) * \text{sigmoid}'(z^O_j) * \frac{\partial z^O_j}{\partial a^H_0} \right) * \text{sigmoid}'(z^H_0) * x_0$$

($\frac{\partial z^O_j}{\partial a^H_0}$ hier nicht aufgelöst, da der Wert variiert, je nach z^O_j)

Und für den Bias gilt dasselbe Prinzip wie vorher:

$$\frac{\partial \text{Verlust}}{\partial b^H_0} = \left(\sum_{j=0}^N -2(\hat{y}_j - a^O_j) * \text{sigmoid}'(z^O_j) * \frac{\partial z^O_j}{\partial a^H_0} \right) * \text{sigmoid}'(z^H_0) * 1$$

Bei der Implementation ist zudem noch ein besonderes Augenmerk darauf zu legen, dass man erst alle Gewichtveränderungen berechnet, und dann die Gewichte aktualisiert. Das liegt daran, dass die unveränderten Gewichte erst noch in weiteren Kalkulationen benutzt werden müssen, und eine vorzeitige Veränderung zu verfälschten Ergebnissen führen würde. [7][10][12]

Zusammengefasst, wird also pro Trainingsbeispiel der Verlust berechnet und basierend darauf die Gewichte allesamt so verändert, dass der Verlust geringer wird.

Dabei kann es jedoch passieren, dass Δw zu groß ist und das lokale Minimum der Verlustfunktion übergangen wird:

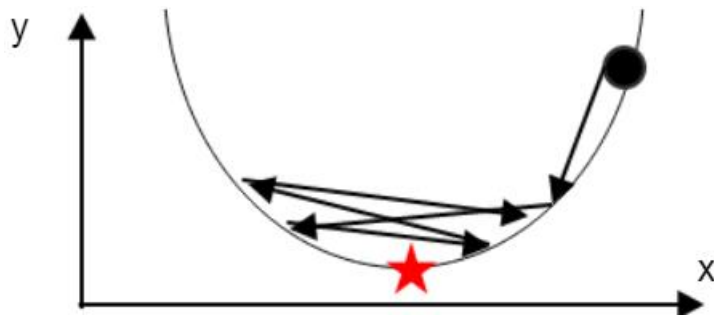


Fig. 13

Dafür wird Δw mit einer weiteren Konstante multipliziert: der Learningrate. Diese vergrößert, bzw. verkleinert, die "Schrittlänge" beim Trainieren. Bei einer großen Learningrate sind weniger Trainingsdurchläufe von nöten, da pro Durchlauf Δw größer ist, jedoch ist die Wahrscheinlichkeit auch höher, dass ein optimales Minimum übergangen wird (siehe Fig. 14). Eine zu kleine Learningrate kann hingegen dazu führen, dass das NN beim Trainieren "stecken bleibt", d.h., dass es sich auf einer Suboptimalen Lösung festsetzt. Die Learningrate wird meistens im Bereich von 0.0 bis 1.0 gewählt und ist einer der wichtigsten Hyperparameter^[4] für ein NN. [12][15]

Convolutional Neural Networks

a. Was ist ein Convolutional Neural Network?

Angenommen, es soll eine Gesichtserkennungssoftware geschrieben werden. Dann kann ein NN aus Inputs wie „Ist ein Auge vorhanden?“, „Ist daneben noch ein Auge?“ und „Ist darunter eine (menschliche) Nase?“ erkennen, ob es sich um ein Menschliches Gesicht handelt oder nicht. Das Problem dabei ist, dass ein NN von sich aus soetwas nicht (bzw. nur schwer) erkennen kann. Es bräuchte einen Menschen, der sich das Bild anschaut und dem NN die Inputs bereitstellt. Das ist viel zu aufwändig, weswegen die Convolutional Layers (und damit Convolutional Neural Networks) erfunden wurden. Sie sind wie ein Filter, der die unnötigen Informationen herausfiltert und die wichtigen Features („Merkmale“) zum Klassifizieren an das NN weitergibt. Sie erkennen Zusammenhänge zwischen zueinander nahen Pixeln, wobei die globale Position dieser Zusammenhänge relativ egal ist und sie dadurch gut kleine Variationen wie Verschiebung oder Drehung ignorieren können.

Ähnlich wie NNs sind CNNs in Schichten aufgebaut, jedoch gibt es verschiedene Schichten innerhalb der CNNs, wie Convolutional Layers oder Pooling Layers, die alle zusammenarbeiten um die sog. Features aus dem Bild zu extrahieren, damit die Dense Layer/s (ein NN) zum Schluss nur noch die Klassifizierung übernehmen müssen.

Die einzelnen Schichten werden im folgenden erklärt:

b. Die Convolutional Layer

Eine Convolutional Layer in einem CNN übernimmt die sog. Convolution („Faltung“) des Bildes. Dabei werden die Features in dem Bild erkannt, und die daraus entstehenden Featuremaps an die nächste Schicht weitergegeben. Ein Feature kann in der ersten Convolutional Layer z.B. eine horizontale Kante oder eine Ecke sein, in der folgenden vielleicht ein Kreis, ein Quadrat oder ein Kreuz und in der letzten sogar ein ganzes Gesicht. Man kann also sagen, dass die Features, die eine Convolutional Layer erkennt, immer komplexer werden, je tiefer die Schichten reichen. Aber wie erkennt denn nun eine Convolutional Layer Features? Mit ihren sog. Filtern.

Ein Filter ist eine meistens 3x3 oder 5x5 Pixel große Matrix an Dezimalzahlen, die über das Bild, bzw. einen Bildausschnitt, gelegt wird und dann angibt, wie genau die Pixelwerte des Bildausschnittes mit ihren Werten übereinstimmen, also ob das Muster, das der Filter erkennen soll, vorhanden ist. [16][17]

Als Beispiel wird ein 2x2 Pixel großes Bild angenommen, dessen Pixelwerte 1 oder -1 sein können. Als Filter wird ein vorgefertigter Filter für eine Diagonale von oben links nach unten rechts genommen.

Filter:

1	-1
-1	1

Fig. 14

Beispielbilder:

1	-1	-1	1
-1	1	1	-1

Beispielbild 1

Beispielbild 2

Fig. 15

Angenommen der Filter wird nun auf Beispielbild 1, das offensichtlich besagtes Muster des Filters enthält, angewandt:

$$z_{i,j} = \sum_{m=0}^{kernelwidth} \sum_{n=0}^{kernelheight} (input_{i+m,j+n} * kernel_{m,n}) + bias_{i,j}$$

$$o_{i,j} = a(z_{i,j})$$

$$output_{0,0} = 5$$

(Mit der ReLU-Funktion als Aktivierungsfunktion und einem Bias von 1.0)

Erkennt dieser das Muster und zeigt dies mit einer hohen Ausgabe.

Ist der Input jedoch ein Bild wie Beispielbild 2, das offensichtlich das Muster des Filters nicht enthält, so ist der Output des Filters 0.0 bzw. eher niedriger, was bedeutet, dass das Muster in gegebenem Bildausschnitt nicht erkannt wurde.

$$output_{0,0} = 0$$

Wie in den NNs gibt es auch in CNNs Aktivierungsfunktionen. Meist wird dafür die ReLU-Funktion benutzt. [16]

Jetzt ist ein Bild, das einem Convolutional Neural Network als Input gegeben wird, nicht immer nur 2x2 Pixel groß, weswegen die Filter über das gesamte Bild bewegt werden müssen. Dabei wird in der oberen linken Ecke angefangen und schrittweise der Filter nach rechts bewegt. Ist er an einer Kante angekommen, so wird er einen Schritt nach unten und wieder an den linken Rand versetzt. Dann wird er wieder nach rechts bewegt usw. bis das komplette Bild vom Filter „begangen“/„untersucht“ wurde. Die Schritte in denen sich der Filter bewegt sind ein weiterer Hyperparameter und bei Convolutional Layers oft auf 1 Pixel gesetzt. Mit jedem Schritt hat der Filter aber auch eine Ausgabe (ob nun ein Muster gefunden wurde, oder nicht). Aus diesen Ausgaben wird Stück für Stück der Output der aktuellen Convolution konstruiert.

Ein Beispiel aus der echten Welt wäre dieses Bild:



Fig. 16

Wird nun auf dieses Bild ein 3x3 Filter für horizontale Kanten angewandt

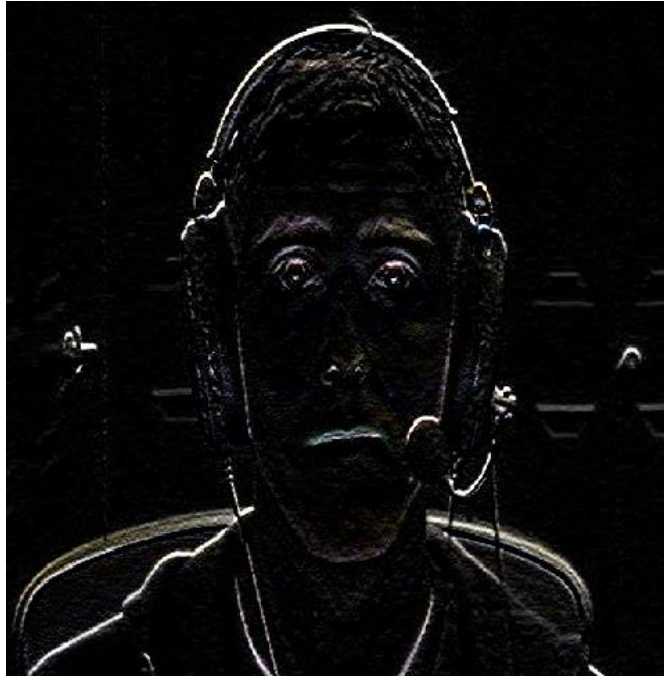


Fig. 17

(Die Formatierung der Seitenverhältnisse liegt an meinem Programm (layerApplyingTool.py), das kommt nicht von dem Filter)

ist sehr deutlich zu sehen, dass die horizontalen Features, wie sein Mund, deutlich herausstechen und vertikale Features, wie die Schlitze in den Schranktüren hinter der Person, komplett ausgeblendet werden. Diese sind wiederum in der Version mit dem Filter für vertikale Kanten hervorgehoben.



Fig. 18

Ein Filter kann also Stellen an denen ein bestimmtes Muster/Feature vorkommt ausfindig machen und in seinem Outputbild markieren.

Schaut man sich das Outputbild mit dem vertikalen Filter aber an, so wird schnell klar, dass z.B. der Mund gar nicht (oder nur sehr leicht) wahrgenommen wurde. Das heißt es reicht für eine Convolutional Layer nicht, nur einen Filter zu haben, da dieser nur eine Art von Mustern erkennen kann. Stattdessen werden mehrere Filter verwendet.

Eine Convolutional Layer wendet nun also nicht mehr nur einen Filter auf das Inputbild an, sondern mehrere, was wiederum bedeutet, dass der Output dann nicht mehr nur ein Bild ist, sondern ein Stapel Bilder. Der Stapel kann dann in der nächsten Convolutional Layer nicht mehr mit einlagigen Filtern bearbeitet werden, sondern braucht Filter, die genauso viele Schichten haben wie der Input. Das Grundprinzip bleibt dabei gleich, nur dass jetzt jeder Pixel in jeder Schicht mit seinem jeweiligen Partner multipliziert und dann alle Produkte aller Schichten addiert werden. [18]

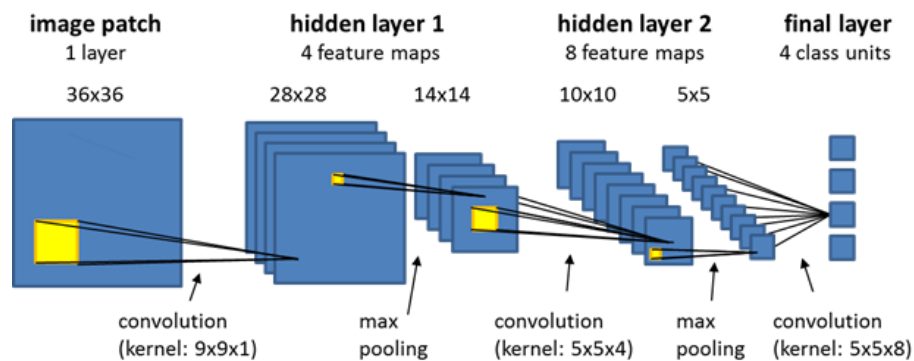


Fig. 19

In Fig.20 wird das Inputbild mit vier einschichtigen 9x9 Filtern bearbeitet (einschichtiger 9x9 Filter = 9x9x1 Filter). Deswegen ist der Output ein vierschichtiger 28x28 Stapel Bilder (28x28x4). Dieser wird durch eine Pooling Layer verkleinert (wird noch erklärt) und dann von der nächsten Convolutional Layer mit acht 5x5x4 Filtern zu acht Featuremaps „gefaltet“.

Zusammenfassend also: Eine Convolutional Layer nimmt einen Stapel (ggf. einlagig) Bilder als Input und bearbeitet ihn mit Filtern, sodass der Output ein Stapel Featuremaps ist, der so viele Bilder enthält wie es Filter in der Schicht gab.

c. Die Pooling Layer

Auf eine Convolutional Layer folgt meistens auch eine Pooling Layer. Ihre Aufgabe ist es das Bild/den Bilderstapel zu verkleinern, sodass das NN am Ende weniger Inputs hat, dabei aber alle wichtigen Signale beizubehalten, damit keine wichtigen Informationen verloren gehen. [16]

Es gibt zwei verschiedene Pooling Methoden: Average Pooling und Max Pooling. Beide haben einen Kernel den sie, wie in den Convolutional Layers, über das Bild bewegen. Der Kernel ist in einer Pooling Layer meist 2x2 Pixel groß und bewegt sich mit einer Schrittlänge von 2 Pixel über das Bild, sodass die Bildgröße halbiert wird (mit leichten Abweichungen bei ungeraden Größen. Stichwort Same-, Valid-, etc. Padding). Die Berechnungen, die die beiden Methoden durchführen, sind jedoch unterschiedlich. [19]

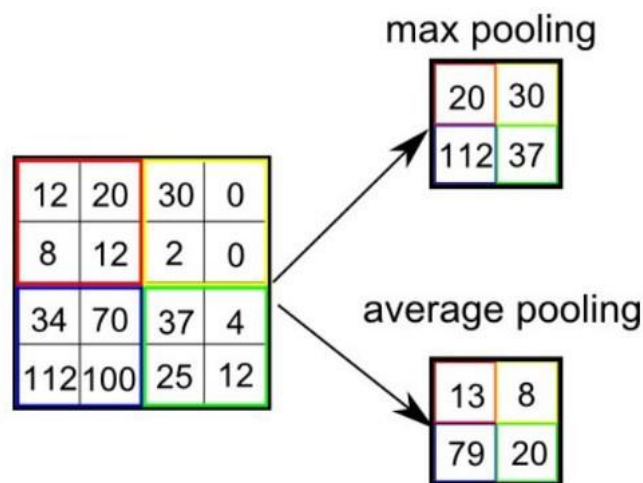


Fig. 20

Beim Average Pooling werden die Werte, die vom Kernel bedeckt werden, addiert und durch die Anzahl an Werten dividiert, sodass der Durchschnitt aller vom Kernel bedeckten Werte genommen wird. [19]

$$output_{i,j} = \frac{\sum_{m=0}^{kernelwidth} \sum_{n=0}^{kernelheight} input_{(i*stride)+m,(j*stride)+n}}{kernelwidth * kernelheight}$$

Durch diese Art von Pooling wird eine Bildverkleinerung durchgeführt und Features werden weicher übernommen. [20]

Normalerweise wird aber Max Pooling benutzt. Hierbei wird von allen Werten, die der Kernel gerade bedeckt, der größte ausgewählt und als Output gegeben.

$$output_{i,j} = \max(\text{alle inputs die der kernel bedeckt})$$

Bei dieser Methode werden nur die relevantesten Signale an die nächste Schicht weitergegeben und das Bild entrauscht. [16][19]

Um die gesammelten high-level Features zu klassifizieren schließt das Netzwerk mit einer (oder mehreren) „Fully Connected Layers“ (oder „Dense Layers“, wie keras sie nennt) ab. Diese Schichten sind aufgebaut wie ein normales NN, wobei der Input direkt aus der Convolution kommt. Es nimmt also den Output der letzten Schicht der Featureextrahierung als Inputs für die Inputneuronen, welche zu den/dem Outputneuron(en) über gewichtete Verbindungen verbunden sind. Diese Gewichte können dann, über normale Backpropagation, lernen welches der high-level Features für welche Klasse spricht. [21][22]

d. Die Flattening Layer

Da der Output der Convolution aber ein Bild ist, muss dieser erst für das NN verständlich formatiert werden. Dafür ist die sog. Flattening Layer zuständig. Sie konvertiert den Output den die letzte Schicht der Convolution gibt in einen eindimensionalen Array aus Dezimalzahlen. [23]

e. Das Trainieren von Convolutional Neural Networks

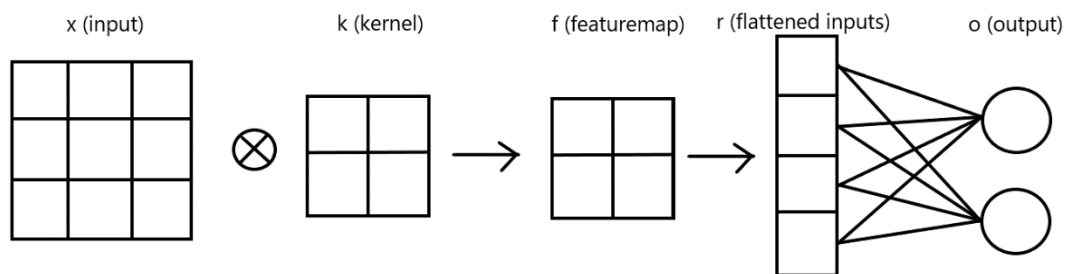


Fig. 21

Für das Training eines CNNs wird das in Fig.22 gezeigte simple CNN verwendet. Der Input (x) ist ein einlagiges Bild welches mit dem Kernel (k) in einer Convolution Operation transformiert wird. Der Output dieser Operation ist dann die Featuremap (f) und wird nach dem Flattening der Fully Connected Layer als Input übergeben. Dieses unternimmt die Klassifizierung.

Damit ein NN lernen kann braucht es Parameter, die es optimieren kann. Diese waren im normalen NN die Gewichte, und sind in CNNs die Pixelwerte der Kernels (und natürlich die Gewichte der Fully Connected Layers). Das was diese Gewichte

unterscheidet ist das sog. „weight sharing“. Das heißt, dass dasselbe Gewicht nicht nur zwei Neuronen (hier Pixel) verbindet, sondern noch andere Paare.

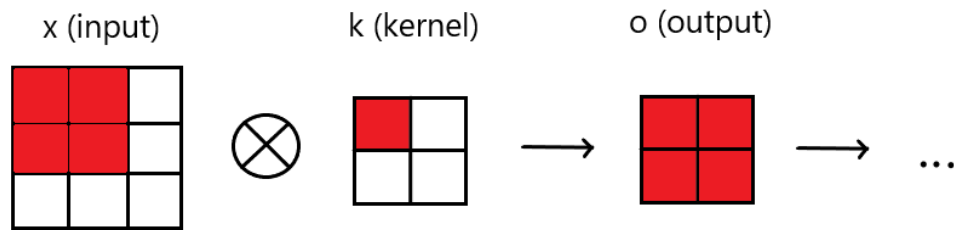


Fig. 22

(Hier sind alle Pixel in rot markiert, die durch das rote Gewicht verbunden sind)

Denn, um eine Convolution durchzuführen, wird in der Berechnung eines jeden Outputpixels jedes Gewicht des Kernels verwendet (siehe Funktionen zur Berechnung der Outputpixel), dadurch ist jeder Pixel der Outputfeaturemap mit denselben Gewichten zu jeweils unterschiedlichen Pixeln des Inputs verbunden. Man kann also sagen: die Pixel der Outputfeaturemap teilen (eng. „to share“) sich die Gewichte (eng. „weights“) des Kernels. Durch diese Methode müssen deutlich weniger Parameter erlernt werden und die erforderliche Rechenleistung wird gesenkt. [24]

Um diese Gewichte nun zu trainieren, kann wieder der Backpropagation Algorithmus angewendet werden. In der Fully Connected Layer können die Gewichte wie schon bekannt trainiert werden. Um die Kernels/Filter zu lernen ist das Prinzip zwar unverändert, aber die Rechnung leicht anders.

$$\frac{\partial E}{\partial k_{i,j}} = \sum_{m=0}^{\text{width } o} \sum_{n=0}^{\text{height } o} \frac{\partial E}{\partial o_{m,n}} * \frac{\partial o_{m,n}}{\partial z_{m,n}} * \frac{\partial z_{m,n}}{\partial k_{i,j}}$$

Wie vorher ist die Verlustfunktion in Abhängigkeit von dem zu lernenden Gewicht zu berechnen. Dadurch, dass der Wert von $k_{i,j}$ jeden Outputpixel zu jeweils einem Inputpixel verbindet, wird eine Änderung des Gewichtes den Wert eines jeden Outputpixels verändern, und so über jeden Outputpixel den Gesamtverlust beeinflussen. Deswegen wird $\frac{\partial E}{\partial k_{i,j}}$ unter Berücksichtigung des Einflusses aller Outputpixel berechnet.

Die partielle Ableitung des Verlustes in Abhängigkeit von jeweils einem Outputpixel kann in diesem Beispiel mit den schon bekannten Formeln berechnet werden, da die Outputpixel einfach der Input des neuronalen Netzes sind.

$$\begin{aligned}\frac{\partial E}{\partial k_{i,j}} &= \sum_{m=0}^{width\ o} \sum_{n=0}^{height\ o} \left(\sum_{p=0}^{outputs} \frac{\partial E}{\partial a_p^o} * \frac{\partial a_p^o}{\partial z_p^o} * \frac{\partial z_p^o}{\partial o_{m,n}} \right) * \frac{\partial o_{m,n}}{\partial z_{m,n}} * \frac{\partial z_{m,n}}{\partial k_{i,j}} \\ &= \sum_{m=0}^{width\ o} \sum_{n=0}^{height\ o} \delta_{n,m}^o * \frac{\partial o_{m,n}}{\partial z_{m,n}} * \frac{\partial z_{m,n}}{\partial k_{i,j}}\end{aligned}$$

Die partielle Ableitung des Outputs zu einem der Gewichte ist auch in den Convolutional Layers ein Input. Die Formel sieht nun so aus:

$$\frac{\partial E}{\partial k_{i,j}} = \sum_{m=0}^{width\ o} \sum_{n=0}^{height\ o} \delta_{n,m}^o * a'(z_{m,n}) * x_{i+n,j+m}$$

Mit dieser Methode lassen sich die Filter in der Schicht direkt hinter der Fully Connected Layer berechnen. Für die Schichten dahinter muss die Verkettungsregel, sowie das Beispiel CNN erweitert werden.

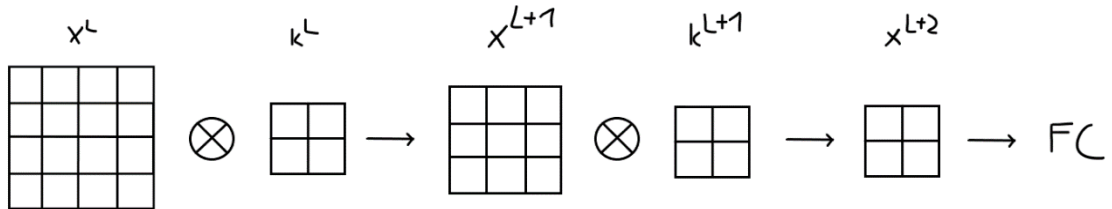


Fig. 23

Jetzt ist also $\frac{\partial E}{\partial k_{i,j}^L}$ in einer hinteren Schicht gesucht.

$$\frac{\partial E}{\partial k_{i,j}^L} = \sum_{m=0}^{width\ x^{L+1}} \sum_{n=0}^{height\ x^{L+1}} \frac{\partial E}{\partial x_{m,n}^{L+1}} * \frac{\partial x_{m,n}^{L+1}}{\partial z_{m,n}^{L+1}} * \frac{\partial z_{m,n}^{L+1}}{\partial k_{i,j}^L}$$

Die Gewichte $k_{i,j}^{L-1}$ haben nur direkten Einfluss auf den Output x^L , was bedeutet, dass nun der Verlust in Abhängigkeit von allen Pixeln dieser Featuremap gesucht ist.

$$\frac{\partial E}{\partial k_{i,j}^L} = \sum_{m=0}^{width\ x^{L+1}} \sum_{n=0}^{height\ x^{L+1}} \left(\sum_{m'=0}^{width\ k^{L+1}} \sum_{n'=0}^{height\ k^{L+1}} \frac{\partial E}{\partial x_{m-m',n-n'}^{L+2}} * \frac{\partial x_{m-m',n-n'}^{L+2}}{\partial z_{m-m',n-n'}^{L+2}} * \frac{\partial z_{m-m',n-n'}^{L+2}}{\partial x_{m,n}^{L+1}} \right) * \frac{\partial x_{m,n}^{L+1}}{\partial z_{m,n}^{L+1}} * \frac{\partial z_{m,n}^{L+1}}{\partial k_{i,j}^L}$$

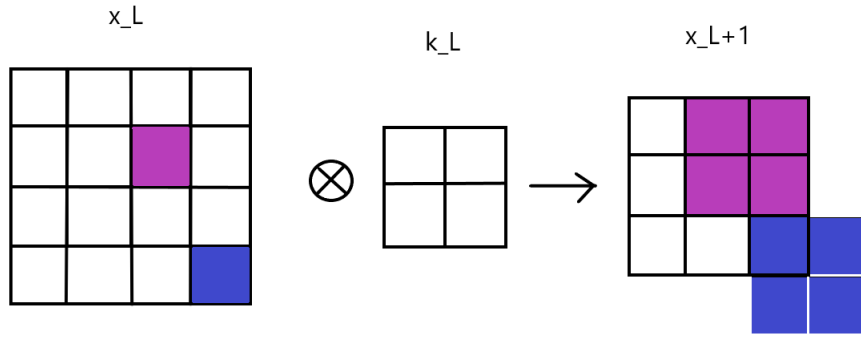


Fig. 24

Da ein Pixel der Featuremap x^L , wie in Fig.25 zu sehen, nicht Einfluss auf alle Outputpixel hat, wie es bei den Gewichten der Fall war, sondern nur auf einen kleineren Bereich, muss auch nur über besagten Bereich iteriert werden. Der Einflussbereich ist genau so groß wie der Kernel, wenn auch manchmal außerhalb des Outputs, weswegen über die Dimensionen des Kernels iteriert wird.

$\frac{\partial E}{\partial x^{L+2}_{m-m',n-n'}}$ ist in diesem Beispiel der Verlust in Abhängigkeit von den Outputs der letzten Convolutional Layer, also den Inputs des NNs, d.h. hier kann jetzt die Backpropagation der Convolutional Layers und die der Fully Connected Layers anknüpfen und die Formel ist komplett. Wenn jedoch das CNN noch tiefer reicht, also noch mehr Convolutional Layers hat, kann hier dieselbe „Erweiterung“ angewandt werden, wie in den letzten beiden Formeln, wodurch sich der Backpropagation Algorithmus auch hier auf beliebig viele Schichten anwenden lässt.

$$\frac{\partial E}{\partial k^L_{i,j}} = \sum_{m=0}^{width\ x^{L+1}} \sum_{n=0}^{height\ x^{L+1}} \left(\sum_{m'=0}^{width\ k^{L+1}} \sum_{n'=0}^{height\ k^{L+1}} \delta^{L+2}_{m-m',n-n'} * \frac{\partial x^{L+2}_{m-m',n-n'}}{\partial z^{L+2}_{m-m',n-n'}} * \frac{\partial z^{L+2}_{m-m',n-n'}}{\partial x^{L+1}_{m,n}} \right) * \frac{\partial x^{L+1}_{m,n}}{\partial z^{L+1}_{m,n}} * \frac{\partial z^{L+1}_{m,n}}{\partial k^L_{i,j}}$$

Um $\frac{\partial z^{L+2}_{m-m',n-n'}}{\partial x^{L+1}_{m,n}}$ aufzulösen, wird es erweitert.

$$\frac{\partial E}{\partial k^L_{i,j}} = \sum_{m=0}^{width\ x^{L+1}} \sum_{n=0}^{height\ x^{L+1}} \left(\sum_{m'=0}^{width\ k^{L+1}} \sum_{n'=0}^{height\ k^{L+1}} \delta^{L+2}_{m-m',n-n'} * \frac{\partial x^{L+2}_{m-m',n-n'}}{\partial z^{L+2}_{m-m',n-n'}} \right) * \frac{\partial}{\partial x^{L+1}_{m,n}} \left(\sum_{i'=0}^{width\ k^{L+1}} \sum_{j'=0}^{height\ k^{L+1}} (x^{L+1}_{m-m'+i',n-n'+j'} * k^{L+1}_{i',j'}) + b_{m-m',n-n'} \right) * \frac{\partial x^{L+1}_{m,n}}{\partial z^{L+1}_{m,n}} * \frac{\partial z^{L+1}_{m,n}}{\partial k^L_{i,j}}$$

$\frac{\partial}{\partial x^{L+1}_{m,n}}$ ist nur dann nicht 0, wenn $m-m'+i'=m$ und $n-n'+j'=n$. Das ist dann der Fall, wenn $m'=i'$ und $n'=j'$. Die anderen partiellen Ableitungen können wie schon bekannt aufgelöst werden, was dann in der folgenden finalen Formel resultiert:

$$\frac{\partial E}{\partial k_{i,j}^L} = \sum_{m=0}^{width x^{L+1}} \sum_{n=0}^{height x^{L+1}} \left(\sum_{m'=0}^{width k^{L+1}} \sum_{n'=0}^{height k^{L+1}} \delta_{m-m',n-n'}^{L+2} * a'(z_{m-m',n-n'}^{L+2}) * k_{m',n'}^{L+1} \right) * a'(z_{m,n}^{L+1}) * x_{i+m,j+n}^{L+1}$$

Damit ist Backpropagation durch Convolutional Layers abgeschlossen. Mit diesen Formeln lässt sich der Verlust durch beliebig viele Convolutional Layers zurückverfolgen und die Filter dementsprechend aktualisieren.

Eine weitere wichtige Komponente in CNNs sind die Pooling Layers, durch die der Verlust auch hindurchpropagiert werden muss. In Pooling Layers gibt es keine lernbaren Parameter, da diese lediglich die Größe der Featuremaps reduzieren. Der Verlust in Abhängigkeit zu den Outputs der Pooling Layer kann mit den schon bekannten Formeln berechnet werden.

Bei Max Pooling wird die größte Aktivierung der vorherigen Schicht unverändert weitergegeben, d.h. dass der Verlust in Abhängigkeit eines Outputpixels der Pooling Layer dem größten Inputpixel (einer Poolingoperation) eins zu eins zugeschrieben werden kann. Die Pixel, deren Werte nicht weitergegeben worden sind, haben keinen Einfluss auf den Verlust gehabt, wodurch die partielle Ableitung des Verlustes in Abhängigkeit von diesem Pixel gleich 0 ist.

Bei Average Pooling trägt jeder Inputpixel zum Output bei und hat dadurch auch Einfluss auf den Verlust. Die partielle Ableitung ist dadurch immer $\frac{1}{kernelwidth*kernelheight}$. [24]

$$\frac{x_0 + x_1 + \dots + x_N}{N} = (x_0 + x_1 + \dots + x_N) * \frac{1}{N} = \frac{1}{N}x_0 + \frac{1}{N}x_1 + \dots + \frac{1}{N}x_N$$

$$\frac{\partial}{\partial x} = \frac{1}{N}$$

Wobei $N = kernelwidth*kernelheight$.

Zusammenfassend kann man also sagen, dass der Backpropagation Algorithmus in CNNs, vom Prinzip her, gleich ist mit Backpropagation in NNs. Das was den Unterschied macht ist das weight sharing, wodurch die erforderliche Rechenleistung gesenkt wird, und das Arbeiten mit, teilweise dreidimensionalen, Matrizen, den Bildern, anstatt eindimensionalen Arrays.

f. Implementation eines CNN in Python mit tensorflow und keras

Nach der ganzen Theorie folgt jetzt die praktische Anwendung. Im folgenden wird ein CNN in der Programmiersprache Python mit den externen Modulen keras und tensorflow implementiert und auf den CIFAR 10-Datensatz trainiert und getestet.

In diesem Beispiel wird die supervised-learning Methode angewandt, was bedeutet, dass zum Trainieren ein Trainingsdatensatz mit dazugehörigen Beschriftungen (sog. „Labels“) von Nöten ist. Dieser wird von Tensorflow glücklicherweise bereitgestellt. Zusätzlich zu den Trainingsdaten wird ein weiterer Datensatz benötigt: der Validierungsdatensatz. Dieser Datensatz wird zum Testen während des Trainings verwendet, um zu zeigen, wie gut sich das CNN bei Daten schlägt, die es noch nie gesehen hat. In anderen Worten: Wie gut das CNN generalisieren kann. Auf Basis der Daten aus dem Validationsdatensatz wird daher keine Backpropagation durchgeführt. Um die Pixelwerte der Inputbilder zu normalisieren werden sie durch 255 dividiert. [25]

```
1 (train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
2 train_images, test_images = train_images / 255.0, test_images / 255.0
```

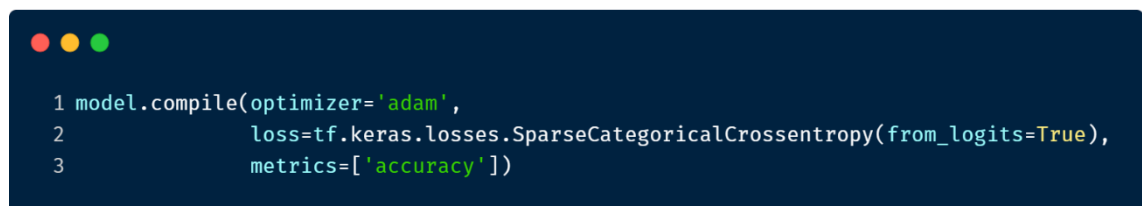
Fig. 25

Jetzt, da die Datensätze bereit sind um das CNN zu trainieren, muss das CNN, das „model“, erstellt werden.

```
1 model = models.Sequential([
2     layers.Conv2D( 32, (3, 3), activation="relu", padding="same", input_shape=(32, 32, 3) ),
3     layers.BatchNormalization(),
4     layers.Conv2D( 32, (3, 3), activation="relu", padding="same" ),
5     layers.BatchNormalization(),
6     layers.MaxPooling2D( (2, 2), strides=2 ),
7     layers.SpatialDropout2D( 0.1 ),
8
9     layers.Conv2D( 64, (3, 3), activation="relu", padding="same" ),
10    layers.BatchNormalization(),
11    layers.Conv2D( 64, (3, 3), activation="relu", padding="same" ),
12    layers.BatchNormalization(),
13    layers.MaxPooling2D( (2, 2), strides=2 ),
14    layers.SpatialDropout2D( 0.1 ),
15
16    layers.Flatten(),
17    layers.Dense( 256, activation='relu' ),
18    layers.Dropout( 0.5 ),
19    layers.Dense( 10, activation='softmax' )
20 ])
```

Fig. 26

Die Architektur, auf die ich mich nach *langem* Trial and Error festgelegt habe, besteht aus zwei Convolutional Layers mit jeweils 32 3x3 Filtern, der ReLU Aktivierungsfunktion und Same-Padding, gefolgt von einer Max Pooling Layer. Das zweite Pack Convolutional Layers ist dem ersten identisch bis auf die Anzahl an Filtern, die hier verdoppelt sind. Nach dieser Konfiguration ist das Bild (32->16->8) 8x8 Pixel groß, und damit klein genug für die Fully Connected Layers. Den Abschluss bilden eine Fully Connected Layer („Dense“ Layer) mit 256 Neuronen und eine weitere Fully Connected Outputlayer mit zehn Neuronen. In der letzten Schicht wird die Softmax Aktivierungsfunktion verwendet. Ihr Vorteil ist, dass alle Outputs der Outputneuronen sich zu 1.0 aufaddieren. Die Layers zwischendurch, wie *BatchNormalization* und *Dropout*, helfen dem CNN dabei schneller und besser das Problem zu generalisieren und damit die Validation Accuracy zu verbessern (Darauf werde ich nicht eingehen, das würde zu viel werden).



```
1 model.compile(optimizer='adam',  
2               loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
3               metrics=['accuracy'])
```

Fig. 27

Nach der Erstellung des CNNs folgt die Vorbereitung auf die Trainingsphase. Hier wird der aktuelle Standardoptimizer „Adam“ verwendet, da er häufig eine bessere Optimierung erzielt als andere Optimizer (wie z.B. Stochastic Gradient Descend, der in der NN Backpropagation vorgestellt wurde: $w_{\text{neu}} = w - \eta * \Delta w$). Die hier angegebene Verlustfunktion ist die Categorical Cross-Entropy Verlustfunktion. Sie ist die Standardverlustfunktion für mehrklassige Klassifizierungsprobleme. Die *metrics* sind Metriken nach denen das model während des Trainings und dem Testen ausgewertet wird. [26][27]

```
1 model.fit( train_images, train_labels, epochs=20,  
2           validation_data=(test_images, test_labels) )
```

Fig. 28

Nachdem das model auf das Training vorbereitet wurde kann es auch trainiert werden. Die Trainingsmethode bekommt als Parameter die Trainingsdatensätze, die am Anfang des Programms geladen wurden, sowie die Validierungsdatensätze. Das model wird dann für die spezifizierte Anzahl an Epochen (Iterationen über den Datensatz) mit den Trainingsdatensätzen trainiert und währenddessen mit den Validierungsdatensätzen validiert.

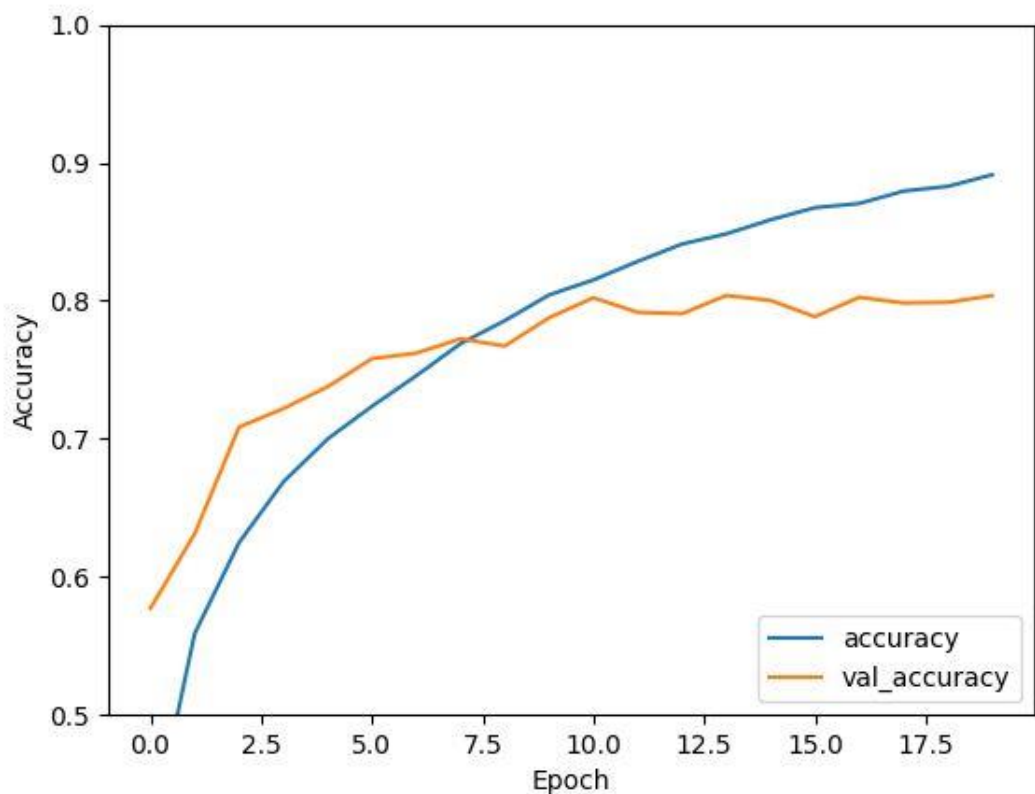


Fig. 29

Fig.30 ist eine Visualisierung dieses Trainingsprozesses und zeigt, wie sich Accuracy und Validation Accuracy über die Epochen entwickeln.

```
1 predictions = model.predict( test_images ).tolist()
```

Fig. 30

Als nächstes wird das trainierte Model alle Bilder aus dem Validationsdatensatz klassifizieren. Die Labels werden dann in einer Liste *predictions* gespeichert, da sie später noch verwendet werden.

```
1 class_names = ["airplane", "automobile", "bird", "cat", "deer", "dog",  
                "frog", "horse", "ship", "truck"]  
2 e = Editor()  
3 for i in range( len( predictions ) ):  
4     output = class_names[predictions[i].index(max(predictions[i]))]  
5     actual_label = class_names[test_labels[ i ][0]]  
6  
7     info = f"Actual Label : {actual_label}\nPredicted Label : {output}"  
8  
9     e.newEntry( f"#{i} {str(actual_label)}          // {output}",  
                test_images[ i ], info, actual_label = output )  
10 e.mainloop()
```

Fig. 31

Mit allen klassifizierten Bildern in einer Variable müssen diese nur noch praktisch visualisiert werden. Dazu wird ein neues *Editor*-Objekt^[5] erstellt und mit einer *For*-Loop über *predictions* iteriert. Jedes Element in *predictions* ist eine Liste mit Wahrscheinlichkeiten für jede gegebene Klasse, d.h. der größte Wert einer jeden Liste kennzeichnet die wahrscheinlichste Klasse für das gegebene Bild. Mit diesen Informationen kann also der Index des höchsten Wertes als Index für *class_names* genommen werden, um den Output in ein Label umzuwandeln. Dieser Wert wird dann in *output* gespeichert. In *actual_label* wird das tatsächliche Label des Bildes gespeichert. *info* ist eine Variable, die im *Editor*-Objekt eigentlich ein paar Informationen anzeigen sollte. Das hat jedoch irgendwann einfach nicht mehr funktioniert, aber ich lasse es drin, falls es sich mal anders entscheiden sollte. Am Ende des *For*-Loops sind alle nötigen Informationen über ein Trainingsbild gesammelt und der Eintrag wird dem Editor hinzugefügt. Zum Schluss, nach dem *For*-Loop muss noch der Loop für das Editorfenster gestartet werden, da dieses sich sonst schließt.

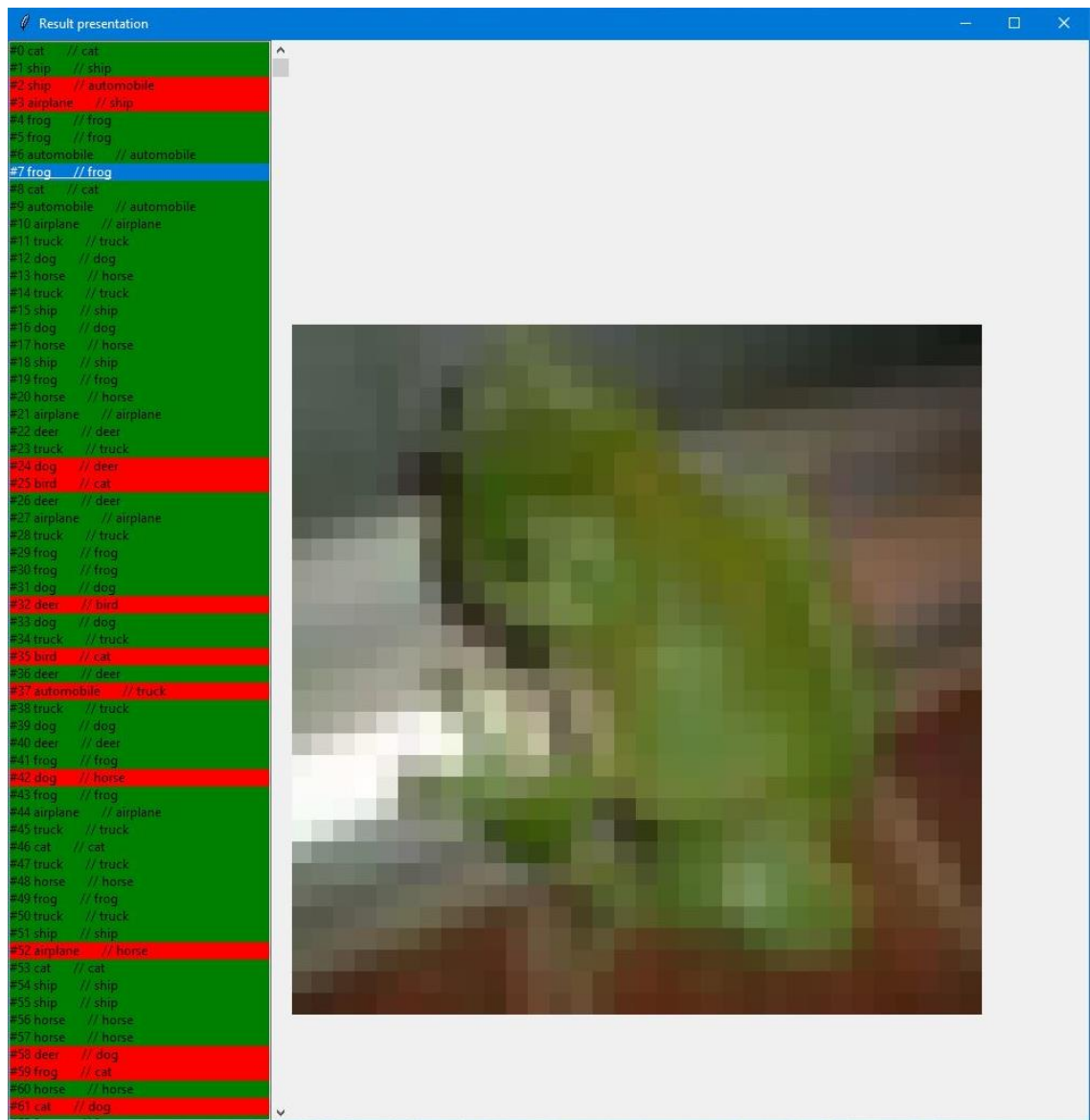


Fig. 32

Fazit

Neuronale Netze sind also schichtenweise, durch Gewichte, verbundene künstliche Neuronen, die nichtlineare Zusammenhänge zwischen Inputdaten erkennen können. Sie werden durch Algorithmen wie dem Backpropagation Algorithmus mit Hilfe des Gradientenabstiegsverfahren trainiert, indem die Gewichte und Biase in kleinen Schritten immer so verändert werden, dass der Verlust ein, am besten globales, Minimum erreicht. Convolutional Neural Networks sind auch schichtenweise aufgebaut. Sie haben spezielle Schichten, die sog. high-level Features aus Inputbildern extrahieren können, indem in den ersten Schichten einfache Features, wie Linien und Kurven, und in den hinteren Schichten high-level Features, wie ganze Autos oder Gesichter, erkannt und weitergegeben werden. Zudem verringern andere Schichten die Größe des Bildes, sodass Pixel, die keine wichtigen Informationen tragen, einfach entfernt werden. Am Ende werden die Features in ein NN gegeben, welches diese dann z.B. verschiedenen Klassen zuordnet. Vorteile für CNNs sind zum Einen, dass sie sog. weight sharing verwenden, wodurch die Anzahl der zu trainierenden Parameter und die erforderliche Rechenleistung dafür deutlich gesenkt werden. Zum Anderen analysieren sie nur die Beziehungen zwischen nahe beieinander liegenden Pixeln, wodurch die Position des Objektes im Bild selber egal wird. Das Training funktioniert auch bei CNNs mit dem Backpropagation Algorithmus, aber mit ein paar Unterschieden, da hier unter anderem mit Bildern gearbeitet wird.

Zum Arbeiten mit Bildern (Klassifizierung, Objekterkennung, ...) eignen sich CNN also sehr gut und erreichen sogar schon übermenschliche Genauigkeiten in einigen Bereichen. Meine Implementation findet sich, in Bezug auf Validationsgenauigkeit, in den top 50, aber ist nur ein Beispiel, wie man die Strukturierung eines CNNs angehen könnte. [16][28]

Fußnote

[1] CIFAR 10: Der CIFAR 10 Datensatz kommt von dem „Canadian Institute For Advanced Research“ und besteht aus 60.000 RGB Bilder im Format 32x32 Pixel. Die Bilder zeigen insgesamt zehn verschiedene Klassen an Objekten (daher CIFAR 10). Diese Klassen sind: Flugzeuge, Autos, Vögel, Katzen, Rehe, Hunde, Frösche, Pferde, Schiffe und LKW. [1]

[2] Perzeptron: Das was ein Perzeptron im wesentlichen von einem NN unterscheidet sind, laut data-sience-blog.com, die Aktivierungsfunktionen, die in NN komplexer sind.

[3] Verlustfunktion: Die Verlustfunktion gibt an wie genau das NN Vorhersagen treffen kann. Ein Verlust von 0.0 kann also nur von einem perfekten NN erreicht werden. Dies ist jedoch meistens nicht möglich.

[4] Hyperparameter: Hyperparameter in Bezug auf NN sind bestimmte Werte und Eigenschaften eines NNs, die vor dem Training gesetzt werden. Beispiele für Hyperparameter sind: die Learningrate, die Anzahl an Hiddenlayers, die Anzahl an Trainingsdurchgängen etc.

[5] *Editor*-Objekt: Der Editor ist eine von mir programmierte Klasse, in der die Ergebnisse eines Modeltrainings visualisiert werden können. Alle Bilder werden aufgelistet, sodass sie ausgewählt werden können. Dann wird das ausgewählte Bild dargestellt und angezeigt, ob das vergebene Label richtig/falsch ist.

[6] Der verschwindende Gradient: Je mehr Schichten einem NN hinzugefügt werden, desto mehr Aktivierungsfunktionen gibt es, durch die der Verlust hindurchpropagiert werden muss. Die Ableitungen von Funktionen wie Sigmoid werden dabei so klein, dass das effektive Training der hinteren Gewichte nicht mehr möglich ist. [29]

Quellen

- [1] Alex Krizhevsky | The CIFAR-10 dataset |
<https://www.cs.toronto.edu/~kriz/cifar.html> | 3.1.2021
- [2] Benjamin Aunkofer | Funktionsweise künstlicher neuronaler Netze |
<https://data-science-blog.com/blog/2018/08/31/funktionsweise-kunstlicher-neuronaler-netze/> | 7.1.2021
- [3] - | Künstliches Neuron | https://de.wikipedia.org/wiki/Künstliches_Neuron |
9.1.2021
- [4] - | Nervenzelle | <https://de.wikipedia.org/wiki/Nervenzelle> | 7.1.2021
- [5] - | Sigmoid Function | https://en.wikipedia.org/wiki/Sigmoid_function |
10.1.2021
- [6] - | Aufbau einer Nervenzelle | <https://www.studienkreis.de/biologie/nervenzelle-aufbau/> | 7.1.2021
- [7] 3Blue1Brown | Backpropagation Infinitesimalrechnung | Deep Learning, Kapitel 4 |
<https://www.youtube.com/watch?v=tIeHLnjs5U8> | 10.12.2020
- [8] - | Lineare Separierbarkeit |
https://de.wikipedia.org/wiki/Lineare_Separierbarkeit | 19.1.2021
- [9] - | - | <https://www.geogebra.org> | 23.1.2021
- [10] - | Machine Learning Glossary | <https://ml-cheatsheet.readthedocs.io/en/latest/> |
29.1.2021
- [11] Jason Brownlee | 14 Different Types of Learning in Machine Learning |
<https://machinelearningmastery.com/types-of-learning-in-machine-learning/> |
29.1.2021
- [12] Marcel Schöni | Was ist das Gradientenabstiegsverfahren? |
<https://zkmaBlog.com/2017/02/11/was-ist-das-gradientenabstiegsverfahren/> |
29.1.2021
- [13] Julian Moeser | Künstliche Neuronale Netze – Aufbau & Funktionsweise |
<https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/> | 6.2.2021
- [14] SAGAR SHARMA | Activation Functions in Neural Networks |
<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6> | 11.2.2021

- [15] Jason Brownlee | Understand the Impact of Learning Rate on Neural Network Performance | <https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/> | 12.2.2021
- [16] Roland Becker | Convolutional Neural Networks – Aufbau, Funktion und Anwendungsgebiete | <https://jaai.de/convolutional-neural-networks-cnn-aufbau-funktion-und-anwendungsgebiete-1691/> | 14.2.2021
- [17] Luis Serrano | A friendly introduction to Convolutional Neural Networks and Image Recognition | <https://www.youtube.com/watch?v=2-OI7ZB0MmU> | 15.2.2021
- [18] - | Convolutional Neural Network Algorithms | https://docs.ecognition.com/v9.5.0/eCognition_documentation/Reference%20Book/23%20Convolutional%20Neural%20Network%20Algorithms/Convolutional%20Neural%20Network%20Algorithms.htm | 15.2.2021
- [19] Sumit Saha | A Comprehensive Guide to Convolutional Neural Networks – the ELI5 way | <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53> | 16.2.2021
- [20] - | Average Pooling | <https://paperswithcode.com/method/average-pooling> | 16.2.2021
- [21] jamesmf | - | <https://stats.stackexchange.com/questions/182102/what-do-the-fully-connected-layers-do-in-cnns/182122> | 18.2.2021
- [22] Brandon Rohrer | How convolutional neural networks work, in depth | https://www.youtube.com/watch?v=JB8T_zN7ZC0 | 18.2.2021
- [23] Jiwon Jeong | The Most Intuitive and Easiest Guide for Convolutional Neural Network | <https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480> | 18.2.2021
- [24] Jefkine | Backpropagation In Convolutional Neural Networks | <https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/> | 20.2.2021
- [25] Primusa | - | <https://stackoverflow.com/questions/51344839/what-is-the-difference-between-the-terms-accuracy-and-validation-accuracy> | 28.2.2021
- [26] Jason Brownlee | Gentle Introduction to the Adam Optimization Algorithm for Deep Learning | <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/> | 1.3.2021

- [27] Jason Brownlee | How to Choose Loss Functions When Training Deep Learning Neural Networks | <https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/> | 2.3.2021
- [28] Rodrigo Beneson | What is the class of this image? | [https://rodrigob.github.io/are we there yet/build/classification datasets results.html](https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html) | 6.3.2021
- [29] Chi-Feng Wang | The Vanishing Gradient Problem | <https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484> | 10.3.2021

Ich erkläre hiermit, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

....., den.....

(Ort)

(Datum)

.....

(Unterschrift)