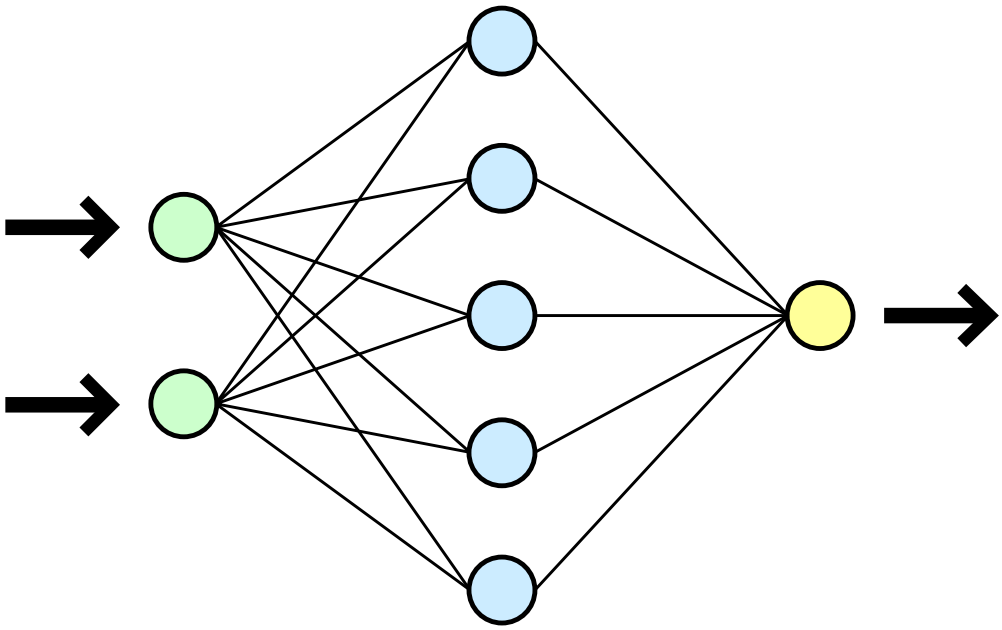
Machine learning

oder
„Des Gehirn Dings-do“[1]

Fundamental Concepts



Lugmayr Gabriel
2019-2020

[1] nach Dominik F.

I. INHALTSVERZEICHNIS

0. VORWORT

Dieses „Skript“ entstand hauptsächlich dadurch, dass ich mich für den Bereich des Maschinellen Lernens interessierte und etwas darüber lernen wollte. Mit der Playlist von „deeplizard“ fand ich eine hervorragende Informationsquelle welche ich im Zuge meines Lernens verschriftlichte. Deshalb sind Struktur und Semantik größtenteils ident. Aus diesem Grund möchte ich „deeplizard“ einen großen Dank aussprechen. Allerdings ist dieses Skript nicht zu 100% ident mit der Playlist. Einige Stellen habe ich ausgelassen und andere genauer ausgeführt. Dieses Skript ist in Deutsch geschrieben, dennoch findest du hier auch oft englische Fachbegriffe, da diese im Bereich der Informatik, vor allem bei Machine learning, sehr gebräuchlich sind und man diese kennen sollte. Ich habe versucht die Begriffe konsistent zu halten, dennoch ist mir dies leider nicht an jeder Stelle gelungen. Falls dir Begriffe beim Lesen unbekannt sind, schau in das Glossar (letztes Kapitel). Hier werden viele Begriffe im Zusammenhang mit Machine Learning erklärt

1. MACHINE LEARNING INTRO

1.1 DEFINITION

> Machine learning ist die Methode, Algorithmen zur Analyse von Daten zu verwenden, aus diesen Daten zu lernen und anhand dieses Lernens Vorhersagen über neue unbekannte Daten zu treffen.

1.2 MACHINE LEARNING Δ LOGIK-ALGORITHMEN

Wie unterscheidet sich nun machine learning von traditionellen Logik-Algorithmen?
Der Unterschied liegt, wie in der Definition oben geschrieben, in dem „Aus den Daten lernen“. Beim maschinellen Lernen wird die Maschine nicht manuell mit einem bestimmten Code zum Ausführen einer bestimmten Aufgabe programmiert, sondern sie wird mit Daten trainiert. Diese Daten durchlaufen Algorithmen, welche der Maschine die Fähigkeit geben, eine Aufgabe durchzuführen ohne dass sie davor explizite Anweisungen erhält. So können für gewisse Anwendungen einfacher und enorm bessere Resultate erzielt werden.

1.2.1 BEISPIEL: SATZBEDEUTUNGSANALYSE

Eine Textstelle soll mit „Positiv“ oder „Negativ“ klassifiziert werden. Also ob die Textstelle eine positive oder negative Emotion ausdrückt

1.2.1.1 PROGRAMMIERANSATZ: TRADITIONELL

Beim Traditionellen Programmieransatz würde nach bestimmten Wörtern gesucht werden die allgemein mit positiven oder negativen Emotionen assoziiert werden.

Zum Beispiel so:

Diese Wörter sind willkürlich von dem Entwickler gewählt. Wenn wir eine Liste von positiven und negativen

```
//pseudocode
let positive = [
    "happy",
    "thankful",
    "amazing",
    "great"
];

let negative = [
    "can't",
    "won't",
    "sorry",
    "unfortunately",
    "bad"
];
```

Wörtern haben, zählt ein simpler Algorithmus die Häufigkeit der negativen und positiven Wörter. So kann der Artikel auf Basis der Wörter, die ihm bekannt sind, klassifizieren, ob die Textstelle positiv oder negativ ist.

Dieser Ansatz hat mehrere Schwachstellen: Beispielsweise ist es enorm schwer (wenn nicht gar unmöglich) einen vollständigen Datensatz von richtig klassifizierten Wörtern zu bekommen oder zu erstellen. Auch ist es nicht sinnvoll den Text als positiv oder negativ anhand der Häufigkeit der positiven oder negativen Worte zu klassifizieren.

1.2.1.2 PROGRAMMIERANSATZ: MACHINE LEARNING

Der Algorithmus wird mit als positiv oder negativ klassifizierten Daten gefüttert. So lernt er, wie ein positiver oder negativer Text aussieht beziehungsweise welche Faktoren in dem Text auftreten müssen, damit er positiv oder negativ ist. Durch diesen Lernprozess kann der Algorithmus neue unklassifizierte Textstellen als positiv oder negativ erkennen. Als Entwickler gibst du also explizit an, welche Wörter positiv bzw. negativ sind, sondern nur welche Sätze positiv oder negativ sind. Aus diesen Sätzen kann der Algorithmus viel genauer bestimmen, was einen Satz positiv oder negativ macht.

```
// pseudocode
let articles = [
    {
        label: "positive",
        data: "The lizard movie was great! I really liked..."
    },
    {
        label: "positive",
        data: "Awesome lizards! The color green is my fav..."
    },
    {
        label: "negative",
        data: "Total disaster! I never liked..."
    },
    {
        label: "negative",
        data: "Worst movie of all time!..."
    }
];
```

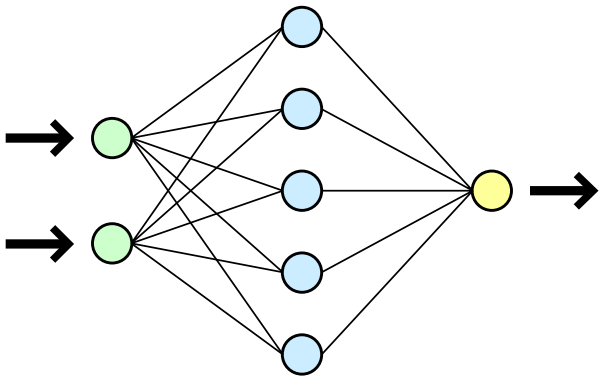
2. DEEP LEARNING ERKLÄRT

2.1 DEFINITION

Deep Learning ist ein Teilbereich des maschinellen Lernens, der Algorithmen verwendet, die von der Struktur und Funktion der neuronalen Netzwerke des Gehirns inspiriert sind.

2.2 KONZEPT

Bei Deep Learning handelt es sich, wie bei machine learning (siehe oben), immer noch um Algorithmen, die von Daten lernen. Allerdings basieren die Algorithmen beim Deep learning weitgehend auf der Struktur und der Funktionsweise des menschlichen neuronalen Netzes. Die neuronalen Netze, die wir beim Deep Learning verwenden, sind jedoch keine echten biologischen neuronalen Netze. Sie teilen einfach einige Eigenschaften mit biologischen neuronalen Netzen. Aus diesem Grund nennen wir sie künstliche neuronale Netzwerke (ANNs). Im Deep Learning wird der Begriff Künstliches Neuronales Netzwerk (ANN) austauschbar mit „Netz“, „neuronales Netz“ oder „Modell“ verwenden.

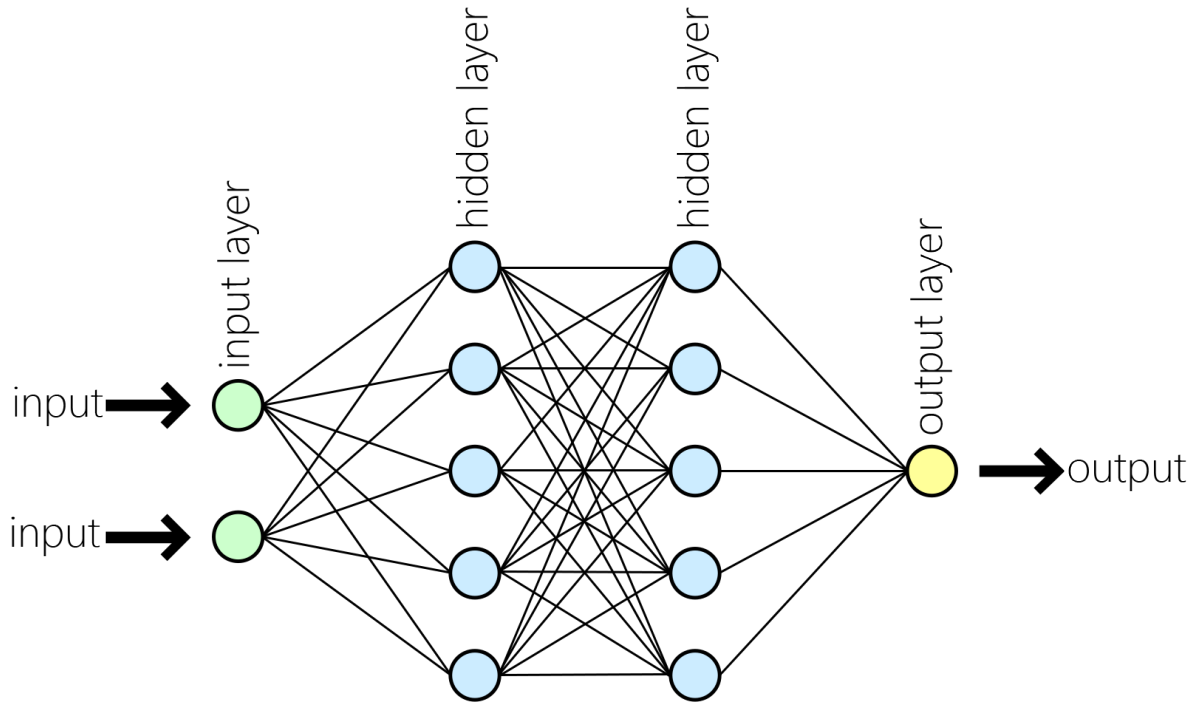


2.3 DAS „DEEP“ IN DEEP LEARNING

Um den Begriff Deep Learning zu verstehen, müssen wir erst verstehen, wie ANNs aufgebaut sind. Sobald das klar ist, können wir sehen, dass Deep Learning eine bestimmte Art von ANN verwendet, nämlich ein Deep Net (= Deep Artificial Neural Network).

Vorweg:

1. ANNs sind mit sogenannten „Neuronen“ aufgebaut
2. Neuronen in einem ANN sind in „Layers“ eingeteilt
3. Layers *in* einem ANN (Alle außer dem Input und Output Layer) heißen „Hidden Layer“
4. Wenn ein ANN mehr als einen Hidden Layer hat ist es ein Deep ANN



3. ARTIFICIAL NEURAL NETWORK (ANN) ERKLÄRT

Ein Artificial Neural Network ist ein Computersystem, das aus einer Sammlung verbundener Einheiten, den Neuronen, besteht. Es wird in Schichten organisiert

3.1 DEFINITION

3.2 ALLGEMEIN

Die verbundenen neuronalen Einheiten (Neuronen) bilden das Netzwerk. Jedes Neuron kann ein Signal zu einem anderen Neuron schicken. Ein Neuron verarbeitet alle Signale, die die Neuronen des vorherigen Layers ihm geschickt haben. Ein Neuron wird auch als Nodes bezeichnet.

Nodes sind in Layers unterteilt. Es gibt 3 Arten von Layers in jedem ANN:

1. Input Layer
2. Hidden Layer(s)

3. Output Layer

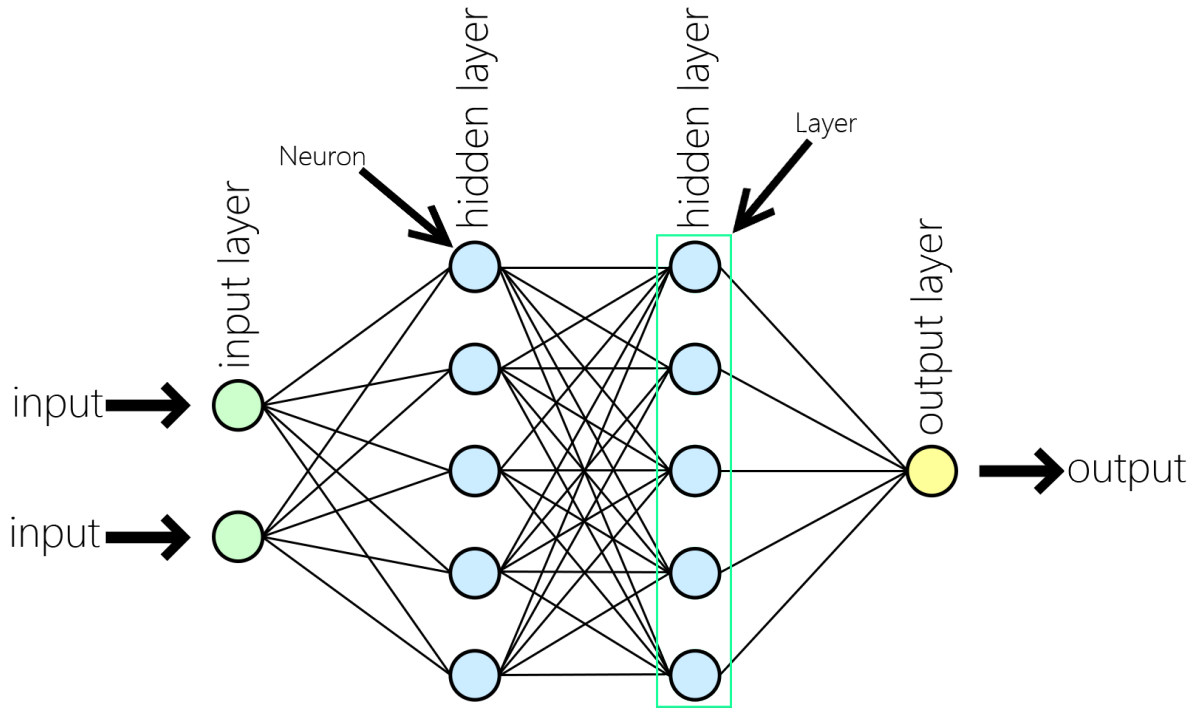
Verschiedene Layer führen verschiedene Arten von Transformationen an ihren Eingängen durch. Daten fließen durch das Netzwerk, beginnend bei dem Input Layer, durch die Hidden Layers und kommt schließlich zum Output Layer. Das nennt man „Forward pass“ durch ein Netzwerk. Alle Layer zwischen Input und Output sind Hidden Layers.

Betrachten wir die Anzahl der Knoten, die in jedem Schichtentyp enthalten sind:

1. Input Layer – Ein Node für jede Komponente der Eingabedaten
2. Hidden Layer(s) – Beliebig wählbare Anzahl von Nodes für jedes Hidden Layer
3. Output Layer – Ein Node für jeden der möglichen gewünschten Ausgänge

3.3 VISUALISIEREN EINES ANN

Wie du sicher schon weiter oben bemerkt hast wird ein ANN üblicherweise so illustriert:



Dieses ANN Hat insgesamt 4 Layer. Der Linke ist der Input Layer und der Rechte der Output Layer. Die zwei Layer in der Mitte sind Hidden Layers.
Nodes (= Neuronen) in jedem Layer:

1. Input Layer: 2 Nodes
2. Hidden Layer: 5 Nodes
3. Hidden Layer: 5 Nodes
4. Output Layer 1 Node

Weil das neuronale Netz zwei Nodes im Input Layer hat muss jeder Input in dieses Netz 2 Dimensionen haben. Zum Beispiel Höhe und Gewicht.
Da dieses Netzwerk zwei Knoten im Output Layer hat, dass es für jeden Eingang, der durch das Netzwerk geleitet wird (Forward pass!) (von links nach rechts (von Input zu Output)), zwei mögliche Ausgänge. Es könnten zum Beispiel Übergewicht oder Untergewicht die zwei Output Klassen sein. Die Output Klassen heißen auch Prediction Classes.

3.4 KERAS SEQUENTIAL MODEL

3.4.1 KERAS INTRODUCTION

Keras ist eine einsteiger- und benutzerfreundliche Open Source Deep-Learning-Bibliothek, geschrieben in Python.

3.4.2 KERAS INSTALLATION

Da davon auszugehen ist, dass du Keras nicht installiert hast, hier eine kurze Dokumentation wie ich Keras bei mir Installiert habe (14.11.2019) (Nur CPU ohne GPU) (Auf einer Windows 10 VM):

1. Ich habe Keras mit Anaconda installiert. Also ist zuerst Anaconda zu installieren:

   Installiere Anaconda für einen einzelnen Benutzer (Für alle Benutzer kann Probleme verursachen. Z.B: Du kannst keine Module mehr installieren, weil Anaconda nicht die benötigten Berechtigungen hat)
2. Installiere Keras und Tensorflow:
   2.1. Öffne Anaconda Prompt
   2.2. Erstelle ein neues conda Environment wo wir unsere Module installieren:
   `conda create --name PythonCPU`
   2.3. Aktiviere das conda Environment mit:
   `activate PythonCPU`
   2.4. (zum Deaktivieren: `conda deactivate`)
   2.5. Downgrade Python zu einer Keras & Tensorflow kompatiblen Version. Um Python auf 3.6 zu downgraden verwende
   `conda install python=3.6`
   2.6. Installiere Keras & Tensorflow
   `conda install -c anaconda keras`
   2.7. Installiere Spyder (eine IDE)
   `conda install spyder`
   2.8. Führe Spyder aus
   `spyder`
   2.9. Um sicherzugehen, dass alles korrekt installiert wurde, führe das in der Python Konsole (in spyder) aus:
   ```
   import numpy as np
   import tensorflow
   import keras
   ```
   Wenn keine Module Import Fehler erscheinen, hat die Installation richtig funktioniert

3.5.3 BUILD SEQUENTIAL MODEL

In Keras können wir ein sogenanntes sequenzielles Model erstellen. Keras definiert ein sequenzielles Model als einen sequenziellen Stack von linearen Layers. Das ist, was wir erwartet haben, da wir gelernt haben das Neuronen in Layers organisiert sind.
Das Sequenzielle Model ist Keras' Implementation von einem Artificial Neural network. So wird ein sehr simples

sequenzielles Model mit Keras erstellt:
Zuerst importieren wir die benötigten keras Klassen:
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Activation
Dann erstellen wir uns eine Variable namens Model, die wir gleichsetzen mit einer Instanz von einem Sequential object.

model = Sequential(layers)

An den Konstruktor übergeben wir ein Array mit Dense objects. Jedes dieser Objekte namens „Dense“ ist eigentlich ein Layer

```
layers = [
    Dense(3, input_shape=(2,), activation='relu'),
    Dense(2, activation='softmax')
]
```

Das Begriff „Dense“ bedeutet, dass diese Layers vom Typ „Dense“ sind. Dense ist ein spezieller Typ eines Layer, es gibt aber auch noch viele andere Typen.

Fürs erste verstehe, dass Denses die grundlegendste Art von Layer in einem ANN ist und dass jeder Output eines Dense Layers mit jeder Eingabe in die Schicht berechnet wird.

Wenn wir die Verbindungen in dem Bild eines ANNs (Oben), die von dem Hidden Laer zu dem Output Layer führen betrachten, sehen wir das jeder Node in dem Hidden Layer zu allen Nodes in dem Output Layer eine Verbindung hat.

Der erste Parameter, der an den Konstruktor des Dense Layer in jeder Schicht übergeben wird, sagt uns wie viele Neuronen der Dense Layer haben soll

Der zweite Parameter sagt uns wie viele Neuronen unser Input Layer hat. Braucht nur der erste Layer damit das Netz die Form der Daten, mit denen es arbeiten soll, weiß

Als letztes folgt die sogenannte Activation function (= Aktivierungsfunktion).

Mehr über das erfährst du später. Fürs erste merke dir, dass eine Aktivierungsfunktion eine nichtlineare Funktion ist, die typischerweise einem Dense Layer folgt.

4. LAYER EINES NEURONALEN NETZES

4.1 VERSCHIEDENE LAYERARTEN

Im vorherigen Kapitel (3) haben wir gesehen, dass die Neuronen in einem ANN in Layers organisiert sind. Als Beispiel wurde der Dense Layer genommen, welcher auch als vollständig vernetzter (fully connected) Layer bekannt ist. Allerdings gibt es verschiedene Arten. Hier einige Beispiele:

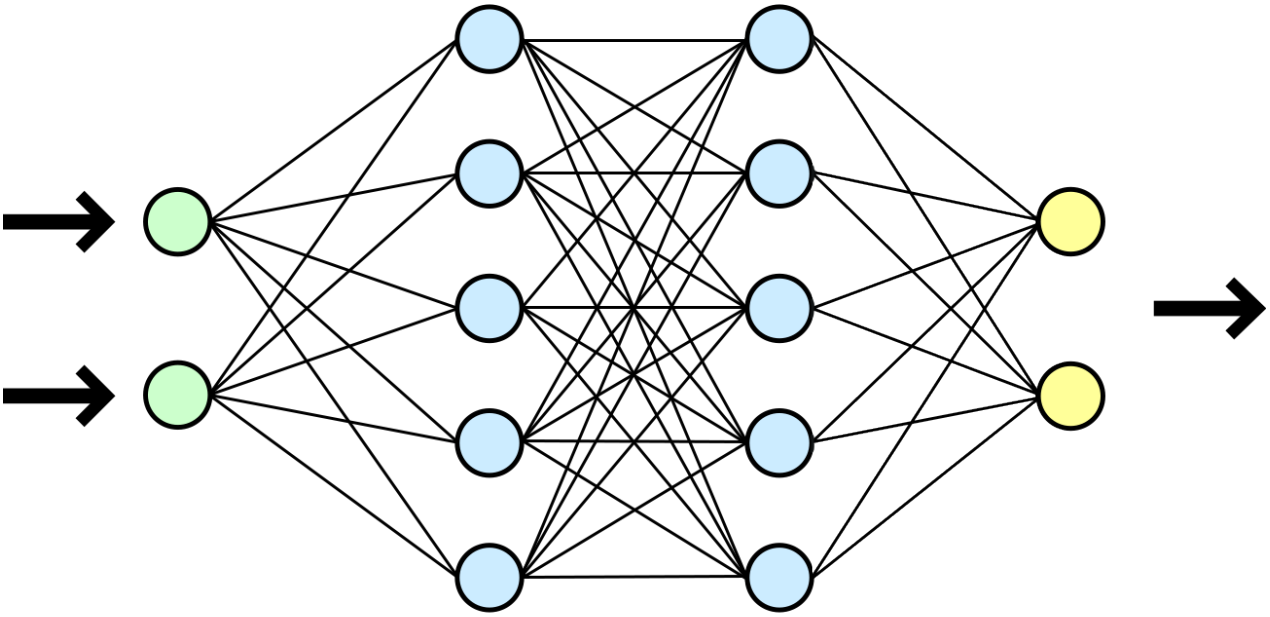
- Dense (fully connected) Layer
- Convolution Layer
- Pooling Layer
- Recurrent Layer
- Normalization Layer

4.1.1 WARUM GIBT ES VERSCHIEDEN LAYERARTEN?

Verschiedene Layer Transformieren ihren Input verschieden. Darum sind einige Layer für einige Aufgaben besser geeignet als andere Layer. Z.B.:

- Ein Convolutional Layer wir üblicherweise in Netzen benutzt, die mit Bilddaten arbeiten
- Recurrent Layers werden in Netzen benutzt, die mit Zeitreihen arbeiten
- Dense Layers verbinden jeden Eingang vollständig mit jedem Ausgang innerhalb seiner Schicht.

4.2 BEISPIEL EINES ANNS



Wir sehen, dass der erste Layer, der Input Layer, aus 2 Neuronen besteht. Jedes dieser Neuronen in diesem Layer steht für ein individuelles Merkmal von einem Beispiel in unseren Trainingsdaten. -> Jedes einzelne Beispiel in unserem Datensatz hat 2 Dimensionen. Das bedeutet, wenn wir ein Beispiel von unserem Datensatz unserem Netz übergeben, jeder der 2 Werte an das entsprechende Neuron im Input Layer übergeben werden. Wir sehen das jedes der 2 Input Nodes mit jedem Node im nächsten Layer verbunden ist. Jede Verbindung zwischen dem ersten und dem zweiten Layer überträgt die Ausgabe vom vorherigen Neuron auf den Input des nächsten Neurons. Die Beiden mittleren Layers mit jeweils 5 Nodes sind Hidden Layers da sie zwischen Input und Output Layer positioniert sind.

4.3 LAYER WEIGHTS

Jede Verbindung zwischen zwei Knoten hat ein zugeordnetes Gewicht, das einfach eine Zahl ist.

Jedes Gewicht repräsentiert die Stärke der Verbindung der beiden Nodes. Wenn ein Node in der Input Schicht einen Input empfängt, wird dieser Input über eine Verbindung an den nächsten Node weitergeleitet und mit dem Gewicht multipliziert, das dieser Verbindung zugeordnet ist.

Für jedes Neuron im zweiten Layer wird dann eine gewichtete Summe über jede der ankommenden Verbindungen berechnet. Diese Summe wird dann an eine Aktivierungsfunktion übergeben, welche eine Transformation der gegebenen Summer durchführt. Zum Beispiel transformiert eine Aktivierungsfunktion die Summe zu einer Nummer zwischen 0 und 1. Die eigentliche Transformation variiert abhängig von der benutzten Aktivierungsfunktion.

Node Output = Aktivierungsfunktion(Gewichtet Summer der Inputs)

4.4 FORWARD PASS DURCH EIN ANN

Sobald wie die Ausgabe für ein bestimmtes Neuron errechnet haben, ist die errechnete Ausgabe der Input der Neuronen im nächsten Layer. Der Prozess wird solange wiederholt, bis der Output Layer erreicht wird. Die Anzahl der Neuronen im Output Layer ist abhängig von der Anzahl der Output/Prediction Klassen. In unserem Beispiel haben wir 2 mögliche Prediction Klassen

Angenommen unser Netz hat die Aufgabe, 2 Arten von Tieren zu klassifizieren. Jeder Node im Output Layer würde eine der 2 Möglichkeiten darstellen. Zum Beispiel könnten wir nach Katze oder Hund klassifizieren. Die Kategorien/Klassen hängen davon ab wie viele Klassen wir in unserem Datensatz haben.

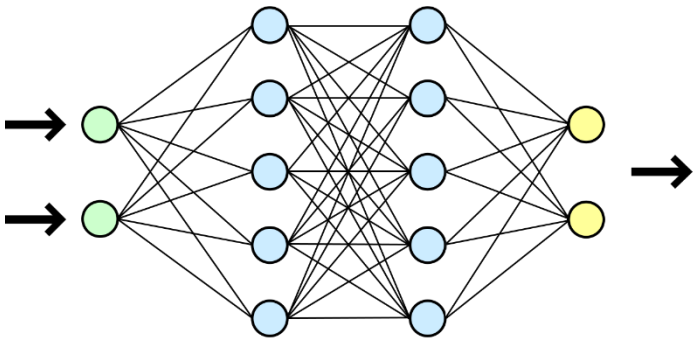
Für ein Beispiel aus dem Datensatz wird der gesamte Prozess von der Eingangsschicht bis zur Ausgangsschicht als Forward Pass durch das Netzwerk bezeichnet.

4.5 FINDEN DER OPTIMALEN WEIGHTS

Während dem Lernvorgang werden die Gewichte an allen Verbindungen aktualisiert und optimiert damit die Eingangsdaten im besser und präziser klassifiziert werden können. Mehr über die Findung der optimalen weights folgen später.

4.6 DAS BEISPIEL-SEQUENTIAL MODEL MIT KERAS

Wir starten mit dem Definieren von einem Array aus Dense Objekten, unseren Layers. Dieses Array wird dann an den Konstruktor des sequential Model weitergegeben.
Zur Erinnerung, so sieht unser Netz aus:



Definieren der Layer:

```
layers = [
   Dense(5, input_shape=(2,), activation='relu'),
   Dense(5, activation='relu'),
   Dense(2, activation='softmax')
]
```

Beachte das das erste Dense Objekt nicht der Input Layer ist. Das erste Dense Objekt ist der erste Hidden Layer. Der Input Layer ist durch den ersten Parameter des Konstruktors des ersten Dense Objektes festgelegt.

Unser Input hat 2 Dimensionen. Deshalb ist unser Input shape spezifiziert als input_shape=(2,).

Unser erster Hidden Layer hat genau wie der zweite 5 Node. Der Output Layer hat 2 Nodes.

Fürs Erste merke dir einfach, dass wir eine Aktivierungsfunktion namens relu für beide unserer Hidden Layers und für den Output Layer eine Aktivierungsfunktion namens softmax. activation='relu'; activation='softmax';

Unser finales Produkt schaut folgendermaßen aus:

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Activation


layers = [

        Dense(5, input_shape=(2,), activation='relu'),

        Dense(5, activation='relu'),

        Dense(2, activation='softmax')

    ]


model = Sequential(layers)
```

So wird ein ANN mit Keras geschrieben.

5. AKTIVIERUNGSFUNKTIONEN

5.1 WAS IST EINE AKTIVIERUNGSFUNKTION?

> In einem künstlichen neuronalen Netz ist eine Aktivierungsfunktion eine Funktion, die die Eingänge eines Neurons auf seinen entsprechenden Ausgang überträgt

Betrachtet man eine der vorherigen Bilder eines ANNs macht dies Sinn. Wir nahmen die gewichtete Summe einer jeden Input-Verbindung für jeden Node in dem Layer und übergaben diese gewichtete Summe an eine Aktivierungsfunktion

Node Output = Aktivierungsfunktion(gewichtet Summe der Eingänge)

Die Aktivierungsfunktion führt eine Operation durch, um die Summe in eine Zahl umzuwandeln, die normalerweise zwischen einer Untergrenze und einer Obergrenze liegt. (z.B. zwischen 0 und 1). Diese Operation ist meistens eine nichtlineare Transformation.

5.2 DIE AUFGABE EINER AKTIVIERUNGSFUNKTION

Was hat es auf sich mit dieser Aktivierungsfunktionstransformation? Was ist die Intuition? Um das zu erklären, sehen wir uns einige Beispiele an.

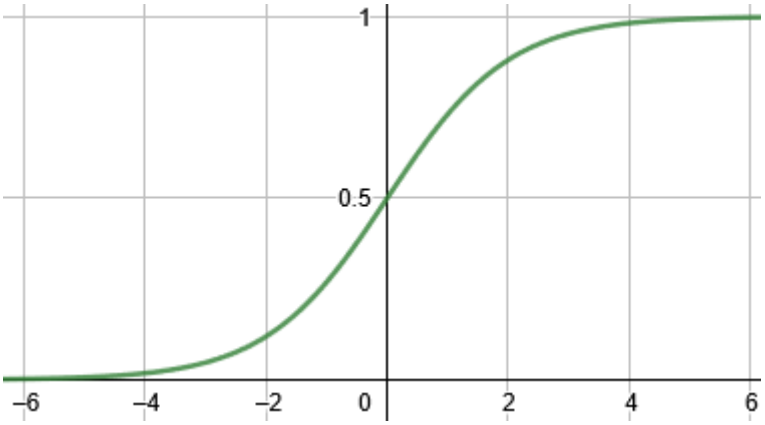
5.2.1 SIGMOID AKTIVIERUNGSFUNKTION

Sigmoid nimmt den Input und macht folgendes:

- Die meisten Negative Inputs transformiert Sigmoid in eine Nummer sehr nahe bei 0
- Die meisten Positiven Inputs transformiert Sigmoid in eine Nummer sehr nahe bei 1
- Inputs die relativ nahe bei 0 sind transformiert Sigmoid in eine Nummer zwischen 0 und 1

Mathematisch betrachtet ist Sigmoid einfach ein logistisches Wachstum:

$$sigmoid(x) = \frac{e^x}{e^x + 1}$$



Für Sigmoid ist 0 die Untergrenze (das Infimum) und 1 die Obergrenze (das Supremum)

5.2.2 INTUITION EINER AKTIVIERUNGSFUNKTION

Eine Aktivierungsfunktion ist biologisch inspiriert von der Aktivität unseres Gehirns, wenn verschiedene Neuronen aufgrund verschiedener Reize feuern (bzw. aktiviert werden).

Zum Beispiel, wenn du etwas Angenehmes riechst, so wie frische Waffeln mit Eis, feuern bestimmte Neuronen in deinem Gehirn und werden aktiviert. Wenn du etwas Unangenehmes riechst, so wie abgelaufene Milch, feuern andere Neuronen.

Tief in den Hirnregionen feuern bestimmte Neuronen entweder oder sie tun es nicht. Dies wird in einem ANN mit 1 für feuern und 0 für nicht feuern repräsentiert.

Mit der Sigmoidfunktion in einem ANN konnten wir sehe, dass ein Neuron zwischen 0 und 1 sein kann. Je näher der Output aus der Sigmoidfunktion bei 1 ist, desto aktivierter ist dieses Neuron. Je näher er bei 0 ist, desto weniger aktiviert ist dieses Neuron.

5.2.3 RELU AKTIVIERUNGSFUNKTION

Es ist aber nicht immer der Fall, dass unserer Aktivierungsfunktion den Input in eine Nummer zwischen 0 und 1 transformiert. Tatsächlich tut eine der gebräuchlichsten Aktivierungsfunktionen namens Relu (= Rectified Linear Unit) genau das nicht. Relu transformiert den Input zu einem maximum von entweder 0 oder in den

```
// pseudocode
if (smell.isPleasant()) {
    neuron.fire();
}
```

Input selbst.

relu(x) = max(0,x)

Wenn der Input weniger oder gleich 0 ist, dann gibt relu 0 aus. Wenn der Input höher als 0 ist, gibt relu einfach den Input aus.

```
// pseudocode
function relu(x) {
    if (x <= 0) {
        return 0;
    } else {
        return x;
    }
}
```

Die Idee dahinter ist, dass je positiver (bzw. höher) ein Neuron ist, desto aktivierter ist es.

Mit Sigmoid und Relu kennst du jetzt schon 2 Aktivierungsfunktionen, es gibt aber noch andere Aktivierungsfunktionen die die Daten anders Transformieren als diese beiden.

5.3 WARUM BENUTZEN WIR AKTIVIERUNGSFUNKTIONEN?

Um zu verstehen, warum wir Aktivierungsfunktionen verwenden, müssen wir zuerst Lineare Funktionen verstehen.

Stell dir vor, dass f eine Funktion auf einem Satz X ist.

Stell dir vor, dass a und b sind in X.

Stell dir vor, dass x eine reale Zahl ist.

Die Funktion f gilt als lineare Funktion, wenn und nur wenn:

$$f(a+b) = f(a) + f(b)$$

und

$$f(xa) = xf(a)$$

Eine wichtige Eigenschaft von linearen Funktionen ist, dass die Zusammensetzung von zwei linearen Funktionen auch eine lineare Funktion ist. Das bedeutet, selbst für sehr tiefe Neural Networks, dass wenn wir nur lineare Transformation von unseren Daten während dem Forward pass machen, die gelernte Zuordnung von Input zu Output ebenfalls linear ist.

Typischerweise sind die Mappings, die wir mit unseren Deep Neural Networks lernen wollen, komplexer als einfache lineare Mappings.

5.3.1 BEWEIS, DASS RELU NICHT LINEAR IST

Für jede reale Nummer x definieren wir eine Funktion f:

$$f(x) = relu(x)$$

Angenommen, a ist eine reale Nummer and $a < 0$

Mit dem Fakt, dass $a < 0$ ist können wir sehen, dass

$$f(-1a) = \max(0, -1a) > 0$$

Und das

$$(-1)f(a) = (-1)\max(0, a) = 0$$

Dass lässt uns zu dem Schluss kommen:

$$f(-1a) \neq (-1)f(a)$$

➔ Die Funktion f ist nicht linear

5.4 AKTIVIERUNGSFUNKTIONEN IN CODE MIT KERAS

Lass uns nun sehen wie man eine Aktivierungsfunktion in einem Keras Sequential model spezifiziert.

Zuerst müssen unsere benötigten Klassen importiert werden mit:

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Activation
```

Dann gibt es 2 verschiedene Wege

1) :
```
model = Sequential([
        Dense(5, input_shape=(3,), activation='relu')
    ])
```
In diesem Fall haben wir einen Dense Layer und spezifizieren relu als unsere Aktivierungsfunktion.
2) Als zweiten Weg kann man die Layers und Aktivierungsfunktionen zu unserem Model hinzufügen nachdem es instanziiert worden ist.:
```
model = Sequential()
model.add(Dense(5, input_shape=(3,)))
model.add(Activation('relu'))
```

6. TRAINIEREN EINES NN

6.1 WAS IST TRAINIEREN IN EINEM ANN?

Wenn wir ein Netz trainieren, versuchen wir im Grunde nur ein Optimierungsproblem zu lösen. Wir versuchen die Gewichte (siehe 4.3, 4.5) in dem Netz zu optimieren. Unsere Aufgabe ist es, die Gewichte zu finden, die unsere Input Daten zu dem Korrekten Output führt. Dieses „Mapping“ muss das Netz lernen. In 4.3 wurde vermittelt, dass jede Verbindung zwischen Nodes mit einem willkürlichen Gewicht versehen ist. Während dem Training werden diese Gewichte iterativ geupdated und deren Optimalwert nähergebracht.

```
# pseudocode
def train(model):
    model.weights.update()
```

6.2 OPTIMIERUNGSALGORITHMUS

Die Gewichte werden mithilfe eines sogenannten Optimierungsalgorithmus optimiert. Der Optimierungsprozess hängt von dem verwendeten Optimierungsalgorithmus. Es wird auch der Begriff Optimizer (= Optimierer) verwendet, um auf den verwendeten Algorithmus zu weisen. Der meist genutzte Optimizer heißt stochastic gradient descent (= stochastischer Gradientenabstieg), oder SGD.

Wenn ein Optimierungsproblem haben, müssen wir auch ein Optimierungsziel haben. Was ist also das Ziel von dem SGD Algorithmus für die Optimierung der Gewichte?
Das Ziel von SGD ist es, eine sogenannte „Loss function“ (= Verlustfunktion) zu minimieren. -> SGD aktualisiert die Gewichte so, dass die Loss function einen möglichst kleinen Wert hat.

6.3 LOSS FUNCTION

Eine übliche Loss function ist mean squared error (MSE). Es gibt aber noch einige andere Loss functions die wir stattdessen verwenden können. Als Deep learning developers ist es unsere Aufgabe zu entscheiden, welche Loss function wir am besten verwenden. Zunächst betrachten wir einmal die allgemeinen Loss functions. Später komme ich noch genauer auf Loss functions zurück

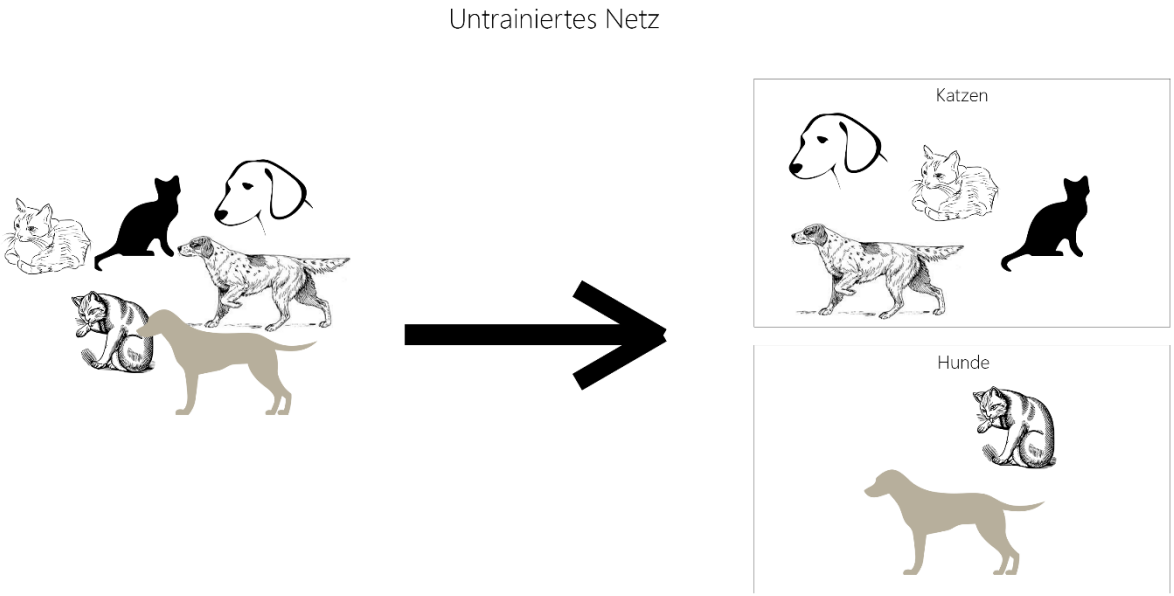
Was ist der Loss über den wir reden? Nun, während dem Training versorgen wir unser ANN mit Daten und deren dazugehörigen Labels. Stell dir beispielsweise vor, wir haben ein neuronales Netz, das Erkennen soll, ob auf einem Bild entweder ein Hund oder eine Katze ist. Wir werden unser Netz mit Bildern von Katzen und Hunden mit deren dazugehörigen Labels (Hund bzw. Katze) „füttern“.
Angenommen, wir füttern das Netz mit einem Bild von einer Katze. Wenn der Forward pass fertig ist und die Bilddaten durch das Netz gejagt wurde wird uns unser Netz uns unser Netz vorhersagen, ob das Bild eine Katze oder ein Hund ist.

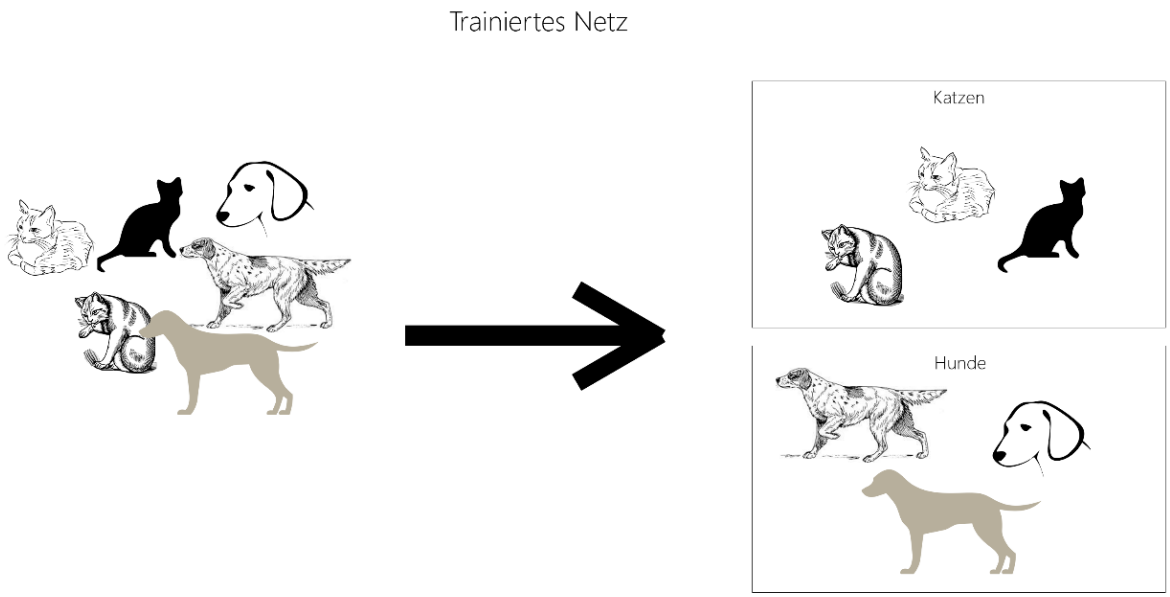
Der Output besteht dabei aus wie wahrscheinlich es ist, dass auf dem Bild eine Katze bzw. ein Hund ist. Es wird zum Beispiel sagen, dass das Bild zu 75% eine Katze und zu 25% ein Hund ist. In diesem Fall weißt das Netz dem Bild eine höhere Wahrscheinlichkeit zu, dass es eine Katze ist und kein Hund. Diese Herangehensweise ist sehr ähnlich dem, wie Menschen Entscheidungen treffen. Alles ist eine Vorhersage!

Der Loss ist der Fehler oder der Unterschied zwischen dem, was das Netz für das Bild vorhersagt und dem wahren Label des Bildes. SGD wird versuchen, diesen Fehler zu minimieren, um möglichst präzise Vorhersagen von dem Netz zu bekommen.

Nachdem wir unsere ganzen Daten durch unser Netz gejagt haben werden wir dieselben Daten immer wieder durchjagen. Dieser Prozess des fortlaufenden senden der gleichen durch das Netz ist das Training. Während diesem Prozess wird das Netz lernen.

Das bedeutet, dass das Netz anfangs „dumm“ ist und seine Entscheidungen zufällig trifft. Je öfter es aber trainiert, desto besser werden die Ergebnisse.





7. WIE LERNT EIN ANN

7.1 WAS IST EINE EPOCHE?

Im letzten Kapitel hast du über den Lernprozess erfahren und gesehen, dass jeder Datenpunkt, der für das Training verwendet wird, durch das Netzwerk gejagt wird. Sobald wir alle Datenpunkte in unserem Datensatz durch das Netz gejagt haben, sprechen wir von einer abgeschlossenen Epoche.

> Eine Epoche bezieht sich auf einen einzigen Durchlauf des gesamten Datensatzes während des Trainings

Beachte, dass es viele Epochen im Trainingsprozess gibt.

7.2 WAS BEDEUTET ES ZU LERNEN?

Wenn ein Netz initialisiert wird, werden die Gewichte zufällig gesetzt. Sobald wir eine Ausgabe erhalten, kann der Loss für diese spezifische Aufgabe berechnet werden, indem man sich die Differenz aus Vorhersage und echten (gelabelten) Wert nimmt.

7.2.1 GRADIENT DER LOSS FUNCTION

Nachdem der Loss berechnet wurde, wird der Gradient von diesem Loss mit Bezug auf jedes der Gewichte berechnet. Es ist zu beachten, dass Gradient nur ein Wort für die Ableitung einer Funktion aus mehreren Variablen ist.

Fahren wir mit dieser Erklärung fort und konzentrieren uns nur auf die Gewichte im Netz.

An diesem Punkt angelangt, haben wir den Loss eines einzelnen Outputs berechnet. Nun berechnen wir den Gradienten von diesem Loss mit Bezug auf unser einzelnes, gewähltes Gewicht. Diese Berechnung wird durchgeführt mithilfe einer Technik namens Backpropagation. Backpropagation behandeln wir später.

Wenn wir einmal den Wert des Gradienten unserer Loss function haben, können wir diesen Wert benutzen, um unser Gewicht upzudaten. Der Gradient sagt uns, in welche Richtung sich der Verlust auf das Minimum bewegt. Unsere Aufgabe ist es das Gewicht in die Richtung zu bewegen, die den Verlust senkt.

7.2.2 LERNRATE

Danach multiplizieren wir den Gradientenwert mit etwas namens learning rate (= Lernrate). Eine Lernrate ist

> Die Lernrate sagt uns, wie große Schritte wir in die Richtung des Minimums gehen.

eine kleine Nummer die gewöhnlich zwischen 0.01 und 0.0001 liegt.

Näheres über die Lernrate erfährst du später.

7.2.3 AKTUALISIERUNG DER GEWICHTE

Das neue Gewicht entsteht also dadurch, dass wir den Gradienten mit der Lernrate multiplizieren und dieses Produkt dann vom alten Gewicht abziehen

neues Gewicht = altes Gewicht – (Lernrate * Gradient)

Bis jetzt konzentrierten wir uns nur auf ein einzelnes Gewicht, um das Konzept zu erklären, dieser Prozess aber geschieht mit jedem der Gewichte, wenn Daten durchlaufen.

Der einzige Unterschied besteht darin, dass wenn der Gradient der Loss function berechnet wird, der Wert des Gradienten bei jedem Gewicht verschieden sein wird, da der Gradient für jedes Gewicht berechnet wird.

Stell dir nun vor, dass alle diese Gewichte schrittweise mit jeder Epoche aktualisiert werden. Die Gewichte werden zunehmend näher and den Optimalwert kommen während SGD die Loss function minimiert.

7.2.3 DAS NETZ LERNT :]

Dieses aktualisieren der Gewichte ist im Wesentlichen das, was gemeint wird, wenn gesagt wird, dass das Netz lernt. Es lernt welche Werte jedem Gewicht zuzuordnen sind basierend darauf, wie sich diese schrittweisen Änderungen auf die Loss function auswirken

7.3 TRAINING IN KERAS

Wenn wir ein Netz trainieren wollen, müssen wir als erstes das Netz bauen. (Wer hätt's gedacht)

Als erstes müssen die benötigten Klassen importiert werden:

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

import numpy as np
```

Als nächstes definieren wir uns unser Netz:

```
model = Sequential([

            Dense(16, input_shape=(1,), activation='relu'),

            Dense(32, activation='relu'),

            Dense(2, activation='sigmoid')

        ])
```

Bevor wir es trainieren, müssen wir unser Netz Kompilieren:

```
model.compile(

        Adam(lr=0.0001),

        loss='sparse_categorical_crossentropy',

        metrics=['accuracy']

        )
```

Wir übergeben der compile() Funktion den optimier, die loss function und die Metriken die wir sehen wollen. Beachte, dass der Optimizer den wir hier spezifiziert haben „Adam heißt". Adam ist eine Variante des SGD. In dem Adam Konstruktor spezifizieren wir die Lernrate. In dem Fall ist die Lernrate 0.0001.

Zum Trainieren brauchen wir auch Trainingsdaten. Dazu erstellen wir einfach ein numpy Array. Hierbei soll das Netz lernen, dass 0 zu eins und 1 zu 0 wird. Also quasi die Zahl invertiert. (Anm.: Ist für echtes maschinelles Lernen völlig sinnfrei, da dieses Problem einfacher und wesentlich besser mit einem traditionellen Programmieransatz gelöst werden kann.)

```
train_examples =
np.array([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1])
labels=np.array([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
```

Schließlich passen wir unser Netz an die Daten an. Das Anpassen des Netzes an die Daten bedeutet, das Netz auf diese Daten zu trainieren. Das tun wir mit folgendem Code:

```
model.fit(
        train_examples,
        labels,
        batch_size=2,
        epochs=20,
        shuffle=True,
        verbose=2
        )
```

train_examples ist ein numpy Array mit den Trainingsbeispielen

labels ist ein numpy Array mit den dazugehörigen labels für die Trainingsbeispiele

batch_size=2 spezifiziert wie viele Trainingsbeispiele wir auf einmal durch unser Netz jagen

epochs=20 bedeutet das das gesamte Trainingsset 20x durch das Modell gejagt werden

shuffle=True bedeutet das die Trainingsbeispiele vor dem durchlauf im Netz gemischt werden.

Verbose=2 gibt an, wie viel Protokollierung wir sehen, wenn das Netz trainiert.

Wenn wir diesen Code starten erhalten wir solch einen Output:

```
40/40 - 1s - loss: 0.7156 - accuracy: 0.3000
Epoch 2/20
40/40 - 0s - loss: 0.7090 - accuracy: 0.5000
Epoch 3/20
40/40 - 0s - loss: 0.7019 - accuracy: 0.5000
Epoch 4/20
40/40 - 0s - loss: 0.6953 - accuracy: 0.5000
Epoch 5/20
40/40 - 0s - loss: 0.6889 - accuracy: 1.0000
Epoch 6/20
40/40 - 0s - loss: 0.6825 - accuracy: 1.0000
Epoch 7/20
40/40 - 0s - loss: 0.6764 - accuracy: 1.0000
Epoch 8/20
40/40 - 0s - loss: 0.6706 - accuracy: 1.0000
Epoch 9/20
40/40 - 0s - loss: 0.6642 - accuracy: 1.0000
Epoch 10/20
40/40 - 0s - loss: 0.6583 - accuracy: 1.0000
Epoch 11/20
40/40 - 0s - loss: 0.6526 - accuracy: 1.0000
Epoch 12/20
40/40 - 0s - loss: 0.6466 - accuracy: 1.0000
Epoch 13/20
40/40 - 0s - loss: 0.6410 - accuracy: 1.0000
Epoch 14/20
40/40 - 0s - loss: 0.6350 - accuracy: 1.0000
Epoch 15/20
40/40 - 0s - loss: 0.6300 - accuracy: 1.0000
Epoch 16/20
40/40 - 0s - loss: 0.6252 - accuracy: 1.0000
Epoch 17/20
40/40 - 0s - loss: 0.6202 - accuracy: 1.0000
Epoch 18/20
40/40 - 0s - loss: 0.6151 - accuracy: 1.0000
Epoch 19/20
40/40 - 0s - loss: 0.6101 - accuracy: 1.0000
Epoch 20/20
40/40 - 0s - loss: 0.6048 - accuracy: 1.0000
```

Der Output gibt uns folgende Werte für jede Epoche:

1. Epochen Nummer
2. Zeitaufwand in Sekunden
3. Loss
4. Genauigkeit

Du wirst bemerkt haben, dass mit jeder Epoche der loss niedriger wird und die Genauigkeit höher wird. Das bedeutet, dass unser Netz richtig lernt.

8. LOSS

Wir lernten bereits im Kapitel 6 was eine loss function ist. Die loss function ist das, was SGD zu minimieren versucht, indem es die Gewichte im Netz iterativ (schrittweise) aktualisiert.

Am Ende jeder Epoche wird der loss mithilfe des Outputs des Netzes und den echten, richtigen Labels berechnet. Stell dir vor unser Netz soll Katzen und Hunde klassifizieren. Dabei ist das Label für Katze ist 0 und das für Hund ist 1.

Angenommen wir füttern das Netz mit einem Bild einer Katze und erhalten einen Output von 0.25. In diesem Fall ist die Differenz zwischen der Vorhersagung des Netzes und dem echten Label: 0.25 - 0.00 = 0.25. Diese Differenz heißt auch error.
Dieser Prozess wird für jeden Output wiederholt. Für jede Epoche wird der Fehler über alle einzelnen Outputs summiert.

Es gibt verschiedene loss functions. In diesem Kapitel werde ich MSE vorstellen.

Diese allgemeine Idee, die ich gleich für die Berechnung eines einzelnen Beispiels zeigen werde (in 8.1), gilt für alle verschiedenen Arten von loss functions. Die Implementation was wir wirklich mit jeden der Errors (Differenzen) machen hängt von dem Algorithmus ab, den unsere gewählte loss function verwendet. Zum Beispiel benutzen wir den Mittelwert der quadrierten Errors bei der Berechnung mit MSE, andere loss functions aber benutzen andere Algorithmen um den Wert des loss zu bestimmen.

Wenn wir unseren gesamten Datensatz auf einmal durch das Netz jagen, dann wird der Prozess, den wir gerade zur Berechnung des Verlustes durchlaufen haben, am Ende jeder Epoche während des Trainings stattfinden.

Wenn wir unseren Datensatz in mehrere Batches teilen und einen nacheinander durch das Netz jagen, dann wird der loss für jede Batch Size berechnet.

Da mit beiden Methoden der loss von den Gewichten abhängt, erwarten wir, dass sich der Wert des loss jedes Mal, wenn die Gewichte upgedated werden, sich ändert.

Da das Ziel von SGD darin besteht, den Verlust zu minimieren, wollen wir, dass der loss kleiner wir je mehr Epochen geschehen sind.

8.1 MEAN SQUARED ERROR (MSE)

Für ein einzelnes Beispiel berechnet MSE die Differenz (den error) zwischen der Output Prediction und dem Label. Dieser Fehler wird dann korrigiert. Für einen einzelnen Input wird also dies gemacht:

MSE(Input) = (Output – Label)(Output - Label)

Wenn wir mehrere Beispiele auf einmal (einen Batch) durchs Netz jagen, dann nehmen wir den Mittelwert der quadrierten Errors über alle Beispiele.

8.2 LOSS FUNCTION MIT KERAS

Wie kann nun eine loss function in Keras verwendet werden?

Ich erkläre dies anhand des Beispiels, welches wir in 7.3 geschrieben haben:

```
model = Sequential([
            Dense(16, input_shape=(1,), activation='relu'),
            Dense(32, activation='relu'),
            Dense(2, activation='sigmoid')
        ])
```

Wenn wir unser Netz erstellt haben, können wir es so kompilieren:

```
model.compile(
        Adam(lr=0.0001),
        loss='sparse_categorical_crossentropy',
```

```
        metrics=['accuracy']
        )
```

Wenn wir auf den zweiten Parameter in compile() schauen, können wir sehen, dass unsere spezifizierte loss function loss='sparse_categorical_crossentropy' ist.

In diesem Beispiel benutzen wir eine loss function namens sparse categorical crossentropy, aber es gibt noch viele andere.

Die aktuell (19.11.2019) verfügbaren loss functions in Keras sind:

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- huber_loss
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity
- is_categorical_crossentropy

9. LEARNING RATE/LERNRATE

Im Kapitel 7 habe ich die Lernrate schon grob erwähnt. Sie ist eine Nummer, die wir mit dem Gradienten multiplizieren. Jetzt erfährst du mehr Details über die Lernrate

9.1 EINFÜHRUNG DER LERNRATE

Wir wissen das das Ziel für SGD während des Trainings die Minimierung des loss zwischen dem richtigen Ergebnis (dem Label) und dem vorhergesagten Output des Netzes ist. Der Weg zu diesem minimierten loss braucht einige Schritte.

Wir wissen, dass wenn wir mit dem Trainingsprozess starten, wir zufällige Gewichte setzen und dann diese Gewichte schrittweise updaten damit wir dem minimierten loss näherkommen.

Die Größe dieser Schritte hängt von der Lernrate ab. Konzeptionell können wir uns die Lernrate von unserem Netz als die Schrittgröße vorstellen.

Kurze Wiederholung: Wir wissen, dass während dem Training, nachdem der loss für unsere Inputs berechnet worden ist, der Gradient/die Steigung in Bezug auf jedes der Gewichte in unserem Netz.

Wenn wir einmal die Werte dieser Gradienten haben, kommt die Lernrate ins Spiel. Die Gradienten werden dann mit der Lernrate multipliziert.

Gradienten * Lernrate

Die Lernrate ist üblicherweise eine kleine Zahl zwischen 0.01 und 0.0001. Die Zahl kann aber variieren.
-> jeder Wert, den wir für die Gradienten wird sehr klein, wenn wir ihn mit der Lernrate multipliziert haben.

9.2 AKTUALISIEREN DER GEWICHTE

Mit diesem Wert, der nach der Multiplikation mit der Lernrate herauskommt, aktualisieren wir also unsere Gewichte indem wir diesen Wert von ihnen abziehen

Neues_Gewicht = Altes_Gewicht – (Lernrate * Gradient)

Wir verwerfen also unsere alten Gewichte und ersetzen sie durch die aktualisierten neuen Gewichte.

Die der „richtige“ Wert der Lernrate erfordert herumprobieren und testen. Die Lernrate ist ein weiterer „Hyperparameter“ den wir für jedes Netz testen und tunen müssen bevor wir wissen, wo er die besten Resultate erzielt. Wie schon vorhin erwähnt ist es typisch ihn zwischen 0.01 und 0.0001 zu setzen.

Wenn wir die Lernrate zu hoch setzen riskieren wir ein Overshooting (= übertreffen, überschreiten) des optimalen Wertes. Dies tritt auf, wenn wir einen zu großen Schritt in die Richtung der minimierten loss function machen und dieses Minimum Verfehlen.

9.3 LERNRATEN IN KERAS

Wir benutzen (wieder) das Netz von Kapitel 7.3.

```
model = Sequential([
            Dense(16, input_shape=(1,), activation='relu'),
            Dense(32, activation='relu'),
            Dense(2, activation='sigmoid')
        ])
model.compile(
        Adam(lr=0.0001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
        )
```

Bei dem kompilieren des Netzes können wir sehen, dass der erste Parameter unseren Optimizer spezifiziert. In diesem Fall benutzen wir Adam.

Optional können wir dem Optimizer unsere Lernrate mitgeben mit dem Schlüsselwort `lr`. In diesem Beispiel ist 0.0001 unsere Lernrate.

Im letzten Absatz habe ich erwähnt, dass der lr Parameter optional ist. Wenn wir ihn nicht explizit angeben, dann nimmt Keras automatisch die Default learning rate für den jeweiligen Optimizer. Die Default learning rate findest du in der Keras Dokumentation.

Du kannst dem Optimizer auch noch anders mitteilen:

```
model.optimizer.lr = 0.01
```

Hier setzen wir die Lernrate auf 0.01. Wenn wir jetzt unsere Lernrate ausgeben, sehen wir, dass sie sich von 0.0001 auf 0.01 geändert hat.

10. TRAININGS, TESTING & VALIDATION SETS

10.1 DATENSÄTZE FÜR DEEP LEARNING

In diesem Kapitel geht es um die verschiedenen Datensäte die wir während dem Training und Testing eines Neuronalen Netzes benutzen.

Für Trainings- und Testzwecke teilen wir unsere Daten in 3 Teile. Nämlich in das

- Training Set
- Validation Set
- Test Set

10.1.1 TRAINING SET

Das Training Set ist, wie der Name schon sagt, der Datensatz, der zum Training des Netzes benutzt wird. Während jeder Epoche wird unser Netz immer wieder mit den gleichen Trainingsdaten, und lernt in jeder Epoche von denselben Daten.

Die Hoffnung dabei ist, dass wir später mit unserem Netz richtige Vorhersagen über Daten, die es noch nie gesehen hat treffen. Diese Vorhersagen trifft es basierend auf was es über die Trainingsdaten gelernt hat.

10.1.2 VALIDATION SET

Die Validationsdaten sind separiert von den Trainingsdaten. Sie dienen dazu, unser Netz während dem Training zu validieren. Dieser Validationsprozess gibt uns Informationen, welche uns helfen können unsere Hyperparameters zu adjustieren.

Während dem Trainieren mit den Trainingsdaten wird das Netz simultan mit den Validierungsdaten überprüft.

Wir wissen von den vorherigen Kapiteln, dass während dem Trainingsprozess das Netz den Output für jeden Input klassifiziert. (In dem Trainingsdatensatz) Nachdem diese Klassifizierung erfolgt ist, wird der loss berechnet und die Gewichte aktualisiert. In der nächsten Epoche werden die gleichen Inputs neu klassifiziert.

Während dem Training wird das Netz jeden Input des Validationsdatensatz auch klassifizieren. Es klassifiziert dies aber nur anhand von dem, was es von dem Trainingssatz gelernt hat und die Gewichte werden nach durchlauf der Validationsdaten nicht geupdatet.

Da die Trainingsdaten separat von den Validationsdaten gehalten werden, validiert sich das Netz nur anhand von Daten, mit denen es noch nie trainiert hat.

Einer der Hauptgründe, warum wir einen Validierungsdatensatz brauchen, ist sicherzustellen, dass unser Modell nicht zu stark an die Daten im Trainingssatz abgestimmt ist. Also das das Netz die Daten nicht „auswendig“ lernt. Das nennt man Overfitting. Overfitting bedeutet, dass unser Netz extrem gut die Daten in unserem Trainingssatz klassifizieren kann, aber nicht fähig ist die Eigenschaften der Daten zu generalisieren und akkurat Daten zu klassifizieren, die es noch nie gesehen hat. Overfitting und Underfitting werde ich im Detail später erklären.

Wenn wir also während dem Training auch den Validierungsdatensatz durch unser Netz laufen lassen und die Ergebnisse für diesen etwa so gut sind wie die vom Trainingsdatensatz, wissen wir, dass unser Netz nicht Overfitted ist.

Mit dem Validierungsdatensatz können wir feststellen, wie gut unser Netz
während des Trainings generalisiert

10.1.3 TEST SET

Der Testdatensatz ist ein Satz von Daten der benutzt wird um das Netz zu testen nachdem es trainiert wurde. Der Testdatensatz ist separat von Trainings- und Validierungsdatensatz.

Nachdem unser Netz trainiert und validiert wurde (mit Trainings, und Validierungsdatensatz), benutzen wir unser Netz um den Output von den ungelabelten Daten im Testdatensatz vorherzusagen.

Ein großer Unterschied zwischen dem Test Set und den anderen beiden Datensetzen ist, dass das Test Set nicht gelabelt sein sollte. Der Trainings- und Validierungsdatensatz müssen gelabelt sein damit wir unsere Metriken während des Trainings, wie der loss und die accuracy (= Genauigkeit), sehen.

Wenn unser Netz also über die Daten im Testdatensatz vorhersagen trifft, ist das der gleiche Prozess wie, wenn es in „echt“ verwendet werden würde.

Das Test Set bietet eine finale Überprüfung, dass unser Netz
gut generalisiert bevor es im Arbeitsumfeld eingesetzt wird.

Wenn wir zum Beispiel ein Netz verwenden, um Daten zu klassifizieren, ohne im Voraus zu wissen, was die Labels der Daten sind, oder wenn wir niemals die exakten Daten, die es klassifizieren wird, sehen, dann könnten wir unserem Netz auch keine gelabelten Daten geben.

Das ganze Ziel eines ANN’s ist ja, Daten zu klassifizieren ohne zu wissen, was die Daten sind. (Wenn man schon wüsste was die Daten sind bräuchte man sie ja nicht mehr klassifizieren)

Das letztendliche Ziel von maschinellem Lernen und Deep Learning ist es,
Artificial Neural Networks zu erstellen, die in der Lage sind,
gut zu generalisieren

10.2 DATENSÄTZE VON ANN: ZUSAMMENFASSUNG

Datensatz	Aktualisierung der Gewichte	Beschreibung
Training Set	Ja	Trainiert das Netz. Ziel des Trainings ist es, das Netz an die Trainingsdaten anzupassen und

		gleichzeitig auf unbekannte Daten gut generalisieren
Validation Set	Nein	Wird benutzt um nachzuprüfen wie gut ein Netz generalisiert
Test Set	Nein	Wird benutzt um die finale Fähigkeit des Netzes vor dem Echteinsatz zu generalisieren zu überprüfen

11. VORHERSAGEN IN EINEM ANN

11.1 DATEN OHNE LABELS

Im Wesentlichen geben wir bei einer Vorhersage unsere ungelabelten Testdaten an das Netz. Diese Vorhersagen sind abhängig von dem, was das Netz vorher gelernt hat.

Vorhersagen basieren auf dem, was das Netz während des Trainings gelernt hat.

Wenn wir zum Beispiel ein Netz zum Klassifizieren von Hunderassen (basierend auf Bilder von Hunden) programmieren, gibt das Netz die Rasse aus, von welcher es denkt, dass sie am wahrscheinlichsten ist.

Jetzt stell dir vor, dass unser Test Set Bilder von Hunden enthält, die das Netz noch nie gesehen hat. Wir geben diese Daten unserem Netz, und „fragen“ es, welchen Rasse jeder Hund auf dem Bild hat. Vergiss nicht, dass unser Netz keinen Zugriff auf Labels für diese Bilder hat.

Dieser Prozess zeigt uns, was das Netz gut gelernt, und was es nicht gut gelernt hat. Stell dir vor, wir trainieren unser Netz nur anhand von Bildern von großen Hunden trainieren aber unser Testdatensatz enthält einige Bilder von kleinen Hunden. Wenn wir nun ein Bild eines kleinen Hundes an unser Netz geben, wird es nicht gut vorhersagen können, welche Rasse der Hund hat, weil es nicht auf kleinere Hunde trainiert wurde.

Das bedeutet, dass wir sichergehen müssen, dass unseres Trainings und Validierungsdatensatz repräsentativ für die aktuellen Daten sind, für die das Netz vorhersagen treffen soll.

11.2 EINSATZ DES NETZES IN DER ECHTEN WELT [PRODUKTION]

Neben dem Vorhersagen über unsere Trainingsdaten können wir mit unserem Netz auch reale Daten vorhersagen.

Wir können zum Beispiel ein Netz zum Klassifizieren von Hunden in eine Website einbinden, die jeder aufrufen kann, Bilder seines Hundes hochladen und erfahren, welche Rasse der Hund hat.

Solch ein hochgeladenes Bild ist, logischerweise, nicht in unserem Trainings-, Validierungs- oder Testdatensatz.

11.3 BENUTZUNG VON KERAS FÜR VORHERSAGEN

Angenommen wir haben folgenden code:

```
predictions = model.predict(
    scaled_test_samples,
    batch_size=10,
    verbose=0
```

)

Das erste item, das wir hier haben ist eine Variable, welche wir predictions genannt habe. Wir nehmen an, dass wir bereits unser Netz gebaut und trainiert haben. Unser Netz in diesem Beispiel ist das Objekt namens „model“. Wir weisen predictions model.predict zu.

Diese predict() Funktion ist die Funktion, die wir aufrufen, wenn wir wollen, dass das Netz vorhersagen macht. Der predict Funktion übergeben wir eine Variable namens scaled_test_samples. Diese Variable beinhaltet unsere Testdaten. Wir setzen unsere batch_size willkürlicher Weise zu 10. Wir setzen die verbosity, welche dafür verantwortlich ist, wie viel am Bildschirm ausgegeben wird, zu 0, damit nichts angezeigt wird.

Um das Netz auszugeben benutzen wir diesen Code:

```
for p in predictions:
    print(p)
```

Würden wir diesen Code mit echten Daten ausführen würde in etwa so etwas herauskommen

[0.7410683 0.2589317]

[0.14958295 0.85041702]

...

[0.87152088 0.12847912]

[0.04943148 0.95056852]

Für dieses Netz haben wir 2 Output Kategorien und geben jede Voraussage für jedes Beispiel in unserem Testdatensatz aus.
Wir sehen, dass wir in der Ausgabe 2 Spalten haben. Diese stehen für die beiden Output Kategorien und zeigen uns die Wahrscheinlichkeit für jede der Kategorien. Lass uns die Kategorien der einfachheitshalber 0 und 1 nennen.

Zum Beispiel ist das erste Beispiel mit 74%iger Wahrscheinlichkeit in der Kategorie 0 und nur zu 26%tiger Wahrscheinlichkeit in der Kategorie 1

Das zweite Beispiel in unserem Testdatensatz ist mit 74%iger Wahrscheinlichkeit in der Kategorie 1 und mit 15%tiger Wahrscheinlichkeit in der Kategorie 0

12. OVERFITTING IN EINEM ANN

In diesem Kapitel geht es darum, was es heißt, wenn ein Netz als Overfitted bezeichnet wird. Es werden auch einige Techniken vorgestellt, um auftretendes Overfitting zu reduzieren.

Grob wurde hier das Konzept von Overfitting schon in 10.1.2 vorgestellt. Jetzt wird dies genauer ausgeführt.

Overfitting passiert, wenn unser Netz extrem gut darin wird, unsere Daten im Trainings Set vorherzusagen aber es schlecht darin ist, Daten zu klassifizieren an denen es nicht trainiert hat.

12.1 WIE MAN OVERFITTING ERKENNT

Overfitting erkennt man anhand der gegebenen Metriken für unsere Trainings- und Validierungsdaten während des Trainings. Wir haben bereits gesehen, dass wir bei der Festlegung eines Validierungsdatensatzes während des Trainings Metriken für die Validierungsgenauigkeit und -verlust sowie die Trainingsgenauigkeit und -verlust erhalten.

Wenn die Validierungsmetriken deutlich schlechter sind als die Trainingsmetriken ist das ein Anzeichen von Overfitting. Es ist auch ein Anzeichen von Overfitting, wenn das Netz während des Trainings gut ist, aber die Testdaten falsch klassifiziert.

12.2 REDUZIERUNG VON OVERFITTING

Overfitting ist ein sehr häufig auftretendes Problem. Es gibt aber einige Techniken um es zu reduzieren

12.2.1 MEHR DATEN HINZUFÜGEN

Die leichteste Möglichkeit ist (wenn wir überhaupt noch mehr haben) mehr Daten zum Trainingssatz hinzuzufügen. Desto mehr Daten wir dem Netz übergeben, desto mehr kann es davon lernen. Außerdem, so hoffen wir, erhöhen wir mit mehr Trainingsbeispielen die Diversität in unseren Daten.

Wenn wird zum Beispiel unser Netz darauf trainieren, ob ein Bild eine Katze oder ein Hund ist, und das Netz nur mit Bildern von großen Hunden trainiert wird, dann wird es wahrscheinlich bei echten Beispielen kleine Hunde nicht als Hunde identifizieren können. Wenn wir diesem Netz mehr Daten mit mehr unterschiedlichen Hunderassen (und somit auch Hundegrößen) geben, dann wird die Diversität in den Daten höher und das Netz Overfitted nicht so leicht.

12.2.2 DATENAUGMENTATION

Eine andere Technik, um Overfitting zu minimieren ist Datenaugmentation. Dabei erstellt man zusätzliche Daten durch sinnvolle Änderungen an unseren Daten in unserem Trainings Set. Für Bilddaten beispielsweise können wir solche Veränderungen durchführen:

- Zuschneide
- Rotieren
- Spiegeln
- Vergrößern

Die grundlegende Idee von Datenaugmentation ist, dass sie uns erlaubt, mehr Daten zu unserem Trainingssatz hinzuzufügen, die zwar ähnlich, aber nicht exakt gleich mit den Trainingsdaten sind.

Wenn wir zum Beispiel unser Netz nur an linksschauenden Hunden trainieren wäre es eine sinnvolle Änderung, dieselben Bilder nur gespiegelt dem Trainingssatz hinzuzufügen, um auch rechtsschauende Hunde zu erhalten.

12.2.3 KOMPLEXITÄTSREDUKTION DES NETZES

Ein anderer Weg, um Overfitting zu reduzieren ist es, die Komplexität unseres Netzes zu reduzieren. Die Komplexität können wir mit einfachen Änderungen reduzieren wie einige Layer des Netzes löschen oder die Nummer der Neuronen in den Layers zu reduzieren. Das hilft dem Netz bei der Generalisierung der Daten.

12.2.4 DROPOUTS

Die letzte Methode, die ich hier anführe, um Overfitting zu reduzieren heißt Dropout. Die allgemeine Idee hinter Dropout ist, dass wenn du es einem Netz hinzufügst es eine zufällige Anzahl der Neuronen in einem Layer ignoriert. Das heißt Dropout der Neuronen des Layers. Das beugt vor, dass dropped out Neuronen nicht an der Vorhersage, die das Netz trifft, nicht beteiligt sind.

Diese Technik hilft dem Netz auch Daten, die es noch nie gesehen hat zu generalisieren. Ich werden auf das Konzept von Dropouts noch einmal bei den Regulierungstechniken zurückkommen.

13. UNDERFITTING

13.1 WAS UNDERFITTING IST

In diesem Kapitel geht es darum, was es heißt, wenn ein Netz als Underfitted bezeichnet wird. Es werden auch einige Techniken vorgestellt, um auftretendes Underfitting zu reduzieren. Underfitting ist im Prinzip das Gegenteil von Overfitting.

Ein Netz wird als Underfitted bezeichnet, wenn es nicht fähig ist,
die Trainingsdaten zu klassifizieren

13.2 WIE MAN UNDERFITTING ERKENNT

Beim Underfitting sind unsere Metriken für das Training schlecht. Dass bedeutet, dass die training accuracy (Genauigkeit) schlecht ist und der loss ist hoch. Es ist dann schlecht in der Vorhersage von den Trainingsdaten und demnach auch schlecht in der Vorhersage von Daten, die es noch nie gesehen hat.

13.3 REDUZIERUNG VON UNDERFITTING

13.3.1 DIE KOMPLEXITÄT DES NETZES ERHÖHEN

Eine Sache, die wir tun können, ist, die Komplexität unseres neuronalen Netzes zu erhöhen. Das ist genau die gegenteilige Maßnahme wie beim Overfitting. Wenn unsere Daten sehr komplex sind und unser Netz aber relativ einfach aufgebaut ist, dann kann es sei, dass unser Netz nicht schlau genug ist um die Daten richtig zu Klassifizieren oder komplexe Daten vorherzusagen

So kann die Komplexität des Netzes erhöht werden:

- Die Anzahl der Layer erhöhen
- Die Anzahl der Neuronen in jedem Layer erhöhen
- Die Art und den Ort der Layers verändert.

13.3.2 DEN INPUTS MEHR FEATURES HINZUFÜGEN

Eine andere Technik zur Reduzierung des Underfittings ist es, mehr Features zu den Input Beispielen hinzufügen. (Wenn wir mehr hinzufügen können.) Diese Zusätzlichen Features können dem Netz helfen die Daten besser zu klassifizieren.

Wir haben zum Beispiel ein Netz, dass versucht den Preis einer Aktie anhand der letzten Drei Schlusskurse vorherzusagen. Unser Input hätte also drei Features:

- Tag 1 Schlusskurs
- Tag 2 Schlusskurs
- Tag 2 Schlusskurs

Wenn wir mehr Features zu diesen Daten hinzufügen, wie zum Beispiel die Öffnungspreise dieser Tage, dann kann das dem Netz helfen, mehr über unsere Daten zu lernen und seine Genauigkeit verbessern.

13.3.3 REDUZIEREN DES DROPOUTS

Das ist wieder die Gegenteilige Methode zu 12.2.4. Wie in diesem Kapitel erwähnt, ist Dropout eine Regulierungstechnik die zufällig eine Teilmenge der Neuronen in einem Layer ignoriert. Es verhindert im Wesentlichen, dass diese dropped out Neuronen an einer Vorhersage für die Daten teilnehmen. Wenn wir Dropout verwenden, dann spezifizieren wir, wie viele Prozent der Nodes wir droppen wollen. Wenn wir also eine 50%ige Dropoutrate definieren und wir bemerken, dass das Netz Underfitted, dann können wir den Dropout erniedrigen und testen, welche Metriken wir danach erhalten.

Diese Neuronen werden nur zu Trainingszwecken und nicht während der Validierung ausgelassen. Wenn wir also bemerken, dass unser Netz bessere Ergebnisse für die Validierungsdaten erzielt als bei den Trainingsdaten, dann ist das ein Anzeichen dafür, dass wir die Dropouts verringern sollen.

14. ÜBERWACHTES LERNEN

In diesem Kapitel geht es um Überwachtes (= supervised oder geführtes) Lernen. Bis jetzt wurde jedes Mal, wenn das Trainieren eines Netzes erwähnt wurde, handelte es sich eigentlich schon die ganze Zeit von überwachtem Lernen.

14.1 GELABELTE DATEN

Überwachtes Lernen entsteht, wenn die Daten in unserem Trainingsset gelabelt sind.

Labels werden verwendet, um den Lernprozess zu überwachen

In Kapitel 10 wurde schon erklärt, dass die Trainings und die Validierungsdaten gelabelt sind. Das ist der Fall bei überwachtem Lernen.

Bei überwachtem Lernen ist jedes Datenstück, das während des Trainings an das Modell übergeben wird, ein Paar aus Inputobjekt und dem dazugehörigen Label.

Im Grunde lernt das Netz beim überwachten Lernen, wie man von Inputs zu bestimmten Outputs basierend auf den Trainingsdaten, kommt.

Als Beispiel trainieren wir ein Netz zur Klassifizierung von verschiedenen Reptilienarten auf Basis von Bildern. Nun geben wir dem Netz ein Bild einer Schildkröte. Da wir überwacht Lernen, müssen wir das Netz auch mit einem Label für dieses Bild versorgen, was in diesem Fall einfach „Schildkröte“ ist. Das Netz wird dann den Output dieses Bildes klassifizieren und den loss durch die Differenz von dem vorausgesagtem und echtem (gelabelten) Wert ermitteln.

14.1.1 LABELS SIND ZAHLEN

Um dies zu tun, müssen die Labels in eine Zahl enkodiert werden. In diesem Fall kann das Label von „Schildkröte“ als 0, und das Label für „Schlangen“ als 1 kodiert werden.

Danach wird der loss für alle Daten in unserem Trainingssatz für die spezifizierten Epochen berechnet. Denke daran, dass das Ziel des Netzes während des Trainings darin besteht, den loss zu minimieren. Wenn wir unser Modell einsetzen und es verwenden, um Daten vorherzusagen, auf denen es nicht trainiert wurde, trifft es diese Vorhersagen auf Grundlage von den gelabelten Daten, auf denen es trainiert hat.

Was, wenn wir für unser Netz keine Labels bereitstellen? Dazu gibt es das Gegenteil des überwachten Lernens, das unüberwachtes Lernen. Es gibt auch noch eine andere Technik, das semi-überwachtes Lernen. Diese werden in späteren Kapiteln behandelt.

14.2 ARBEITEN MIT GELABELTEN DATEN IN KERAS

Wir starten wir üblich mit unseren Imports:

```
import tensorflow.keras

from tensorflow.keras import backend as K

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import activation, Dense

from tensorflow.keras.optimizers import Adam
```

Wir erstellen hier ein einfaches sequential Netz mit zwei hidden Dense Layers und einem Outputlayer mit zwei Kategorien

```
model = Sequential([
    Dense(16, input_shape=(2,), activation='relu'),
    Dense(32, activation='relu'),
    Dense(2, activation='sigmoid')
])
```

Jetzt Kompilieren wir unser Netz
Wir gehen davon aus, dass die Aufgabe dieses Netzes darin besteht, anhand von Größe und Gewicht zu klassifizieren, ob eine Person männlich oder weiblich ist.

```
model.compile(
    Adam(lr=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Nachdem wir unser Netz kompiliert haben, haben wir hier ein Beispiel einiger Trainingsdaten, die nur zu Veranschaulichung erstellt sind. (Keine echten Daten, also wird es hier wahrscheinlich nichts zu lernen geben)

```
#height, weight
```

```
train_samples = [
    [150, 67],
    [130, 60],
    [200, 65],
    [125, 52],
    [230, 72],
    [181, 70]
]
```

Die Trainingsdaten sind in train_samples gespeichert. Hier haben wir eine Liste von Trainingsbeispielen, wobei jedes Beispiel aus Gewicht und Größe besteht

Als nächstes haben wir unsere Labels gespeichert in der train_labels Variable. Hier repräsentiert eine 0 einen Mann und eine 1 eine Frau.

```
# 0: male
# 1: female
train_labels = [1, 1, 0, 1, 0, 0]
```

Die Position von den Labels entspricht der Position aller Trainingsdaten in unserer train_samples Variable. Zum Beispiel repräsentiert die erste 1 eine Frau und ist das Label für das erste Element in train_samples.

```
model.fit(
    x=train_samples,
    y=train_labels,
    batch_size=3,
    epochs=10,
    shuffle=True,
    verbose=2
)
```

15. UNÜBERWACHTES LERNEN

15.1 UNGELABELTE DATEN

Im Gegensatz zum überwachten Lernen arbeitet das Netz bei unüberwachten Lernen mit Daten, die nicht gelabelt sind.

Unüberwachtes Lernen tritt bei ungelabelten Daten auf

Wie lernt das Netz dann, wenn die Daten nicht gelabelt sind? Wie kann es sich selbst bewerten um zu verstehen ob es gut oder schlecht arbeitet?

Zunächst einmal ist klarzustellen, dass es bei unüberwachten Lernen nicht möglich ist, die Genauigkeit zu messen. Die Genauigkeit ist bei unüberwachtem Lernen keine typische Metrik, mit der wir einen unüberwachten Lernprozess überwachen.

Im Wesentlichen wir das Netz beim unüberwachten lernen mit einem ungelabelten Datensatz gefüttert und es versucht, eine Art Struktur aus den Daten zu lernen und die nützlichen Informationen oder Features davon zu extrahieren.

Es wird lernen, wie es ein Mapping von gegebenen Inputs auf bestimmte Outputs erstellt, basierend auf dem, was es über die Struktur dieser ungelabelten Daten lernt.

15.2 BEISPIELE FÜR UNÜBERWACHTES LERNEN

15.2.1 CLUSTERING

Eine der beliebtesten Anwendungen von unüberwachten Lernen ist der Einsatz von Clustering-Algorithmen.

15.2.1.1 CLUSTERANALYSE: ERKLÄRT

Das Clustering ist die Aufgabe, eine Menge von Objekten so zu gruppieren, dass Objekte eines Clusters (also einer Gruppe) einander ähnlicher sind als die in anderen Clustern. Die Clusteranalyse ist selbst kein spezieller Algorithmus. Sie ist nur eine zu lösende Aufgabe, die mit unterschiedlichen Algorithmen lösen lässt. Die verschiedenen Algorithmen unterscheiden sich teilweise stark. Sie haben ein unterschiedliches Verständnis davon, was einen Cluster ausmacht und wie man ihn effizient findet. Was der optimale Clusteralgorithmus ist, hängt vom Einsatzzweck ab. Mehr Informationen über Cluster findest du hier oder auf anderen Webseiten.

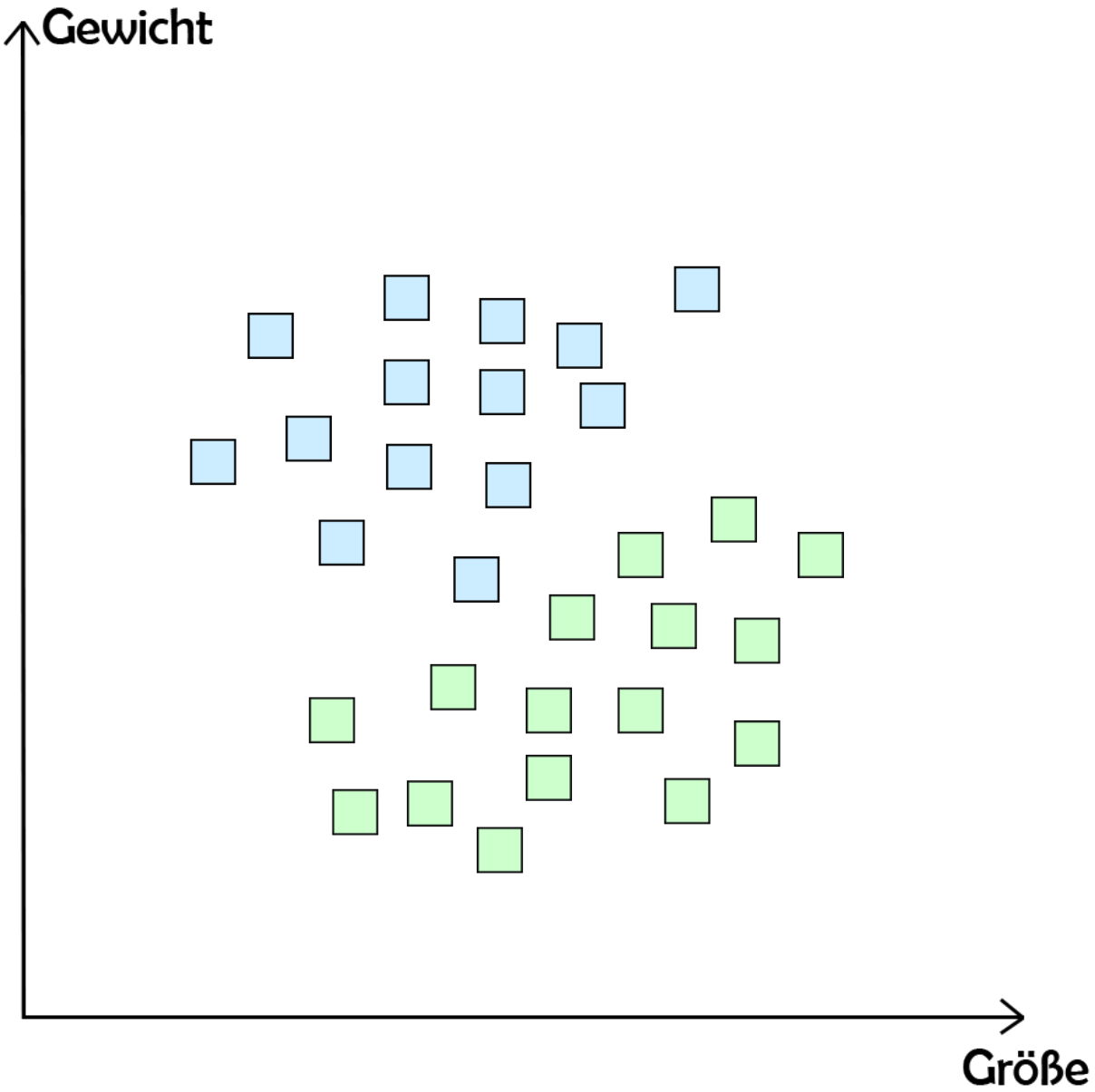
15.2.2 BEISPIEL

Bleiben wir bei unserem Beispiel aus dem vorigen Kapitel: Wir haben die Größe und das Gewicht von Männern und Frauen einer bestimmten Altersgruppe.

Dieses Mal haben wir keine Labels für diese Daten, also besteht jedes Beispiel dieses Datensatzes nur aus Größe und Gewicht. Die Daten sind mit keinen Labels verknüpft, die uns sagen ob diese Person männlich oder weiblich ist.

Nun könnte ein Clustering-Algorithmus diese Daten analysieren und beginnen, die Struktur zu lernen, obwohl die Daten ungelabelt sind. Durch das Lernen der Struktur kann es die Daten in Cluster unterteilen.

Wir können uns vorstellen, dass, wenn wir diese Daten auf einem Diagramm darstellen würde, es vielleicht etwa so aussehen würde:



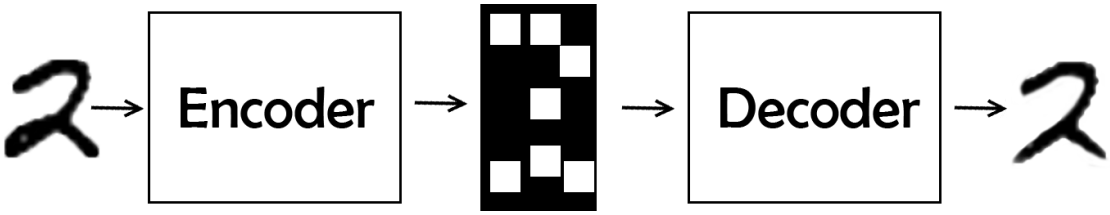
Nichts sagt uns hier explizit die Labels für diese Daten, aber wir können sehen, dass es hier zwei unterschiedliche Cluster gibt, und so können wir daraus schließen, dass diese zwei Cluster Männer und Frauen darstellt

15.2.2 AUTOENCODERS

Beim unüberwachten Lernen werden auch Autoencoders eingesetzt.

Im einfachsten Sinne ist ein Autoencoder ein Artificial Neural network, welches einen Input nimmt und dann eine Rekonstruktion dieses Inputs ausgibt.

Basierend auf allem, was wir bisher über neuronale Netze gelernt haben, erscheint dies seltsam, aber lass mich das an einem Beispiel näher erklären.



(eine nähere Erklärung zu diesem Beispiel hier.)

Angenommen, wir haben einen Satz von Bildern von handgeschriebenen Nummern und wollen sie durch einen Autoencoder geben. (Erinnerung: Ein Autoencoder ist nur ein neurales Netz).

Dieses neurale Netz nimmt das Bild dieser Zahl und wird es enkodieren. Dann, am Ende des Netzes, wird es das Bild wieder dekodieren und gibt die dekodierte Rekonstruktion des Originalbildes aus.

Das Ziel dabei ist, dass das rekonstruierte Bild so nah wie möglich am Originalbild ist.

Wie können wir messen, wie gut dieser Autoencoder bei der Rekonstruktion des Originalbildes ist, ohne es visuell zu inspizieren?

Nun, wir können uns die Loss function für diesen Autoencoder als eine Messung vorstellen, wie ähnlich die rekonstruierte Version des Bildes der Originalversion ist. Je ähnlicher das rekonstruierte Bild dem Originalbild ist, desto geringer ist der Loss.

Da es sich schließlich um ein ANN handelt, wird während des Trainings immer noch eine gewisse Variation des SGD verwendet, und somit ist unser Ziel die Minimierung des Loss.

Während des Trainings wird unser Modell also motiviert, die rekonstruierten Bilder immer näher an die Originalbilder anzugleichen.

15.2.2.1 ANWENDUNGEN VON AUTOENCODERN

Was ist eine Anwendung dieser Methode? Warum wollen wir einfach den Input rekonstruieren?

Nun, eine Anwendung dafür könnte die Rauschentfernung von Bildern sein. Sobald das Netz trainiert wurde, kann es andere, rauschende Bilder nehmen und wird fähig sein, die wichtigen Informationen daraus zu extrahieren und das Bild ohne rauschen zu Rekonstruieren.

16. SEMI-ÜBERWACHTES LERNEN

Semi-überwachtes Lernen ist eine Art Mischung aus überwachten, und unüberwachtem Lernen.

Semi-überwachtes Lernen. Wir haben hierbei also gelabelte und ungelabelte Daten.

16.1 GROßER UNGELABELTER DATENSATZ

Angenommen wir haben Zugriff auf einen großen ungelabelten Datensatz, auf dem wir unser Netz trainieren wollen und auf dem manuelles Labeling der Daten nicht praktisch wäre.

Wir könnten einen Teil dieses großen Datensatzes selbst durchgehen ihn manuell beschriften und diesen Teil nutzen, um unser Modell zu trainieren.
Das ist in Ordnung. Tatsächlich werden auf diese Weise viele Daten, die für neuronale Netze verwendet werden, gelabelt. Wenn wir jedoch Zugang zu großen Datenmengen haben und nur einen kleinen Teil dieser Daten gelabelt haben, dann wäre es doch eine Verschwendung, alle anderen ungelabelten Daten nicht zu verwenden. Es ist doch so, dass mit mehr Daten unser Netz besser lernt.
Was können wir also tun, um unsere verbliebenen, ungelabelten Daten zu verwenden?

Eine Sache, die wir tun können, ist, Pseudo-Labeling. Pseudo-Labeling ist eine Technik in der Kategorie des semi-überwachtem Lernen.

16.2 PSEUDO-LABELING

So funktioniert das Pseudo-Labeling. Wie bereits erwähnt, ist ein Teil unserer Daten gelabelt. Nun werden diese gelabelten Daten als Trainingssatz für unser Netz verwendet. Wir werden unser Netz trainieren, genau wie bei jeden anderen gelabelten Datensatz.

Nur durch den regulären Trainings Prozess wird unser Netz ziemlich gut. Also alles was wir bis alles was wir bis jetzt getan haben ist reguläres überwachtes Lernen.

Danach kommt das unüberwachtes Lernen ins Spiel. Nachdem unser Netz auf den gelabelten Daten trainiert worden ist, benutzen wir unser Netz, um die verbleibenden ungelabelten Daten vorherzusagen und benutzen diese Vorhersagen, um die jetzt noch ungelabelten Daten mit Labels zu versehen.

Dieser Prozess des Labelns der ungelabelten Daten ist das wesentliche beim Pseudo-Labeling.

Nach dem automatischen Labeln der ungelabelten Daten trainieren wir unser Netz auf dem ganzen Datensatz, also den Daten, die wirklich gelabelt worden sind und den Daten, die mit Pseudo-Labeling gelabelt worden sind.

Pseudo-Labeling ermöglicht das Trainieren auf einem wesentlich größeren Datensatz

So sind wir in der Lage, auf Daten zu trainieren, die ansonsten möglicherweise viele Stunden menschlicher Arbeit in Anspruch genommen hätten, um sie zu Labeln.

17. DATA AUGMENTATION

Daten Augmentation (= Datenvermehrung) tritt auf, wenn wir neue Daten erstellen, die nur Änderungen unserer bestehenden Daten sind. Im Wesentlichen erstellen wir neue Daten indem wir passende Änderungen an den Daten in unserem Trainingssatz vornehmen

Zum Beispiel könnten wir Bilddaten erweitern, indem wir die Bilder entweder horizontal oder vertikal spiegeln. Wir können sie rotieren, hinein oder hinauszoomen, sie zuschneiden oder die Farben ändern. Das alles sind übliche Augmentationstechniken.

- Horizontale Spiegelung
- Vertikale Spiegelung

- Rotation
- Vergrößern
- Verkleinern
- Zuschneiden
- Farbvariationen

17.1 WARUM BENUTZT MAN DATA AUGMENTATION

Einfach, deshalb, um dem Trainingssatz mehr Daten hinzuzufügen. Zum Beispiel haben wir einen relativ kleinen Trainingsdatensatz und es ist schwer, an mehr Daten zu gelange. Dann können wir einfach mithilfe von Data Augmentation neue Daten erstellen.

17.1.1 REDUZIERUNG VON OVERFITTING

Darüber hinaus wird Data Augmentation verwendet, um Overfitting zu reduzieren. Das wurde auch schon in Kapitel 12.2.2 erwähnt.

Wenn also das Netz Overfitted und wir mehr Daten hinzufügen sollen/müssen, bietet uns Data Augmentation eine einfache Möglichkeit dazu, wenn wir keinen Zugriff auf weitere Daten haben.

Allerdings sollte man nicht einfach alle Augmentationsarten anwenden, sondern diese mit Verstand wählen, da es zum Beispiel nicht immer sinnvoll ist, ein Bild Vertikal zu Spiegeln

18. ONE-HOT ENCODING

In dem Kapitel 14.1.1 wurde schon erwähnt, dass Labels für Bilder codiert werden. Sie werden als one-hot encoded Vektoren kodiert.

18.1 LABELS

Wir wissen, dass bei überwachtem Lernen gelabelter Input durch das Netz gegeben wird.

Wenn also zum Beispiel unser Netz ein Bildklassifizierer ist, dann werden wir gelabelte Bilder an unser Netz geben. Wenn wir das tun, interpretiert das Netz diese Labels normalerweise nicht als Wörter wie „Katze“ oder „Hund“. Darüber hinaus sind die Vorhersagen unseres Netzes auch keine Wörter wie „Katze“ oder „Hund“. Stattdessen werden unsere Labels meistens kodiert, so dass sie die Form eines Integers (Ganzzahl) oder eines Vektors von Integers annehmen.

18.2 HOT AND COLD VALUES

Eine verbreitete Form von Enkodierung, die für die Kodierung von kategorischen Daten mit Zahlwerten verwendet wird, heißt one-hot encoding.

One-hot encodings wandeln unserer kategorischen Labels in Vektoren (also quasi in Arrays) aus 0en und 1en um. Die Länger dieser Vektoren ist die Anzahl der Kategorien, die unser Netz klassifizieren wird.

Wert	Interpretation
0	Cold
1	Hot

18.2.1 VEKTOREN AUS 0EN UND 1EN

Wenn wir klassifizieren würden, ob auf einem Bild ein Hund oder eine Katze ist, dann währen unsere one-hot encoded Vektoren jeweils von der Länge 2. Wenn wir noch eine Kategorie, zum Beispiel eine Schildkröte, hinzufügen, dann würden wir Klassifizieren, ob auf einem Bild entweder ein Hund, eine Katze oder eine Schildkröte ist. Die dazugehörigen one-hot encoded Vektoren hätten jeweils eine Länge von 3, weil wir ja jetzt 3 Kategorien haben.

Mittlerweile wissen wir also schon, dass Labels als Vektoren kodiert werden. Wir wissen, dass die Länge jeder dieser Vektoren gleich der Anzahl der Kategorien, nach denen das Netz klassifiziert, ist und dass die Vektoren aus 0en und 1en bestehen. Auf den letzten Punkt werde ich jetzt noch etwas eingehen.

18.3 ONE-HOT ENCODINGS FÜR MEHRERE KATEGORIEN

Bleiben wir bei dem Beispiel, bei dem wir nach Katze, Hund oder Schildkröte klassifizieren werden. Da jeder der entsprechenden Vektoren für diese Kategorien eine Länge von 3 hat, können wir uns jedes Element/jeder Index des Vektors als eine der 3 Kategorien vorstellen. Sagen wir, dass bei dem Beispiel „Katze“ für das erste, „Hund“ für das zweite, und „Schildkröte“ für das dritte und letzte Element steht.

Da jede dieser Kategorien ihren eigenen Platz in den Vektoren hat, können wir nun die Idee hinter dem Namen one-hot besprechen.

Bei jedem one-hot encoded Vektor ist jedes Element eine Null außer das Element, das der tatsächlichen Kategorie der Eingabe entspricht. Dieses Element wird als „hot“ bezeichnet.

Eines der Elemente in dem Vektor ist hot

Für unser Beispiel würde das so aussehen:

Label	1. Element	2. Element	3. Element
Katze	1	0	0
Hund	0	1	0
Schildkröte	0	0	1

Für eine Katze ist das erste Element eine 1 und die anderen Elemente 0en. Das ist, wie schon erklärt, weil jedes der Elemente in einem Vektor 0 ist außer das Element, dass zu der aktuellen Kategorie gehört.

18.3.1 EIN VEKTOR FÜR JEDE KATEGORIE

Wir können sehen, dass jedes Mal, wenn das Netz einen Input erhält, welcher eine Katze ist, es ihn nicht als das Wort „Katze“ interpretiert, sondern als diesen Vektor: [1,0,0].
Für Bilder, die als „Hund“ gelabelt sind [0,1,0]
und für „Schildkröte“ als [0,0,1]

Nur zur Verdeutlichung: Sagen wir, wir fügen dem eine weitere Kategorie hinzu, nämlich „Lama“. Nun haben wir 4 Kategorien und so wird jeder one-hot encoded Vektor in diesem Netz jetzt eine Länge von 4 haben.
Die Vektoren sehen jetzt so aus:

Label	Vektor
Katze	[1,0,0,0]
Hund	[0,1,0,0]
Schildkröte	[0,0,1,0]

Lama	[0,0,0,1]

Wir sehen, dass sich die Position der 1en für Katze, Hund und Schildkröte nicht verändert hat. Dadurch, dass wir die Kategorie Lama hinzugefügt haben, ist bei Katze, Hund und Schildkröte einfach eine 0 am Ende hinzugefügt worden. Das letzte Element in dem Vektor entspricht also der Kategorie „Lama“.

Beachte, dass wir die zu den Labeln zugehörige Position gerade willkürlich festgelegt haben. Natürlich kann dies eine andere Reihenfolge sein. Dies hängt einfach von dem Code oder der Library ab, welche one-hot encoded.

19. CONVOLUTIONAL NEURAL NETWORKS (CNNS)

Ein Convolutional Neural Network, auch bekannt als CNN oder ConvNet, ist ein Artificial Neural Network, das bisher am häufigsten zur Analyse von Bildern verwendet wurde.
Auch wenn die Bildanalyse der populärste Einsatzzweck von CNNs ist, kann man sie auch für Datenanalysen oder Klassifizierungsaufgaben einsetzen.

19.1 WAS IST EIN CNN?

Im Allgemeinen können wir uns ein CNN als ein ANN vorstellen, dass einen Art Spezialisierung hat um Muster zu erkennen. Diese Mustererkennung macht CNNs so nützlich für die Bildanalyse.

Wenn CNN eigentlich nur ein NN ist, was ist der Unterschied zu einem Standard Multilayer Perceptron (= MLP)?

CNNs haben hidden Layers namens convolutional layers. Diese Layers mach das CNN zu einem CNN.

CNNs haben Layers namens convolutional Layers

Ein CNN kann, und tut es für gewöhnlich auch, andere, nicht-convolutional Layers haben. Die Basis eines CNN sind aber die convolutional Layers

19.1.1 CONVOLUTIONAL LAYERS

Wie jeder andere Layer erhält der convolutional Layer einen Input, wandelt ihn in einer gewissen Art um und gibt als Output den transformierten (bzw. umgewandelten) Input an den nächsten Layer. Die Inputs eines convolutional Layer heißen Input channels und die Outputs Output channels.

Die, vorhin schon erwähnte, Transformation bei einem convolutional Layer heißt convolutional operation. Dies ist de Begriff, der von der Deep Learning Community verwendet wird. Aus mathematischer Sicht sind es cross-correlations (= Kreuzkorrelation)

19.2 FILTER UND COVOLUTION OPERATIONS

Wie vorhin schon erwähnt, können CNNs Muster in Bildern erkennen.

Bei jedem convolutional Layer muss die Anzahl der „Filter“ spezifiziert werden. Diese Filter sind das, was die Muster erkennt

19.2.1 MUSTER

Was ist ein „Muster“ genau und was heißt es, dass diese Filter Muster erkennen können?

Denk einfach daran, was in einem einzelnen Bild vor sich geht. Mehrere Kanten, Formen, Texturen, Objekte, etc. Das ist es, was mit Muster gemeint ist.

- Kanten

- Formen
- Texturen
- Kurven
- Objekte
- Farben

Eine Art von Muster, das ein Filter in einem Bilde erkennen kann, sind Kanten, dieser Filter wird also Edge detector genannt.

Abgesehen von Kanten können einige Filter auch Ecken erkennen. Einige Kreise und andere Rechtecke. Diese einfachen, geometrischen Filter sind das, was wir als Beginn eines CNN bezeichnen würden.

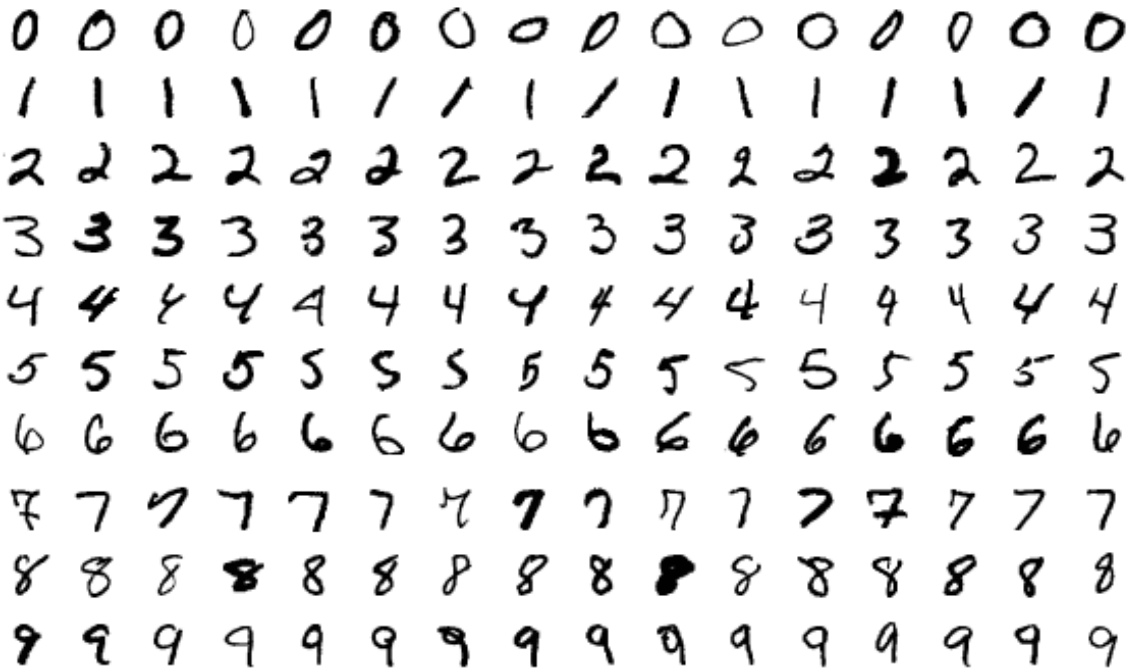
Je tiefer das Netzwerk ist, desto ausgefeilter werden die Filter. In späteren Layers können Filter statt einfachen Objekten komplexe Formen wie Ohren, Augen, Haare, Federn, Schuppen… erkennen.

In noch tieferen Layers können die Filter noch komplexere Muster erkennen wie ganze Hunde, Katzen, Schildkröten und Vögel.

Um zu verstehen, was in so einem convolutional Layer eigentlich passiert, lass uns ein Beispiel ansehen

19.2.2 FILTER (PATTERN DETECTORS)

Angenommen, wir haben ein CNN das Bilder von handgeschriebenen Zahlen akzeptiert (zum Beispiel vom MNIST Datensatz). Und unser Netzwerk klassifiziert, was auf diesem Bild ist (0,1,2,3,4,5,6,7,8,9).



Nehmen wir jetzt an, dass unser erster hidden Layer ein convolutional Layer ist. Wie vorhin gesagt, müssen wir auch angeben, wie viele Filter der Layer haben soll.

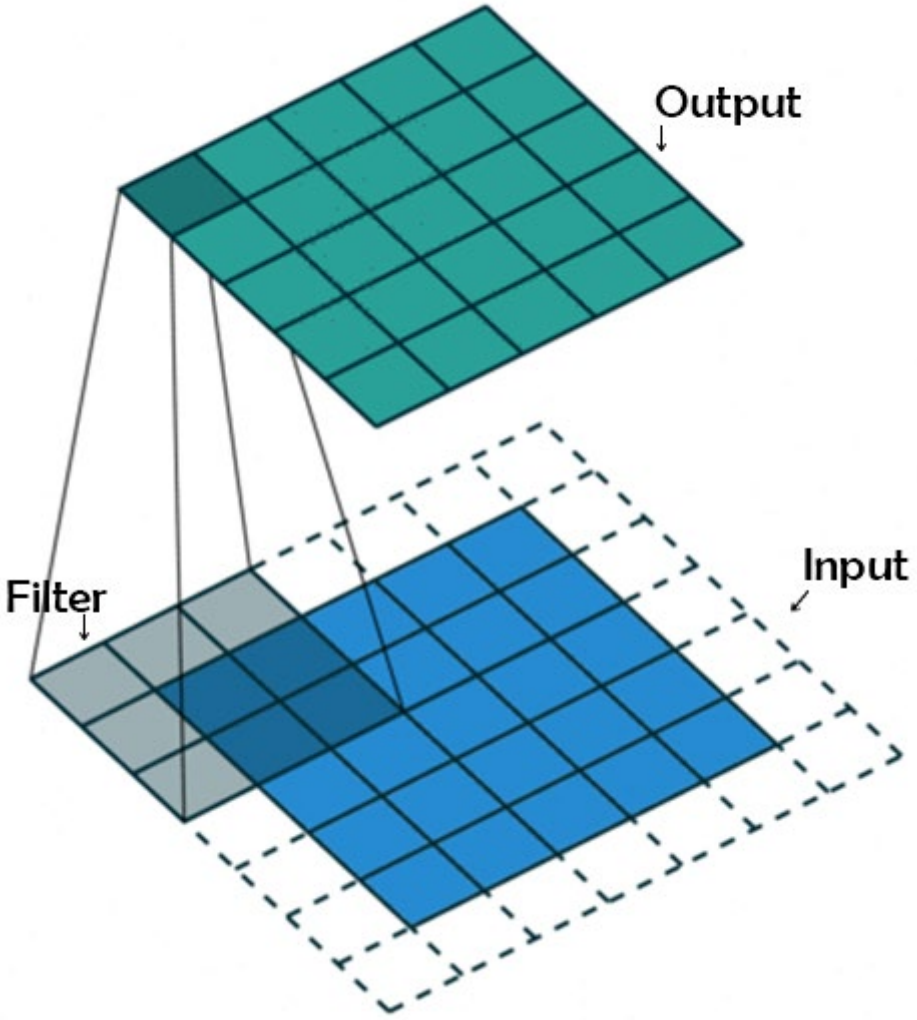
Die Anzahl der Filter bestimmt die Anzahl der Output channels

Ein Filter ist technisch gesehen eine relativ kleine Matrix (Tensor), für die wir die Anzahl der Zeilen und Spalten bestimmen. Die Werte der Matrix werden zufällig gewählt.

Für diesen ersten convolutional Layer bestimmen wir eine Filtergröße von 3 x 3.

19.2.3 CONVOLUTIONAL LAYER

So arbeitet ein Convolutional Layer:



Dieses Bild zeigt ein CNN ohne Zahlen. Wir haben einen blauen Input channel auf der Unterseite, einen, auf der Unterseite schattierten, convolutional Filter und den Output channel

Für jede Position auf dem blauen Input channel führt der 3 x 3 Filter eine Berechnung durch, die den schattierten Teil des blauen Eingangskanals auf den entsprechenden schattierten Teil des grünen Ausgangskanals abbildet.

Dieses convolutional Layer empfängt ein Eingangssignal, und der Filter wird über jeden 3x3 Pixelsatz des Eingangs gelegt, bis er das Gesamte Bild abgedeckt hat. Zur besseren Vorstellung hier ein GIF davon.

19.2.4 CONVOLUTION OPERATION

Dieses Schieben des Filters heißt convolving (=konvolierung). Ein Filter konvoliert also über jeden 3 x 3 Block des Inputs.

Der blaue Input channel ist eine Matrixdarstellung eines Bildes aus dem MNIST-Datensatz. Die Werte dieser Matrix sind die individuellen Pixel des Bildes. Diese Bilder sind Graustufenbilder, also haben wir nur einen Input channel.

- Graustufenbilder haben einen Farbkanal
- RGB-Bilder haben 3 Farbkanäle

Dieser Input wird an einen convolutional Layer gegeben.

Wie gerade besprochen hat der erste convolutional Layer nur einen Filter und dieser Filter konvoliert über jeden 3 x 3 Pixelblock des Inputs. Wenn der Filter auf dem ersten 3 x 3 Pixelblock landet, wird das dot product (=Skalarprodukt) des Filters mit dem 3 x 3 Pixelblock berechnet und gespeichert. Dies geschieht für jeden Pixelblock den der Filter konvoliert.

Zum Beispiel nehmen wir das Skalarprodukt des Filters mit dem ersten 3 x 3 Pixelblock und speichern dieses Ergebnis im Output channel. Dann bewegt sich der Filter zum nächsten 3 × 3-Block, berechnet das Skalarprodukt und speichert den Wert als nächstes Pixel im Output channel.

Nachdem dieser Filter den gesamten Input convolved hat, bleibt uns eine neue Darstellung des Inputs, die jetzt um Output channel gespeichert ist. Dieser Output channel wird als Feature Map bezeichnet.

Dieser grüne Output channel wird der Input channel des nächsten Layers. Dieser Prozess, den wir gerade besprochen haben, passiert nun auch mit diesem neuen Input channel mit dem Filter des nächsten Layers.

Das war eine sehr vereinfachte Illustration und Erklärung, aber wie vorhin erwähnt kann man sich die Filter wie Pattern detectors vorstellen.

19.2.4.1 DOT PRODUCT/SKALARES PRODUKT

Vorhin wurde der Begriff Skalares Produkt schon verwendet, um die Operation oben zu erklären. Technisch gesehen summieren wir die Produkte jedes Elementpaares.

Wenn wir also zwei 3x3 Matrizen haben, sieht das folgendermaßen aus:

$$A=\begin{matrix} a1,1 & a1,2 & a1,3 \\ a2,1 & a2,2 & a2,3 \\ a3,1 & a3,2 & a3,3 \end{matrix}$$

$$B=\begin{matrix} b1,1 & b1,2 & b1,3 \\ b2,1 & b2,2 & b2,3 \\ b3,1 & b3,2 & b3,3 \end{matrix}$$

Dann summieren wir die Summen folgendermaßen auf:

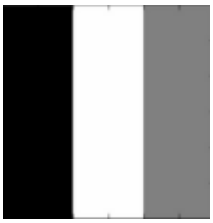
a1,1*b1,1+a1,2*b1,2+⋯+a3,3*b3,3

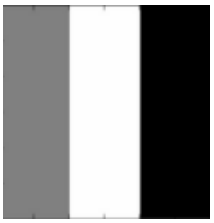
Technisch gesehen ist diese Operation also die Summe der elementweisen Produkte. Diese Operation kann auch unter dem Namen Frobenius inner product oder summation of the Hadamard product.

19.3 INPUT UND OUTPUT CHANNELS

Angenommen dieses Graustufenbild (ein Farbkanal) einer Sieben vom MNIST Datensatz ist unser Input:

Wir haben vier 3 x 3 Filter für unseren ersten convolutional Layer und diese Filter sind mit den Werten, die du unten siehst, gefüllt. Jeder dieser Werte können visuell dargestellt werden. -1 steht für schwarz, 0 steht für



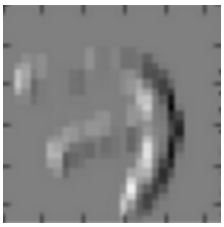


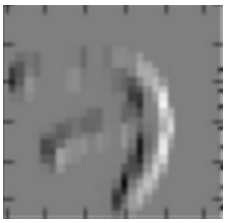
Grau und 1 für weiß

$$\begin{matrix} -1 & -1 & -1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{matrix} \qquad \begin{matrix} -1 & 1 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & 0 \end{matrix} \qquad \begin{matrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ -1 & -1 & -1 \end{matrix} \qquad \begin{matrix} 0 & 1 & -1 \\ 0 & 1 & -1 \\ 0 & 1 & -1 \end{matrix}$$

Wenn wir unser originales Bild der Sieben konvolieren, ist das in etwas das, was für jeden Filter als Ergebnis kommt:

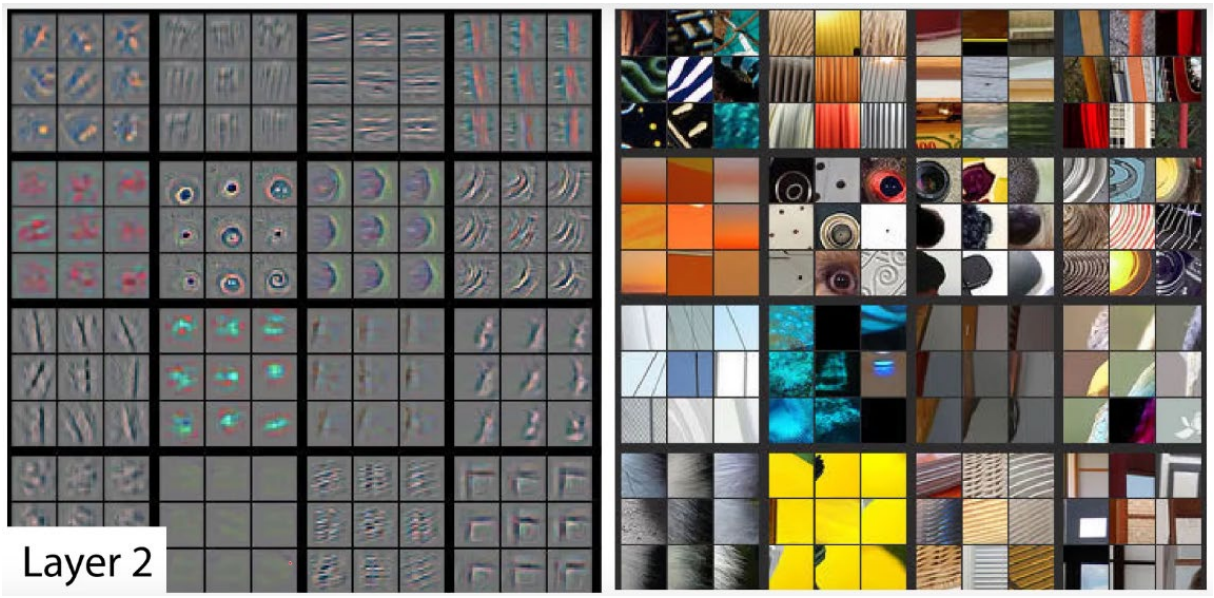
Wir sehen, dass jeder der vier Filter Kanten erkennt. In den Output channels können die hellsten Pixel als das interpretiert werden, was der Filter erkannt hat.





In dem ersten können wir sehen, dass der Filter die oberen horizontalen Kanten der Sieben erkannt hat, der zweiten erkannte die linken vertikalen Kanten, der dritte die unteren horizontalen Kanten und der vierte die rechten vertikalen Kanten.

Diese Filter sind wie vorhin erwähnt sehr einfach und erkennen nur Kanten. Solche Filter sehen wir am Anfang eines CNN. Komplexere Filter findet man in tieferen Layers eines Netzes



Wir sehen die Formen, welche die Filter auf der linken Seite aus den Bildern auf der rechten Seite erkennen konnten. Wir sehen Kreise, Kurven und Ecken. Wenn wir in noch tiefere Layers gehen, können die Filter noch Komplexere Muster erkennen wie ein Hundegesicht.

Das erstaunliche dabei ist, dass diese pattern detectors automatisch vom Netz erzeugt werden. Die Filterwerte beginnen zufällig und ändern sich, wenn das Netz etwas lernt. Die Mustererkennungsfähigkeit der Filter entsteht automatisch.

Pattern detectors entstehen, wenn das Netz lernt.

20. ZERO PADDING IN CNNS

In diesem Kapitel wird erklärt, was die Motivation hinter zero padding ist, was zero padding überhaupt ist, auf welche Probleme wir stoßen können, wenn wir zero padding nicht verwenden und wie wir zero padding mit Keras implementieren.

20.1 PROBLEMATIK DER CHANNEL DIMENSIONEN

Im Kapitel 19, welches von CNNs handelte, haben wir gesehen, dass jeder convolutional Layer eine Anzahl von Filtern hat, die wir definieren, und wir ebenfalls die Dimensionen dieser Filter definieren müssen. Es wurde auch gezeigt, wie diese Filter Bildinput konvolieren.

Wenn ein Filter einen gegebenen Input konvolviert, dann gibt er uns einen Output channel zurück. Dieser Output channel ist eine Matrix von Pixeln mit Werten, die während der convolutions auf dem Input channel berechnet wurden. Wenn das passiert werden die Dimensionen von unserem Bild reduziert.

Bilddimensionen werden reduziert.

Lass uns das anhand des gleichen Bildes einer Sieben überprüfen, das wir in Kapitel 19 verwendet haben. Wir haben hierfür eine 28x28 Matrix von Pixelwerten von der 7 des MNIST Datensatzes

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.4	0.3	0.5	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.4	0.5	0.9	0.9	0.9	0.9	0.9	0.9	1.0	1.0	1.0	1.0	1.0	0.9	0.7	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.7	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.9	1.0	0.8	0.8	0.8	0.8	0.8	0.5	0.2	0.2	0.2	0.2	0.2	0.5	0.9	1.0	1.0	0.7	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.1	0.3	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.8	1.0	1.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.6	1.0	1.0	1.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.5	0.9	0.9	0.9	1.0	1.0	1.0	1.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.5	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.7	1.0	1.0	1.0	1.0	0.9	0.8	0.8	0.3	0.3	0.8	1.0	1.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.9	1.0	0.9	0.9	0.5	0.3	0.1	0.0	0.0	0.0	0.0	0.8	1.0	0.9	0.2	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.7	1.0	0.7	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.9	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.0	1.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	0.9	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8	1.0	1.0	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	1.0	0.9	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	1.0	0.9	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8	1.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

So sieht das Ganze im originalen, 28 x 28 Format aus. Nun wenden wir diesen 3 x 3 Filter an:

0.98	0.28	0.94
0.71	0.05	0.56
0.60	0.33	0.85

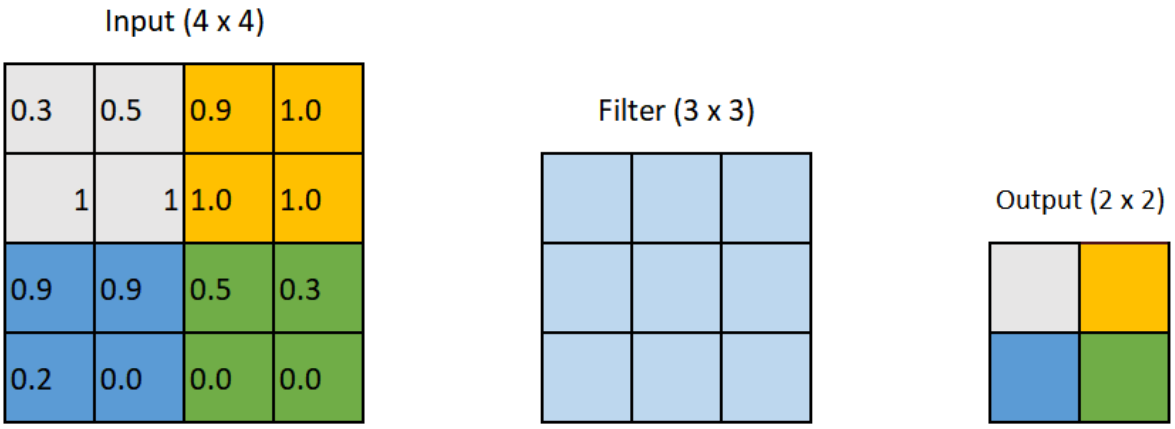
Daraus ergibt sich dieser Output Channel:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	0	0	0	0	0	0	0	0
1	1	2	3	3	3	3	3	3	3	4	4	4	4	4	3	2	2	1	0	0	0	0	0	0	0
1	2	3	4	4	5	5	5	4	4	4	4	4	4	5	5	4	3	2	1	0	0	0	0	0	0
1	2	3	4	3	3	3	3	3	3	3	3	3	3	3	4	4	4	3	1	0	0	0	0	0	0
1	1	2	2	2	2	2	2	1	1	1	1	1	1	1	2	4	5	4	2	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	4	5	3	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	4	5	3	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	2	2	2	4	4	5	4	3	1	0	0	0	0
0	0	0	0	0	0	0	0	1	1	2	2	3	3	3	4	5	5	5	4	3	1	0	0	0	0
0	0	0	0	0	0	1	1	2	3	3	4	4	5	4	4	5	5	5	5	3	2	0	0	0	0
0	0	0	0	0	1	2	3	3	4	4	4	4	3	3	3	4	4	5	4	3	1	0	0	0	0
0	0	0	0	1	2	3	4	4	3	3	3	2	2	1	1	2	3	5	3	2	1	0	0	0	0
0	0	0	0	1	2	3	3	3	2	1	1	0	0	0	0	2	3	5	3	2	0	0	0	0	0
0	0	0	0	1	1	2	2	1	0	0	0	0	0	0	1	3	4	4	3	2	0	0	0	0	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	2	3	4	4	2	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	4	4	3	2	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	3	5	3	2	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	3	4	3	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	3	2	2	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2	1	1	0	0	0	0	0	0	0	0

Wir können sehen, dass der Output nicht die gleiche Größe wie der originale Input hat. Die Output Size ist 26 x 26 während unser originaler Input channel 28 x 28 war. Der Output channel wurde also von 28 x 28 auf 26 x 26 geschrumpft. Warum ist das so?

Bei unserem 28 x 28 Bild passt der 3 x 3 nur in 26 x 26 möglichen Positionen, nicht in alle 28 x 28. Dadurch erhalten wir den resultierenden 26 x 26 Output.

Um dies leichter zu visualisieren, schauen wir uns ein kleineres Beispiel an. Hier haben wir einen 4 x 4 Input und einen 3 x 3 Filter. Schauen wir uns an, wie oft wir den Input mit diesen Filter convolven können und wie die resultierende Output Size sein wird.



Das bedeutet, wenn dieser 3 x 3 Filter diese 4 x 4 Filter konvoliert hat, bekommen wir eine Output Size von 2 x 2.

Wir sehen also wie bei der Sieben vorhin, dass unsere Output dimensions kleiner sind als der Input dimensions.

Wir können schon im Voraus wissen, um wie viel unsere Dimensionen weniger werden. Wenn unser Bild die Größe n x n hat und wir es mit einem f x f Filter konvolieren, dann ist der resultierende Output:

(n - f + 1) x (n-f + 1)

Wenn wir dies an unserem Beispiel anwenden kommen wir auf

(n-f+1)

=(4-3+1)

=2

➔ 2 x 2 Output

Tatsächlich bestätigt es unseren obigen 2 x 2 Ausgangskanal. Das funktioniert auch mit dem obigen Beispiel der Sieben.

20.2 PROBLEME DES DIMENSIONREDUZIERENS

Stell dir noch einmal den Output der Sieben aus dem vorherigen Kapitel vor. Es schaut nicht nach einer großen Sache aus, dass dieser Output ein wenig kleiner ist als der Input, oder?

Wir verloren nicht viel Daten, weil die wichtigen Teile dieses Inputs hier in der Mitte waren. Es würde aber sehr wohl einen großen Unterschied machen, wenn wir bedeutende Daten bei den Ecken des Input-Bildes haben.

Außerdem haben wir dieses Bild nur mit einem Filter konvoliert. Was passiert, wenn dieser Input durchs Netz geht und mit deutlich mehr Filtern konvoliert wird?

Wenn wir zum Beispiel mit einem 4 x 4 Bild starten, dann würde der Output nach ein oder zwei Convolutional Layers bedeutungslos klein werden. Ein weiteres Problem ist, dass wir (wahrscheinlich) wichtige Daten verlieren, weil wir einfach die Informationen an den Ecken wegschmeißen.

Dadurch wird das Bild kleiner und kleiner und enthält schließlich keine wertvollen Informationen mehr. Das ist ein Problem!

Dagegen können wir mit Zero padding vorgehen

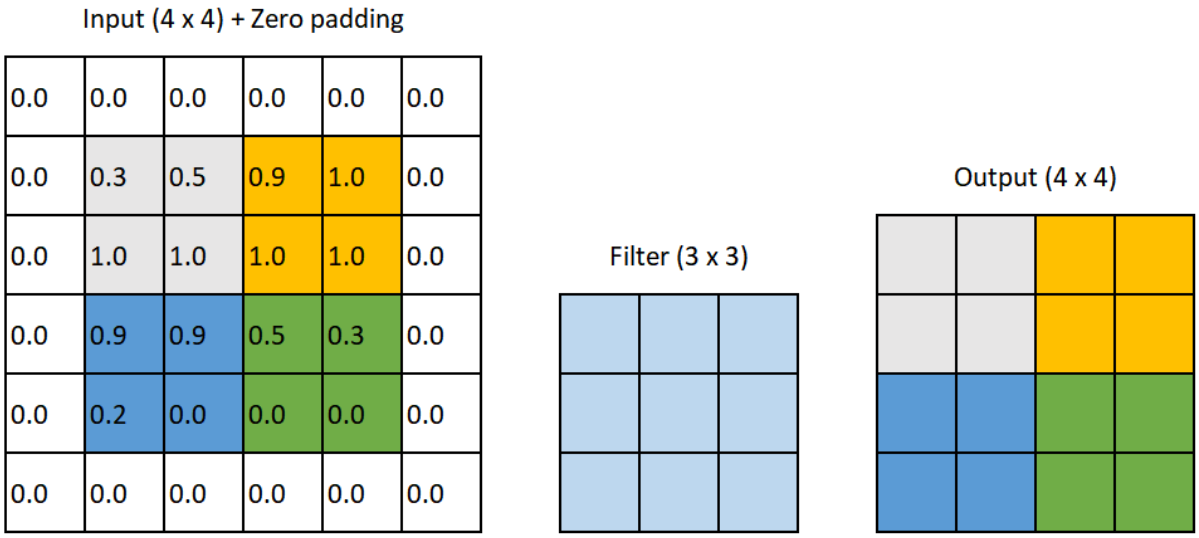
20.3 ZERO PADDING

Zero padding ist eine Technik, die uns erlaubt die originale Input Size zu behalten. Mit jedem Convolutional Layer können wir festlegen, (genau wie bei der Anzahl der Filter) ob wir Padding verwenden oder nicht.

20.3.1 WAS IST ZERO PADDING?

Wir wissen nun, wogegen wir Zero padding einsetzen, aber was ist es eigentlich?

Beim Zero padding werden am Rand der Input Matrix lauter 0en eingetragen. Dies fügt eine Art Padding (vgl.: HTML/CSS) von Nullen um die Außenseiten des Bildes hinzu. Dies ist der Grund warum diese Technik „Zero padding“ heißt. Sehen wir uns unser kleineres Beispiel von vorher mit Zero padding an.



Wir sehen, dass unser Output wirklich 4 x 4 hat. Manchmal müssen auch mehr als eine Schicht 0en ergänzt werden. Ob wir nun eine einfache, zweifache oder dreifache Schicht an 0en hinzufügen müssen hängt von der Input Size und von der Filter Size ab.

Das Gute daran ist, dass die meisten neuronalen Netz-APIs die Größe der Schichten automatisch berechnen. Alles was wir tun müssen, ist zu spezifizieren, ob wir Zero padding verwenden wollen oder nicht.

20.3.2 VALID UND SAME PADDING

Es gibt zwei Arten von padding. Eines trägt den Namen valid und bedeutet einfach so viel wie kein padding. Wenn wir valid padding spezifizieren, dann wird unser Layer kein Zero padding vornehmen und unser Input Size bleibt nicht erhalten.

Die andere Art des Paddings heißt same und sagt aus, dass die Output Size gleich der Input Size sein soll.

Padding Typ	Beschreibung	Auswirkung
Valid	Kein Padding	Dimensionen reduzieren sich

Same	0en um die Ecken	Dimensionen bleiben gleich

20.4 PADDING MIT KERAS

Wir starten mit einigen Imports:

```
import tensorflow.keras

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Activation

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.layers import Conv2D
```

Jetzt erstellen wir völlig willkürlich ein CNN

```
model = Sequential([

   Dense(16, input_shape=(20,20,3), activation='relu'),

   Conv2D(32, (3,3), activation='relu', padding='valid'),

   Conv2D(64, (5,5), activation='relu', padding='valid'),

   Conv2D(128,(7,7), activation='relu', padding='valid'),

   Flatten(),

   Dense(2, activation='softmax')

])
```

Dieses hat einen Dense Layer, danach 3 Convolutional Layers und dann ein Dense Output Layer.

Wir haben spezifiziert, dass die Input Size der Bilder 20x20 ist und unser erster Convolutional Layer eine 3x3 Filtergröße hat. (Dies ist der zweite Parameter -> (3,3), welcher auch als Kernel Size bekannt ist.) Der zweite Convolutional Layer hat eine 5x5 Filtergröße und der dritte eine 7x7 Filtergröße.

Bei diesem Modell spezifizieren wir das padding als 'valid', welches, wie schon erwähnt, kein padding bedeutet.

Das ist der Default Parameter für Convolutional Layers in Keras, also wenn wir für einen Layer kein Padding wollen, brauchen wir dies eigentlich nicht extra spezifizieren.

Da dies kein Padding verwendet, wird die Dimension dieser Layer immer kleiner. Lass uns das mit der summary Funktion nachprüfen

```
model.summary()
```

Model: "sequential"

_____

Layer (type) Output Shape Param #

==========

dense (Dense) (None, 20, 20, 16) 64

_____

conv2d (Conv2D) (None, 18, 18, 32) 4640

_____

conv2d_1 (Conv2D) (None, 14, 14, 64) 51264

_____

conv2d_2 (Conv2D) (None, 8, 8, 128) 401536

_____

flatten (Flatten) (None, 8192) 0

_____

dense_1 (Dense) (None, 2) 16386

==========

Total params: 473,890

Trainable params: 473,890

Non-trainable params: 0

_____

Wir können die Output Größe jedes Layers in der zweiten Spalte sehen. Die ersten zwei Zahlen spezifizieren die Dimensionen der Output höhe und breite. Unser erster Layer hat also eine Input Size von 20x20

Der Output des ersten Convolutional Layer hat nur mehr 18x18 und der nächste Layer verkleinert die Dimensionen zu 14x14. Beim letzten Convolutional Layer sind sie nur mehr 8x8 groß.

Als Kontrast dazu können wir uns ein Netz ansehen, dass Zero-padding verwendet:

```
model = Sequential([
    Dense(16, input_shape=(20,20,3), activation='relu'),
    Conv2D(32, (3,3), activation='relu', padding='same'),
    Conv2D(64, (5,5), activation='relu', padding='same'),
    Conv2D(128, (7,7), activation='relu', padding='same'),
```

```
    Flatten(),
    Dense(2, activation='softmax')
])
```

Dieses Netz ist bis auf "padding='same'" ein exaktes Duplikat vorherigen Netzes. Das bedeutet, dass Zero-Padding verwendet wird. Die summary des Netzes sieht nun so aus:

```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 20, 20, 32)        4640
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 20, 20, 64)        51264
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 20, 20, 128)       401536
_________________________________________________________________
flatten_2 (Flatten)          (None, 51200)             0
_________________________________________________________________
dense_6 (Dense)              (None, 2)                 102402
=================================================================
Total params: 559,906
Trainable params: 559,906
Non-trainable params: 0
_________________________________________________________________
```

Wir können ebenfalls als Input Size 20x20 erkennen, allerdings behalten die Input Dimensions ihre Größe.

21. MAX POOLING

In diesem Kapitel kläre ich, was May Pooling ist, wie es berechnet wird, warum man es verwendet und wie man es mit Keras implementiert.

21.1 INTRODUCTION

May Pooling ist eine Art von Operation, die die Dimensionalität von Bildern reduziert, indem es die Pixel des Outputs des vorigen Convolutional reduziert.

Jetzt schauen wir uns noch einige Beispiele an, um zu sehen, was Max Pooling genau macht und dann klären wir noch, warum es verwendet werden sollte.

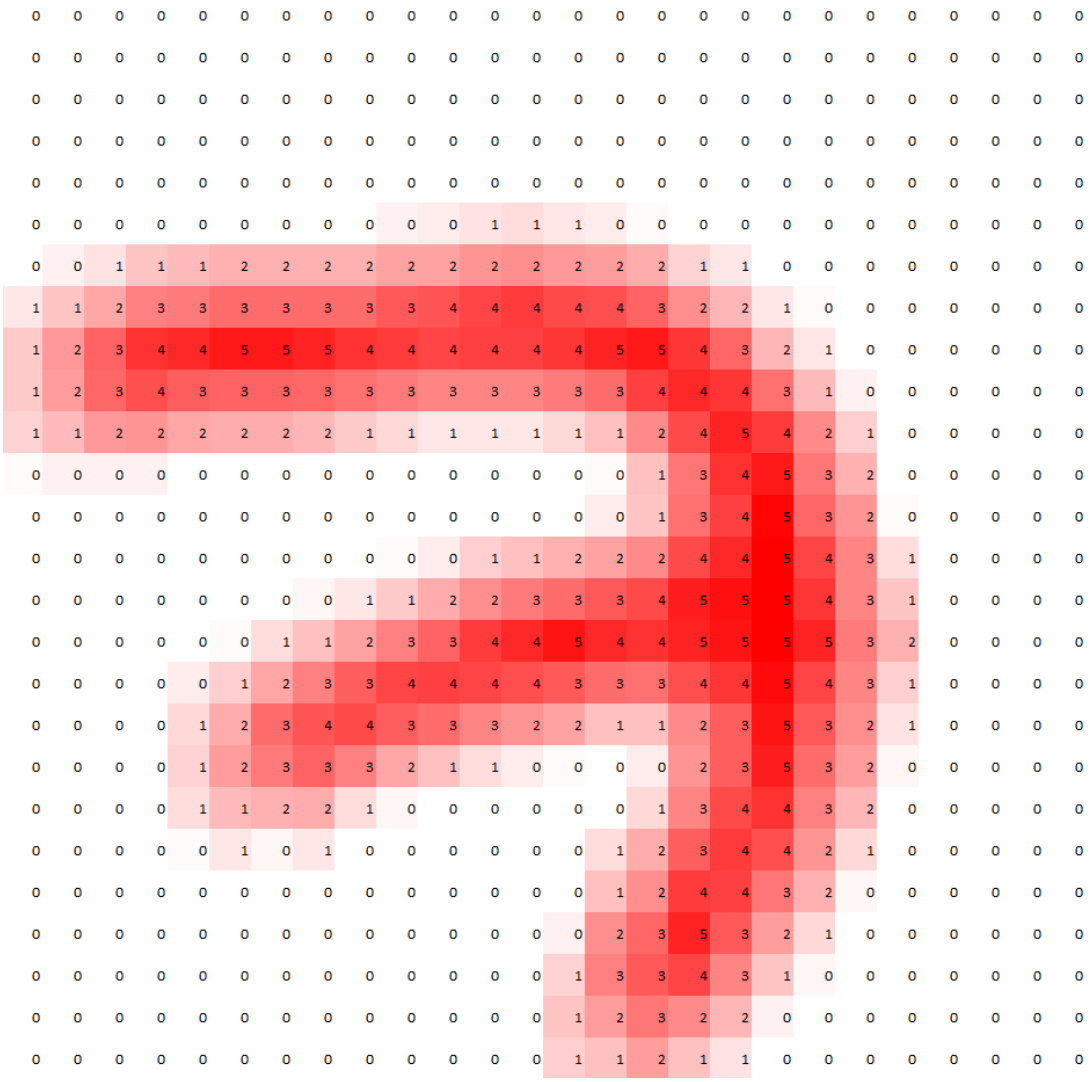
21.2 BEISPIEL ANHAND DES MNIST DATENSATZES

Wir haben schon gelernt, dass jeder Convolutional Layer eine Anzahl von Filtern hat. Diese Filter haben eine spezifizierte Dimension.

Nachdem ein Filter einen Input konvoliert hat, bekommen wir einen Output zurück. Dieser Output ist eine Matrix aus Pixeln, deren Werte während der Konvolierung unseres Bildes berechnet werden. Dies wird als Output Channels bezeichnet.

Wir benutzen wieder das gleiche Bild der Sieben, welches auch in den Kapiteln davor verwendet wurde.

Wir benutzten einen 3x3 Filter, um diesen Output zu erzeugen:



Wie vorhin erwähnt wird Max Pooling nach einem Convolutional Layer. Dieser 26 x 26 Output ist also der Input in die Max Pooling Operation.

Nach dem Max Pooling ist das unser Output:

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0.3	0.6	0.7	0.4	0	0	0	0	0
1.2	2.6	3	3	3.4	3.8	4	3.6	2.3	0.5	0	0	0
2.1	4.2	4.7	4.7	4.2	3.9	4.1	4.7	4.4	2.9	0.3	0	0
1.4	2.2	1.8	1.7	1.1	0.5	0.8	2.4	4.5	4.7	1.6	0	0
0	0	0	0	0.1	1	1.6	2.4	4.4	5.2	2.5	0	0
0	0	0.1	1.3	2.6	4	4.8	4.4	4.9	5.2	2.7	0	0
0	0	1.7	3.5	3.8	3.9	3.6	3	4.1	5	2.5	0	0
0	0	2	3.2	2.6	1.3	0.4	0.8	3.7	4.6	2	0	0
0	0	0.5	0.5	0	0	0	2.3	4	3.6	0.8	0	0
0	0	0	0	0	0	0.9	3.4	4.5	2	0	0	0
0	0	0	0	0	0	1.2	2.8	2.4	0.3	0	0	0

So funktioniert Max Pooling. Wir definieren eine n x n Region als dazugehörigen Filter der May Pooling Operation. Wir definieren eine Schrittlänge (= Stride), welche aussagt, über wie viele Pixel unser Filter iterieren soll, wenn es über das Bild gleitet.

Stride definiert, über wie viele Werte der Filter gleitet

Wir nehmen von dem Convolutional Output die erste 2 x 2 Region und berechnen den Maximalen Wert dieses 2x2 Blockes. Dieser Wert wird in dem Output Channel gespeichert. Der Filter bewegt sich so weit weiter, wie wir den Stride definiert haben. Wir benutzen für dieses Beispiel 2, also bewegen wir uns um 2 Werte in der Matrix weiter. Danach geschieht wieder das Gleiche. Der Maximale Wert wird berechnet, in den Output channel gespeichert und der Filter bewegt sich wieder um 2 weiter.

Wenn wir das rechte Ende erreicht haben, gleitet unser Filter um 2 nach unten. (Weil die Stridegröße ja 2 ist). Danach wird der oben geschilderte Prozess für diese Reihe wiederholt. Dies wird wiederrum so lange wiederholt, bis alle Felder des Bildes abgearbeitet wurden.

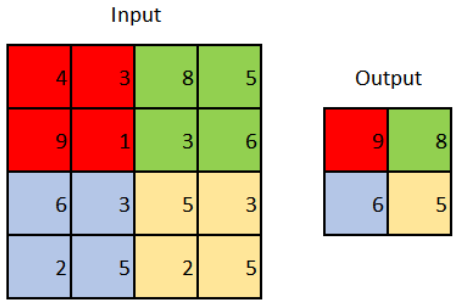
Wir können uns diese 2 x 2 Blöcke als Pools von Nummern vorstellen. Durch diesen Umstand und dadurch, dass wir den maximalen Wert eines Pools berechnen, ergibt sich der Name Max Pooling.

In dem oberen Beispiel ist unser convolutional Output 26 x 26 groß. Während dem May Pooling wird unsere Größe um den Faktor 2 reduziert und der Output hat 13 x 13.

Nur um sicherzugehen, dass diese Operation verstanden wurde, zeige ich sie noch schnell anhand eines leichteren Beispiels

21.3 KLEINERES BEISPIEL

Angenommen wir haben folgendes:



Wir haben einen 4 x 4 Input wenden Max Pooling mit einer 2 x 2 Filtergröße und einem Stride von 2 an.

Wir können sehen, dass in dem Roten Bereich der maximale Wert 9 ist, demnach ist der Output Wert des roten Bereichs auch 9. Als nächstes gleiten wir über 2 Pixel und behandeln somit den grünen Bereich. Hier ist das Maximum 8, also wird 8 in den Output channel eingetragen. Danach gleitet der Filter nach unten in die gelbe Region. Derselbe Prozess geschieht nun für die gelbe und für die blaue Region.

Nun wissen wir, wie Max Pooling funktioniert, doch wieso soll es verwendet werden?

21.4 WARUM WIRD MAX POOLING VERWENDET?

Es gibt einige Gründe warum Max Pooling hilfreich sein kann.

21.4.1 REDUKTION DER RECHENLAST

Da Max Pooling die Auflösung einer Matrix reduziert, kann das Netz auf einmal auf (eigentlich) größere Bereiche des Bildes schauen, was die Parameter in unserem Netz reduziert und somit auch die Rechenlast reduziert

21.4.2 REDUKTION VON OVERFITTING

Weiters kann Max Pooling auch dabei helfen, Overfitting zu reduzieren, da das Netz nur reduzierte Werte sieht und nicht alle.

Mit Max Pooling können wir die aktiviertesten Pixel (also die höchsten Werte) herausheben während wir die Pixel mit den niedrigsten Werten entfernen.

Es gibt auch noch andere Arten von Pooling die denselben Prozess folgen. Diese unterscheiden sich nur in den Operationen auf den Regionen, anstatt den Maximal zu finden.

21.4.2.1 AVERAGE POOLING

Zum Beispiel gibt es noch Average Pooling. Hier wird der Mittelwert der Region anstatt dem Maximalwert genommen.

Derzeit wird Max Pooling öfter als Average Pooling verwendet. Was man Einsetzt hängt vom Anwendungsgebiet ab. Am besten findet man durch experimentieren das für seinen Anwendungszweck geeignetste heraus.

21.5 MAX POOLING IN KERAS

Wie üblich einige Imports:

```
import tensorflow.keras

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import *
```

Danach erstellen wir ein vollkommen willkürliches CNN:

```
model = Sequential([

   Dense(16, input_shape=(20,20,3), activation='relu'),

   Conv2D(32, (3,3), activation='relu', padding='same'),

   MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'),

   Conv2D(64, (5,5), activation='relu', padding='same'),

   Flatten(),

   Dense(2, activation='softmax')

])
```

Es hat einen Input Layer mit 20 x 20 x 3 Dimensionen, danach einen Dense gefolgt von einem Convolutional Layer. Danach kommt ein Max Pooling Layer und danach noch ein Convolutional Layer welcher schlussendlich zum Output Layer führt.

Weil die Convolutional Layer 2D sind benutzen wir auch einen MaxPooling2D Layer. Keras hat aber auch noch 1d und 3d Max Pooling Layers.

Der erste Parameter ist die Pool Size. Das ist die Größe, die wir vorhin Filter genannt haben, in diesem Beispiel haben wir einen 2 x 2 Filter benutzt.

Der nächste Parameter ist strides. Wir haben wieder in unserem vorherigem Beispiel 2 dafür genommen, deshalb haben wir hier auch 2 definiert. Der letzte Parameter ist das Padding welches schon aus dem vorherigen Kapitel (20) bekannt ist.

Wenn wir uns das summary unseres Netzes ansehen, können wir erkennen, dass unsere Dimensionen bei unserem Max Pooling Layer halbiert wurden:

```
model_valid.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 20, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 64)        51264
_________________________________________________________________
flatten_1 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 12802
=================================================================
Total params: 68,770
Trainable params: 68,770
Non-trainable params: 0
_________________________________________________________________
```

Das geschieht aufgrund des 2 x 2 Filters mit einem Stride von 2.

22. BACKPROPAGATION – DIE INTUITION VON BACKPROPAGATION

In diesem Kapitel geht es um Backpropagation und welche Rolle es in dem Trainingsprozesses eines neuronalen Netzes einnimmt.

22.1 STOCHASTIC GRADIENT DESCENT (SGD) (REVIEW)

Hier ein kurzer Überblick was in vorherigen Kapiteln (6, 7) schon erwähnt wurde:

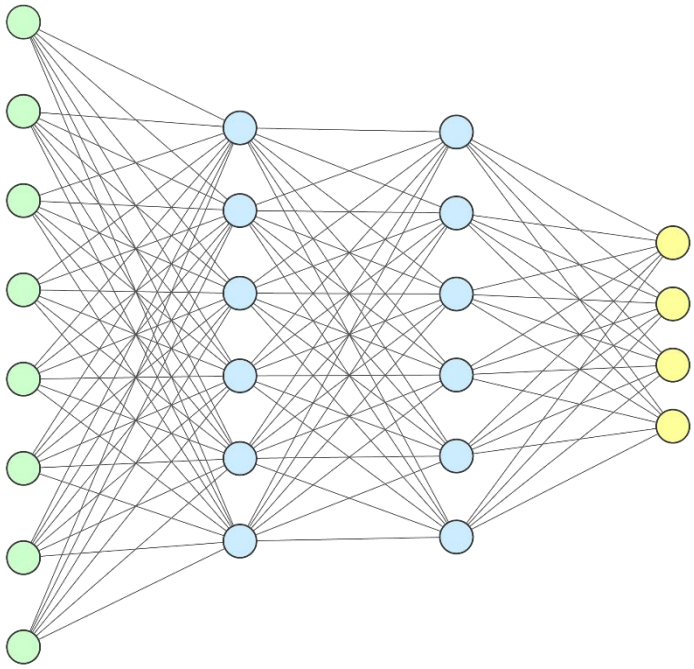
Während dem Training minimiert der stochastic gradient descent (= SGD) die loss function indem es die gewichte mit jeder Epoche updatet.

Dieses updaten geschieht durch das berechnen des Gradienten, oder der Ableitung der Loss function. Das haben wurde aber noch nicht ausführlich erklärt.

Das wird hier erklärt. Der Prozess der Berechnung des Gradienten um die Gewichte zu updaten wird durch backpropagation erreicht.

22.2 FORWARD PROPAGATION

Hier haben wir ein Beispiel Netz mit zwei hidden Layers. Einfachheitshalber wird der Prozess nur anhand eines Inputs erklärt und nicht anhand eines Batchs.



Hier eine kurze wiederholung vom Trainingsprozess: Wenn wir Daten durch unser Netz jagen, propagieren unsere Daten durch das Netz bis sie das Ende erreicht haben.
Jedes Neuron in unserem Netz bekommt einen Input vom vorherigen Layers. Das ist ein eine gewichtete Summe der Verbindungen des Neurons multipliziert mit dem Output des vorherigen Layers.

Diese Summe wird an eine Aktivierungsfunktion gegeben. Das Ergebnis dieser Aktivierungsfunktion ist der Output für ein einzelnes Neuron welches dann Teil des Inputs der Neuronen im nächsten Layer wird. Das geschieht für jeden Layer bis wir den Outputlayer erreichen. Dieser Prozess heißt forward propagation.

Wenn der Output Layer erreicht wird, erhalten wir den Output des Netzes für einen gegebenen Input. Wenn unser Netz zum Beispiel Bilder von Tieren klassifizern soll, würde jeder der Neuronen einem anderen Tiertyp entsprechen und das Neuron, dass am meisten „aktiviert“ ist, ist der Output.

22.2.1 BERECHNUNG DES LOSS

Anhand der Outputs berechnen wir dann den Loss. Wie der Loss berechnet wird, hängt von der verwendeten Loss function ab. Einfachheitshalber stellen wir uns vor, dass er beschreibt wie weit das Netz bei der Klassifizierung des Inputs daneben liegt. Genaueres wurde im Kapitel 6.3 und im Kapitel 8 erklärt.

Es wurde also schon erklärt, dass das Ziel von SGD ist, den Loss zu minimeren. Das geschieht durch die Ableitung, d.h. die Steigung, der Loss function in Bezug auf die weights im Netz.

$$\frac{d(loss)}{d(weights)}$$

Hier kommt backpropagation ins Spiel.

Backpropagation ist das „Werkzeug“, mit dem der Gradentenabstieg der Gradienten der Loss function berechnet.

Wie schon erwähnt heißt das durchjagen von Daten durch das Netz forward propagation. Der Prozess, der in diesem Kapitel behandelt wird, heißt backpropagation. Und ja, du liegst richtig, wenn du denkst, dass dies bedeutet, dass irgendwie rückwärts über das Netz gearbeitet wird.

Der Loss wird für einen Output berechnet und der gradient descent updatet mithilfe von backpropagation die Gewichte, um den Loss zu verringern.

22.3 DIE INTUITION VON BACKPROPAGATION

Um die Gewichte zu aktualisieren, beginnt der gradient descent den Output mit den dazugehörigen Labels zu vergleichen.

Wenn uns unser Netz einen richtigen Output liefert, dann weiß gradient descent, dass der Wert, den das richtige Output-Neuron hat, noch größer werden soll, und alle anderen noch kleiner. Durch dieses Wissen kann SGD den loss verringern.

Wir wissen, dass der Wert eines Output Nodes berechnet wird, indem die gewichtete Summe über jede der ankommenden Verbindungen multipliziert mit dem Output des vorherigen Layers an die Aktivierungsfunktion des Output Layers gegeben wird.

Um nun die Werte der Output-Neuronen so anzupassen, dass ein „besseres“ Ergebnis zustande kommt, müssen entweder die Gewichte der Verbindungen zwischen den Nodes geupdated werden oder der Output des vorherigen Layers verändert werden.

Direkt können wir den Output der Nodes im vorherigen Layer nicht ändern. Indirekt können wir durch einen Schritt zurück (im Layer davor) die Gewichte der Verbindungen zu diesem vorletzten Layer ändern und somit Einfluss auf den Output dieses Layers Einfluss nehmen.

Dieser Prozess wird solange wiederholt, bis wir den Input Layer erreichen. Den Output der Neuronen im Input Layer wollen wir nicht verändern, da er ja unseren eigentlichen Input enthält.

Wir bewegen uns also durch das Netz zurück (= back -> Backpropagation) und updaten so die Gewichte von rechts nach links, um die Werte der Output Nodes leicht zu verändern und somit den Loss zu verringern.

Das Verhältnis, in dem einige Gewichte im Vergleich zu anderen aktualisiert werden, kann höher oder niedriger sein. Je nachdem, wie stark sich die Aktualisierung auf das gesamte Netz auswirkt, um den Loss zu minimieren.

Nach der Berechnung der Ableitung werden die Gewichte, mithilfe der eben errechneten Ableitung, verhältnismäßig zu deren neuen Werten geupdated.

Hier wurde jetzt nur ein einzelner Input erklärt. Der exakt gleiche Prozess passiert aber für alle Inputs für jeden Batch, mit dem wir unser Netz füttern, und die daraus resultierenden Updates der Gewichte sind Durchschnittswerte, welche für jeden individuellen Input berechnet werden.

22.3.1 EINE KURZE ZUSAMMENFASSUNG DIESES PROZESSES

Wenn ein neuronales Netz trainiert wird, werden Daten an das Netz gegeben. Der „Weg“, auf dem diese Daten durch das Netz fließen, ist die Forward Propagation, bei der wir wiederholt die gewichtete Summe der Outputs der vorherigen Layers mit den entsprechenden Gewichten berechnen und diese Summe dann an die Aktivierungsfunktion des nächsten layers weitergeben.

Dies wird getan, bis der Output Layer erreicht wird. Zu diesem Zeitpunkt wird der Loss des Outputs berechnet und der gradient descent versucht dann, diesen Loss zu minimieren.

Gradient descent führt diesen Minimierungsprozess durch, indem zunächst der Gradient der Loss function berechnet wird und dann die Gewichte im Netz entsprechend aktualisiert werden.

23. VERSCHWINDENDER & EXPLODIERENDER GRADIENT

In diesem Kapitel geht es um ein Problem, das durch backpropagation auftritt: Ein instabiler Gradient der am häufigsten als vanishing (verschwindender) Gradient bezeichnet wird.

23.1 INTRODUCTION

Was wissen wir bisher schon über Gradienten im Bezug zu Neuralen Netzwerken?

- Wenn wir das Wort Gradient verwenden, beziehen wir uns normalerweise auf den Gradienten der loss function in Bezug auf die Gewichte im Netz.
- Wie wir in dem Kapitel gesehen haben, das zeigt, wie ein neuronales Netz lernt, wissen wir, was zu tun ist, nachdem der Gradient berechnet wurde -> Wir aktualisieren unsere Gewichte damit!
- Der SGD tut das, mit dem Ziel, die optimalen Gewichte zu bekommen, um den loss des Netzes zu minimieren

Mithilfe dieses Verständnisses werden wir nun über das vanishing Gradienten Problem reden.
Zuerst werden wir klären, was das vanishing Gradienten Problem überhaupt ist, wie es Auftritt und was exploding (=Explodierende) Gradienten sind. Wir werden feststellen, dass das exploding Gradienten Problem ziemliche Ähnlichkeiten mit dem vanishing Gradienten Problem aufweist.

23.2 WAS IST DAS VANISHING GRADIENTEN PROBLEM

Im Allgemeinen ist das vanishing Gradienten Problem ein Problem, welches beim Training eines neuronalen Netzes große Schwierigkeiten bereitet. Dieses Problem betrifft die früheren Schichten eines Netzwerkes.

Erinnere dich, dass während des Trainings der SGD den Gradienten des loss im Bezug auf die Gewichte im Netzwerk minimiert.

Manchmal passiert es, dass der Gradient in frühen Schichten des Netzes sehr gering wird, verschwindend gering -> Daher auch vanishing Gradienten Problem

Was ist also so schlimm an kleinen Gradienten?

23.2.1 KLEINE GRADIENTEN

Wenn also SGD den Gradienten berechnet hat, benutzt es den Wert, um die Gewichte in einem proportionalen Weg zu aktualisieren. Wen der Gradient also extrem klein ist, ist dieses Update auch sehr klein. Dadurch wirkt sich diese Änderung kaum auf das Netz aus. Dadurch hilft dieses neue Gewicht nicht dabei, den Loss zu minimieren.

23.2.1.1 WIE ENTSTEHT DIESES PROBLEM?

Der Gradient des Loss wird in Bezug ein bestimmtes Gewicht das Produkt einiger Ableitungen sein, die von Komponenten abhängen, die sich später im Netzwerk befinden.
Je früher also ein Gewicht im Netz ist, desto mehr Terme sind in dem eben genannten Produkt notwendig, um den Gradienten dieses Gewichtes zu bekommen.

Der Schlüssel dabei ist, zu verstehen, was passiert, wenn die Terme in diesem Produkt kleiner als Eins sind.
Das Produkt von Zahlen, die kleiner als Eins sind, wird eine noch kleinere Zahl ergeben, stimmts?

Wie vorhin erwähnt nehmen wir also diese kleine Zahl und updaten unsere Gewichte damit. Wir multiplizieren also diese Zahl zuerst mit unserer Lernrate, die gewöhnlich zwischen `0.01` und `0.0001` liegt.
Das Produkt dieser Zahlen ist also eine noch kleinere Zahl. Nachdem wir diese (kleine) Zahl berechnet haben, subtrahieren wir diese vom Gewicht. Dieses Ergebnis wird der Wert des aktualisierten Gewichts sein.

Wenn also unser Gradient schon verschwindend klein ist, und wir ihn auch noch mit der Lernrate multiplizieren, ist diese Zahl extrem klein und wirkt sich kaum auf das Gewicht aus.

Im Wesentlichen gerät das Gewicht in einen Zustand der Starre. Es bewegt sich nicht, lernt nicht und trägt daher nicht dazu bei, den Loss zu verringern

23.3 EXPLODIERENDER GRADIENT

Das ist das Gegenteil des gerade beschriebenen Problems, der Gradient wird nicht extrem klein, sondern extrem groß.

Im Prinzip unterscheidet sich das exploding Gradienten Problem und das vanishing Gradienten Problem nur insofern, dass hier die Werte deutlich über 1 liegen und deshalb verändert sich das Gewicht zu stark und wir verpassen den Optimalen Wert eines Gewichtes, da wir uns mit zu großen Schritten bewegen und wir uns deshalb immer weiter von diesem optimalen Wert entfernen.

24. GEWICHTSINITIALISIERUNG

In diesem Kapitel geht es darum, wie die Gewichte in einem ANN initialisiert werden, wie diese Initialisierung den Trainingsprozess beeinflusst und was wir dagegen machen können.

Wir wissen, dass die Gewichte die Neuronen zwischen den Schichten verbinden. Zunächst werden wir besprechen, wie diese Gewichte initialisiert werden und wie sich diese initialisierten Werte negativ auf den Trainingsprozess auswirken können.

Mit diesem Wissen im Kopf, werden wir lernen, wie diese Initialisierung beeinflussen können. Dann werden wir sehen, wie wir in Keras spezifizieren können, wie die Gewichte initialisiert werden.

24.1 WIE WERDEN GEWICHTE INITIALISIERT?

Wenn wir ein Netz bauen und kompilieren, werden die Werte der Gewichte zufällige Zahlen sein. Eine zufällige Zahl pro Gewicht. Normalerweise werden diese Zufallszahlen normalverteilt, so dass die Verteilung dieser Zahlen einen Mittelwert von 0 und eine Standardabweichung von 1 hat.

Wie beeinflusst diese zufällige Initialisierung unser Training?
Um dies zu sehen, betrachten wir folgendes Beispiel:

24.1.1 ZUFÄLLIGES INITIALISIERUNGSBEISPIEL

Angenommen, unser Input Layer hat 250 Nodes. Einfachheitshalber stellen wir uns vor, dass der Wert jeder der 250 Nodes 1 ist.

Nun konzentrieren wir uns nur auf die Gewichte, die den Input Layer mit einem einzelnen Node im ersten hidden Layer verbindet. Insgesamt gibt es 250 Gewichte, die diesen Node in unserem ersten Hidden Layer mit allen (250) Nodes im Input Layer verbindet.

Nun wurde jedes dieser Gewichte zufällig generiert und normalverteilt (mit einem Mittelwert von 0 und einer Standardabweichung von 1.) Was bedeutet dies also für die gewichtete Summe z, die dieser Knoten als Eingabe akzeptiert?

Beachte, dass in unserem Fall alle Eingabeknoten einen Wert von 1 haben, so dass jedes Gewicht in z mit einer 1 multipliziert wird, so dass z nur zu einer Summe der Gewichte wird.

Zurück dazu, wie diese zufällige Initialisierung z beeinflusst. Genauer gesagt wie es die Varianz von z beeinflusst.

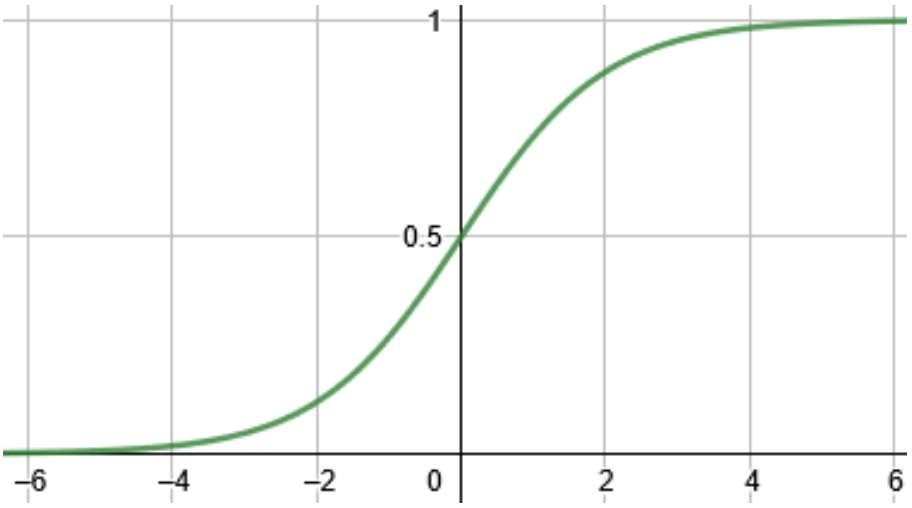
Nun, z, als Summe normalverteilter Zahlen mit einem Mittelwert von 0, wird ebenfalls um 0 normalverteilt sein, aber seine Varianz, und ebenso seine abgeleitete Standardabweichung werden größer als 1 sein.

Das ist deshalb so, weil die Varianz einer Summe von eigenständigen zufälligen Zahlen die Summe der Varianzen von jeder dieser Zahlen ist. Da die Varianz von allen 250 Zahlen 1 ist, ist die Varianz z die Summe dieser Varianzen, also 250.

Nimmt man die Quadratwurzel dieses Wertes, so sieht man, dass z eine Standardabweichung von 15.811 hat.

Wenn wir uns also die Normalverteilung von z ansehen, sehen wir, dass sie breiter ist als eine Normalverteilung mit der Standardabweichung von 1.

Mit dieser größeren Standardabweichung nimmt der Wert von z eher eine Zahl an, die deutlich größer oder kleiner als 1 ist. Wenn wir diesen Wert an unsere Aktivierungsfunktion weitergeben, wissen wir, dass die meisten positiven Eingaben, insbesondere diejenigen, die deutlich größer als 1 sind, auf den Wert 1 gemappt werden. In ähnlicher Weise werden die meisten negativen Eingänge auf 0 gemappt.



24.1.2 PROBLEME VON ZUFÄLLIGER INITIALISIERUNG

Wenn der erwünschte Output unserer Aktivierungsfunktion auf der entgegengesetzten Seite liegt dann wird SGD während des Trainings, wenn es die Gewichte versucht zu aktualisieren, nur sehr kleine Änderungen im Wert dieser Aktivierungsausgabe vornehmen und sie kaum noch in die richtige Richtung bewegen.

Dadurch wird die Lernfähigkeit des Netzes behindert, und das training bleibt in diesem langsamen und ineffizienten Zustand stecken.

Angesichts dieser zufälligen Gewichtsinitialisierung, die Probleme verursacht, können wir etwas tun, um das zu bessern? Können wir ändern wie die Gewichte initialisiert werden?

Ja, können wir ;)

24.2 XAVIER INITIALIZATION

Die Probleme, die wir besprochen haben, entstehen also dadurch, dass die gewichtete Summe eine Varianz annimmt, die viel größer oder kleiner als 1 ist.
Um dieses Problem zu lösen, müssen wir also die Varianz zwingen, kleiner zu werden.

Wie wird das erreicht?

Nun, da die Varianz der Eingabe für einen bestimmten Knote durch die Varianz der Gewichte bestimmt wird, die mit diesem Knoten aus der vorherigen Schicht verbunden sind, müssen wir die Varianz dieser Gewichte verkleinern, wodurch die Varianz der gewichteten Summe verkleinert wird.

Einige Forscher haben einen Wert für die Varianz der Gewichte identifiziert, der ziemlich gut zu funktionieren scheint, um die vorhin besprochenen Probleme zu mildern. Der Wert für die Variant der Gewichte, die mit einem bestimmten Knoten verbunden sind, ist 1/n, wobei n die Anzahl der Gewichte ist, die mit diesem Knoten aus der vorherigen Schicht verbunden sind.

Anstatt also die Verteilung dieser Gewichte um 0 mit einer Varianz von 1 zu zentrieren, was wir früher hatten, sind sie jetzt immer noch um 0 zentriert, aber mit einer deutlich geringeren Varianz 1/n.

Um diese Gewichte zu einer Varianz von 1/n zu bringen, müssen wir nach der zufälligen Generierung der Gewichte – zentriert um 0 mit einer Varianz von 1 – jedes mit $\sqrt{\frac{1}{n}}$ multiplizieren. Dadurch wird die Varianz 1/n.

Diese Art von Initialisierung wird Xavier initialization oder Glorot initialization genannt.

Es ist wichtig zu beachten, dass, wenn wir relu als Aktivierungsfunktion verwenden, der beste Werte für die Varianz 2/n und nicht 1/n ist.

Dieser Initialisierungsprozess geschieht auf einer Pro-Schicht-Basis, da wir n als die Anzahl der Gewichte, die mit einem bestimmten Knoten aus der vorherigen Schicht verbunden, definiert haben

24.3 GEWICHTSINITIALISIERUNG MIT KERAS

Lass uns nun sehen, wie wir einen Gewichtsinitialisierer für unser eigenes Netz in Keras verwenden können. Wir werden dieses willkürliche Netz verwenden welches 2 hidden Dense Layers und einen Output Layer mit 2 Nodes hat.

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense
```

```
from tensorflow.keras.layers import Activation


model = Sequential([
    Dense(16, input_shape=(1,5), activation='relu'),
    Dense(32, activation='relu', kernel_initializer='glorot_uniform'),
    Dense(2, activation='softmax')
])
```

Nahezu alles, was in diesem Netz gezeigt wird, wurde schon in vorherigen Kapiteln erklärt, also werde ich sie an dieser Stelle nicht mehr erklären. Stattdessen fokussieren wir uns auf das eine neue Ding, das wir noch nicht gesehen haben. Nämlich der kernel_initializer Parameter im zweiten hidden Layer

Diesen Parameter benutzen wir, um zu spezifizieren, welchen Typ von Gewichtsinitialisierer wir für einen spezifischen Layer verwenden wollen. Hier haben wir den Wert auf glorot_uniform gesetzt. Das ist die Xavier/Glorot Initialisierung, welche eine gleichmäßige Verteilung verwendet. Du kannst auf glorot_normal für eine Xavier/Glorot Initialisierung mit Normalverteilung verwenden.

Wenn wir gar nichts angeben, initialisiert Keras die Gewichte in jedem Layer standardmäßig mit der glorot_uniform Initialisierung. Das gilt auch für andere Layerarten, nicht nur für Dense. Convolutional Layers verwenden zum Beispiel auch glorot_uniform standardmäßig.

Das heißt, dass wir eigentlich nichts angeben müssen und Keras trotzdem die Gewichte mit der Xaver Initialisierung initialisiert.

25. BIAS

Keine Angst, wir werden hier Bias (= Vorurteil) nicht von einem sozialen oder politischen Standpunkt besprechen, sondern wir werden speziell den Bias (also die Voreingenommenheit) innerhalb von ANN's besprechen.

25.1 HINTERGRUND

Wenn du etwas über Artificial Neural Networks lest, hast du wahrscheinlich schon irgendwo etwas von Vorurteilen gesehen. Manchmal wird diese nur als Bias bezeichnet, manchmal auch als Bias Nodes, Bias Neurons, oder Bias Units.

Lass uns mit der offensichtlichsten Frage starten: Was ist Bias in einem ANN?

Danach werden wir sehen, wie Bias in einem Netz implementiert wird und sehen uns dann ein Beispiel an, um zu sehen welchen Einfluss Bias in einem neuralen Netz hat.

25.2 BIAS IN EINEM NEURONALEN NETZ

25.2.1 WAS IST BIAS?

Zuerst werden wir Bias an einer Per-Neuron Basis besprechen. Wir können uns vorstellen, dass jedes Neuron seinen eigenen Bias hat, und so wird das gesamte Netzwerk aus mehreren Biases bestehen.

Nun sind diesen Bias zugeordneten Werte erlernbar, ebenso wie die Gewichte. Genau wie stochastic gradient Descent die Gewichte über Backpropagation während des Trainings aktualisiert, lernt und aktualisiert SGD auch die Biases.

Nun können wir uns konzeptionell vorstellen, dass der Bias bei jedem Neuron eine ähnliche Rolle wie die eines Schwellenwertes (= threshold value) hat. Der Wert des Bias bestimmt nämlich, ob die Aktivierungsausgabe eines Neurons durch das Netzwerk nach vorne geleitet (siehe forward propagation) wird oder nicht.

In anderen Worten bestimmt das Bias, ob ein Neuron feuert oder nicht. Es lässt uns wissen, wann ein Neuron sinnvoll aktiviert ist.

Wie wir in wenigen Augenblicken sehen werden, führt die Hinzufügung von Biases dazu, dass die Flexibilität eines Netzes zu Anpassung an die gegebenen Daten erhöht wird.

25.2.2 WO IST BIAS IN EINEM ANN

Jetzt wissen wir also, was Bias ist, aber wo genau passt es in das Schema eines neuronalen Netzes?

Wie wir in vorhergehenden Kapiteln besprochen haben, wissen wir, wie jedes Neuron eine gewichtete Summe des Inputs von dem vorherigen Layer erhält, und diese gewichtete Summe an eine Aktivierungsfunktion übergibt.

Nun, das Bias für ein Neuron passt genau hier hinein. Anstatt die gewichtete Summe direkt an die Aktivierungsfunktion zu übergeben, übergeben wir der Aktivierungsfunktion die gewichtete Summe und den Bias Term.

Ok, was soll das bringen?

Schauen wir uns dazu ein einfaches Beispiel an, das die Rolle dieses Bias in der Praxis veranschaulicht.

25.2.3 BEISPIEL ZUR VERANSCHAULICHUNG VON BIAS

Stell dir vor, wir haben ein neuronales Netz, das einen Input Layer mit nur zwei Nodes hat. Stell dir vor, dass das erste Node als Wert 1 hat und das zweite als Wert 2.

Jetzt richten wir unsere Aufmerksamkeit auf ein einzelnes Neuron innerhalb des ersten Hidden Layers, welche direkt der Eingabeschicht folgt.

Als Aktivierungsfunktion verwenden wir relu, und wir generieren zufällige Gewichte für unsere connections.

Lass uns nun sehen, was der Output dieses Nodes sein würde ohne Bias.

Die gewichtete Summe, die dieser Knoten erhält, ist gegeben durch:

```
(1*-0.55) + (2*0.10) = -0.35
```

Dieses Ergebnis übergeben wir an relu. Wir wissen, dass der Wert von relu entweder 0 oder der Input selbst, abhängig davon, ob 0 oder der Input höher ist. In unserem Fall haben wir:

```
relu(-0.35) = 0
```

Bei einem Aktivierungsoutput von 0 gilt das Neuron nicht aktiviert bzw. es feuert nicht.

Im Wesentlichen ist hier 0 die Schwelle für eine gewichtete Summe bei der Bestimmung, ob ein Neuron feuert oder nicht.

Was ist, wenn wir diese Schwelle verschieben wollen? Was, wenn wir anstelle von 0 festgelegt hätten, dass ein Neuron feuern soll, wenn sein Input größer oder gleich -1 ist?

Das ist der Moment, wo Bias ins Spiel kommt.

Erinnere dich daran, dass wir vorhin gesagt haben, dass Bias zu der gewichteten Summe hinzugefügt wird bevor diese an die Aktivierungsfunktion übergeben wird. Der Wert den wir als Bias festlegen ist das Gegenteil von dem sogenannten threshold Wert (= Schwellenwert).

Wenn wir mit unserem Beispiel fortfahren, wollen wir den threshold von 0 zu -1 schieben. Der Bias Wert muss also das Gegenteil von -1 sein, also 1-

Die gewichtete Summe von -0.35 plus unserem Bias von 1 ergibt 0.65

```
(1*-0.55) + (2*0.10) +1 = -0.35
```

Wenn wir diesen Wert an relu übergeben, sehen wir:

```
relu(0.65) = 0.65
```

Das Neuron feuert also.

Dass Netz ist jetzt etwas flexibler bei der Anpassung der Daten, da es jetzt eine größere Bandbreite an Werten hat, die es als aktiviert oder nicht aktiviert sieht.

Wir könnten den gleichen Prozess auf umgekehrt durchführen, um die Ausgabewerte von Neuronen, die wir als aktiviert betrachten, einzuschränken. Wenn wir zum Beispiel festlegen, dass ein Neuron als aktiviert betrachtet werden sollte, wenn sein Ausgangswert größer oder gleich 5 ist, dann wäre unser Bias = -5

25.3 CONCLUSION

Bei unserem Beispiel haben wir uns explizit unser Bias ausgewählt. In der Praxis ist dies nicht der Fall. Genau wie wir uns auch nicht explizit die Gewichte aussuchen und steuern.

Vergiss nicht, dass Biases wie weights lernfähige Parameter sind. Nachdem die Biases mit zufälligen Zahlen (oder Nullen oder irgendein anderer Wert) initialisiert worden sind werden sie während dem Training geupdated, so dass unser Netz lernen kann, wann ein Neuron aktiviert ist und wann nicht.

26. LERNFÄHIGE PARAMETER IN EINEM ANN

Wie sich herausstellt, haben wir bereits viel über lernfähige Parameter in einem neuronalen netz gelernt, aber wir haben zu dem allgemeinen Thema noch keine formale Einführung erfahren.

In diesem Kapitel starten wird damit, zu definieren, was lernfähige Parameter sind. Danach werden wir sehen, wie die Gesamtanzahl von lernfähigen Parametern in einen ANN berechnet werden kann und werden dies anhand eines leichten Beispiels illustrieren.

26.1 WAS SIND LERNFÄHIGE PARAMETER?

Lassen uns sehen, ob wir anhand des Namens die Funktion von lernfähigen Parametern ableiten können.

Das ist nicht schwer, ein lernfähiger Parameter ist einfach ein Parameter, der vom Netz während des Trainings gelernt wird.

Während dem Trainingsprozess versucht SGD die Gewichte und Biases in unserem Netz zu lernen und zu optimieren. Diese Gewichte und Biases sind also in der Tat lernfähige Parameter.

Tatsächlich werden alle Parameter innerhalb unseres Netzes, die während dem Training mithilfe von SGD gelernt werden, als lernfähige Parameter bezeichnet.

Beachte, dass diese Parameter auch als trainierbare Parameter bezeichnet werden.

26.2 BERECHNUNG DER ANZAHL VON LERNFÄHIGEN PARAMETERN

Jetzt wissen also was lernfähige Parameter sind. Aber wie können wir die Anzahl dieser für einen Layer oder ein ganzes ANN berechnen?

Um dieses Ergebnis zu finden, zählen wir im Wesentlichen nur die Anzahl der Parameter innerhalb jeden Layers und summieren sie dann, um die Gesamtanzahl der Parameter innerhalb des gesamten Netzes zu erhalten.

Um die Anzahl der Parameter in einem einzelnen Layer zu berechnen brauchen wir:

- Die Anzahl der Inputs dieses Layers
- Die Anzahl der Outputs dieses Layers
- Die Information, ob dieser Layer Biases verwendet oder nicht

Beachte, dass wir hier von einem fully connected Netz mit lauter Dense Layers ausgehen. In dem nächsten Kapitel werden wir uns auch ansehen, wie dies für Convolution Neural Networks funktioniert.

Wenn wir einmal unsere benötigten Informationen haben, multiplizieren wir die Anzahl der Inputs eines Layers mit der Anzahl der Outputs dieses Layers. Eine andere Möglichkeit, über die Outputs nachzudenken, besteht darin, einfach über die Anzahl der Knoten innerhalb der Schicht zu denken. Die Anzahl der Nodes ist gleich mit der Anzahl der Outputs.

Wenn wir nun die Inputs mit den Outputs multiplizieren, erhalten wir die Anzahl der Gewichte, die in diesen Layer einfließen.

Danach müssen wir verstehen, ob der Layer Biases hat oder nicht. Wenn er Biases verwendet, dann addieren wir einfach die Anzahl der Biases zu der eben berechneten Anzahl der Gewichte. Die Anzahl der Biases ist gleich mit der Anzahl der Nodes in dem Layer.

Das gibt uns die Anzahl von lernfähigen Parametern in einem Dense Layer. Um die Anzahl aller Lernfähigen Parameter in einem Netz zu bekommen, müssen wir einfach dieselbe Berechnung für alle anderen Layers in dem Netz ausführen und alle Ergebnisse aufsummieren.

26.3 LERNFÄHIGE PARAMETER BEISPIEL

Angenommen wir haben ein fully connected Netz mit 3 Layer:

Layer	Anzahl der Nodes
Input	2
Hidden	3
Output	2

Zusätzlich nehmen wir an, dass unser Netz Biases verwendet. Das heißt, dass Biases in dem Hidden und dem Output Layer existieren.

Lass uns nun die Anzahl der lernfähigen Parameter in jedem Layer berechnen.

Zunächst einmal ist zu beachten, dass der Input Layers hat keine Lernfähigen Parameter, weil er nur aus den Inputdaten besteht und der Output dieses Layers nur als Input des nächsten Layers betrachtet wird.

Lass uns nun die Anzahl der lernfähigen Parameter in dem hidden Layer berechnen.

Im ersten Hidden Layer haben wir 2 Inputs, welche die Outputs des Input Layers sind. Nun brauchen wir auch die Anzahl der Outputs dieses Layers.

Die Anzahl der Outputs ist die Anzahl der Neuronen. Das bedeutet in unserem Fall, dass wir 3 Outputs haben. Wir multiplizieren diese beiden Zahlen miteinander, was uns sechs Gewichte gibt.

Als nächstes fügen wir unsere Biases hinzu. Der Hidden Layer hat drei Neuronen, was bedeutet, dass er auch drei Biases hat. Addieren wir 3 und 6, können wir sehen, dass dieser Hidden Layer 9 lernfähige Parameter hat.

Bei unserem Output Layer machen wir das gleiche.
-> 3 Inputs * 2 Outputs + 2 Biases = 8 Lernfähige Parameter

Wenn wir nun unsere 8 Parameter aus dem Hidden Layer mit den 9 Parametern aus den Output Layer addieren sehen wir, dass unser ganzes Netz insgesamt 17 lernfähige Parameter hat. Während des Trainings wird SGD also diese 17 Parameter optimieren.

27. LERNFÄHIGE PARAMETER IN EINEM CNN

In diesem Kapitel geht es auch um die Lernfähigen Parameter, allerdings nicht in einem fully connected Netz, sondern in einem Convolutional Neural Network.

Wir starten damit, zu besprechen, was lernfähige Parameter in einem CNN sind und wie die Anzahl dieser berechnet wird. Nachdem das gemacht ist, werden wir – wie im vorherigen Kapitel – uns diese Berechnung auch an einem konkreten Beispiel anschauen.

27.1 WAS SIND LERNFÄHIGE PARAMETER IN EINEM CNN?

Im Allgemeinen sind es dieselben Parameter wie bei einem fully connected Layer. Also die Gewichte und die Biases. Wir müssen dabei aber bedenken, dass ein CNN und ein standardmäßiges fully connected ANN verschiedene Netztypen sind, was unsere Berechnung beeinflusst.

27.2 WIE DIE ANZAHL DER LERNFÄHIGEN PARAMETER BERECHNET WIRD

Wie bei einem normalen Netz berechnen wir die Anzahl der lernfähigen Parameter per Schicht, und Summieren dann die Anzahl von jeder Schicht über alle Schichten auf.

```
//pseudocode (javascript)

Let sum = 0;
network.layers.forEach(function(layer){
   sum += layer.getLernableParameters().length;
})
```

Für einen Dense Layer bekommen wir die Anzahl der lernfähigen Parameter indem wir Inputs * Outputs + Biases rechnen. Lass uns nun sehen, was ein convolutional Layer hat, was ein Dense Layer nicht hat.

Ein convolutional Layer hat Filter, auch bekannt als Kernels. Als Architekten unseres Netzes müssen wir bestimmen, wie viele Filter in einem convolutional Netz sind und wie groß diese sind.

Mit diesem Wissen werden wir die Formel für die Berechnung zu Bestimmung der Anzahl von lernfähigen Parametern in einem convolutional Layer bestimmen.

Was ist also der Input für einen convolutional Layer? Das hängt davon ab, was für eine Layerart der vorherige Layer war.

- Wenn der vorherige Layer ein Dense Layer war, dann ist der Input für diesen conv Layer nur die Anzahl der Nodes in dem vorherigen Layer
- Wenn der vorherige Layer ein Convolutional Layer war, dann ist der Input die Anzahl der Filter dieses vorherigen Convolutional Layer

Was ist der Output eines convolutional Layer

- Die Anzahl der Filter multipliziert mit deren Größe

Zum Schluss noch die Anzahl der Biases, welche einfach gleich hoch sind wie die Anzahl der Filter in dem Layer.

Insgesamt ergibt sich also die Rechnung: `Inputs * Outputs + Biases`

Das ist also die gleiche Formel wie bei einem ANN bestehend aus Denses. Der Unterschied zum Convolutional Layer besteht darin, dass die Inputs und Outputs die Anzahl der Filter und die Filtersize ist.

27.3 BEISPIEL IN EINEM CNN

Stell dir vor, wir haben ein CNN mit einem Input Layer, zwei convolutional Hidden Layers und einen Dense Output Layer.

Unser Input Layer besteht aus Bilddaten mit der Größe 20x20x3, wobei 20x20 die Breite und die Länge des Bildes sind und 3 die Anzahl der Farbkanäle. Die drei Farbkanäle sagen uns, dass unsere Bilder aus RGB Farben bestehen. Diese Kanäle werden die Input Features.

Unser erster Convolutional Layer besteht aus 2 Filtern mit einer Größe von 3x3. Das zweite Convolutional Layer hat 3 Filter ebenfalls mit einer Größe von 3x3 und unser Output Layer ist ein Dense Layer mit 2 Nodes.

Wir gehen davon aus, dass das Netz Biases enthält und dass wir um gesamten Netzwerk Zero-Padding verwenden, damit die Dimensionen der Bilder gleichbleiben.

- Input Layer — Bilder mit der Größe von 20x20x3
- Hidden conv Layer — 2 Filter mit einer Größe von 3x3
- Hidden conv Layer — 3 Filter mit einer Größe von 3x3
- Dense Output Layer — 2 Nodes

27.3.1 INPUT LAYER

Beim Input Layer greifen dieselben Regeln wie einem normalen ANN. Der Input Layer hat keine lernfähigen Parameter da er nur die Input Daten enthält.

27.3.2 CONV LAYER 1

Bei diesem Layer haben wir 3 Inputs von unserem Input Layer. Und wie viele Outputs? Nun, lass uns das herausfinden. Erinnere dich daran, dass die Anzahl der Outputs die Anzahl der Filter mal der Anzahl der Filtergröße entspricht. Wir haben zwei Filter mit einer Größe von 3x3. Also `2 * 3 * 3 = 18`. Wenn wir nun unsere drei Inputs mit unseren 18 Outputs multiplizieren, kommen wir auf 54 Gewichte. Und wie viele Biases? Nur zwei, da die Anzahl der Biases der gleich ist mit der Anzahl der Filter. Das ergibt uns insgesamt 56 Lernfähige Parameter in diesem Layer.

27.3.3 CONV LAYER 2

Wie viele Inputs bekommt dieser Layer? Wir haben zwei, was wir aus der Anzahl der Filter im vorherigen Layer schließen. Wie viele Outputs? Nun ja, wir haben 3 Filter mit einer Größe von 3x3. Deshalb rechnen wir `3 * 3 * 3 = 27`. Multiplizieren wir das mit unseren 2 Inputs kommen wir auf 54 Gewichte in diesem Layer. Fügen wir nun unsere 3 Bias Werte hinzu, kommen wir auf 57 Lernfähige Parameter in diesem Layer

27.3.4 OUTPUT LAYER

Wie viele Inputs hat der Output Layer? Jetzt könnten wir 3 denken, da das die Anzahl der Filter in dem letzten Convolutional Layer ist. Das ist aber nicht ganz richtig. Bevor du nämlich einen Output eines convolutional Layers an einen Dense Layer übergeben, müssen wir ihn „flatten“ indem wir den Output mit den Dimensionen der Daten aus dem conv Layer mit der Anzahl der Filter in diesem Layer multiplizieren.

Da wir Zero Padding verwenden, sind die Dimensionen unserer Bilddaten noch immer 20x20. Wenn wir also 20x20 mit den 3 Filtern multiplizieren erhalten wir insgesamt 1200 Inputs, welche an diesen Dense Layer übergeben werden.

Da es sich hierbei um einen Dense Layer handelt, ist die Anzahl der Outputs gleich der Anzahl der Nodes in diesem Layer, was in unserem Fall 2 sind. Multiplizieren wir also 1200 mit 2 kommen wir auf 2400 Gewichte. Zählen wir nun noch unsere 2 Biases hinzu, kommen wir auf 2402 Lernbare Parameter.

27.3.5 DAS ERGEBNIS

Wenn wir alle Parameter aus jeder Schicht aufsummieren erhalten wir insgesamt 2515 lernfähige Parameter in dem Gesamten Netz.

Wir können also sehen, dass dieser Prozess in einem CNN ähnlich funktioniert wie in einem fully connected Netz.

28 REGULARISIERUNG IN EINEM ANN

In diesem Kapitel geht es darum, was eine Regulierung in einem ANN ist und warum sie uns helfen kann, wenn wir sie einem Netz hinzufügen.

In dem Kapitel 12.2.4 haben wir kurz Dropouts behandelt und erklärt, dass es sich dabei um eine Regularisierungstechnik handelt aber haben noch nicht besprochen, was eine Regularisierung ist.

Im Allgemeinen ist Regularisierung eine Technik, die uns hilft, Overfitting oder die Varianz in unserem Netz zu reduzieren, indem die Komplexität reduziert wird. Die Idee dabei ist, dass gewisse Komplexitäten in unserem Netz es unfähig werden lassen, gut zu Generalisieren, obwohl es mit den Trainingsdaten gut umgehen kann.

Wenn wir also unserem Netz Regularisierung hinzufügen, dann tauschen wir im Wesentlichen einen Teil der Fähigkeit des Netzes, gut mit den Trainingsdaten umzugehen, gegen die Fähigkeit, besser zu Generalisieren und so besser bei unbekannten Daten abschneiden.

Eine Regularisierung zu implementieren, bedeutet einfach einen Term zu unserer Loss function hinzuzufügen, der große Gewichte abschwächt. Diese Idee werden wir gleich weiter ausführen.

28.1 L2 REGULARISIERUNG

Die gebräuchlichste Regularisierungstechnik heißt L2 Regularisierung. Wir wissen, dass solch eine Technik grundsätzlich einen Term zu unserer Loss function hinzufügt, der große Gewichte „straft“.

28.1.1 L2 REGULARISIERUNGSTERM

Bei der L2 Regularisierung ist der Term, den wir zu dem Loss hinzufügen die Summe der quadrierten Normen der Gewichtsmatrizen.

$$\sum_{j=1}^{n} \left|\left|\omega^{[j]}\right|\right|^2$$

Multipliziert mit einem kleinen konstanten Wert:

$$\frac{\lambda}{2m}$$

28.1.2 NORMEN SIND POSITIV

Wenn dir Normen im Allgemeinen nicht bekannt sind, verstehe, dass eine Norm nur eine Funktion ist, die jedem Vektor in einem Vektorraum eine streng positive Länge oder Größe zuweist. Der Vektorraum, mit dem wir hier arbeiten, hängt von den Größen unserer Gewichtsmatrizen ab.

Anstatt jetzt bei der linearen Algebra zu bleiben, fahren wir mit der allgemeinen Idee der Regularisierung fort. Da Normen ein grundlegendes Konzept der linearen Algebra sind, findest du im Internet eine Menge Informationen dazu falls du sie besser verstehen willst.

Um es zu vereinfachen, solltest du vorerst einfach wissen, dass die Norm für jede unserer Gewichtsmatrizen nur eine positive Zahl sein wird.

Stell dir vor, dass v ein Vektor in einem Vektorraum ist. Die Norm von v ist $||v||$, und dass $||v| \geq 0$ sein muss.

28.1.3 HINZUFÜGEN DES TERMS ZU DEM LOSS

Lass uns nun sehen, wie die L2 Regularisierung aussieht.
Wir haben:

$$loss + \left(\sum_{j=1}^{n} \left|\left|\omega^{[j]}\right|\right|^2\right) * \frac{\lambda}{2m}$$

In der Tabelle unten kannst du die Definition für die Variablen einsehen

Variable	Definition
n	Anzahl der Layer
$\boldsymbol{\omega}^{[j]}$	Gewichtsmatrix für den j'ten Layer
m	Anzahl der Inputs
$\boldsymbol{\lambda}$	Rugulierungsparameter

Der Term λ heißt Regulierungsparameter und ist ein weiterer Hyperparameter, welchen wir uns aussuchen, testen und dann tunen (also anpassen) müssen, um die richtigen/besten Werte für unser spezielles Netz zu finden.

Zusammengefasst ist also die Regularisierung eine Technik, die große Gewichte abschwächt. Sie wird implementiert einfach durch die Addition eines Terms zu unserer bereits existierenden loss function.

28.2 DER EINFLUSS VON REGULARISIERUNG

Wie hilft uns also die Regularisierung?

Nun, am Beispiel der L2-Regularisierung würde es, wenn wir λ zu groß setzen würden, das Netz dazu verleiten, die Gewichte nahe Null zu setzen, weil es das Ziel von SGD ist, die loss function zu minimieren. Erinnere dich daran, dass unsere originale loss function nun mit der Summe der quadrierten Matrix Normen summiert wird.

$$\sum_{j=1}^{n} \left|\left|\omega^{[j]}\right|\right|^2$$

Was dann mit

$$\frac{\lambda}{2m}$$

Multipliziert wird.

Wenn λ also zu hoch ist, dann wird der Term $\frac{\lambda}{2m}$ auch relativ groß sein. Wenn wir dann die Summe der quadrierten Normen mit diesem Term multiplizieren, dann ist das Product (je nachdem wie groß die Gewichte sind) auch relativ groß. Das bedeutet, dass das Netz versuchen wird, die Gewichte klein zu halten damit der Wert dieser Gesamten Funktion relativ klein bleibt, um den loss zu minimieren.

Intuitiv könnten wir denken, dass diese Technik vielleicht die Gewichte so nahe an Null setzen wird, dass sie im Grunde genommen die Auswirkungen einiger unserer Schichten auf Null reduzieren könnte. Wenn das der Fall ist, dann würden sie unser Netz konzeptionell vereinfachen, wodurch die Komplexität des Netzes geringer wird, was wiederum die Varianz und das Overfitting reduzieren könnte.

29. BATCH SIZE

In diesem Kapitel wirst du lernen, was es bedeutet, eine Batch Size zu definieren, und wie man die Batch Size für ein Netz in Keras angibt.

In dem Kapitel 7.2.2 (beziehungsweise in 7.3)sahen wir, dass wenn wir ein Netz trainieren, die Batch Size spezifizieren müssen.

29.1 INTRODUCING BATCH SIZE

Vereinfacht ausgedrückt ist die Batch Size die Anzahl der Trainingsdaten, die auf einmal durch das Netz gegeben wird. Beachte, dass ein Batch im Allgemeinen auch als Mini-Batch referenziert wird.

Erinnere dich, dass eine Epoche ein einziger Durchgang über das gesamte Trainingsset ist. Die Batch Size und eine Epoche sind also nicht das gleiche. Lass uns das an einem Beispiel verdeutlichen

29.1.1 BATCHES IN EINER EPOCHE

Lass uns annehmen, dass wir 1000 Bilder von Hunden haben, mit denen wir unser Netz trainieren möchten, mit dem Ziel verschiedene Hunderassen zu erkennen. Nehmen wir an, wir definieren eine Batch-Size von 10. Das bedeutet, dass 10 Bilder von Hunden als eine Gruppe (also als ein Batch) auf einmal an das Netz übergeben werden.

Dadurch, dass eine Epoche ja einen gesamten Durchlauf der Trainingsdaten beschreibt, besteht eine Epoche aus 100 Batches. `1000 Bilder / 10 Batches = 100 Batches/Epoche`

Wir wissen also nun, was ein Batch ist, aber was bringt das? Warum geben wir nicht ein Element nach dem anderen an das Netz, anstatt sie in Gruppen von Batches zu unterteilen?

29.1.2 WARUM BENUTZEN WIR BATCHES?

Zum einen wird unser Netz jede Epoche umso schnelle abgeschlossen haben, umso größer die Batch Size ist. Das liegt daran, dass – abhängig von den Systemressourcen – unser Computer in der Lage sein kann, mehr als ein einziges Element gleichzeitig zu verarbeiten.

Der Kompromiss besteht jedoch darin, dass, selbst wenn unser Rechner große Batches verarbeiten kann, sich die Qualität des Netzes verschlechtern kann, wenn wir den Batch größer einstellen, was letztlich dazu führen kann, dass das Netz nicht in der Lage ist, Daten, die es noch nie gesehen hat, zu generalisieren und richtig verarbeiten.

Im Allgemeinen ist die Batch Size ein weiterer Hyperparameter, welchen wir anhand der Leistung unseres Netzes anpassen und testen müssen. Dieser Parameter muss auch im Hinblick darauf getestet werden, wie unser Rechner in Bezug auf Ressourcennutzung bei der Verwendung verschiedener Batch Sizes funktioniert.

Wenn wir unsere Batch Size zu einer relativ großen Zahl setzen, sagen wir 100, hat unser Computer eventuell nicht genug Rechenleistung, um alle 100 Bilder parallel zu verarbeiten. In so einem Fall sollten wir die Batch Size verringern.

29.1.3 MINI-BATCH GRADIENT DESCENT

Beachte außerdem, dass bei der Verwendung von Mini-Batch Gradient Descent, der normalerweise von den meisten Neuronalen Netzwerk-APIs wie Keras standardmäßig verwendet wird, das Gradientenupdate auf einer Pro-Batch Basis erfolgt.

Anders ist es bei Stochastic Gradient Descent (SGD), welcher Gradientenupdates per Beispiel implementiert. Batch Gradient Descent updatet die Gradienten auf einer Per-Epochen-Basis.

29.2 ARBEITEN MIT DER BATCH SIZE IN KERAS

Wir arbeiten hier mit dem gleichen Netz, dass wir schon in einigen Kapiteln verwendet haben. Das ist einfach ein willkürliches Sequential Model.

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import Activation

from tensorflow.keras import regularizers


model = Sequential([

    Dense(16, input_shape=(1,), activation = 'relu'),

    Dense(32, activation = 'relu', kernel_regularizer =
regularizers.l2(0.01)),

    Dense(2, activation = 'sigmoid')

])
```

Konzentrieren wir unsere Aufmerksamkeit auf die Stelle, an der wir model.fit() aufrufen. Wir wissen, dass dies die Funktion ist, die wir aufrufen, um unser Modell zu trainieren, und wir haben dies in Kapitel 7.3 schon in Aktion gesehen.

```
model.fit(
    scaled_train_samples,
    train_labels,
    validation_data = valid_set,
    batch_size = 10,
    epochs = 20,
    shuffle = True,
    verbose = 2
)
```

Diese fit() Funktion akzeptiert einen Parameter namens batch_size. Das ist die Stelle, an der wir unsere Batch Size für das Training definieren. In diesem Beispiel haben wir sie willkürlich auf 10 gesetzt.

Jetzt, während des Trainings dieses Netzes, werden wir 10 Trainingsdatenelemente auf einmal an das Netz geben, bis wir alle Trainingsdaten 1x verwendet haben. Dadurch schließen wir eine Epoche ab. Dieser Vorgang passiert jetzt für jede Epoche 1x.

30. FINE-TUNING NEURALER NETZE

In diesem Kapitel geht es darum, was fine-tuning ist und wie wir daraus einen Vorteil ziehen können, wenn wir ein ANN bauen und Trainieren.

30.1 INRODUCING FINE-TUNING UND TRANSFER LEARNING

Fine-tuning ist eng mit dem Begriff transfer learning verknüpft. Transfer learning entsteht, wenn wir das Wissen, das bei der Lösung eines Problems gewonnen wurde, auf ein neues, aber verwandtes Problem anwenden.

Zum Beispiel könnte das Wissen, dass durch das Erlernen des Erkennens von Autos gewonnen wurde, bei dem Erkennen von Lastwagen angewendet werden.

Fine-tuning ist eine Möglichkeit, transfer learning anzuwenden oder zu nutzen. Insbesondere ist das fine-tuning ein Prozess, bei dem ein Modell, das bereits für eine bestimmte Aufgabe trainiert wurde, dann so abgestimmt oder optimiert wird, dass es eine zweite, ähnliche Aufgabe erfüllt.

30.1.1 WARUM FINE-TUNING BENUTZEN?

Unter der Annahme, dass die ursprüngliche Aufgabe der neuen Aufgabe ähnlich ist, können wir einen Vorteil daraus ziehen, was das bereits entworfene und trainierte ANN gelernt hat, anstatt es von Grund auf neu entwickeln zu müssen.

Wenn wir ein Netz von Grund auf entwickeln, müssen wir normalerweise viele Lösungsansätze durch Trial-and-Error ausprobieren.

Zum Beispiel müssen wir aussuchen, wie viele Layer wir verwenden, was für Layerarten wir verwenden, in welcher Reihenfolge die Layer sein sollen, wie viele Nodes in jedem Layer sein sollten, wie viel Regularisierung wir verwenden wollen, welchen Wert unsere Lernrate haben soll usw.

- Anzahl der Layer
- Art der Layer
- Reihenfolge der Layer
- Anzahl der Nodes pro Layer
- Höhe der Regularisierung
- Lernrate

Der Aufbau und die Validierung unseres Netzes können für sich genommen schon eine gewaltige Aufgabe sein, je nachdem, auf welchen Daten wir es trainieren.

Das macht den fine-tuning Ansatz so attraktiv. Wenn wir schon ein trainiertes Netz finden, dass bereits eine Aufgabe gut erfüllt und diese Aufgabe wenigstens im Ansatz unserer Aufgabe ähnelt, dann können wir aus allem, was dieses Netz bereits gelernt hat nutzen und auf unsere spezifische Aufgabe anwenden.

Wenn die beiden Aufgaben unterschiedlich sind, dann gibt es natürlich einige Informationen, die das Netz gelernt hat, die möglicherweise nicht auf unsere Aufgabe zutreffen, oder es gibt neue Informationen, die das Modell aus den Daten bezüglich der neuen Aufgabe lernen muss, die nicht aus der vorherigen Aufgabe gelernt wurden.

Zum Beispiel wird ein auf Autos trainiertes Netz noch nie eine Lkw-Ladefläche gesehen haben, so dass dieses Merkmal etwas Neues ist, über das das Netz lernen müsste. Denk jedoch über alles nach, was unser Netz für die Nachrüstung von Lastwagen von dem ursprünglich an Autos ausgebildeten Netz verwenden könnte.

Dieses bereits trainierte Modell hat gelernt, Kanten, Formen und Texturen zu verstehen, und objektiver, Scheinwerfen, Türgriffen, Windschutzscheiben, Reifen usw.

Das klingt gut, aber wie können wir diese Technik implementieren?

30.2 WIE KANN FINEGETUNED WERDEN?

Zurück zu unserem Beispiel: Wenn wir ein Netz haben, dass bereits auf die Erkennung von Autos trainiert wurde und wir wollen dieses Netzt fine-tunen, um Lkws zu erkennen, dann importieren wir zuerst das originale Netz, das für die Autos verwendet wurde.

Einfachheitshalber sagen wir, dass wir den letzten Layer unseres Netzes löschen. Der letzte Layer war dazu zuständig, zu klassifizieren, ob ein Bild ein Auto ist oder nicht. Nach dem löschen wollen wir einen neuen Layer hinzufügen, welcher klassifizieren soll, ob ein Bild ein Lkw ist oder nicht.

Bei einigen Problemen möchten wir vielleicht mehr als nur die letzte Schicht entfernen, und wir möchten vielleicht mehr als nur eine Schicht hinzufügen. Das hängt davon ab, wie ähnlich die zu lösende Aufgabe ist.

Layers am Ende unseres Netzes haben möglicherweise Merkmale gelernt, die sehr spezifisch für die ursprüngliche Aufgabe sind, während die Schichten am Anfang normalerweise allgemeinere Merkmale wie Kanten, Formen und Texturen kennen.

Nachdem wir die Struktur des bereits existierenden Netzes verändert haben, wollen wir die Layers „einfrieren“. Das nennt man Freezing weights.

30.2.1 FREEZING WEIGHTS

Mit Freezing meinen wir, dass wir die Gewichte für einen Layer während des Trainings nicht updaten wollen. Wir wollen alle diese Gewichte so beibehalten, wie sie nach dem Training an der ursprünglichen Aufgabe waren. Wir wollen nur die Gewichte in unseren neuen oder modifizierten Layers updaten.

Nachdem wir das getan habe, müssen wir nur mehr unser neues Netz an unseren neuen Trainingsdaten trainieren. Noch einmal: Während dieses Trainingsprozesses werden die Gewichte von allen Layers, die wir übernommen haben, gleichbleiben und nur die neuen Gewichte werden aktualisiert.

31. BATCH NORMALIZATION

In diesem Kapitel geht es um Batch Normalization, welche auch als Batch norm bekannt ist, und wie sie auf das Training von ANNs angewendet wird. Außerdem werden wir sehen wie wir Batch norm in Keras implementieren

31.1 NORMALIZATION TECHNIKEN

Bevor wir zu den Details von Batch Normalization kommen, wollen wir uns zunächst kurz über reguläre Normalisierungstechniken reden.

$$z = \frac{x - mean}{std}$$

Im Allgemeinen wollen wir beim Training eines neuronalen Netzes unsere Daten in der Vorverarbeitung in irgendeiner Weise vorzeitig normalisieren oder standardisieren. Dies ist der Schritt, in dem wir unsere Daten für das Training vorbereiten.

Normalisierung und Standardisierung haben das gleiche Ziel, nämlich die Daten zu transformieren, um alle Datenpunkte gleich zu skalieren.

Ein typischer Normalisierungsprozess besteht darin, numerische Daten auf eine Skala von Null bis Eins herunter zu skalieren, und ein typischer Standardisierungsprozess besteht darin, den Mittelwert des Datensatzes von jedem Datenpunkt zu subtrahieren und dann diese Differenz durch die Standardabweichung des Datensatzes zu dividieren.

Dadurch werden die standardisierten Daten gezwungen, einen Mittelwert von Null und eine Standardabweichung von 1 anzunehmen. In der Praxis wird dieser Standardisierungsprozess oft auch nur als Normalisierung bezeichnet.

31.2 DER SINN HINERT NORMALISIERUNGSTECHNIKEN?

Im Allgemeinen läuft dies alles darauf hinaus, unsere Daten auf eine Art bekannte oder Standardskala zu setzen. Warum tun wir das?

Nun, würden wir die Daten nicht normalisieren, können wir uns vorstellen, dass wir einige numerische Datenpunkte in unserem Datensatz haben, die sehr hoch sein könnten, und andere, die sehr niedrig sein können.

Zum Beispiel: Angenommen wir haben Daten über die Anzahl der Kilometer, die Personen in den letzten 5 Jahren mit dem Auto gefahren sind. Wir haben einige die die 100 000 Kilometer gefahren sind und andere, die nur 1000 Kilometer gefahren sind. Diese Daten haben eine relativ große Bandbreite und liegen nicht unbedingt im gleichen Maßstab.

Darüber hinaus kann jedes der Merkmale für jedes unserer Trainingsbeispiele ebenfalls stark variieren. Wenn wir ein Merkmal haben, das dem Alter einer Person entspricht, und das andere Merkmal der Anzahl der

Kilometer, die diese Person in den letzten fünf Jahren mit dem Auto gefahren ist, dann können wir wiederum feststellen, dass diese beiden Daten, Alter und gefahrene Kilometer, nicht auf der gleichen Skala liegen.

Die größeren Datenpunkte in diesem nichtnormalisierten Datensatz können in den Netz Instabilität verursachen, weil die relativ großen Daten durch die Layer des Netzes nach unten abfallen können, was unausgeglichene Gradienten zur folge haben kann, was sich wiederum in dem exploding Gradienten Problem (Kapitel 23.3) auswirken kann.

Vorerst sei im Klaren darüber, dass diese unausgeglichenen, nichtnormalisierten Daten Probleme in unserem Netz verursachen können, die das Training drastisch erschweren. Darüber hinaus können nichtnormalisierte Daten unser Trainingstempo erheblich beeinträchtigen.

Wen wir unsere Inputs jedoch normalisieren, setzen wir jedoch alle unsere Daten auf die gleiche Skala, um die Trainingsgeschwindigkeit zu erhöhen und um das gerade besprochene Problem zu vermeiden, weil wir diese relativ große Spanne zwischen den Datenpunkten nicht haben.

Das ist gut, allerdings gibt es ein weiteres Problem, welches selbst bei normalisierten Daten auftritt. Wir wissen, wie die Gewichte in unserem Netz während des Trainings über jede Epoche durch den Prozess des stochastischen Gradientenabstiegs aktualisiert werden.

31.2.1 GEWICHTE, DIE DIE WAAGE KIPPEN

Was ist, wenn während des Trainings eines der Gewichte drastisch höher wird als die anderen Gewichte?

Nun, dieses große Gewicht wird dann dazu führen, dass der Output des entsprechenden Neurons extrem groß ist, und dieses Ungleichgewicht wird sich wiederum weiter durch das Netz kaskadieren (= aufschaukeln) und Instabilität verursachen. An dieser Stelle kommt Batch Normalisierung ins Spiel.

31.3 ANWENDEN DER BATCH NORM AUF EINEN LAYER

Batch norm wird auf von uns ausgewählte Layer in unserem Netz angewandt.

Wenn Batch norm zu einem Layer hinzugefügt wird, dann ist das erste, was Batch norm tut, das Normalisieren des Outputs von der Aktivierungsfunktion. Erinnere dich an das Kapitel 5 über Aktivierungsfunktionen daran, dass die Ausgabe eines Layers an eine Aktivierungsfunktion übergeben wird, die die Ausgabe in Abhängigkeit von der Funktion selbst in irgendeiner Weise transformiert, bevor sie an den nächsten Layer als Input weitergegeben wird.

Nach der Normalisierung des Outputs von der Aktivierungsfunktion multipliziert Batch norm diesen normalisierten Output mit einem willkürlichen Parameter und addiert danach einen weiteren zufälligen Parameter zu diesem Produkt

Schritt	Ausdruck	Beschreibung
1	$z = \frac{x - mean}{std}$	Normalisiert den Output x einer Aktivierungsfunktion
2	$z * g$	Multipliziert den normalisierten Output z mit einem beliebigen Parameter g
3	$(z * g) + b$	Addiert den beliebigen Parameter b zu dem resultierenden Produkt (b*g)

31.3.1 TRAINIERBARE PARAMETER

Durch diese Berechnung mit den zwei beliebigen Werten bekommen wir eine neue Standardabweichung und einen neuen Mittelwert der Daten. Diese zwei beliebig gesetzten Parameter, g und b, sind trainierbar, was bedeutet, dass sie während des Trainings gelernt und optimiert werden.

Dieser Prozess sorgt dafür, dass die Gewichte innerhalb des Netzwerks nicht durch extrem hohe oder niedrige Werte unausgewogen werden, da die Normalisierung in den Gradientenprozess einbezogen wird.

Diese Ergänzung unseres Netzes um die Batch norm kann die Geschwindigkeit, mit der das Training stattfindet, erheblich steigern und die Fähigkeit, den Trainingsprozess durch übergroße Gewichte übermäßig zu beeinflussen, verringern.

Als wir über die Normalisierung unserer Inputdaten vor dem Training sprachen, haben wir verstanden, dass diese Normalisierung mit den Daten geschieht, bevor sie and die Eingabeschicht weitergegeben werden.

Mit Batch norm, können wir auch die Outputdaten von der Aktivierungsfunktion für individuelle Layer in unserem Netz normalisieren.

31.3.2 NORMALISIERUNG PER BATCH

Alles was wir gerade über Batch Normalization besprochen haben, tritt auf einer Per-Batch Basis auf, deshalb der Name Batch norm.

Diese Batches werden durch die Batch Size bestimmt, die wir für unser Netz bestimmen.

Da wir nun verstehen, was Batch norm ist, lass uns sehen, wie wir Batch norm zu einem Netz mit Keras hinzufügen können.

31.4 BATCH NORM MIT KERAS

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Activation, BatchNormalization


model = Sequential([

    Dense(16, input_shape=(1,5), activation = 'relu'),

    Dense(32, activation = 'relu'),

    BatchNormalization(axis=1),

    Dense(2, activation = 'softmax')

])
```

Hier haben wir ein Netz mit zwei Hidden Layers mit 16 und 32 Nodes, die beide relu() als Aktivierungsfunktion verwenden, und einen Output Layer mit zwei Output Kategorien, welcher softmax() als Aktivierungsfunktion verwendet.

Das einzig neue and diesem Netz ist die Zeile zwischen dem letzten Hidden Layer und dem Output Layer.

```
BatchNormalization(axis=1)
```

So wird die Batch Normalization in Keras spezifiziert. Nach der Schicht, für die wir den Ativierungsoutput normalisieren wollen, geben wir ein BatchNormalization Objekt. Dazu müssen wir BatchNormalization von tensorflow.keras.layers importieren.

Der einzige Parameter den wir für BatchNormalization spezifizieren, ist der axis Parameter, welcher aussagt, welche Achse der Daten Normalisiert werden soll.

Es gibt noch einige andere Parameter, die wir optional spezifizieren können, wie beta_initializer und gamma_initializer. Diese sind die Initialisierer für die willkürlich festgelegten Parameter, die wir erwähnt haben, als wir beschrieben haben, wie die Batch Norm funktioniert.

Diese werden standardmäßig von Keras auf 0 und 1 gesetzt, wir können diese aber optional verändern.

Ω. GLOSSAR

Nr.	Ausdruck	Beschreibung
1	Aktivierungsfunktion	Eine Aktivierungsfunktion ist eine Funktion, die die gewichtete Summe der Inputs entgegennimmt, diese Transformiert und als Output weitergibt
2	Autoencoder	Ein Autoencoder ist ein ANN, welches einen Input nimmt und dann eine Rekonstruktion dieses Inputs ausgibt
3	Batch norm/Batch Normalization	Die Daten werden alle auf eine Standardskala gemappt
4	Clustering	Beschreibt die Teilung von Daten in Gruppen
5	Data Augmentation	Data Augmentation beschreibt die Erschaffung von neuen Daten durch Änderung bestehender.
6	Dot Product	Beschreibt das Skalarprodukt und wird in CNNs angewandt
7	Dropout	Zufällig wird ein bestimmter Prozentsatz von Neuronen in jeder Trainingsiteration "getötet". Dadurch wird sichergestellt, dass einige der gelernten Informationen zufällig entfernt werden, wodurch das Risiko von Overfitting reduziert wird.
8	Error/loss/Fehler	Definiert, wie weit die tatsächliche Ausgabe des aktuellen ANNs von der korrekten Ausgabe entfernt ist. Beim Trainieren des Netzes ist es das Ziel, den Fehler zu minimieren und die Ausgabe so nah wie möglich an den korrekten Wert zu bringen
9	error/loss/Fehler	Mit einer Ableitung kann man sich die Steigung einer Funktion/eines Graphen, an einem beliebigen Punkt ausrechnen
10	Feature Maps	Nachdem ein Filter in einem CNN den gesamten Input konvoliert hat, bleibt uns eine neue Darstellung des Inputs, die jetzt im Output channel gespeichert ist. Das nennt man eine Feature Map
11	Flattening	Bevor der Output eines Convolutional Layers an einen Dense Layer weitergegeben werden kann, muss er geflattet werden. Flatten beschreibt das Konvertieren von mehreren Dimensionen in eine Dimension
12	Forward Pass	Der Forward Path nimmt die Inputs, leitet sie durch das Netz und erlaubt jedem Neuron, auf einen Bruchteil der Eingabe zu reagieren. Die Neuronen erzeugen ihre Outputs und leiten sie an den nächsten Layer weiter, bis schließlich der Output-Layer erreicht ist
13	Freezing Weights	Freezing Weights sind Gewichte, die während des Trainings nicht geupdatet werden sollen
14	Gradient/Ableitung	Um die optimalen Gewichte für die Neuronen zu finden, führen wir Backpropagation durch, wobei wir von der Vorhersage des Netzwerks zu den Neuronen zurückgehen, die diese Vorhersage generiert haben. Die Backpropagation verfolgt die Ableitungen der Aktivierungsfunktionen in jedem aufeinanderfolgenden Neuron, um Gewichte zu finden, die den Loss auf ein Minimum bringen, wodurch die beste Vorhersage generiert wird. Dies ist

		ein mathematischer Prozess, der Gradientenabstieg (Gradient Descent) bezeichnet wird
15	Hyperparameter	Hyperparameter werden vor Beginn des Trainings festgelegt, und können während des Trainings nicht angepasst werden. In der Praxis muss man Probieren, welche Einstellungen der Hyperparameter am besten für sein Netz und Problem passt
16	Input Features	Sind die Merkmale der Trainingsdaten. Wenn man zum Beispiel ein Netz hat, dass Mann und Frau unterscheiden soll, könnte man als Input Features die Größe, das Gewicht und die Haarlänge haben
17	Kernel/Filter	Ein Filter ist eine relative kleine Matrix, die im Laufe des Trainings Kanten, Formen, Ecken oder andere Muster erkennen soll
18	lernfähige Parameter	Parameter, die während des Trainings optimiert werden
19	Neuron/Node/Perceptron	Die Grundeinheit eines NN. Akzeptiert einen Input und generiert einen Output
20	pseudo-labeling	Ein Neuronales Netz wird anhand von Daten trainiert, Labelt mit den gelernten Informationen neue Daten und lernt wiederum von diesen neu gelabelten Daten
21	Semi-Überwachtes Lernen	Semi-Überwachtes Lernen ist eine Mischung aus überwachtem und unüberwachtem Lernen. Hierbei haben wir gelabelte und ungelabelte Daten. Wird vor allem bei großen Datensätzen angewandt
22	Threshold	Beschreibt den Schwellenwert, wann ein Neuron feuert
23	Trainingssatz	Ein Datensatz von Inputs, der zum Training eines neuronalen Netzes verwendet wird. In der Norm sind die dazugehörigen Outputs bekannt
24	Überwachtes Lernen	Überwachtes Lernen beschreibt das Training an einem Datensatz, der vollständig gelabelt ist.
25	Unüberwachtes Lernen	Unüberwachtes Lernen beschreibt das Training an einem Datensatz, der nicht gelabelt ist.
26	Weight	Jedes Neuron wird mit einem Gewicht (in Form einer Zahl) versehen. Die Gewichte definieren zusammen mit der Aktivierungsfunktion den Output jeden Neurons.
27	Zero padding	Zero padding ist eine Technik, die in einem CNN erlaubt, die originale Input Size beizubehalten, indem am Rand der Matrix lauter 0en eingetragen werden