Universität
Augsburg
University

Bachelorarbeit

Titel der Bachelorarbeit: Image Analysis for Food Safety and Health

Name des Autors: TEMKENG Thibaut

Datum der Abgabe: Ende Oktober

Betreuung: Name der Betreuer/ des Betreuers:Shou Liu

Fakultät für Embedded Intelligence for Health Care and Wellbeing

Erklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde, sowie die Satzung der Universität Augsburg zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Ort, den Datum

Inhaltsverzeichnis

KNN	Künstliches neuronales Netz
CNN	Convolutional Neural Network
KI	Künstliche Intelligenz
NN	neuronales Netz
ConvL	Convolutional Layer
FCL	Fully Connected Layer
PooL	Pooling Layer
ILSVRC	Large Scale Visual Recognition Challenge
DNN	Tiefe neuronale Netze
ML	Maschine Lernen
DSC	Depthwise Separable Convolution

1 Einleitung

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen von neuronalen Netzwerken beschrieben, beginnend mit künstlichen neuronalen Netzen, gefolgt von einem Abschnitt über Faltungsneuronale Netzwerke und zum Abschluss ein Abschnitt über die verwendeten Datensätze und Bibliotheken.

2.1 Künstliche neuronale Netze

2.1.1 Neuron

Ein künstliches Neuron[18] ist eine mathematische Funktion, die das Verhalten vom biologischen Neuron nachbildet. Künstliche Neuronen sind elementare Einheiten in einem Künstliches neuronales Netz (KNN). Das künstliche Neuron empfängt einen oder mehrere Inputs und bildet sie auf einen Output ab. Normalerweise wird jeder Eingabe x_i separat mit einem Gewicht w_i multipliziert und danach aufsummiert und zum Schluss wird die Summe durch eine Funktion geleitet, die als Aktivierungs- oder Übertragungsfunktion bekannt ist.Eine schematische Darstellung eines künstlichen Neurons ist in Abbildung 1 zu sehen.

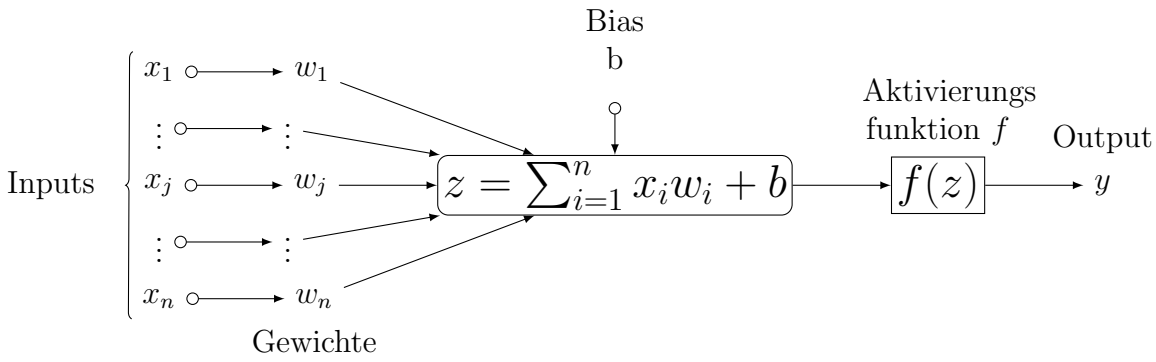


Abbildung 1: Funktionsweise eines künstlichen Neurons

Künstliche Neurone können aufgestapelt werden, um eine Schicht(*Layer*) zu bilden.Ein KNN besteht aus einer oder mehreren Schichten und je nach seiner Position in einem neuronales Netz (NN) wird eine Schicht anders genannt: Eingangsschicht(*Input Layer*) bzw. Ausgabeschicht(*Output Layer*), wenn das Layer die Eingangsdaten bzw. Ausgabedaten des neuronalen Netzes darstellt und versteckte Schicht(*Hidden Layer*), wenn es keine Eingang- oder Ausgabeschicht ist.Ein kurzer Überblick über die Darstellung von KNNs kann sich in Abbildung 18 verschafft werden.

2.1.2 Merkmalskarten(Feature-Maps)

Die Ausgabe einer Schicht wird als Aktivierungskarten oder Feature-Map(s) bezeichnet. Die Anzahl von Feature-Maps in einer Schicht ist gleich die Anzahl der Ausgabekanäle(*output channels*) bzw: Tiefe(*depth*) dieser Schicht. In einer Faltungsschicht ist ein

Feature-Map immer zweidimensional, während die Dimension eines Feature-Maps in einem Fully Connected Layer (FCL) nur von der des Inputs abhängt, genauer gesagt,für ein n-dimensionales Inputs ist ein Feature-Map $(n-1)$dimensional. Wie die Feature-Maps berechnet werden, hängt sehr vom Schichttyp ab. Es wird beispielsweise in einer Faltungsschicht die Faltungsoperation mehrmals auf die Eingabe angewendet und jedes Mal wird ein neues Feature-Map erhalten.

2.1.3 Filters

Ein Filter ist eine kleine Matrix, die die Extraktion von Features ermöglicht. Die modernen Architekturen von Convolutional Neural Network (CNN) verwendet immer mehr Filter (mindestens 2000), um bessere Ergebnisse zu erzielen. Wegen der zufällige Initialisierung von Filtern und ihrer großen Anzahl im CNN kann es vorkommen, dass mehrere Filter das gleiche Feature extrahieren, solche Filter werden als Redundant bezeichnet, oder dass Filter genau das Gegenteil von dem filtern, was wir wollen, nämlich unwichtige Informationen, deshalb werden während des Trainings die Filterparameter ständig geändert. Zwei nichtlineare Filter, d. h. Filter, die nicht proportional zueinander sind, lernen unterschiedliche Dinge.Man kann also daraus schließen, dass je größer die Anzahl an Filter ist, desto mehr Features gefiltert werden können oder besser sind die Ergebnisse, aber das ist in Allgemein falsch, da die guten Ergebnisse sehr stark davon abhängen, wie die Filter im CNN aufgestapelt sind. Zu viele Filtern in einem CNN können dazu führen, dass das CNN sowohl gute Features als auch schlechte schneller lernt und zu wenige Filter schränken die Kapazität des CNN ein, wichtige Features zu extrahieren, denn irgendwann wird es vorkommen, dass jedes Filter etwas Wichtiges gelernt hat, aber noch mehr Informationen werden benötigt, um die Trainingsdaten zu erklären. Es ist also wichtig die richtige Anzahl von Filtern zu finden.

-

2.1.4 Entwicklung von Künstlichen Neuronalen Netzen

2.2 Convolutional Neural Network

2.2.1 Feedforward

2.2.1.1 Input Layer

Die Eingangsschicht stellt die Eingangsdaten dar. Hier müssen die Eingangsdaten dreidimensional sein.Also die Eingangsdaten von CNN haben immer die folgende Form $W \times H \times D$ wobei (W, H) der räumlichen Dimension und D die Tiefe der Daten entspricht. Z.B $100 \times 100 \times 3$ für ein RGB-Bild und $224 \times 224 \times 1$ für ein Graustufenbild.

2.2.1.2 Faltungsschicht

Die wichtigste Sicht bzw. der Hauptschicht in einem CNN ist die Faltungsschicht(*Convolution Layer*). Die Eingabedaten eines CNN besteht auf jedenfalls aus wichtigen und unwichtigen Informationen.Als wichtige und unwichtige Informationen haben wir z.B die starke

Präsenz der weißen Farbe in einer Muschelsuppe bzw. die Präsenz eines Menschen, wenn man verschieden Ernährungsklasse klassifizieren möchte. Während des Trainings eines CNN wird versucht, diese relevanten Informationen aus den Daten zu entnehmen und die irrelevanten auszuschließen. Alles was ein CNN aus den Eingabedaten nutzt, um die Daten zu bestimmter Klassen zuzuordnen, wird als Feature bezeichnet. Das einzige bzw. das Hauptziel einer Faltungsschicht besteht darin, die Features aus seinen Inputdaten herauszuziehen. Dass eine Faltungsschicht in der Lage ist, Features selbst zu extrahieren, ohne dass man sie hinweist, was wichtig ist und was nicht, ist wirklich beeindruckend,Aber die Art und Weise, wie sie es tut, ist noch beeindruckender.

In einem Convolutional Layer (ConvL) wird die sogenannte Faltungsoperation(*convolution operation*) durchgeführt,dabei wird das komponentenweises Produkt(*Hadamard-Product*) zwischen einem kleinen Bereich der Eingabedaten und einem Kernel durchgeführt und dann die ganze aufsummiert. Eine Illustration der Faltungsoperation ist in der Abbildung 4a zu sehen. Die Resultante der Faltungsoperation wird die Aktivierung des Neurons genannt.Sollte diese Aktivierung des Neurons null sein, dann sagt man, dass das Neuron "*nicht aktiv*" ist und sonst ist es *aktiv*.Die null Aktivierung bedeutet, dass das vom Filter gesuchte Feature nicht gefunden wurde. Je stärker oder größer die Aktivierung eines Neurons ist, desto höher ist die Wahrscheinlichkeit, dass das vom Filter gesuchte Feature gefunden ist(siehe Abbildung 4b). Um diese Faltungsoperation durchzuführen, muss einige Hyperparameter vordefiniert sein, und zwar:

- **Anzahl und Größe von Filtern**
  Die Anzahl an Filters gibt nicht nur an, wie viele Filter in dem ConvL verwendet werden, sondern auch ,wie oft die Faltungsoperation auf die Eingabe durchgeführt wird, d.h die Anzahl der Feature-Maps oder die Tiefe der Schicht. Anstatt das ganze Bild zu betrachten, wenn man auf der Suche nach einem Feature ist, was mit der Tatsache gleichbedeutend ist, dass jedes Neuron der Schicht mit allen Neuronen der vorherigen verbunden ist, wird nur einen lokalen Bereich des Bildes betrachtet, wir verbinden also jedes Neuron nur mit einem lokalen Bereich der Bild. Dieser lokaler Bereich wird als Empfangsfeld(*receptive field*) des Neurons bezeichnet und entspricht der Filtergröße.Es ist zu beachten, dass die Filtergröße nur der räumlichen Dimension des Filters entspricht. Die Tiefe des Filters ist gleich die vom Input, also für ein $(100, 100, 3)$ Bild haben alle Filter die Tiefe 3. Die Verwendung von solchen kleinen Filtern ist die Hauptidee hinter einer Faltungschicht. Filter haben im Allgemein eine kleine räumliche Dimension wie z.B 2×2, 3×3 oder 5×5, sonst verliert man einen großen Vorteil von ConvL, der darin besteht, die Speicheranforderung deutlich zu reduzieren, indem es die Gewichte verteilt.

- **Schrittgröße**(*Stride*)

  Da ein Filter nur einen kleinen Bereich des Bilds wahrnehmen kann,wird eine Schrittgröße verwendet, um die Bewegung des Filters auf dem Bild zu steuern.Das Filter wird über das Bild von links nach rechts, von oben nach unten bewegt. Sei $S := (n, m)$ die Schrittgröße, dann wird das Filter von n Pixeln nach rechts für

Abbildung 2: Einfluss der Schrittgröße auf die Größe der Feature-Maps

(a) *Schrittgröße=(1,1)* (b) *Schrittgröße=(2,2)*

jede horizontale Bewegung des Filters und m Pixeln nach unten für jede vertikale Bewegung des Filters bewegt(siehe Abbildung 3a und 3b).Wie die folgende Tabelle zeigt,hängt die räumliche Dimension des Outputs sehr von der Schrittgröße ab.

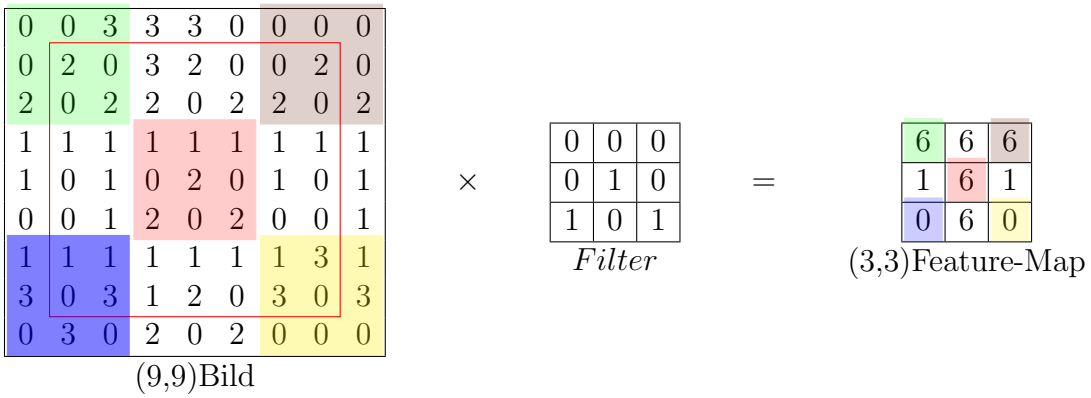
Imagegröße	***Stride***	***Filtergröße***	***Output (räumliche Dim.)***	***Mit Padding***
$(100, 100, 3)$	$(1, 1)$	$(1, 1)$	$(100, 100)$	$(100, 100)$
$(100, 100, 3)$	$(1, 1)$	$(3, 3)$	$(98, 98)$	$(100, 100)$
$(100, 100, 3)$	$(2, 2)$	$(3, 3)$	$(49, 49)$	$(50, 50)$
$(100, 100, 3)$	$(1, 1)$	$(4, 4)$	$(97, 97)$	$(100, 100)$
$(100, 100, 3)$	$(1, 1)$	$(10, 10)$	$(91, 91)$	$(100, 100)$

Tabelle 1: Auswirkung von Schrittgröße,Filtergröße und Padding auf Output eines $(100, 100, 3)$ Bild.

- ***Padding***
  Mit einem großen Filter lernt man in der Regel ”mehr” als mit einem kleinen.Aber wie es in der Tabelle 1 zu sehen ist, habe man eine Reduktion der Dimension, wenn ein Filter mit Filtergröße > 1 angewendet wird und um die Dimension zu behalten, wird vor der Anwendung der Filter auf das Bild eine Füllung(*padding*) an den Rändern des Bilds gemacht. Diese Füllung ermöglicht erstens den Entwurf immer tiefer Netzwerke, denn es gibt nach jedem ConvL einen kleinen Dimensionverlust und zweitens,dass die Information an Rändern nicht zu schnell verschwunden werden.Um die Ränder einer Eingabe zu füllen, werden sehr oft Nullen (*zero padding*) verwendet, oder die Pixel, die an der Grenze liegen, werden wiederholt.

In einem CNN mit mehreren ConvLs beschäftigen sich die ersten ConvLs mit dem Erlernen einfacher Merkmale wie Winkel, Kanten oder Linien und je tiefer das CNN ist, desto komplexer sind die extrahierten Merkmale. Das liegt daran, dass jede nachfolgende Schicht einen größeren Bereich des Originalbildes betrachten oder „sehen “ kann.Nehmen wir an,dass die zwei ersten Layers eines CNNs Filter von Größe 3×3 und ein *Stride* $= (1, 1)$ verwenden.Die erste Schicht betrachte immer 3×3 benachbarte Pixel des Originalbildes und speichert seine Aktivierung auf einem Pixel. Jetzt wenn die zweite ConvL 3×3 benachbarte Pixel betrachtet, betrachtet sie eigentlich 5×5 benachbarte Pixel des Originalbildes.Wir können uns deshalb vorstellen, dass irgendwo in späteren Schichten Filter das gesamte Originalbild betrachten.

(a) Matrixdarstellung



(b) Pixeldarstellung

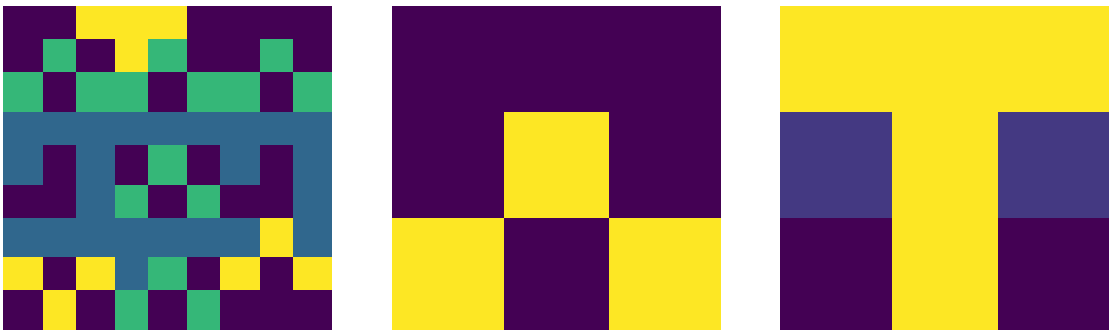


Abbildung 4: Faltungsoperation mit einem 3×3-Filter und Schrittgröße $= 3$

2.2.1.3 Aktivierungsfunktion

Das neuronale Netzwerk wird während dem Training mit sehr vielen Daten gespeist und das sollte in der Lage sein, aus diesen Daten zwischen relevanten und irrelevanten Informationen Unterschied zu machen. Die Aktivierungsfunktion auch Transferfunktion oder Aktivitätsfunktion genannt, hilf dem NN bei der Durchführung dieser Trennung. Es gibt sehr viele Aktivierungsfunktionen und in folgenden werden wir sehen, dass eine Aktivierungsfunktion je nach zu lösende Aufgaben vorzuziehen ist.

$$\begin{cases} Y = f(\Sigma(Gewicht * Input + Bias)) \\ f := Aktivierungsfunktion \end{cases}$$

Binäre Treppenfunktion ist extrem einfach, siehe Abbildung 5, definiert als $f(x) =$

$\begin{cases} 1, & \text{if } x \geq a \text{ (a:= Schwellenwert)} \\ 0, & sonst \end{cases}$. Sie ist für binäre Probleme geeignet, also Probleme wo man mit *ja* oder *nein* antworten sollte.Sie kann leider nicht mehr angewendet werden, wenn es mehr als zwei Klassen klassifiziert werden soll oder wenn das Optimierungsverfahren gradientenbasierend ist, denn Gradient immer null.

Abbildung 5: Binäre Treppenfunktion

(a) Binäre Treppenfunktion (b) Ableitung Binäre Treppenfunktion

Lineare Funktion ist definiert als $f(x) = ax$, $f'(x) = a$, siehe Abbildung 7. Sie ist monoton, null zentriert und differenzierbar. Es ist jetzt möglich,nicht mehr nur binäre Probleme zu lösen und mit gradientenbasierenden Optimierungsverfahren während der Backpropagation Parameter anzupassen, denn Gradient nicht mehr null, also sie ist besser als binäre Funktion .Nutzt ein mehrschichtiges Netz die lineare Aktivierungsfunktion, so kann es auf ein einschichtiges Netz überführt werden und mit einem einschichtigen Netz können komplexe Probleme nicht gelöst werden.Außerdem ist der Gradient konstant. Der Netzfehler wird also nach einigen Epochen nicht mehr minimiert und das Netz wird immer das Gleiche vorhersagen.

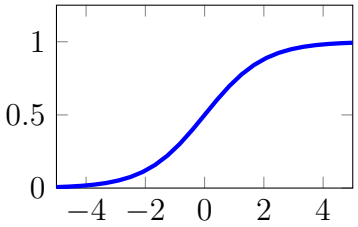
Abbildung 7: Lineare Funktion

(a) Lineare Funktion: $f(x) = x$ (b) Ableitung Lineare Funktion: $f'(x) = 1$

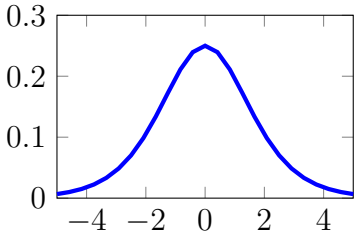
Logistische Funktion ist definiert als $f(x) = \frac{1}{1+exp(-x)}$,$f'(x) = \frac{exp(x)}{(1+exp(x))^2}$, siehe Abbildung 9. Sie ist differenzierbar, monoton, nicht linear und nicht null zentriert(hier nur positive Werte).Zwischen $[-3, +3]$ ist der Gradient sehr hoch.Kleine Änderung in der Netzinput führt also zu einer großen Änderung der Netzausgabe. Diese Eigenschaft ist bei Klassifikationsproblemen sehr erwünscht.Die Ableitung ist glatt und von Netzinput abhängig. Parameter werden während der Backpropagation je nach Netzinput angepasst. Außerhalb von $[-3, 3]$ ist der Gradient fast gleich null, daher ist dort eine Verbesserung der Netzleistung fast nicht mehr möglich.Dieses Problem wird Verschwinden des

Gradienten *(vanishing gradient problem)* genannt.Außerdem konvergiert das Optimierungsverfahre sehr langsam und ist wegen wegen der exponentiellen (e^x) Berechnung rechenintensiv.

Abbildung 9: Logistische Aktivierungsfunktion:$sigmoid(x)$.



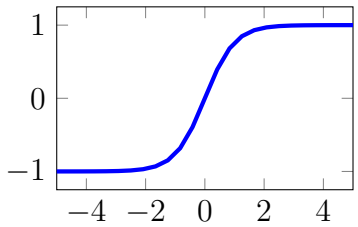
(a) Logistische Aktivierungsfunktion.



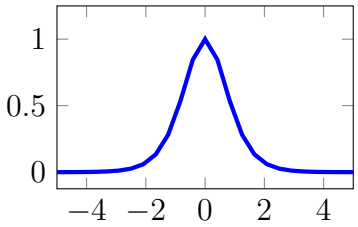
(b) Ableitung der Logistische Funktion.

Tangens Hyperbolicus ist definiert als $tanh := 2sigmoid(x) - 1$, siehe Abbildung 11.Außer dass sie null zentriert ist, hat sie die gleichen Vor- und Nachteile wie die Sigmoid Funktion.Sättigung fehl noch

Abbildung 11: Tangens Hyperbolicus.



(a) Tangens Hyperbolicus.

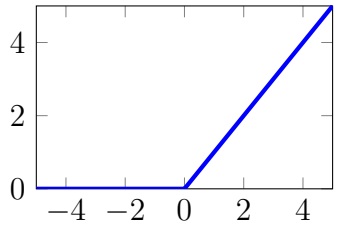


(b) Ableitung der Tangens Hyperbolicus.

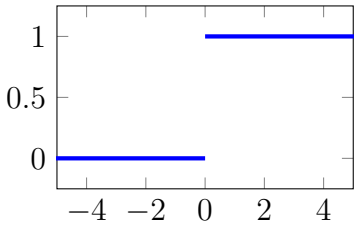
Rectified Linear Unit(ReLU) ist definiert als $f(x) = max(x, 0)$, siehe Abbildung 13. Sie ist sehr leicht zu berechnen. Es gibt keine Sättigung wie bei *Sigmoid* und *tanh*. Sie ist nicht linear,deshalb kann den Fehler schneller propagiert werden.Ein größter Vorteil der ReLU-Funktion ist, dass nicht alle Neurone gleichzeitig aktiviert sind, negative Eingangwerte werden zu null,daher hat die Ausgabe von Neuronen mit negativen Eingangwerten keine Einfluss auf die Schichtausgabe, diese Neurone sind einfach nicht aktiv.Das Netz wird also spärlich und effizienter und wir haben eine Verbesserung der Rechenleistung. Es gibt keine Parameteranpassungen, wenn die Eingangwerte negative sind, denn der Gradient ist dort null.Je nachdem wie die Bias initialisiert sind, werden mehrere Neuron töten,also nie aktiviert und ReLU ist leider nicht null zentriert. ReLUs haben die wünschenswerte Eigenschaft, dass sie keine Eingangsnormalisierung benötigen, um eine Sättigung zu verhindern.

Leaky ReLU Funktion ist definiert als $f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.01x, & sonst \end{cases}$, siehe Abbildung 15. Sie funktioniert genauso wie die ReLU-Funktion, außer dass sie das Problem des

Abbildung 13: ReLU Aktivierungsfunktion



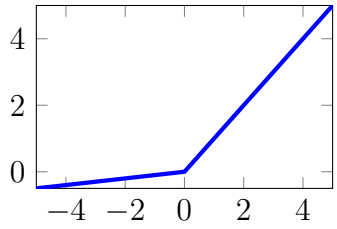
(a) ReLU Aktivierungsfunktion



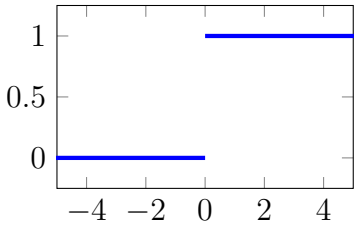
(b) Ableitung der ReLU Funktion

toten Neurons und sie ist null zentriert.Es gibt somit immer eine Verbesserung der Netzleistung, solange das Netz trainiert wird.Wenn das Problem von Leaky ReLU nicht gut gelöst wird, wird empfohlen, die *Parametric ReLU* (PReLU) Aktivierungsfunktion zu verwenden, die während der Training selber lernt, Problem der toten Neurone zu lösen.

Abbildung 15: Leaky ReLU Funktion



(a) Leaky ReLU Funktion



(b) Ableitung der Leaky ReLU Funktion

Softmax ist definiert als $f(x_1, x_2, \cdots, x_n) = \frac{(e^{x_1}, x_2, \cdots, e^{x_n})}{\sum_{i=1}^{n} e^{x_i}}$.Die Softmax-Funktion würde die Ausgänge für jede Klasse zwischen null und eins zusammendrücken und auch durch die Summe der Ausgänge teilen. Dies gibt im Wesentlichen die Wahrscheinlichkeit an, dass sich der Input in einer bestimmten Klasse befindet.

In allgemein wird die ReLU aufgrund des Problems der toten Neurone nur in versteckte Schichten und die Softmax-Funktion bei Klassifikationsproblemen und Sigmoid-Funktion bei Regressionsproblemen in Ausgabeschicht verwenden.

2.2.1.4 Pooling Layer

Die Funktionsweise von Pooling-Schichten ist sehr ähnlich zu der von ConvLs. Das Filter wird über die Inputdaten bewegen und dabei anstatt die Faltungsoperation durchzuführen, werden die Inputdaten Blockweise zusammengefasst. Ein Pooling-Layer besitzt nur eine Schrittgröße und eine Filtergröße. Das Filter in PooL ist in Gegensatz zu Filtern in ConvL nicht lernbar ist, es gibt nur an, wie groß der Block, der zusammengefasst wird, sein muss. Als Standard werden ein 2×2 Filter und eine 2×2 Schrittgröße verwendet, was die Dimension der Inputdaten um mindestens die Hälfte reduziert. Interessanter dabei ist ,dass die wichtigen Informationen oder Muster nach der Pooling-Layer vorhanden bleiben und damit haben wir nicht sowohl eine Erhöhung der Rechengeschwindigkeit als

auch eine Reduzierung der Netzparameter.Noch dazu sind PooLs invariant gegenüber kleiner Veränderung wie Parallelverschiebung.

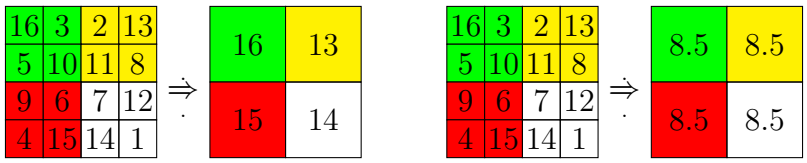


Abbildung 17: Funktionsweise der Pooling-Sicht mit Pooling_size= $(2, 2)$ und $Stride = 2$

Je nachdem wie die Blöcke in Pooling Layer (PooL) zusammengefasst werden, haben die PooLs unterschiedliche Namen. Werden die Werte eines Blockes durch den Maximalwert des Blocks, dann sprechen wir von Max-Pooling-Layer (siehe Abbildung 17 links), wenn sie durch den Mittelwert des Blocks ersetzt, wird von Average-Pooling-Layer (siehe Abbildung 17 rechts) und wenn die Filtergröße gleich die räumliche Dimension der Eingangdaten ist, sprechen wir von Global-Max-Pooling-Layer und Global-Average-Pooling-Layer, es wird also alle Neurone in einem Kanal zu einem Neuron, die Ausgabedimension solche Schicht entspricht der Anzahl der Kanäle bzw. Tiefe der Inputdaten.

Das Global-Pooling-Layer wird sehr oft angewendet, um das Vorhandensein von Merkmale in Daten aggressiv zusammenzufassen. Es wird auch manchmal in Modellen als Alternative zur Flatten-Schicht, die mehrdimensionale Daten zu eindimensionale umwandelt, beim Übergang von ConvLs zu einem FCL verwendet.

2.2.1.5 Multi-layer Perzeptron (Fully Connected Layer)

Nachdem die relevanten lokalen Merkmale durch die Wiederholung von Conv und Pooling-Schichten extrahiert werden sind, wenden sie in einem FCL kombiniert, um das Ergebnis jeder Klasse zu berechnen. Die FCLs des CNN ermöglichen, Informationssignale zwischen jeder Eingangsdimension und jeder Ausgangsklasse zu mischen, so dass die Entscheidung auf dem gesamten Bild basieren kann und ihm eine Klasse zugewiesen werden kann. Die FCLs funktionieren eigentlich genau wie ConvLs, außer dass jedes Neuron in FCL mit allen Neuronen und nicht mit einem kleinen Bereich von Neuronen im vorherigen Layer verbunden ist.Ein NN mit nur FCLs sieht wie in Abbildung18 aus.

Aufgrund der hohen Anzahl von Verbindungen zwischen Neuronen in einem FCL wird viel Speicher und Rechenleistung benötigt und verlangsamt auch das Training, es ist auch einer der Gründe, weshalb die FCLs meist nur in der letzten Schicht von CNN zur Klassifizierung verwendet werden und die Anzahl der Neuronen in letzter Schicht entspricht der Anzahl von Klassen.

2.2.2 Backforward

2.2.2.1 Fehlerfunktion

Das Training von CNNs besteht darin, den vom CNN begangenen Fehler zu korrigieren bzw. zu minimieren, daher wird es sehr oft als ein Optimierungsverfahren betrachtet.

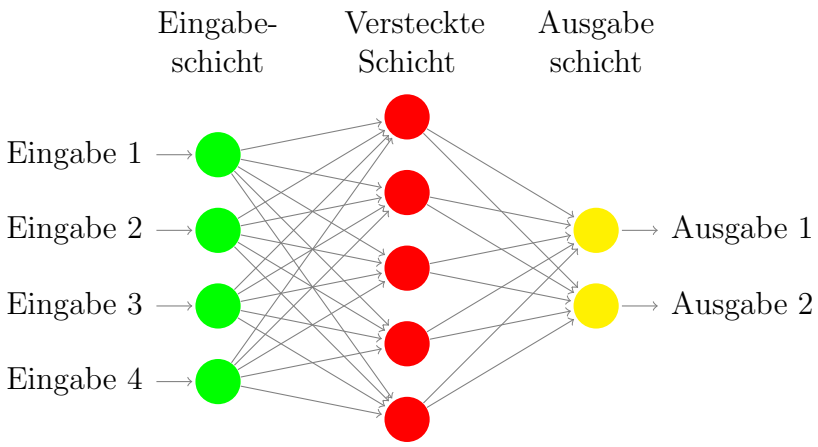


Abbildung 18: Darstellung eines neuronalen Netzes

Wie gut die Vorhersage des neuronalen Netzes gerade ist, wird durch eine Fehlerfunktion auch Kostenfunktion genannt quantifiziert oder angegeben. Die Fehlerfunktion bringt die Ausgabewerte des NNs mit den gewünschten Werten in Zusammenhang. Sie ist ein nicht-negativer Wert und je kleiner dieser Wert wird, desto besser ist die Übereinstimmung des CNNs.Es wird in Laufe des Trainings von CNNs versucht, diese Kostenfunktion mit Gradient basierten Verfahren zu minimieren(siehe Absatz 2.2.2.3)

Die meisten benutzten Kostenfunktionen sind die Kreuzentropie (*cross-entropy*, Gleichung 2.2)(CE) und die mittlere quadratische Fehler (*mean squared error*,Gleichung 2.1)(MSE).

$$MSE(Y, \widehat{Y}) = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \widehat{Y}_i)^2 \tag{2.1}$$

$$CE(Y, \widehat{Y}) = -\frac{1}{n} \sum_{i=1}^{n} Y_i \log(\widehat{Y}_i) \tag{2.2}$$

Y :Satz von echten Labels n :Die Batchgröße $\widehat{Y}$:Vorhersagesatz

Im Gegenteil zu CE Fehlerfunktionen,die sich nur auf Wahrscheinlichkeitsverteilungen anwenden lassen, können die MSE auf beliebige Werte angewendet werden. Nach [10, Pavel et al] ermöglicht die CE-Verlustfunktion ein besseres Finden lokaler Optima als die MSE-Verlustfunktion und das soll daran liegen, dass das Training des MSE Systems schneller in einem schlechten lokalen Optimum stecken bleibt, in dem der Gradient fast null ist und damit keine weitere Reduzierung der Netzfehler ermöglicht.Im Allgemein ist die CE Kostenfunktion für die Klassifikationsprobleme und die MSE Fehlerfunktion für die lineare Regression-Probleme besser.

2.2.2.2 Gradient

Der Gradient einer Funktion ist die erste Ableitung einer Funktion und in mehrdimensionalem Raum ist der Gradient einer Funktion der Vektor, dessen Einträge die ersten partiellen Ableitungen der Funktion sind. Der Gradient an einem Punkt gibt die Richtung der steilsten Anstieg der Funktion an diesem Punkt. Also da wir die Kostenfunktion minimieren möchten, sollen wir lieber immer in die Gegenrichtung des Gradienten gehen.In auf dem Gradient basierten Optimierungsverfahren wird der Gradient benutzt, um die lokalen oder globalen Extremwerte(hier das Minimum) zu erreichen. Da wir jetzt die Richtung des Minimums herausgefunden haben, bleibt noch zu bestimmen ,wie wir in diese Richtung gehen sollen.

2.2.2.3 Lernrate

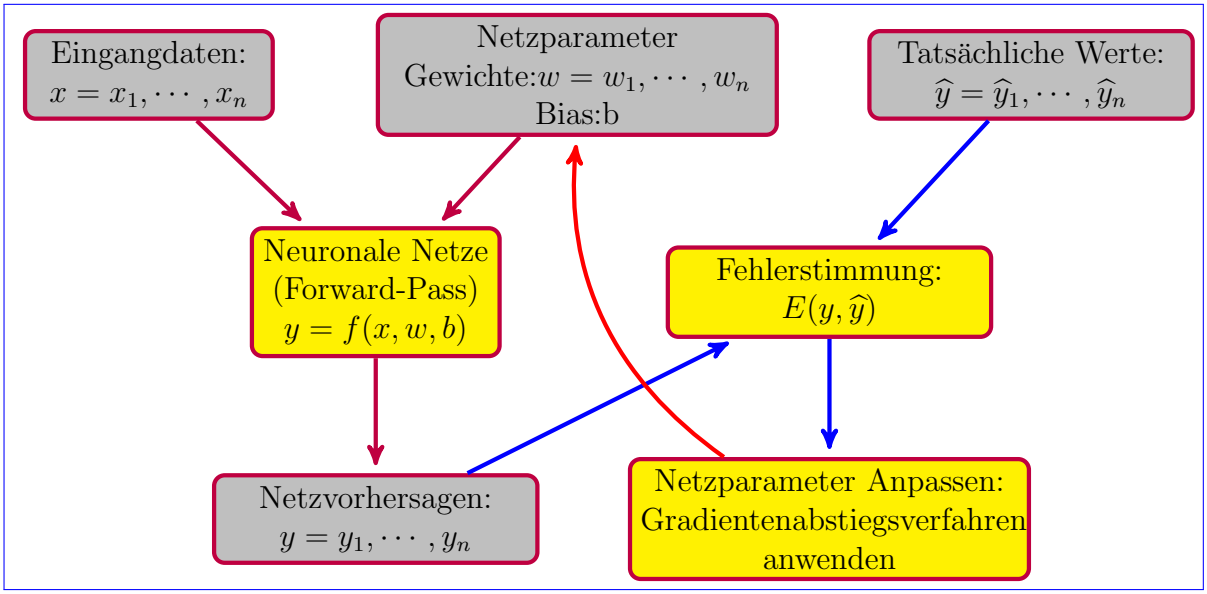
Die Lernrate oder Schrittweite beim maschinellen Lernen ist ein Hyperparameter, der bestimmt, inwieweit neue gewonnene Informationen alte Informationen überschreiben sollen[13], in anderen Worten wie schnell wir ans Ziel Kommen. Je nachdem wie die Lernrate gesetzt wird, werden bestimmte Verhalten beobachtet (Siehe Absatz 4.2) und sie nimmt sehr oft Werte zwischen 0.0001 und 0.5: Die Lernrate muss allerdings im Intervall $]0, 1[$ Werte annehmen, sonst ist das Verhalten des NN nicht vorhersehbar bzw. konvergiert das Verfahren einfach nicht. Für jeden Punkt x aus dem Parameterraum gibt es eine optimale Lernrate $\eta_{opt}(x)$, sodass das globale oder lokale Minimum sofort nach der Parameteranpassung erreicht wird. Da $\eta_{opt}(x)$ am Trainingsanfang leider nicht bekannt ist, wird die Lernrate in die Praxis vom Programmierer basiert auf seine Kenntnisse mit NNs oder einfach zufällig gesetzt.

vGradientenabstiegsverfahren Aktuelle leistungsfähige DNN bestehen fast immer aus Million Variable (lernbarer Parameter). Wir können uns ein DNN als eine Gleichung mit Millionen von Variablen vorstellen, die wir lösen möchten. Mit Hilfe der Daten wollen wir uns in einem Raum, dessen Dimension größer als eine Million ist, bewegen, um die optimalen Parameter(Parameter, die die Trainingsdaten korrekt abbilden) zu finden. Aufgrund der unendlichen Anzahl von Punkten im solchen Räume wäre es nicht sinnvoll,einen Punkt zufällig auszuwählen, dann überprüfen, ob er optimal ist und, wenn nicht, nochmals einen anderen Punkt zufällig auszuwählen. Genauer zu diesem Zeitpunkt kommen Gradientenabstiegsverfahren zum Einsatz. Die Gradientenabstiegsverfahren sind Verfahren, die auf dem Gradient basieren, um Optimierungsprobleme zu lösen.Hier wird Gradientenabstiegsverfahren verwendet, um sich der optimalen Parametern anzunähern oder sie zu finden.

Ablauf eines Gradientenverfahrens im DNN

Das Gradientenabstiegsverfahren kann in drei Hauptschritte aufgeteilt werden.Die Abbildung 19 stellt das Backpropagation-Verfahren bildlich dar. Beim ersten Schritt wird ein zufälliger Punkt aus dem Parameterraum ausgewählt und davon ausgehend wird der Parameterraum exploriert. Das entspricht der Netzparameterinitialisierung am Trainingsanfang. Der zweite Schritt besteht darin, die Abstiegsrichtung zu bestimmen.Dazu

werden zuerst die Eingangdaten in das NN eingespeist(*Forwardpropogation*), danach wird der Fehler zwischen den Netzvorhersagen und den korrekten Werten berechnet und ein Fehler gibt es (fast) immer, denn die Initialisierung wird zufällig gemacht und die Wahrscheinlichkeit, dass wir von Anfang an die optimalen Werte finden, ist verschwindend klein und zuletzt wird der Gradient(2.2.2.2) der Kostenfunktion in abhängig von den gegebenen Eingangdaten und den erwarteten Werten berechnet. Beim letzten Schritt wird die Schrittweite bestimmen.Die Lernrate (2.2.2.3) oder die Schrittweite wird vor Trainingsbeginn festgelegt oder während des Trainings abhängig von aktuellem Netzzustand allmählich adaptiert.



$f(x, w, b)$: Netzfunktion. $E(y, \widehat{y})$: Kostenfunktion.
η: Lernrate. $\longrightarrow$: Backward-Pass.
$\longrightarrow$: Forward-Pass $\longrightarrow$: Fehlerfunktion

Abbildung 19: Ablauf der Backpropagation

Variante des Gradientenverfahrens

Bisher existiert drei Variante des Gradientenabstiegsverfahren, die sich nur durch die Größe der Daten, die sie verwendet, um den Gradienten der Kostenfunktion berechnet, unterscheidet.

Stochastic Gradient Descent (SGD) : Bei SGD wird jeweils ein Element bzw. Sample aus der Trainingsmenge durch das NN durchlaufen und den jeweiligen Gradienten berechnen, um die Netzwerkparameter zu aktualisieren.Diese Methode wird sehr oft online Training genannt,denn jedes Sample aktualisiert das Netzwerk. SGD verwendet geringer Speicherplatz und die Iterationen sind schnell durchführbar.Zusätzlich kann die

Konvergenz für großen Datensatz wegen der ständigen Aktualisierung der Netzwerkparameter beschleunigt werden.Diese ständigen Aktualisierung hat die Schwankung der Schritte in Richtung der Minima zur Folge, was die Anzahl der Iteration bis zum Erreichen des Minimums deutlich ansteigt und dabei helfen kann, aus einem unerwünschten lokalen Minimum zu entkommen.Ein großer Nachteil dieses Verfahren ist der Verlust der parallelen Ausführung, es kann jeweils nur ein Sample ins NN eingespeist werden. Der Algorithmus 1 zeigt den Ablauf von SGD.

Input: loss function E, learning rate η, dataset X, y und das Modell $F(\theta, x)$
Output: Optimum θ which minimizes E

while *converge* **do**
  Shuffle X, y
  for x_i, y_i *in* X, y **do**
    $\tilde{y} = F(\theta, x_i)$
    $\theta = \theta - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \frac{\partial E(y_i, \tilde{y_i})}{\partial \theta}$
  end
end

Algorithm 1: Stochastic Gradient descent(SGD) [2].

Batch Gradient Descent (BGD) : BGD funktioniert genauso wie SGD, außer dass der ganze Datensatz statt jeweils ein Element aus dem Datensatz zur Netzwerkparameteraktualisierung genutzt wird.Jetzt kann das Verfahren einfach parallel ausgeführt werden, was den Verarbeitungsprozess des Datensatzes stark beschleunigt. BGD weist weniger Schwankungen in Richtung des Minimums der Kostenfunktion als SGD auf, was das Gradientenabstiegverfahren stabiler macht.Außerdem ist das BGD recheneffizienter als das SGD, denn nicht alle Ressourcen werden für die Verarbeitung eines Samples, sondern für den ganzen Datensatz verwendet.BGD ist leider sehr langsam,denn die Verarbeitung des ganzen Datensatz kann lange dauern und es ist nicht immer anwendbar, denn sehr große Datensätze lassen sich nicht im Speicher einspeichern.Der Algorithmus 2 zeigt den Ablauf von BGD.

Input: loss function E, learning rate η, dataset X, y und das Modell $F(\theta, x)$
Output: Optimum θ which minimizes ϵ

while *converge* **do**
  $\tilde{y} = F(\theta, x)$
  $\theta = \theta - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \epsilon(y, \tilde{y})}{\partial \theta}$
end

Algorithm 2: Batch Gradient descent [2].

Mini-batch Stochastic Gradient Descent(MSGD): MSGD ist eine Mischung aus SGD und BGD.Dabei wird der Datensatz in kleine Mengen (*Mini-Batch oder Batch*) möglicherweise gleicher Größe aufgeteilt.Je nachdem wie man die Batch-Große setzt, enthalten wir SGD oder BGD wieder. Das Training wird Batchweise durchgeführt, d.h. es wird jeweils ein Batch durch das NN propagiert, der Verlust jedes Sample im Batch wird berechnet und dann deren Durchschnitt benutzt, um die Netzwerkparameter zu anzupassen.MSGD verwendet den Speicherplatz effizienter und kann von Parallelen Ausführung profitieren. Noch dazu konvergiert MSGD schneller und ist stabiler. In die Praxis wird fast immer das MSGD Verfahren bevorzugt.Der Algorithmus 3 zeigt den Ablauf von MSGD

Input: loss function E, learning rate η, dataset X, y und das Modell $F(\theta, x)$
Output: Optimum θ which minimizes E

while *converge* **do**
  Shuffle X, y
  for *each batch of* x_i, y_i *in X, y* **do**
    $\tilde{y} = F(\theta, x_i)$
    $\theta = \theta - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \frac{\partial E(y_i, \tilde{y_i})}{\partial \theta}$
  end
end

Algorithm 3: Mini-Batch Stochastic Gradient descent(MSGD) [2].

2.3 Datensätze und Bibliothek

2.3.1 Datensätze

1. **Food-101**[25]
   a) **Food-101-original** ist ein Datensatz von 101 Lebensmittelkategorien mit 101.000 Bilder. Für jede Klasse werden 250 manuell überprüfte Testbilder sowie 750 Trainingsbilder bereitgestellt. Die Trainingsbilder wurden bewusst nicht gereinigt und enthalten daher noch etwas Rauschen. Dies geschieht meist in Form von intensiven Farben und manchmal falschen Etiketten. Alle Bilder wurden so skaliert, dass sie eine maximale Seitenlänge von 512 Pixel aufweisen.
   b) **Food-101 von Tensorflow** Dieser Datensatz enthält die gleiche Anzahl von Bildern wie der soeben beschrieben, Aber der Datensatz wird nicht aufgeteilt und nicht gereinigt und enthalten daher noch etwas Rauschen. Dies geschieht meist in Form von intensiven Farben und manchmal falschen Etiketten.Für unsere Arbeit wurde jede Kategorie in drei(Trainings-, Auswertungs- und Testdatensatz) aufgeteilt ist, die $800, 100$ bzw. 100 Bilder umfasst.
2. **Flowers-102** Der Datensatz *Oxford Flowers 102* ist ein konsistenter Datensatz von 102 Blumenkategorien, die in Großbritannien häufig vorkommen. Jede Klasse

besteht aus 40 bis 258 Bildern. Die Bilder haben große Variationen in Maßstab, Pose und Licht. Darüber hinaus gibt es Kategorien, die innerhalb der Kategorie große Unterschiede aufweisen, und mehrere sehr ähnliche Kategorien. Der Datensatz ist unterteilt in einen Trainingssatz, einen Validierungssatz und einen Testsatz. Das Trainingsset und das Validierungssatz bestehen jeweils aus 10 Bildern pro Klasse (insgesamt je 1020 Bilder). Das Testset besteht aus den restlichen 6149 Bildern (mindestens 20 pro Klasse).

2.3.2 Bibliotheken

1. **Keras**[1] ist eine hochleistungsfähige neuronale Netzwerk-API. Keras ist in Python[2] geschrieben wurde und auf TensorFlow[3] oder Theano[4] laufen kann. Noch dazu erleichtert Keras die Implementierung von neuronalen Netzwerken und ermöglicht schnelle Experimente zu ermöglichen.Für diese Arbeit benutzen wir die Version 2.2.5 von Keras.

2. **TensorFlow** ist eine End-to-End-Open-Source-Plattform für maschinelles Lernen. Es verfügt über ein umfassendes, flexibles Menge aus Tools, Bibliotheken und Ressourcen.Für diese Arbeit haben wir die Version 1.14.0 von TensorFlow verwendet.

3 Kompression von DNN

Die neueren maschinellen Lernmethoden verwenden immer tiefer neuronale Netze wie z.B *Xception(134 Layers),MobileNetV2(157 Layers), InceptionResNetV2(782 Layers)*, um Ergebnisse auf dem neuesten Stand der Technik zu erzielen. Aber die Verwendung von sehr tiefer NNs bringt mit sich nicht nur eine deutliche Verbesserung der Modellleistung, sondern auch einen bedeutenden Bedarf an Rechenleistung und an Speicherplatz, was der Einsatz solcher Modelle auf Echtzeitsystemen mit begrenzten Hardware-Ressourcen schwierig macht.Es wurden bisher mehrere Ansätze untersucht, um die dem NN zugewiesenen Ressourcen effizienter zu nutzen:

- Die Modellbeschneidung(*Network pruning*), die die redundanten und die nicht relevanten Verbindungen zwischen Neuronen entfernt.

- Die Destillation von NNs, die ermöglicht, die großen Modellen in kleinen zu komprimieren.

- Die Quantisierung von NN, die für die Darstellung von einzelnem Netzparameter weniger als 32 Bits nutzt.

- Huffman-Codierung, die eine komprimierte Darstellung des Netzwerks ermöglicht.

[1] https://keras.io/
[2] https://www.python.org/
[3] https://www.tensorflow.org/
[4] http://deeplearning.net/software/theano/

Im folgenden werden nur die Beschneidung, die Quantisierung von NN und die Anwendung von Huffman auf NN mehr eingegangen werden.

3.1 Pruning Network

Wie oben schon erwähnt, wird beim *Pruning* neuronaler Netzwerke versucht, unwichtige oder redundante Verbindungen oder komplette Neuronen aus dem Netzwerk zu entfernen, um ein Netz mit möglichst geringer Komplexität zu erhalten. Mit unwichtigen Verbindungen werden die Parameter (Gewichte und Bias) gemeint, die fast null sind, denn Parameter mit Nullwert haben keinen Einfluss auf das Output des Neurons, sie sind einfach überflüssig. Während oder nach dem Training gibt es mehrere Parameter, die nicht wirklich oder nicht zu viel zum Neuronenergebnis beitragen, obwohl sie keinen Nullwert haben, deshalb ist es zum Reduzieren der Netzwerkdichte notwendig, anderen Maßstäbe als den Nullwert anzulegen, um Verbindungen zu entfernen.

Das Pruning-Verfahren bietet einige Vorteile wie Reduzierung der Speicher- und Hardwarekosten, die Trainingsbeschleunigung , die schnellere Antwortzeit und das Verringern der Wahrscheinlichkeit der Overfitting(4.6.1). Es ist sehr wichtig zu beachten, dass die Anwendung vom Pruning-Verfahren auf ein NN nur Sinn macht, wenn das NN teilweise oder komplett trainiert ist, sonst macht das Pruning nur eine Reduktion der Anzahl der Netzwerkparameter.

Es gibt zwei Hauptszenarien für das Pruning von NN. Die erste besteht darin, die irrelevanten Verbindungen in einem komplett trainierten NN zu entfernen. Mit komplett trainierten NN wird gemeint, dass die erwünschte Genauigkeit schon erreicht ist. Im zweiten Szenario wird Pruning während des Trainings durchgeführt, das wird seht oft als *iteratives Pruning* bezeichnet.Dabei wird vor Trainingsbeginn bestimmte Dinge festgelegt,wie z.B. ab wann wird das Netzwerk beschnitten und wie oft es durchgeführt werden soll. Das erste Szenario ist einfacher anzuwenden, denn man muss nur darauf warten, bis es keine Verbesserung der Genauigkeit des NN mehr gibt und dann Pruning auf das NN anwenden, aber damit verliert man großen Vorteile des Pruning, die die Beschleunigung des Trainings und Reduzierung der Overfitting-Wahrscheinlichkeit sind. Aus diesen Gründen wird in der Praxis das zweite Szenario bevorzugt, aber die richtigen Hyperparameter zu finden, ist kompliziert. Eine zu früh Beschneidung kann z.B. die Genauigkeit des NN zu sehr verschlechtern, denn es kann sein, dass eine Verbindung, die nur nach einer späteren Gewichtsanpassung zum Ergebnis eines Neurons hätte beigetragen können, entfernt würde, aber es bietet eine Beschleunigung des Trainings. Was eine zu spät Beschneidung angeht, haben wir fast die gleichen Nachteile wie im ersten Szenario und eine sehr häufiges Pruning kann das Training auch verlangsamen.Das ganze Prozess musst sehr oft leider mehrmals angewendet, um die richtigen Hyperparameter zu finden, sodass es sich wirklich nicht mehr lohnt. Ein graphischer Ablauf der Beschneidung ist in der Abbildung 20 zu sehen.Wenn wir zu viel auf einmal schneiden, könnte das Netzwerk so sehr beschädigt werden, dass es nicht mehr wiederhergestellt werden kann.

Die Hauptarbeit bei Pruning-Verfahren ist sicherlich, die guten Kriterien für die Bewertung der Wichtigkeit von Parametern zu finden und es ist vielleicht einer der Gründe, warum Pruning-Verfahren bisher nicht so populär ist, obwohl es immer noch zu schwierig

ist, tiefe NNs beispielsweise auf mobile Geräten mit eingeschränkten Ressourcen durchzuführen.

Bisher gibt es viele Kriterien für die Bewertung der Wichtigkeit eines Parameters, die sich miteinander unterscheiden und je nach Anwendung Vor- und Nachteile aufweisen. Im Folgenden werden einige Kriterien vorgestellt.

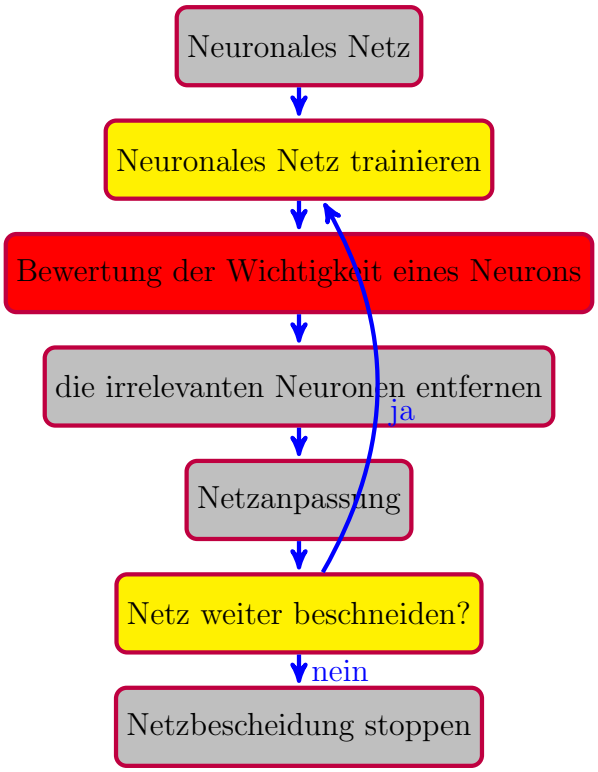


Abbildung 20: Netzbeschneidung während des Trainings

- **Schwellenwert**
  Für die Bewertung der Wichtigkeit eines Parameters verwenden [6, Han et al] einen Schwellenwert θ , also alle Parameter mit einem Wert in $[-\theta, \theta]$ wenden eliminiert.So ein Kriterium macht das Pruning schneller und effizienter, weil es sehr einfach ist, die zu eliminierenden Parameter zu finden.Da nach dem Pruning immer ein Genauigkeitsverlust auftritt, muss das NN erneuert trainiert werden, um seine Genauigkeit wiederherzustellen. Aber dieser Ansatz hat viele Nachteile. Erstens muss der optimale Schwellenwert gefunden werden und dafür muss das ganze Prozess(Schwellenwert auswählen, NN beschneiden, Verlust vergleichen) mehrmals wiederholt werden, um nur den optimalen Schwellenwert zu finden, deshalb können wir nie sicher sein, dass wir den optimalen Schwellenwert gefunden haben.Zweitens wird eine große Reduktion der Rechenkosten nur in FCLs und nicht in ConvLs beobachtet[7]. Die modernen Architekturen von CNNs können also bezüglich der Rechenkosten aus dem Verfahren keinen Vorteil ziehen,denn sie bestehen am meistens nur aus Faltungschichten.Drittens muss spärliche Bibliotheken oder spezielle

Hardware verwendet werden, damit die Bewertung des beschnitten NN effektiv wird.

- **Filter und Feature-Map Pruning**
  Anstatt die Gewichte in einem Filter elementweise zu entfernen,schlagen [7, Li et al] vor, das ganze Filter und die entsprechende Feature-Map zu entfernen.Dieses Vorgehen bringt mit sich im Vergleich mit dem oben vorgestellten Ansatz mehrere Vorteile.Es werden z.B. keine Bibliotheken, die eine Beschleunigung durch spärliche Operationen über CNNs ermöglichen, mehr benötigt.Noch dazu werden die Rechenkosten deutlich reduziert. Bei diesem Ansatz werden die weniger nützlichen Filter aus einem vollständig trainierten CNN entfernen. Zur Auswahl der zu entfernenden Filter wird zuerst die l_1-Norm(3.1) jedes Filter im NN berechnet und dann werden die m Filter mit der kleinsten l_1-Norm in jeder Schicht entfernen, wobei m ein Hyperparameter ist, der die Anzahl der zu löschenden Filter angibt.

$$\begin{aligned} ||F||_1 &= \sum_{i=1}^{h} \sum_{j=1}^{w} |F_{i,j}| \\ (w,h) &:= \text{ Breite und Höhe des Filters} \end{aligned} \tag{3.1}$$

Der Grund, warum nur die Filter mit einer kleineren l_1-Norm entfernt werden, liegt daran, dass sie neigen dazu, Feature-Maps mit geringen Aktivierungen im Vergleich zu den anderen Filtern in der selben Schicht zu erzeugen und das kann gut in der Abbildung 21 festgestellt werden. Wobei das Feature-Map links aus der Abbildung 21 ist das selbe mit dem aus Abbildung 4b und das Feature-Map rechts erhalten wir, indem wir das Filter aus Abb. 4a mit drei multiplizieren.

Abbildung 21: Einfluss der Intensität des Filters auf Feature-Map

Noch interessanter an diesem Verfahren ist, dass das Pruning und das neue Training des Netzwerks auf einmal auf mehrere Schichten durchgeführt werden, was die Beschneidungszeit noch weiter beschleunigt und noch effizienter ist, wenn es um sehr tiefe Netzwerke wie *InceptionResNet* oder *GoogleNet* angeht. Zur Beschneidung von Filter auf mehrere Schichten kann man die zu entfernenden Filter auf jeder Schicht entweder unabhängig von anderen Schicht oder nicht auswählen. Sollte man die zu löschenden Filter auf jeder Schicht unabhängig von anderen Schichten bestimmen, so muss man mit höheren Rechenkosten rechnen, denn es

werden Filter, die in vorherigen Schichten schon ausgenommen wurden, in der Berechnung der Summe der absoluten Gewichte noch miteinbezogen.Nach Li et al [7] ist die zweite Strategie nicht global optimal und kann trotzdem unter bestimmten Umständen zu besseren Ergebnissen führen. Die Abbildung 22 zeigt, wie die Entfernung eines Filters die Sichtausgabe beeinflusst.

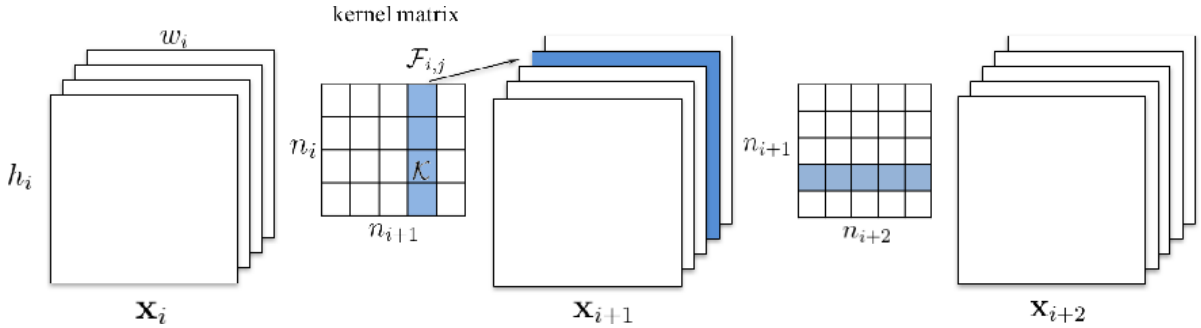


Abbildung 22: Das Beschneiden eines Filters führt zum Entfernen der entsprechenden Feature-Map und der zugehörigen Kernel in der nächsten Ebene[7].

- **Automatische Beschneidung**
  Die bisher beschriebenen Kriterien oder die meisten von ihnen fordern, dass das beschnittene Netzwerk noch trainiert wird, um die Endgewichte für die verbleibenden spärlichen Verbindungen zu lernen. Das ganze macht das Pruning-Verfahren noch schwieriger, denn es wird nur dann unterbrochen, wenn der Genauigkeitsverlust wirklich groß ist, was zu viel Zeit in Anspruch nimmt. Um dieses Problem zu lösen, schlagen *Manessi et al*[8] eine differenzierbare Technik vor, die es ermöglicht, das Pruning durchzuführen, wenn die Gewichte des Netzwerks während der Trainingsphase angepasst werden und die optimalen Schwellwerte automatisch zu suchen. Dieses Verfahren ist inspiriert von dem von Han et al[6] und löst viele Probleme, die das Verfahren von *Han et al* nicht lösen könnten oder wegen der Art und Weise, wie sie das Pruning durchführen, entstanden.Erstens werden die Schwellenwerte während des Trainings wie die Gewichte auch gelernt, es kann also eine große Menge von unterschiedlichen Schwellenwerten ausprobiert werden und anstatt der selbe Schwellenwert auf alle Layers anzuwenden, wird jeder Schicht eine Schwelle zugewiesen. Da Nicht alle Layers gleich empfindlich gegenüber dem Pruning sind und die Verwendung eine selbe Schwelle auf jede Schicht davon ausgeht, dass sie sind, muss dieses Verfahren optimaler. Zweitens, anstatt die Trainingszeit zu erhöhen, weil auch neue Parameter gelernt werden müssen, wird sie reduziert,das liegt daran, dass die Netzwerkparameter irgendwann mehr oder weniger spärlich werden, was Feedforward- und Backpropagationzeit erheblich reduziert.Es ist wichtig zu betonen,dass das Netzwerk nur einmal trainiert werden muss, um eine selbe bzw. bessere Genauigkeit wie bzw. als die anderen Methoden zu erzielen.

Aufgrund seiner Automatisierung und Effizienz wird das Pruning in Zukunft sicherlich immer häufiger eingesetzt werden.

3.2 Quantisierung von neuronalen Netzwerken

Im Allgemein wird von Quantisierung gesprochen, wenn es versucht wird, sich den Werten einer großen Menge mit denen einer kleineren Menge zu nähern. Im Maschine Lernen (ML) ist die Quantisierung der Prozess der Transformation eines ML-Programms in eine approximierte Darstellung mit verfügbaren Operationen mit geringerer Genauigkeit.

Methoden

- Matrixfaktorisierung
- Vektorquantisierung
- low-bit Quantisierung

Matrixfaktorisierung
Die Matrixfaktorisierung versucht, eine Matrix durch die Singulärwertzerlegung(SVD) zu approximieren. Die Singulärwertzerlegung zur Annäherung an eine Matrix $A \in \mathbb{R}^{m\times n}$ verwendet eine Matrix $B \in \mathbb{R}^{m\times n}$ vom Rang $k \leq \min(m,n)$, siehe Gleichung(3.2), wobei U und V orthogonale Matrizen und S eine Diagonalmatrix, deren Diagonaleinträge Singulärwerte heißen und bei Betrachtung von oben nach unten monoton abnehmen. Sollte man B direkt speichern, so hat man keine Kompression gemacht, sondern nur Informationen und Rechenzeiten verschwendet, denn die Matrizen A und B haben die gleiche Dimension, also liegt der Hauptvorteil von der Matrixfaktorisierung darin, wie die Matrix B gespeichert wird.

$$\begin{aligned} &A = USV^T \\ &B = \widehat{U}\widehat{S}\widehat{V}^T = U[:,:k]S[:k,:k]V[:k,:]^T \\ &||A-B||_F = \sqrt{\textstyle\sum_{i=1}^{m}\sum_{j=1}^{n}(A-B)_{i,j}^2} := \text{Frobenius Norm} \end{aligned} \tag{3.2}$$

Eigentlich werden $\widehat{U}, \widehat{S}$ und $\widehat{V}$ statt B gespeichert, damit wird $mk + kk + kn$ statt mn Speicherplatz verwendet und da $\widehat{S}$ Diagonalmatrix ist, kann auch einfach nur die Diagonaleinträge gespeichert werden, was zu einer Kompressionsrate von $\dfrac{mn}{k(m+n+1)}$ führt. Die Verwendung von Matrizen von kleinen Rang hebt die Tatsache hervor, dass es redundante Parameter in NN-Parametern gibt.Dies Redundanzeigenschaft wird ausgenutzt, um die Geschwindigkeit zu erhöhen[9, Denton et al].Im Sinne der Frobeniusnorm ist B die bestmögliche Approximation von A mit Singulärwertzerlegung durch eine Matrix von Rang k. Angenommen bildet die zu approximierenden NN-Parameter ein Bild, dann zeigt die Abbildung 23 ,wie man die NN-Parameter approximieren kann.

Wie man in der Abbildung 23 sieht, reichen relativ wenige Singulärwerte (ca. 5) schon aus, um die groben Konturen zu erkennen. Details erkennt man bei ca. 10 Singulärwerten schon. Generell wird das Bild mit mehr Singulärwerten schärfer.

Quantisierung mit weniger Bits(Low-bit Quantization)
Im Bereich des *Deep Learning* ist das Standard numerische Format für Forschung und Einsatz bisher 32-Bit Fließkommazahlen oder FP32, denn es bietet ein bessere Genauigkeit, aber die anderen Formaten wie 8-,4-,2- oder 1-Bits werden auch verwendet, obwohl

SW: Singulärwerte

Abbildung 23: Parameter Approximieren durch Matrixfaktorisierung

sie mehr oder weniger ein Verlust an Genauigkeit aufweisen. Die Verwendung von weniger genauen numerischen Formaten hat nicht nur einen kleinen Verlust der Netzleistung zur Folge, sondern auch die Verwendung von deutlich reduzierter Bandbreite und Speicherplatz. Noch dazu beschleunigt die Quantisierung die Berechnungen, denn die ganzzahlige Berechnung zum Beispiel ist schneller als die Fließkommaberechnung.

Bei Quantisierung mit wenige Bits geht es mehr oder weniger um die Abbildung eine großen Bereiches auf einen kleinen und dazu werden zwei Hauptwerte benötigt: der dynamische Bereich des Tensors und ein Skalierungsfaktor. Angenommen haben wir einen dynamischen Bereich $[0, 500]$ und einen Skalierungsfaktor:5, dann ergibt sich der neue Bereich $[0, 100]$, es wird also Werte zwischen $[5k, 5(k+1)]$ oder $[5k - 0.5, 5k + 0.5]$ auf $5k$ abgebildet. Es ist sinnvoller, die Skalierungsfaktors unter Berücksichtigung der Anzahl und der Verteilung der Werte in dynamischen Bereich des Tensors auszuwählen, sonst können zu viel Informationen verloren gehen.

Sobald ein anderes Format mit weniger Bits als FP32 verwendet, spricht man in ML von Quantisierung. Als Hauptvorteile von Quantisierung haben wir:Erstens wenn wir ein Modell mit FP32-Format durch das gleiche Modell mit 16-Bit Float(FP16), 8-Bit Integer(INT8) oder 4-Bit Integer(INT4) ersetzen, reduzieren wir den Speicherbedarf um die Hälfte,um ein Viertel bzw. um ein Achtel.Zweitens sind die Hardwares so programmiert, dass die integer Operationen im Vergleich zu FP32-Operationen schneller und energieeffizienter sind und drittens wird die Bandbreite durch die Verwendung von kleineren Modellen und dynamischen Werten stark reduziert.

Zur einer effizienteren Auswahl der Skalierfaktoren pro Schicht bzw. pro Filter ist es notwendig, Statistiken über das NN zu sammeln und das kann *offline* oder *online* gemacht werden. Bei der *Offline* Berechnung werden vor der Bereitstellung des Modells einigen Statistiken gesammelt, entweder während des Trainings oder durch die Ausführung einiger Epochen auf dem trainierten FP32-Modell und basierend auf diesen Statistiken werden die verschiedenen Skalierfaktoren berechnet und nach der Bereitstellung des Modells festgelegt.Bei dieser Methode besteht die Gefahr, dass zur Laufzeit die Werte, die außerhalb des zuvor definierten Bereichs auftreten, abgeschnitten werden, was zu einer Verschlechterung der Genauigkeit führen kann.Bei der *online* werden die *Min/Max-*

Werte für jeden Tensor dynamisch zur Laufzeit berechnet. Bei dieser Methode kann es nicht zu einer Beschneidung kommen, jedoch können die zusätzlichen Rechenressourcen, die zur Berechnung der Min/Max-Werte zur Laufzeit benötigt werden, unerschwinglich sein.[16]

Es gibt zwei Quantifizierungsszenarien.Die erste ist das vollständige Training eines Modells mit einer gewünschten niedrigeren Bit-Genauigkeit(*bit precission*)(kleiner als 32 Bits).Das Training mit sehr geringer Genauigkeit ermöglicht ein potenziell schnelles Training und Inferenz, aber der Hauptproblem mit diesem Ansatz ist, dass Netzparameter nur bestimmte Werte annehmen können, so ist die Aktualisierung der Netzparameter bzw. das Backpropagation nicht mehr wohldefiniert [17]. Das zweite Szenario quantisiert ein trainiertes FP32-Netzwerks mit einer geringeren Bit-Genauigkeit ohne vollständiges Training.Im Allgemeinen gilt: Je geringer die Bitgenauigkeit, desto größer ist der Verlust der Genauigkeit und um diesen Leistungsabfall zu überwinden, wird sehr oft das NN erneuert trainiert oder wird auf eine hybride Quantisierung, die zur Quantisierung verschiedene Formaten z.B. INT8 für die Gewichte und FP32 für die Aktivierung gleichzeitig verwendet, zurückgegriffen.

Vor kurzem haben [17, Yoni et al] zur Quantisierung das lineare Quantisierungsproblem als ein *Minimum Mean Squared Error* (MMSE) Problem formuliert und gelöst.Sie sind in der Lage, die trainierten Modelle zu quantisieren, ohne das NN erneuert trainieren zu haben um seine Genauigkeit wiederherzustellen und benutzen dabei nur das INT4-Format. Obwohl diese Methode minimalen Verlust der Genauigkeit(*accuracy*) aufweist, liefert sie Ergebnisse auf dem neuesten Stand der Technik und nach [17] weist dieser Ansatz den geringeren Genauigkeitsverlust als alle Quantisierungsverfahren auf.

Wie oben erwähnt, eine Quantisierung mit zu weniger Bit führt zu einem großen Genauigkeitsverlust, aber [17, Yoni et al] haben gezeigt, dass Die Quantisierung mit INT4-Format zu einem besseren Ergebnis führen kann. Das sollte daran liegen, dass sie fürs Approximieren eines Tensors, der gegenüber der Quantisierung sehr empfindlich ist, statt nur einen Tensor mehrere Tensors im INT4-Format benutzen.

3.3 Huffman Codierung

Die Huffman-Codierung ermöglicht eine verlustfreie Datenkompression, indem sie jeder einzelnen Dateneinheit eine unterschiedlich lange Folge von Bits zuordnet.Daraus folgt, dass eine gute Möglichkeit zur besseren Verwaltung der dem Modell zugeordneten Ressourcen ist:Erstmal das Netzwerk zu beschneiden, dann zu quantisieren und am Ende die Huffman-Codierung durchzuführen.

4 Experiment

4.1 Analyse der Ergebnisse mit Hilfe von Metriken

Nach dem Entwurf des NNs muss es noch bewertet werden,um zu überprüfen, ob es unseren Erwartungen (Genauigkeit, Stabilität und Trainingszeit) entspricht. Diese Be-

wertung kann mit Hilfe von verschiedenen Metriken und einem Testdatensatz gemacht werden.Es ist zwingend erforderlich, dass der Datensatz in zwei (Training und Validierung) oder drei (Training, Validierung und Test) unterteilt wird, da es sonst nicht möglich sein wird, abzuschätzen, wie das Modell Konzepte aus dem Datensatz generalisiert, sondern vielmehr, wie es lernt, jedes Element des Datensatzes einer Klasse zuzuordnen und da das Modell trainiert ist, um die Fehlklassifizierung zu reduzieren, wird die Netzwerkleistung bei Trainingsdaten auf jeden Fall immer besser, daher lohnt es sich nicht zur Abschätzung der Modellleistung die Trainingsdaten zu verwenden.

Metriken werden verwendet, um die Netzwerkleistung zu quantifizieren, und es gibt eine große Vielfalt von ihnen. Die am meistens verwendeten Metriken sind die mittlere Genauigkeit(*mean accuracy*) und den Logarithmischen Verlust(*logarithmic loss*).

Die Mittlere Genauigkeit oder Klassifizierungsgenauigkeit ist die Anzahl der korrekten Vorhersagen im Verhältnis zu allen getroffenen Vorhersagen. siehe Gleichung(4.1). Der logarithmische Verlust sieht genauso wie das *Cross-entropy*(2.2). Wenn man die Abbildung 24 genau anschaut, stellt man fest, dass die Genauigkeit nach 60 Epochen flach bleibt, was bedeutet, dass die Netzwerkleistung konstant bleibt, d.h. das Netzwerk keine Verbesserung oder Verschlechterung aufweist.Aber wenn man sich die Validierungsverlustkurve (*validation loss curve*) betrachtet, stellt man fest, dass die Kurve nach 60 Epochen langsam steigt, was bedeutet, dass die Netzwerkleistung sich verschlechtert.

$$acc := \frac{Korrekte\ Vorhersagen}{Alle\ Vorhersagen} \tag{4.1}$$

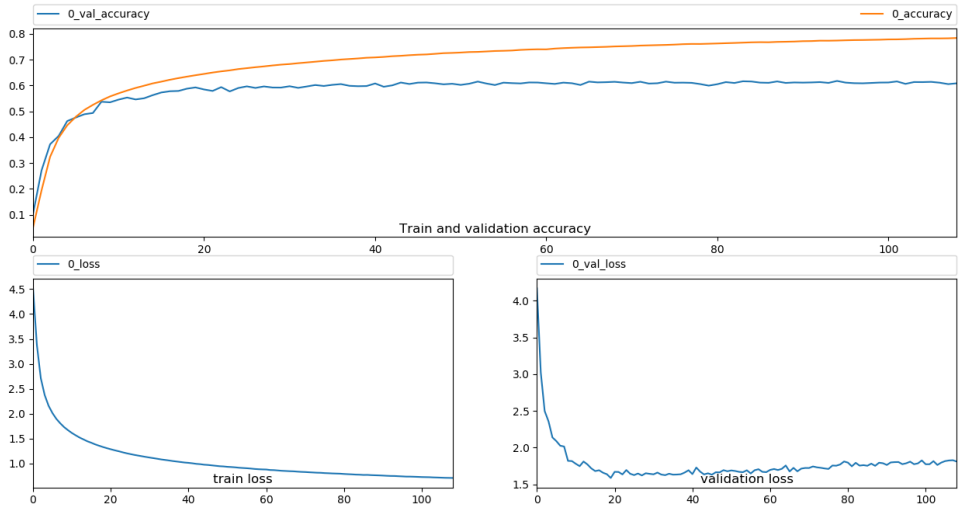


Abbildung 24: Vergleich zwischen der Genauigkeitskurve und der logarithmischen Verlustkurve.

Dieser großer Unterschied liegt daran, dass der logarithmische Verlust die Unsicherheit der Vorhersage in Betracht zieht und die Genauigkeit nicht. Angenommen, das CNN

klassifiziert eine Klasse i zweimal hintereinander falsch;$0,25$ und $0,012$ Prozent zum ersten bzw. zweiten Mal; die Netzwerkleistung hat sich verschlechtert, aber die Genauigkeit bleibt gleich, während sich der logarithmische Verlust verschlechtert.

Die Genauigkeit ist keine zuverlässige Metrik für die tatsächliche Leistung eines Klassifikators, da sie bei einem unausgewogenen Datensatz irreführende Ergebnisse liefert (d.h. wenn die Anzahl der Beobachtungen in verschiedenen Klassen stark variiert). Dies funktioniert nur dann gut, wenn der Datensatz nahezu gleichmäßig auf die Klassen verteilt ist. Datenset nicht gleichmäßig verteilen

4.2 Einfluss der Lernrate

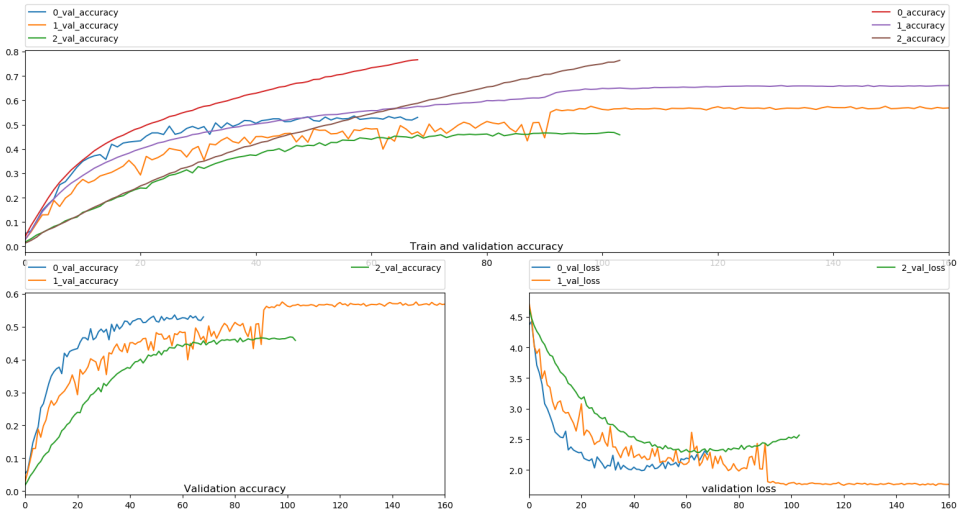
Wie in 2.2.2.3 gesehen, die Lernrate sagt uns, wie schnell wir ans Ziel kommen. sei η_{opt} bzw. η die optimale bzw. die ausgewählte Lernrate für die Lösung unserer Aufgabe.

Gilt $\eta < \eta_{opt}$, so sind wir sichern, ein lokales oder globales Minimum zu erreichen.Aber die Anzahl der benötigten Iterationen bis zum Minimum steigt offensichtlich an und das Verfahren kann leichter in einem unerwünschten lokalen Minimum stecken bleiben .

$\eta > \eta_{opt}$:Hier wird die Anzahl der Iterationen entweder reduziert oder erhöht.Das Verfahren ist im Allgemein nicht stabil, denn es wird über das Minimum ständig hinausgegangen und es ist nicht mehr sichern, zu einem lokalen oder globalen Minimum zu gelangen. Wenn eine hohe Lernrate angewendet wird, kann man versuchen, das Verhalten des CNN zu kontrollieren, indem man die Lernrate nach einer Reihe von Iterationen reduziert, bei denen sich die Metrik (Netzwerkgenauigkeit oder Netzwerkfehler) nicht mehr verbessert hat ,oder indem man Optimierer(4.5) verwendet, die im Laufe des Trainings die Lernrate entsprechend der Eingabe und der aktuellen Lernrate anpassen. Der Grund, warum die Lernrate immer reduziert und nicht erhöht werden muss, sollte daran liegen, dass das CNN irgendwann gut trainiert ist und sollte daher neuen Informationen nicht mehr Gewicht beimessen als dem bereits Gelernten.

- Aus der Abbildung(25) stellt man zuerst fest, dass die Verwendung verschiedener Lernrate zu unterschiedenen Ergebnissen führt und Je höher die Lernrate, desto mehr Schwankungen gibt es.

- Die Reduzierung der Lernrate, wenn sich das Netzwerk nicht mehr verbessert, verbessert die Netzwerkleistung.In der Abbildung gibt es eine jähe Erhöhung der Netzwerkleistung($orange\ Kurve$). Das markiert die Änderung der Lernrate von 0.005 auf 0.00001. Um zu erkennen, dass sich das Netzwerk nicht mehr verbessert, werden sogenannte „$Callback$ “-Funktionen verwendet, die fordern, dass bestimmte Hyperparameter wie der Faktor, um den die Lernrate reduziert wird($new\_lr = old\_lr * factor$) und die Anzahl der Epochen auch $Patience$ genannt, die die überwachte Menge(Z.B. der Validierungsfehler oder Validierungsgenauigkeit) ohne Verbesserung produziert haben. Diese Standard-Callback-Funktionen funktionieren leider nur dann gut, wenn der überwachte Menge (pseudo)-monoton(genau wie die ersten beiden Bilder in der Abbildung 26) ist. Diese Standard-Callback-Funktionen



Imagegröße: (100, 100,3) 0_: Lernrate= 0.0001
1_: Lernrate= 0.005 2_: Lernrate= 0.001

Abbildung 25: MobileNet_v1 mit verschiedenen Lernraten.

nehmen nicht in Betracht die allgemeine Verhalten des Netzwerks im Zeitintervall *Patience*, sondern vergleichen der erste Eintrag in diesem Zeitintervall mit den anderen Einträgen, was zum Fehler führen kann, wie das letzte Bild in der Abbildung 26 gut zeigt.

Um dieses Problem zu lösen, haben wir eine neue Methode entwickelt, die uns sagt, ob sich die Netzwerkleistung verändert hat und wie sehr sich das Netzwerk verändert hat. Das Verfahren funktioniert wie folgt:

- Wir zeichnen eine horizontale Linie, die über den Durchschnittswert im Zeitintervall *Patience* verläuft.Also $y = ax + b, (a = 0, b = Durchschnittswert)$, siehe orange Kurve in der Abbildung 26.
- Dann versuchen wir alle Werte im Zeitintervall *Patience* durch eine affine Funktion($y = ax + b$) zu approximieren, indem wir ein NN trainieren.Siehe blaue Kurve in der Abbildung 26.

vergleich zwischen normale LearningRateScheduler und meine

4.3 Convolution Layers

Heutzutage werden am meisten nur Filter der Größe 1×1 und 3×3 und nicht größer verwendet, obwohl größere Filter globale Informationen effizienter entnehmen und so zu besserer Ergebnisse kommen können. Dafür gibt es viele Gründe. Erstens sind die Berechnungen mit kleinerer Filter viel schneller als mit größerer Filter. Zweitens kann ein

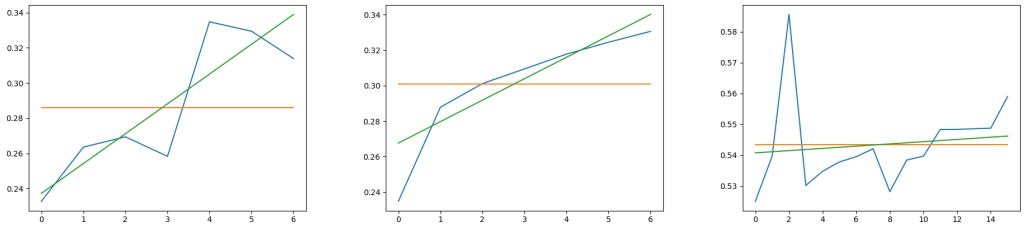


Abbildung 26: Lernraten-Scheduler.

ConvL mit einer großen Filtergröße durch mehrere aufeinanderfolgende ConvLs mit einer kleineren Filtergröße ersetzt werden und dabei werden sogar die Anzahl der benötigten Parameter deutlich reduziert. Sollte man z.B. ein 5×5 Filter auf eine Eingabe von Tiefe d angewendet, so braucht man $5\times5\times d$ Gewichte, um ein Feature-Map zu erzeugen. Nimmt man hingegen zwei aufeinanderfolgende ConvLs mit 3×3 Filtern, so braucht man nur $2(3 \times 3 \times d)$ Gewichte, um ein Feature-Map zu erzeugen. Drittens hat man bei kleinen Filtern nur eine langsame Reduzierung der Bilddimension, was den Entwurf von sehr tiefer CNNs wie *InceptionResNet* ermöglicht. Viertens können kleine Filter sehr lokale Merkmale extrahieren, so dass einfache und komplexe Informationen gleichzeitig erfasst werden.Die Anzahl der extrahierten Features ist enorm und werden gefiltert, wenn man tiefer im Netzwerk geht.

Der Absatz(2.2.1.2) behandelte mehr über die Faltungsoperation bei einem Eingang mit einem Kanal. Im Folgenden werden verschiedene Arten beschrieben, wie die Faltungsoperation an einem Eingang mit mehreren Kanälen durchgeführt werden kann. Für die Erklärung von Konzepten wird folgendes als Eingabe der Faltungsschicht betrachtet.

$$I = \begin{cases} \text{Filtergröße: } F := (F_w, F_h) & n_f\text{: Anzahl von Filtern} \\ \text{Input\_size: } W_{in} \times H_{in} \times D_{in} & P\text{: die Anzahl der Nullauffüllung(Padding)} \end{cases}$$

4.3.0.1 Standard Convolution

Die Standardfaltung funktioniert genau wie in der Abbildung(27).Für eine Eingabe I erzeugt die Standardfaltung Feature-Maps der Größe $(W_{out}, H_{out}, 1)$, wobei W_{out} und H_{out} wie in der Gleichung 4.2 berechnet werden.Für die Berechnung eines Feature-Map wird ein $(F_w \times F_h \times D_{in})$-Filter auf den Input angewendet. Dabei wird zuerst für jeden Kanal $I_i := I[:, :, i]$ die Faltungsoperation mit dem Filter $F_i := F[:, :, i]$ durchgeführt und dann werden die Ergebnisse jeder Faltungsoperation aufsummiert, um das Feature-Map zu bilden.

$$\begin{aligned} W_{out} &= \frac{W_{in} - F_w + 2P}{S} + 1 \\ H_{out} &= \frac{H_{in} - F_h + 2P}{S} + 1 \\ D_{out} &= n_f \end{aligned} \tag{4.2}$$

Die Berechnungskosten der Standardfaltung für eine Eingabe I unter der Annahme, dass die Schrittgröße eins und Padding berechnet werden, entspricht:

$$F_w \times F_h \times D_{in} \times n_f \times W_{in} \times H_{in} \tag{4.3}$$

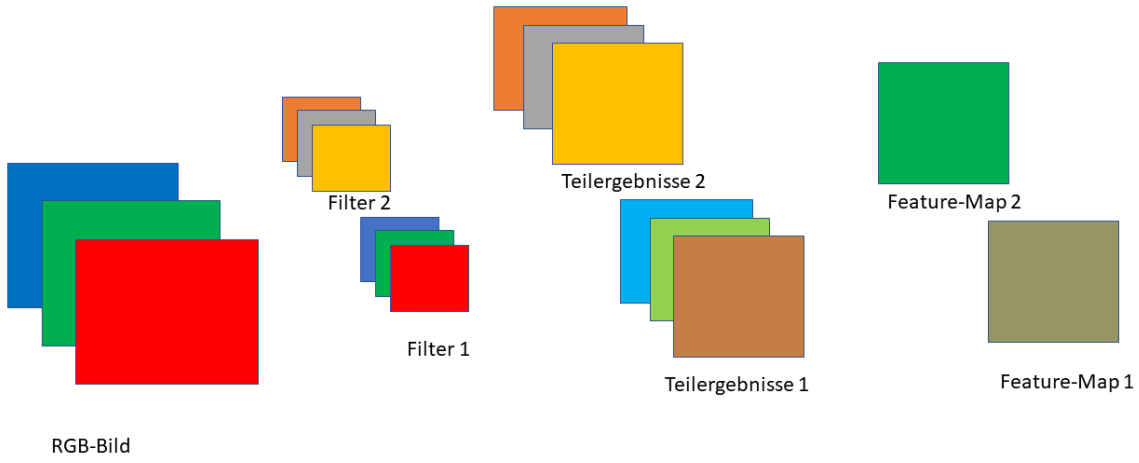


Abbildung 27: Anwendung einer Standardfaltung auf ein Farbbild.

4.3.0.2 Depthwise Convolution

Genau wie die Standardfaltung funktioniert die *Depthwise Convolution*, außer dass die Teilergebnisse von Standardfaltung schon der Feature-Maps von der *Depthwise Convolution* entsprechen.Festzustellen ist, dass die Anzahl von Feature-Maps immer proportional zur Anzahl der Kanäle des Inputs sein muss.Damit die Anzahl an Feature-Maps der *Depthwise Convolution* mehr als die Anzahl der Kanäle des Inputs ist, muss ein so genannte *Depth_multiplier* α definiert werden. Der *Depth_multiplier* $\in \mathbb{N}_{\geq 1}$ gibt an, wie viele Filter pro Kanal verwendet werden müssen. Die Abbildung(28) zeigt die Anwendung der *Depthwise Convolution* mit einem $\alpha = 2$ auf ein Farbbild. Die *Depthwise Convolution* hat für eine Eingabe I unter der Annahme, dass Schrittgröße eins und Padding berechnet werden, einen rechnerischen Aufwand von:

$$F_w \times F_h \times D_{in} \times W_{in} \times H_{in} \tag{4.4}$$

Wenn man die Gleichungen 4.3 und 4.4 betrachtet, stellt man schnell fest, dass *Depthwise Convolution* im Vergleich zur Standardfaltung äußerst effizient ist. Der Nachteil von

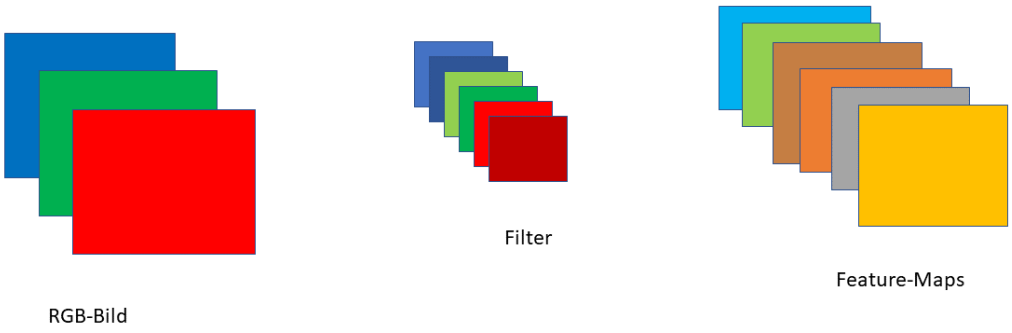


Abbildung 28: Anwendung einer Depthwise Convolution auf ein Farbbild.

Depthwise Convolution ist, dass jede generierte Feature-Map nur einen Kanal in Betracht zieht, was bedeutet, dass Kanäle, die nicht so viele wichtige Informationen enthalten, die gleiche Bedeutung erhalten wie Kanäle, die so viele Informationen enthalten.

4.3.0.3 Pointwise Convolution

Die *Pointwise Convolution* funktioniert genau wie die Standardfaltung, nur dass die Größe der Filter und die Schrittgröße in der Standardfaltung variabel sind und in *Pointwise Convolution* jeweils immer $(1, 1)$. Also jedes erhaltene Teilergebnis ist nur die Skalierung eines Eingangskanals. Die *Pointwise Convolution* hat für eine Eingabe I einen rechnerischen Aufwand von:

$$D_{in} \times W_{in} \times H_{in} \times n_f \tag{4.5}$$

Die Hauptidee bei der Verwendung der *Pointwise Convolution* ist es, die Korrelation zwischen den Eingangskanälen zu finden, oder lineare Kombinationen zu lernen und erleichtert die Reduzierung und Erhöhung der Anzahl der Kanäle, wenn es zu viele oder zu wenige Feature-Maps gibt. Es kann jedoch mit *pointwise Convolution* kein Feature extrahieren.

4.3.0.4 Depthwise Separable Convolution

Die *Depthwise Separable Convolution*(DSC) besteht aus einer *Depthwise Convolution*, gefolgt von einer *Pointwise Convolution*.Sie kann also Features schneller filtern und kombinieren als die Standardfaltung. Die DSC hat einen rechnerischen Aufwand von:

$$F_w \times F_h \times D_{in} \times W_{in} \times H_{in} + D_{in} \times W_{in} \times H_{in} \times n_f \tag{4.6}$$

Von 4.3 und 4.6 aus sieht man, die DSC weniger Berechnung benötigt als Standardfaltung, und zwar $\frac{1}{n_f} + \frac{1}{F_w \times F_h}$. Da die DSC zu wenig Parameter verwendet, werden die CNN, die die DSC verwenden, als quantisiert bezeichnet.Die DSC ersetzt langsam und sicher die Standardfaltung und das kann man mit $Xception$ und $MobileNet$ beobachten.
compare accuracy von Depthwise Separable Convolution and Standard Convolution

4.4 Vergleich zwischen Konvolution Neuronale Netzwerke

4.4.1 AlexNet

Wie in 2.1.4 schon erwähnt, wurde AlexNet von *Krizhevsky* et al [19] im Jahr 2012 entwickelt. AlexNet hat etwa 60 Millionen Parametern und etwa 650.000 Neuronen, besteht aus fünf ConvLs, von denen einige von Max-PooLs gefolgt sind, und drei FCLs(siehe Abbildung 29) [19]. Vor AlexNet gab es keine CNNs mit so vielen Schichten, noch interessanter sind die Techniken zur Verbesserung der Leistung von AlexNet :

- **ReLU**: Vor AlexNet waren *tanh* und *sigmoid* die Standard Aktivierungsfunktionen, aber wegen ihrer Sättigung bei hohen oder sehr niedrigen Werten und der Tatsache, das sie in diesen Bereichen eine Steigung nahe bei null haben, verlangsamen sie stark die Gewichtsanpassungen, was nicht der Fall bei $ReLU$ ist, das eine Steigung gleich null nur bei negativen Werten und bei positiven höheren Werte eine Steigung nicht nahe bei null hat. Das Training von CNN wird durch $ReLU$ um ein Vielfaches schneller als ein Äquivalent mit *tanh* beschleunigt[19].
- **Überlappendes Pooling**: Die normalen PooLs funktioniert wie in 2.2.1.4 ($pool\_size = stride$), aber die Überlappenden verwenden einfach ein Schrittgröße kleiner als das $pool\_size$.Nach [19] verbessern die überlappenden PooL die Netzgenauigkeit und macht das Netz gegenüber Overfitting robuster.
- **Dropout und Data Augmentation** : Ein anderer Vorteil von AlexNet ist die Verwendung von Dropout(4.6.1) in FCLs, das sich heute als die beste oder eine der besten Regulierungsmethoden erweist, und der Vermehrung der Daten durch Spiegelung, die die Wahrscheinlichkeit von Overfitting und die Anzahl von Epochen reduziert.

AlexNet kann aufgrund seiner Größe($\sim$ 240MB) nicht immer auf Systemen mit begrenztem Speicherplatz eingesetzt werden, deshalb ist man seit AlexNet ständig auf der Suche nach neuen Architekturen, die weniger Parameter und Speicherplatz brauchen und gleichzeitig die Ergebnisse auf der Stand der Technik erreichen. Es ist sicherlich in diesem Zusammenhang, dass viele neue effiziente Modelle wie $Xception$ und $MobileNet$ entstanden sind.

4.4.2 Xception

$Xception$([22]) basiert auf $InceptionV3$([23]). Die Inception-Modelle nutzen ein Inception-Block als Baustein. In einem Inception-Block werden mehrere Filter unterschiedlicher

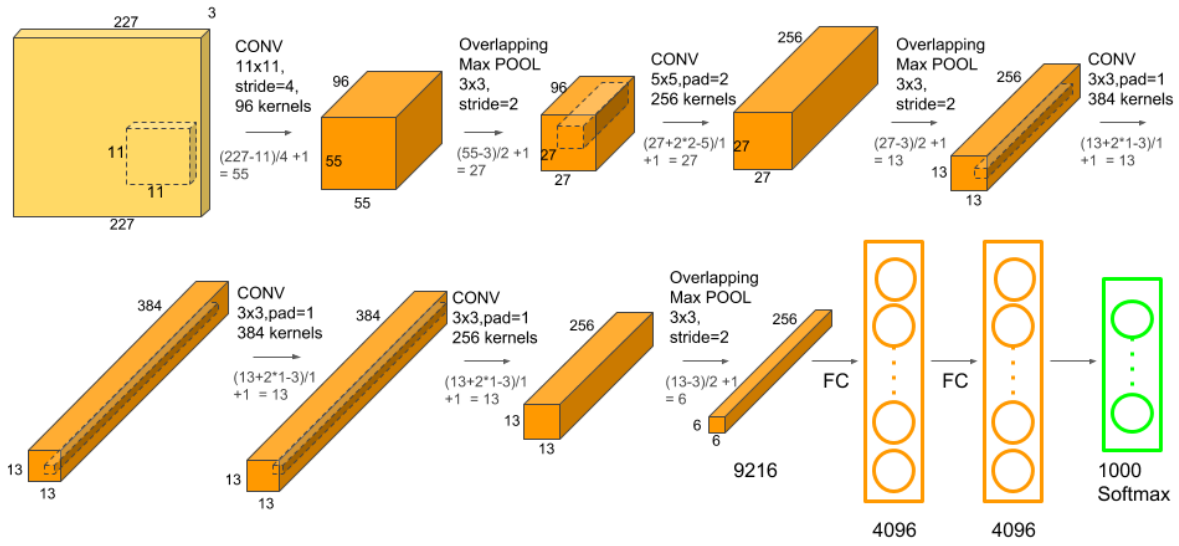


Abbildung 29: AlexNet Architektur source

Größe separat auf eine Eingabe angewendet und die Ergebnisse jeder Faltungsoperation werden zusammengeführt.Die Verwendung von mehreren Filtern unterschiedlicher Größe ermöglicht eine schnellere Feature-Extraktion, da die kleinen Filter sich mit der Extraktion von sehr lokalen Features beschäftigen, während die großen Filter sich mit der Extraktion von globalen Features beschäftigen. Der Xception-Block (30) wird mehrmals wiederholt, um das Xception-Modell zu gestalten. Aus der Abbildung Vergleich Graph sieht man an, dass Xception im Vergleich zu anderen Modellen relativ schnell ist.Die Anzahl der Parameter von Xception ist leider immer noch enorm ($\sim$ 22.000.000) .

4.4.3 MobileNet

MobileNet([24]) wie *Xception* macht die *Depthwise Separable Convolution* zunutze.Wie in der Abbildung(30) zu sehen ist, wurden aber *Batch-Normalisierung und ReLU6*-Layers vor und nach der 1×1 Faltungschicht hinzugefügt, was eine schnelle Feature-Extraktion ermöglicht und die *ReLU6* Aktivierung erhöht die Nichtlinearität der Entscheidungsfunktion. *MobileNet* nutzt etwas $\sim$ 3.500.000 Parameter, was mindestens sechsmal weniger als die Parameteranzahl von *Xception* ist, was auch eine bessere Inferenz ermöglicht.

4.4.4 TemkiNet

Inspiriert von den soeben beschriebenen CNNs haben wir *TemkiNet* und mehrere Variante davon entwickelt.Die Bausteine von *TemkiNet* sind in der Abbildung 32 zu entnehmen

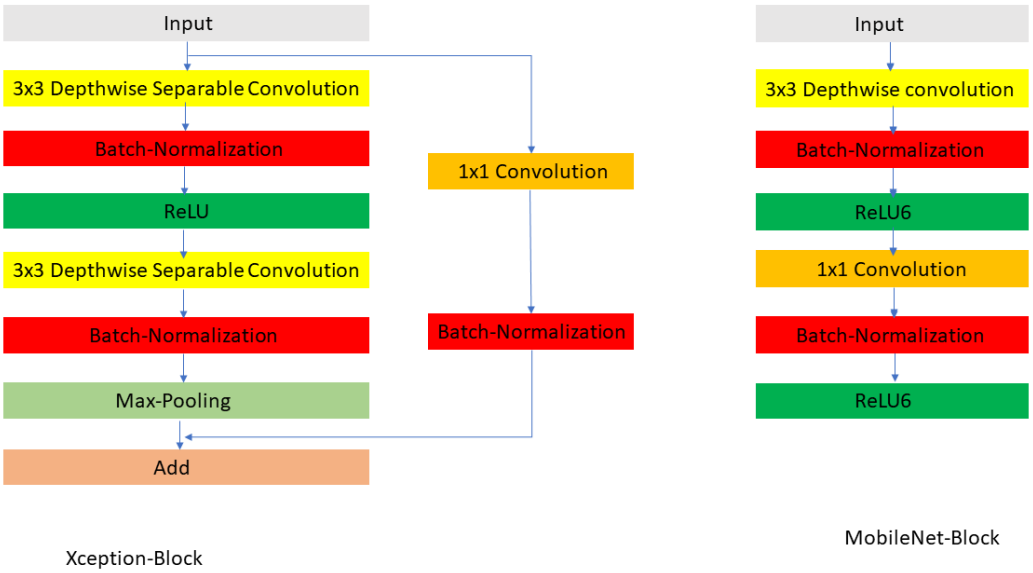


Abbildung 30: Xception- und MobileNet-Baustein.

4.5 Algorithmen zur Optimierung des Gradientenabstiegsverfahren: Optimizer

Die Wahl des Optimierungsalgorithmus für ein CNN kann den Unterschied zwischen guten Ergebnissen in Minuten, Stunden und Tagen ausmachen.Zum besserer Anwendung der Gradientenabstiegsverfahren wurden mehrere Optimierte Lernverfahren entwickelt. Im folgenden wird ein kurzer Einblick über die bekanntesten Lernverfahren(*Optimizer*) gegeben.
Alle heutige Optimizer haben SGD als Vorfahren und der Hauptnachteil von SGD ist , dass es die gleiche Lernrate für die Anpassung aller Netzwerkparameter verwendet und diese Lernrate wird auch während des Trainings nie geändert.

4.5.1 Adaptive Gradient Algorithm (AdaGrad)

AdaGrad bietet während des Netztrainings nicht nur die Möglichkeit, die Lernrate zu verändern, sondern auch für jeden Parameter eine geeignete Lernrate zu finden. Die AdaGrad-Aktualisierungsregel ergibt sich aus der folgenden Formel:

$$\begin{aligned} \alpha_t &= \sum_{i=1}^{t} (g_{i-1})^2 \quad \theta_{t+1} = \theta_t - \eta_t g_t \\ \eta_t &= \frac{\eta}{\sqrt{\alpha_t} + \epsilon} \end{aligned} \tag{4.7}$$

Voreingestellte Parameter$(KERAS)$: $\alpha_0 = 0.0 \quad \eta = 0.001 \quad \epsilon = 10^{-7}$

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1 Conv / s1	$3 \times 3 \times 512$ dw $1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$ $14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Abbildung 31: MobileNet Architektur

Dabei wird am Trainingsanfang eine Lernrate für jeden Parameter definiert und im Trainingsverlauf separat angepasst. Dieses Verfahren eignet sich gut für spärliche Daten, denn es gibt häufig auftretende Merkmale sehr niedrige Lernraten und seltene Merkmale hohe Lernraten, wobei die Intuition ist, dass jedes Mal, wenn eine seltene Eigenschaft gesehen wird, sollte der Lernende mehr aufpassen. Somit erleichtert die Anpassung das Auffinden und Identifizieren sehr voraussehbarer, aber vergleichsweise seltener Merkmale.[14].Wie in der Gleichung (4.7) festzustellen,nach einer bestimmten Anzahl von Iterationen haben wir keine Verbesserung der Netzleistung, denn je größer t wird, desto kleiner η_t wird und irgendwann wird η_t so klein, dass $\eta_t g_t$ fast gleich null ist.

4.5.2 Root Mean Square Propagation(RMSProp)

RMSProp wie AdaGrad findet für jeden Parameter eine geeignete Lernrate und zur Anpassung der Netzparameter basiert der RMSProp Optimizer auf den Durchschnitt der aktuellen Größen der Gradienten statt auf der Summe der ersten Moment wie in AdaGrad.Da $E[g^2]_t$ nicht schneller als α_t(4.7) ansteigt, wird die radikal sinkenden Lern-

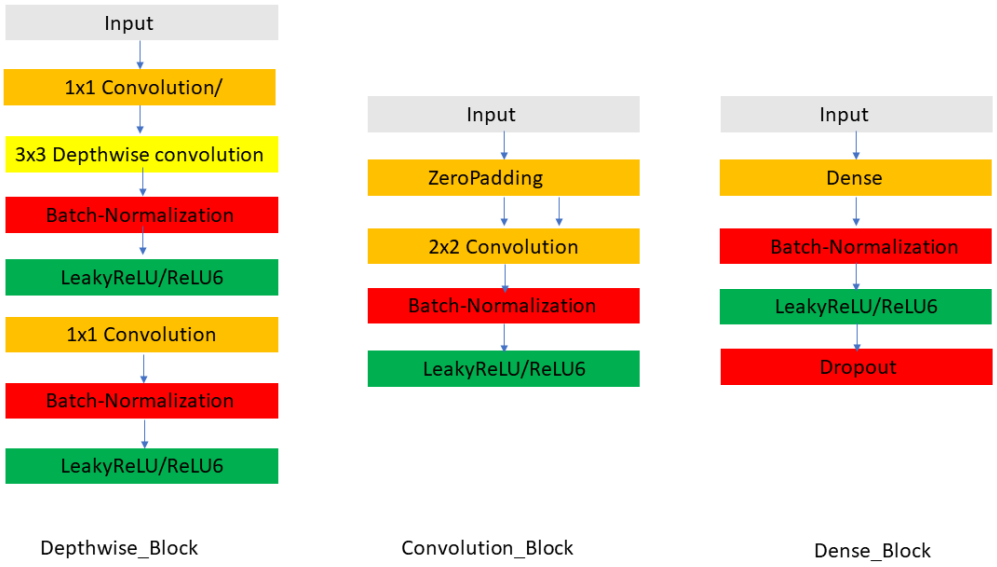


Abbildung 32: TemkiNet-Bausteine.

raten von Adagrad deutlich verlangsamt.Die Parameteranpassungen richten sich nach der folgenden Gleichung:

$$\begin{aligned} E[g^2]_t &= \alpha E[g^2]_{t-1} + (1-\alpha) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t, \quad \epsilon \approx 0 \end{aligned} \tag{4.8}$$

Der RMSProp funktioniert besser bei Online- und nicht-stationären Problemen.

4.5.3 Adaptive Moment Estimation(Adam)

Der Adam[15] Optimizer ist auch ein adaptiver Algorithmus,der die ersten und zweiten Momente der Gradienten schätzt, um individuelle adaptive Lernraten für verschiedene Parameter zu berechnen. Adam weist die Hauptvorteile von AdaGrad, das mit spärlichen Gradienten gut funktioniert, und RMSProp, das einige Probleme von AdaGrad löst und das für nicht-konvexe Optimierung geeignet ist,auf.Wie die Parameteranpassung von Adam Optimizer genau funktioniert, ergibt sich aus der folgenden Gleichung:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad \widehat{m}_t = \frac{m_t}{1-\beta_1^t} \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \quad \widehat{v}_t = \frac{v_t}{1-\beta_2^t} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\widehat{v}_t} + \epsilon} \widehat{m}_t \end{aligned} \tag{4.9}$$

Block	Schrittgröße	Outputgröße
conv_Block	2×2	Bildgröße
Depthwise_Block	2×2	Bildgröße/2
Depthwise_Block	1×1	Bildgröße/2
Depthwise_Block	2×2	Bildgröße/4
2×Depthwise_Block	1×1	Bildgröße/4
Depthwise_Block	2×2	Bildgröße/8
3×Depthwise_Block	1×1	Bildgröße/8
Depthwise_Block	2×2	Bildgröße/16
4×Depthwise_Block	1×1	Bildgröße/16
Depthwise_Block	2×2	Bildgröße/32
2×Depthwise_Block	1×1	Bildgröße/32
Flatten	/	
Dense_Block	/	1024
FC		101/102

Tabelle 2: TemkiNet Architektur

$$\text{Voreingestellte Parameter}(KERAS): \quad \begin{matrix} \beta_1\text{: } 0.9 & \beta_2\text{: } 0.999 \\ \eta\text{: } 0.001 & \epsilon\text{: } 10^{-7} \end{matrix}$$

Zu weiteren Vorteile der Nutzung von Adam gehört auch seine Einfachheit zur Implementierung, effizienter Nutzung der Speicherplatz und seine Invarianz zur diagonalen Neuskalierung der Gradienten.
Kleines Experiment

4.6 Problem beim Training von Convolutional neuronale Netzwerke

4.6.1 Overfitting

Wenn ein von der Maschine gelerntes Modell zu gut auf die Trainingsdaten abgestimmt ist und sehr schlechte Vorhersagen über Daten macht, die es bisher nicht gesehen hat, wird gesagt, dass das Modell an Überanpassung(Overfitting) leidet, anders gesagt, das Modell ist nicht in der Lage, die relevanten Merkmale aus den Trainingsdaten zu verallgemeinern, sondern die ganzen Trainingsdaten auswendig zu lernen.Im folgenden werden einige Mittels vorgestellt, um mit Overfitting umzugehen.

Strategie gegen Overfitting

Data Augmentation Ein großer Datensatz ist entscheidend für die Leistung tiefer neuronaler Netze.Dass ein Datensatz groß oder ausreichend für das Training eines CNNs ist, hängt nur von der Größe des CNNs ab und da die CNNs, die die besten Leistungen aufweisen, Millionen von Parametern haben, ist für jedes Problem des maschinellen Lernens nahezu unmöglich, ausreichende Daten zu finden.Anstatt immer neue Daten zu

sammeln, um die Netzwerkleistung zu verbessern, können wir die Leistung des Modells verbessern, indem wir neue Daten aus bestehenden Daten generieren.
Die populären Techniken oder Transformationen zur Vermehrung des Datensatzes sind die horizontalen oder vertikalen Spiegelungen, Drehungen, Skalierungen, Zuschneiden, Parallelverschiebungen und die Gauß'sches Rauschen. Für diese Arbeit wurden drei An-

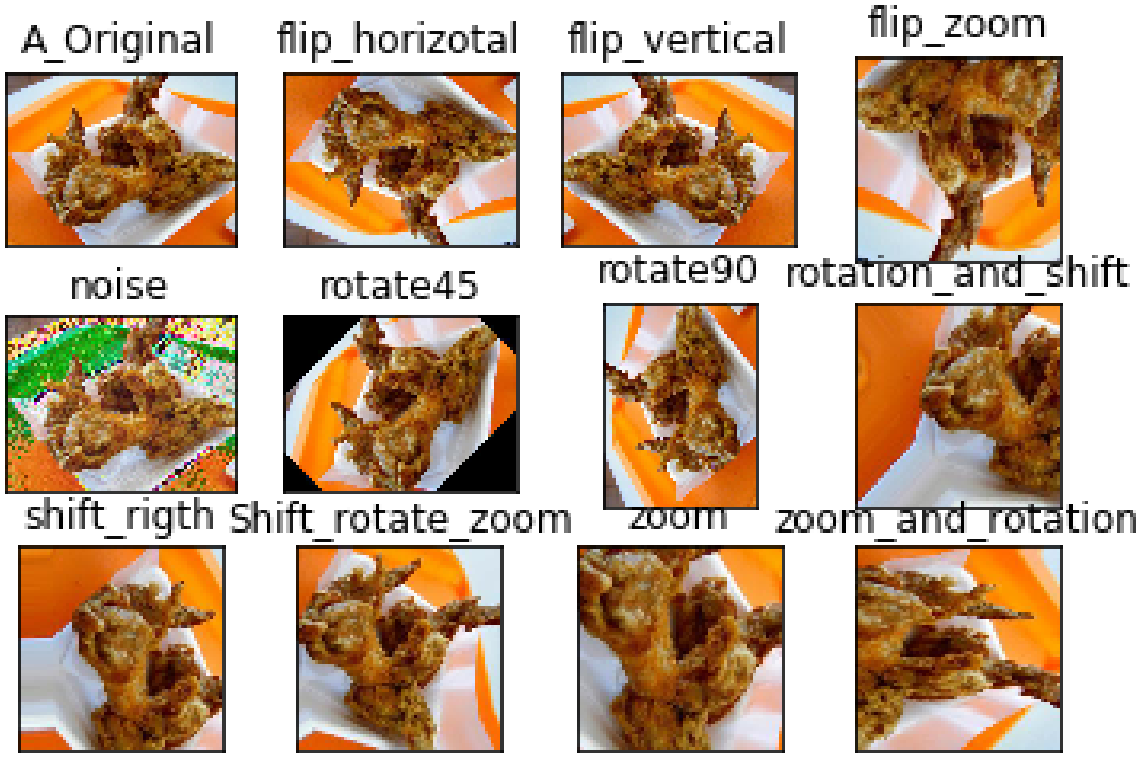


Abbildung 33: Anwendung von *ImageDataAugmentation*

sätze zur Erhöhung des Datensatzes verwendet: Der erste Ansatz besteht darin vor dem Training neue Daten zu erzeugen. Dabei werden die oben erwähnten Techniken vor dem Training angewendet, um zum Trainingszeitpunkt und zur Testzeit einen großen Datensatz zu haben. Der zweite Ansatz verwendet keinen größeren Datensatz, sondern immer das Original. Wenn Samplen aus dem Datensatz in das Netzwerk eingeführt werden, werden sie entweder transformiert oder direkt übertragen.Die *KERAS* Funktion *ImageDataGeneerator* bietet die Möglichkeit, Daten zur Laufzeit umzuwandeln. Vor dem Trainingsanfang werden alle Transformationen, die für jede Sample aus dem Datensatz durchgeführt werden können und beim Einspeisen einer Sample ins Netzwerk wird eine Transformation zufällig auf die Sample durchgeführt. Die Netzwerkeingabe ändern sich also ständig. Das interessanteste an *ImageDataGeneerator* ist, dass es mehrere Transformationen gleichzeitig anwenden(siehe Abbildung 33). Der dritte Ansatz ist die gleichzeitige Anwendung der beiden anderen Ansätze.

In der Abbildung 34 ist zu beobachten, dass die Datenvermehrung die Robustheit des CNN erhöht, Trainingszeit erhöht Es findet die offensichtlichsten Merkmale, die eine Klasse von einer anderen unterscheiden.

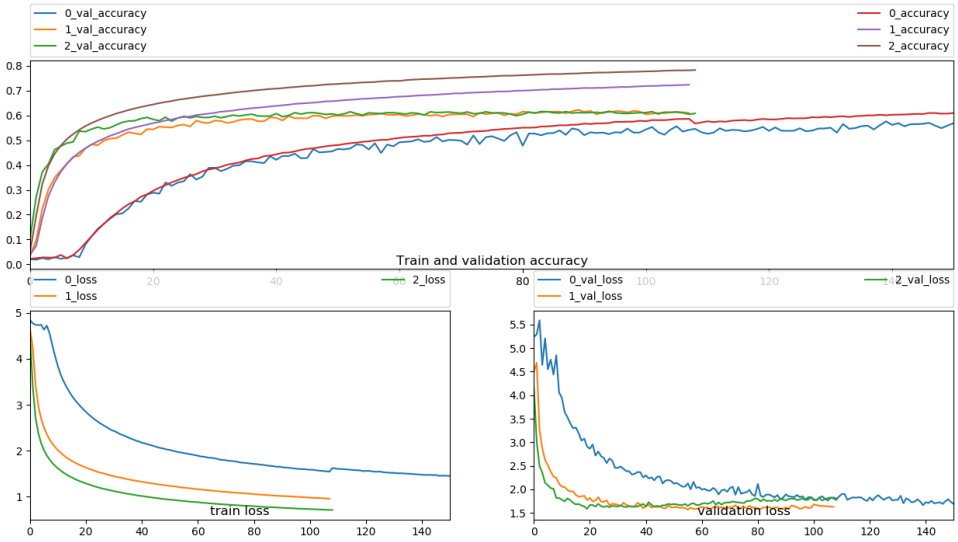
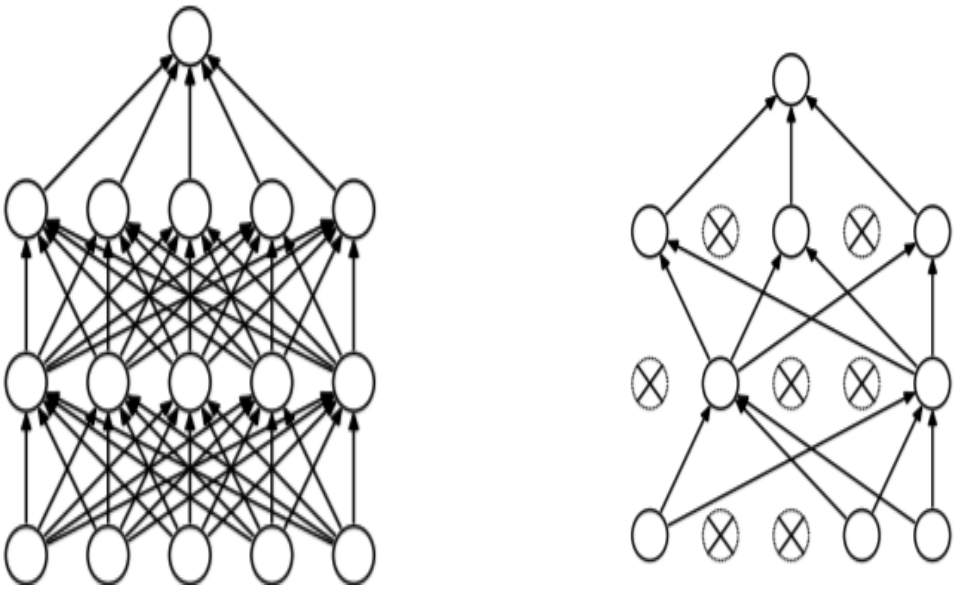


Abbildung 34: Anwendung von *ImageDataAugmentation*

Je mehr Daten verfügbar sind, desto effektiver können die CNNs sein. Es ist also mehr als wichtig über eine große Datenmenge zu verfügen. Leider können die gesammelten Datensätze nicht alle mögliche Szenarios des reellen Lebens abdecken,deshalb ist es auch bedeutend, CNN mit zusätzlichen synthetisch modifizierten Daten zu trainieren. Die CNNs funktionieren glücklicherweise besser oder immer gut, solange nützliche Daten durch das Modell aus dem ursprünglichen Datensatz extrahiert werden können, selbst wenn die erzeugten Daten von geringerer Qualität sind.

Dropout Künstliche Neurone sind von biologischen Neuronen inspiriert, aber die Beiden unterscheidet sich sehr voneinander und einer der wichtigen Unterschiede ist, dass biologische Neuronen unvollkommene Maschinen sind, die sehr oft nicht richtig funktioniert und das ist a priori nie den Fall bei künstlichen Neuronen. Wir könnten also glauben, dass KNN die biologische übertreffen könnten.Es sei denn, dass diese Funktionsstörung von biologischen Neuronen nicht eine Schwäche ist, sondern eher eine Stärke ist.Eine der verblüffenden Entdeckungen in Künstliche Intelligenz (KI) Bereich ist, dass es wünschenswert ist, künstliche Neuronen von Zeit zu Zeit zu Fehlfunktionen zu bringen[1]. Jetzt können wir uns fragen, wie Dysfunktion von Neuronen die Performances CNNs verbessern kann.Die zufällige Hinzufügen von Dysfunktionen in einer Schicht der CNN wird *Dropout* benannt und wurde von [3, Geoffrey E. et al] eingeführt.

Funktionsweise von Dropout .
Genauer gesagt,Dropout bezeichnet die zeitliche zufällige Ausschaltung von Neuronen(versteckt und sichtbar) in einem NN [4]. Wie die Abbildung 35 zeigt, wenn ein Neuron zufällig aus dem NN entfernt wird, werden auch all seine ein- und ausgehenden Verbindungen entfernt. In einer Dropout-Schicht wird ein Neuron N unabhängig von anderen Neuronen mit einer Wahrscheinlichkeit p zurückgehalten, d.h N wird mit einer Wahrscheinlichkeit von p nicht am Ergebnis der Schicht teilnehmen. Während der Testphase werden alle Verbindungen zurückgesetzt, die während des Trainings gelöscht wurden und die ausgehenden Verbindungen gelöschter Neurone mit p multipliziert.



(a) Standard neuronale Netze (b) Netze nach Dropout

Abbildung 35: Neuronales Netz mit Dropout ausgestattet [4].

Verhinderung der Koadaptationen zwischen Neuronen .
Während des Trainings können mehrere Neurone zur Minimierung der Fehlerfunktion so gut zusammenarbeiten, dass die Erkennung bestimmter Merkmale ohne diese Zusammenarbeit nicht mehr möglich ist, aber solche komplexe Koadaptationen können zu einer Überanpassung führen, denn diese komplexe Koadaptationen existieren nicht immer in Testdaten. Da die am Training teilnehmenden Neuronen nach dem Zufallsprinzip nach jeder Epoche ausgewählt werden, haben wir für jedes Training ein neues Modell, was die Neuronen zur Zusammenarbeit zwingt, ohne jedoch voneinander abhängig zu sein, anders gesagt, wird jedes Neuron unabhängig von anderen Neuronen die Muster korrekt lernen können.

Automatische Erhöhung von Training Daten und Regelung .
Noch dazu führt die Ausschaltung von Neuronen, wie es in Abbildung 36 angezeigt ist,

zu einer automatische Erzeugung neuer Trainingsdaten. Die verwendeten Daten in ausgedünnten Modellen sind also nur eine Abstraktion von echten Daten bzw. Rauschdaten und da wir für ein Netz mit n versteckten Einheiten, von denen jede fallen gelassen werden kann, 2^n mögliche Modelle haben, haben wir 2^n mögliche Abstraktion von unseren Daten und das sollte einer der Gründe sein, warum Dropout effektiver als andere rechnerisch kostengünstige Regler ist [4] und warum die Trainingszeit von NNs mit Dropout mindestens verdoppelt wird.

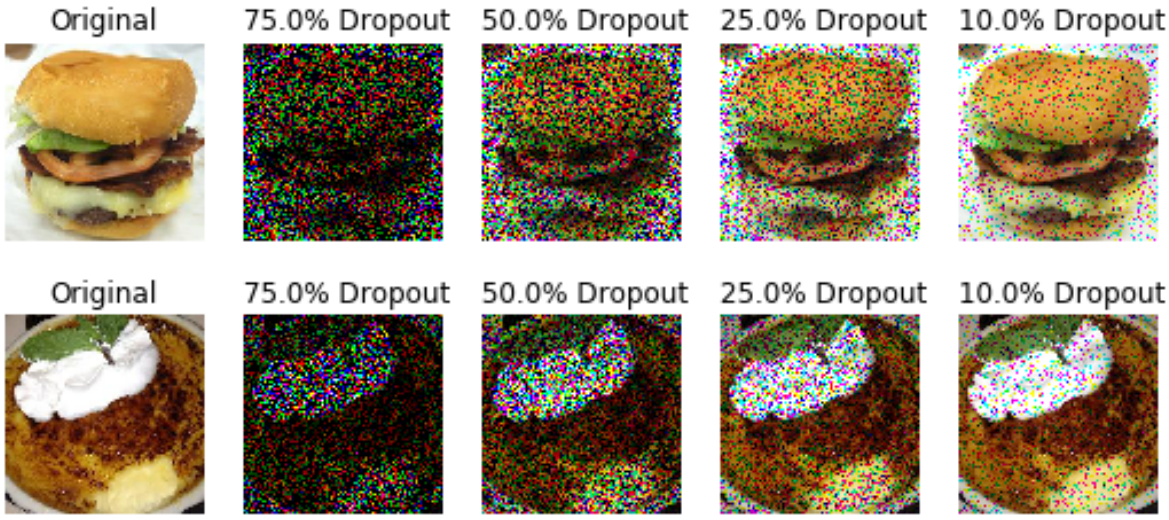


Abbildung 36: Erhöhung des Trainingsdaten durch Dropout

Da heutige CNNs Million von Neurons haben, wäre es unmöglich alle mögliche ausgedünnte Netzwerke zu trainieren, deshalb ist das Modell, das am Ende des Trainings erhalten wird, nur eine durchschnittliche Approximation aller mögliche Modelle, was schon gut, denn es gibt schlechte und gute Modelle.

Batch-Normalisierung Das Training tiefer neuronaler Netze ist sehr kompliziert und ein Grund dafür ist zum Beispiel die Tatsache, dass die Parameter einer Schicht während des Trainings tiefer neuronaler Netze immer unter der Annahme, dass sich die Parameter anderer Schichten nicht ändern, aktualisiert werden und da alle Schichten während des Updates geändert werden, verfolgt das Optimierungsverfahren ein Minimum, das sich ständig bewegt. Ein anderer Grund dafür ist die ständigen Veränderungen im Laufe des Trainings in die Verteilung des Netzinputs, diese Veränderung wird von [12] als interne kovariate Verschiebung(*Internal Covariate Shift*) genannt.Zur Lösung dieser Probleme schlagen *LeCunn et al*[**?**] vor dem Training das Netzinput zu normalisieren. Aber dieser Ansatz bringt nicht so viel, wenn das NN wirklich tief ist, denn nur der Netzinput profitiert von der Normalisierung und die kleinen Veränderungen in versteckte Schichten werden sich immer mehr verstärken, je tiefer man das Netz durchläuft. Mit der Ausbreitung tiefer NNs dehnt Batch-Normalisierung(BN)[12] diese Idee der Datennormalisierung auf versteckte Schichten tiefer NNs aus. Bei der BN werden die Eingaben in einem

Netzwerk standardisiert, die entweder auf die Aktivierungen einer vorherigen Schicht oder auf direkte Eingaben angewendet wird, so standardisiert, dass der Mittelwert in der Nähe von null liegt und die Standardabweichung in der Nähe von eins liegt. Die BN wird über Mini-Batches und nicht über den gesamten Trainingssatz durchgeführt, daher enthalten wir nur Näherungen an tatsächliche Werte der Standardabweichung und des Mittelwerts über das Trainingssatzes, aber wir gewinnen an Geschwindigkeit und an Speicherplatzverbrauch.Die Gleichung (4.10) gibt die formale Beschreibung des BN Algorithmus an.

Batch-Normalisierungstransformation, angewendet auf Aktivierung x über einen Mini-Batch

Input: Werte von x über einer Mini-Batch: $B = \{x_{1...m}\}$
Lernbare Parameter β, γ
Output: $\{y_i = BN_{\beta,\gamma}(x_i)\}$

$$\text{Mini-Batch Mittelwert} : \mu_\beta = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{4.10a}$$

$$\text{Mini-Batch Standardabweichung} : \sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \tag{4.10b}$$

$$\text{Normalisierung}: \widehat{x_i} = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}} \tag{4.10c}$$

$$\text{Skalierung und Verschiebung} : y_i = \gamma\widehat{x_i} + \beta \equiv BN_{\gamma,\beta} \tag{4.10d}$$

Wenn $\gamma = \sqrt{\sigma_\beta^2 + \epsilon}$ und $\beta = \mu_\beta$, bekommen wir die gleiche Verteilung wie vor der Batch-Normalisierung, d.h die Eingabe war also schon normalisiert. Interessanterweise kann das Netz während des Trainings eine bessere Verteilung als die erwünschte finden, denn γ und β sind lernbare Parameter.

Durch die BN kann zum einen eine hohe Lernrate verwendet,was in tiefer NNs ohne BN dazu führen kann, dass die Gradienten explodieren oder verschwinden und in schlechten lokalen Minima stecken bleiben.Die Verwendung einer höhere Lernrate ermöglicht einer schnellere Konvergenz. Zum anderen wird die interne kovariate Verschiebung geringer, was das Training beschleunigt, in einigen Fällen durch Halbierung der Epochen oder besser. Noch dazu wird das Netz durch die BN in gewissem Maße reguliert, daher wird die Verwendung von Dropout bzw. Regulierungstechnik reduziert oder sogar überflüssig und somit eine Verbesserung der Verallgemeinerungsgenauigkeit.

4.6.2 Dataset

5 Literatur

Literatur

[1] P. Kerlirzin, and F. Vallet: Robustness in Multilayer Perceptrons

[2] Md. Zahangir Alom,Tarek M. Taha, Christopher Yakopcic,Stefan Westberg,Mahmudul Hasan,Brian C. Van Esesn , Abdul A. S. Awwal und Vijayan K. Asari The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches

[3] Geoffrey E. Hinton and Nitish Srivastava and Alex Krizhevsky and Ilya Sutskever and Ruslan R. Salakhutdinov: Improving neural networks by preventing co-adaptation of feature detectors

[4] Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov: Dropout: A Simple Way to Prevent Neural Networks from Overfitting

[5] Ian Goodfellow, Yoshua Bengio, Aaron Courville: Adaptive Computation and Machine Learning series Page 342

[6] Song Han, Huizi Mao, William J. Dally Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

[7] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, Hans Peter Graf Pruning Filters for Efficient ConvNets

[8] Franco Manessi, Alessandro Rozza, Simone Bianco, Paolo Napoletano, Raimondo Schettini Automated Pruning for Deep Neural Network Compression

[9] Emily Denton,Wojciech Zaremba,Joan Bruna,Yann LeCun and Rob Fergus Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation

[10] Pavel Golik , Patrick Doetsch, Hermann Ney Cross-Entropy vs. Squared Error Training:a Theoretical and Experimental Comparison

[11] Neuronale Netze:Eine Einführung

[12] Sergey Ioffe, Christian Szegedy Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift

[13] Wikipedia: Learning Rate

[14] John Duchi,Elad Hazan, Yoram Singer: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

[15] Diederik P. Kingma, Jimmy Ba: Adam: A Method for Stochastic Optimization

[16] Compressing Models:Quantization

[17] Yoni Choukroun, Eli Kravchik, Fan Yang, Pavel Kisilev: Low-bit Quantization of Neural Networks for Efficient Inference

[18] wikipedia: Artificial neuron

[19] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton ImageNet Classification with Deep Convolutional Neural Networks

[20] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun: Deep Residual Learning for Image Recognition

[22] François Chollet Xception: Deep Learning with Depthwise Separable Convolutions

[23] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna Rethinking the Inception Architecture for Computer Vision

[24] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

[25] Bossard, Lukas and Guillaumin, Matthieu and Van Gool, Luc:Food-101 – Mining Discriminative Components with Random Forests