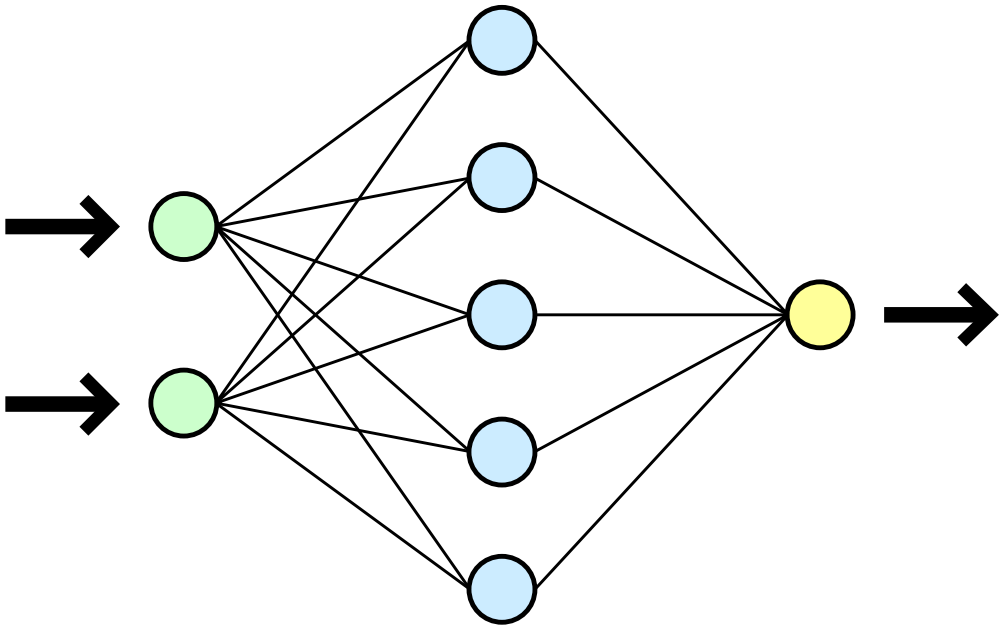
Machine learning

oder

„Des Gehirn Dings-do“

Theorie



Lugmayr Gabriel

2019

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" gelang 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 es nicht zu 100% gleich. Einige Stellen habe ich ausgelassen und andere genauer ausgeführt. Dieses „Skript" ist in Deutsch geschrieben, dennoch findest du hier im Kontext des maschinellen Lernens auch oft englische Fachbegriffe.

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:

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

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

Diese Wörter sind willkürlich von dem Entwickler gewählt. Wenn wir eine Liste von positiven und negativen 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!..."
    }
];
```

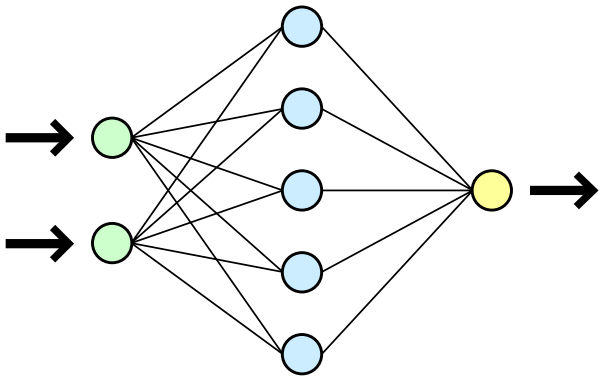
2. Deep Learning erklärt

2.1 Definition

Deep Learning ist ein Teilbereich des maschinellen Lernens, der Algorithmen verwendet, die von der Struktur und Funktion der

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.

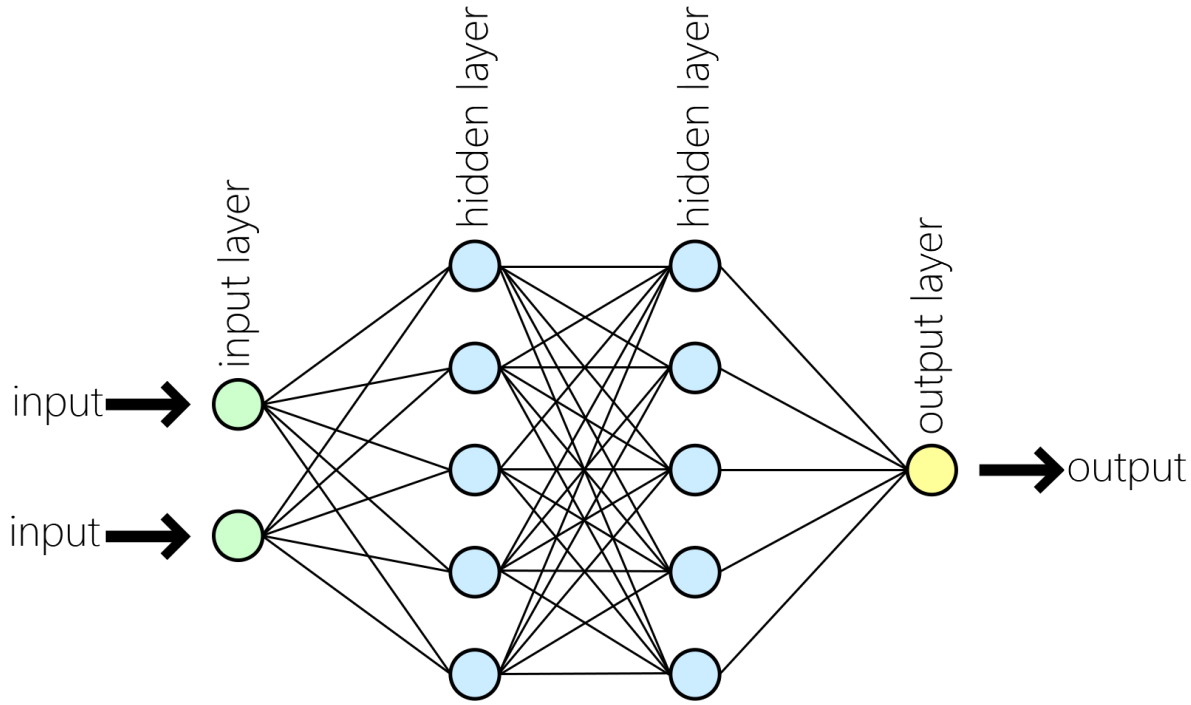


2.3 Das „Deep" in Deep Learning

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



2 Artificial Neural Network (ANN) erklärt

3.1 Definition

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

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

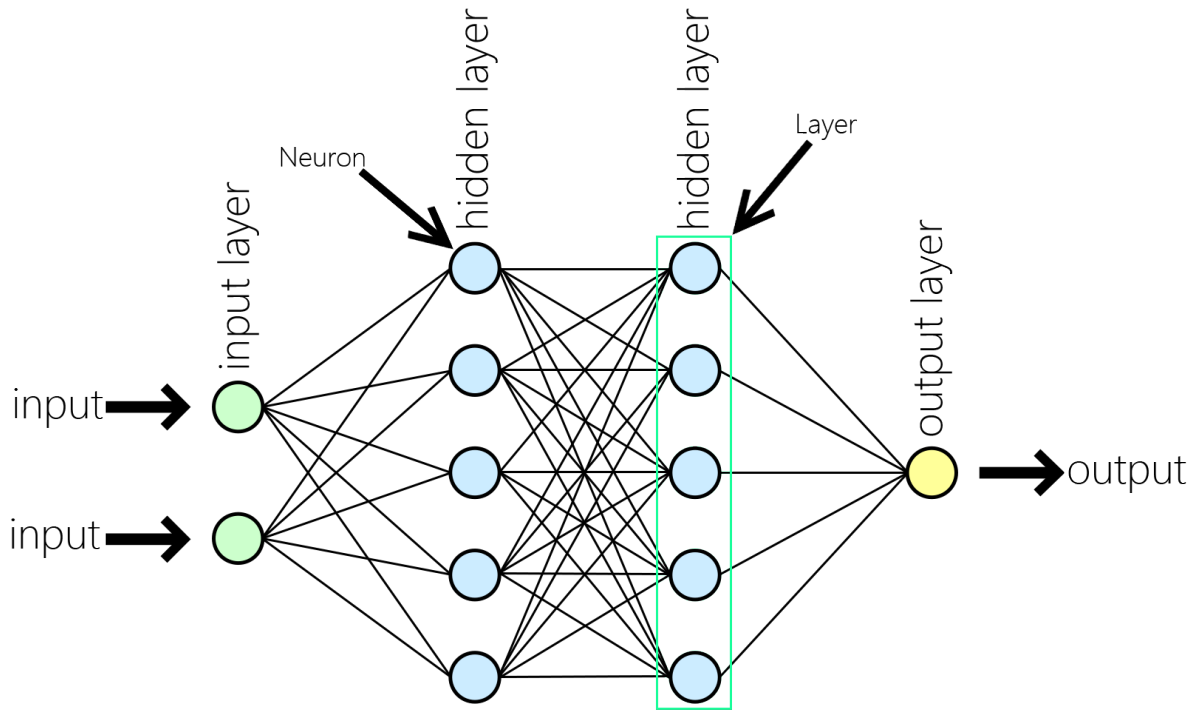
Verschiedene Layer führen verschiedene Arten von Transformationen an ihren Eingängen durch. Daten fließen durch das Netzwerk, beginnend bei dem Input Layer, durch die Hidden Layers und kommt schließlich zum Output Layer. Das nennt man „Forward pass" durch ein Netzwerk. Alle Layer zwischen Input und Output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

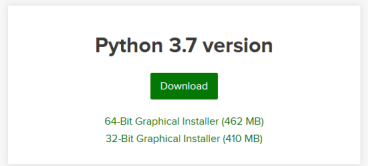
3.4.1 Keras Introduction

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 sequentielles Model erstellen. Keras definiert ein sequentielles Model als einen sequentiellen Stack von linearen Layers. Das ist, was wir erwartet haben, da wir gelernt haben das Neuronen in Layers organisiert sind.
Das Sequentielle Model ist Keras‘ Implementation von einem Artificial Neural network. So wird ein sehr simples sequentielles 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 das 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:

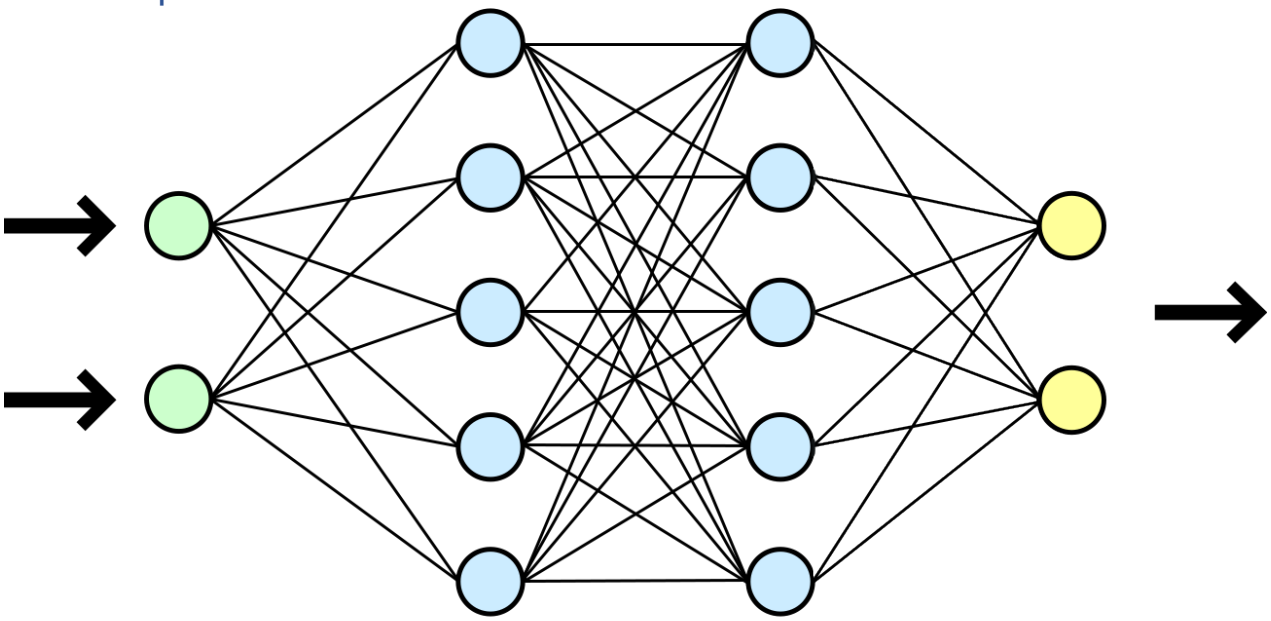
- Dense (fully connected) Layer
- Convolution Layer
- Pooling Layer
- Recurrent Layer
- Normalization Layer

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.

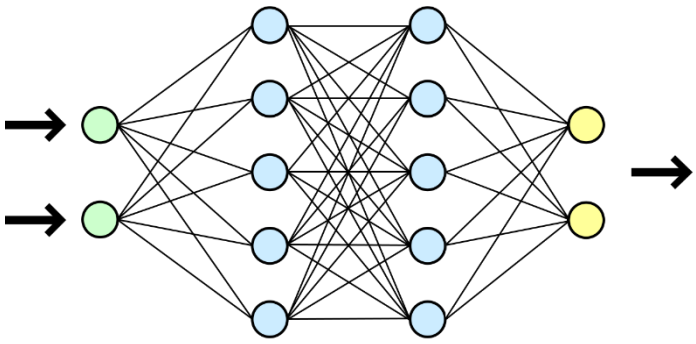
Für ein Beispiel aus dem Datensatz wird der gesamte Prozess von der Eingangsschicht bis zur Ausgangsschicht als Forward Pass durch das Netzwerk bezeichnet.

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

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 Was tut eine Aktivierungsfunktion?

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

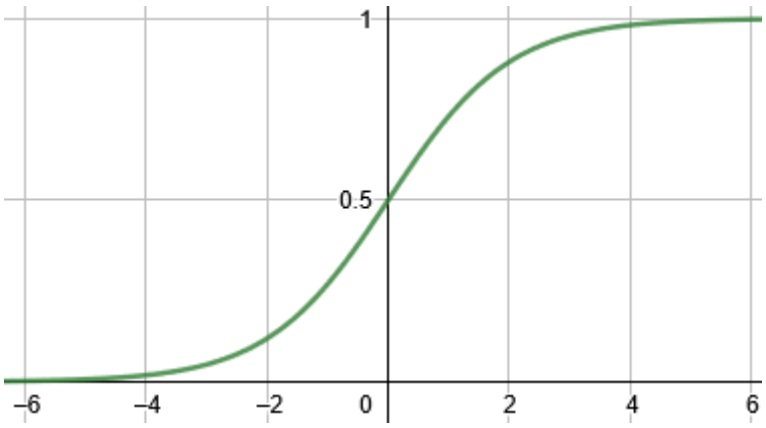
5.2.1 Sigmoid Aktivierungsfunktion

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

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

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 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

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

Verbindung zwischen Nodes mit einem willkürlichen Gewicht versehen ist. Während dem Training werden diese Gewichte iterativ geupdated und deren Optimalwert nähergebracht.

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!

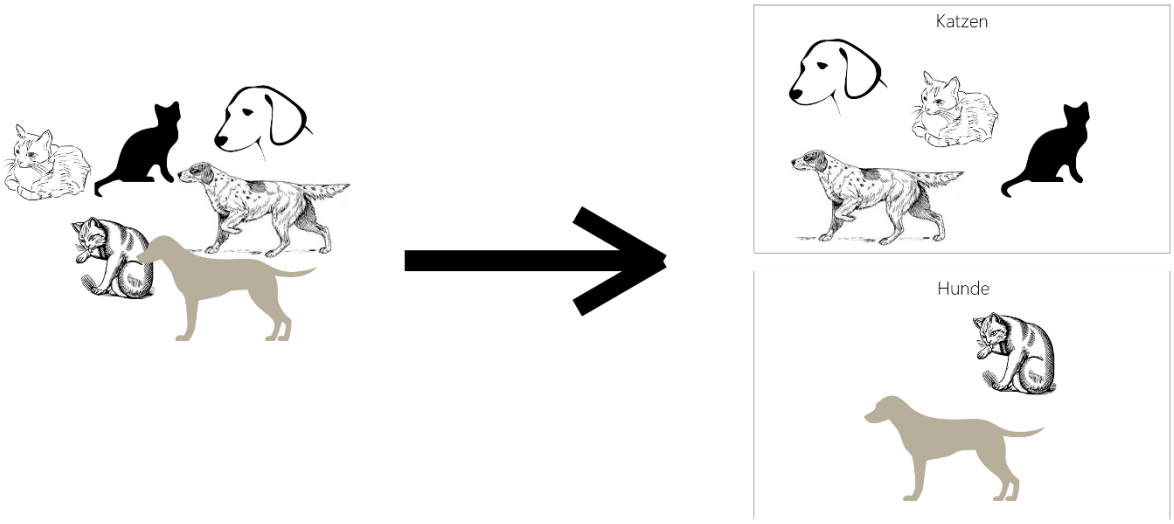
Der Loss ist der Fehler oder der Unterschied zwischen dem, was das Netz für das Bild vorhersagt und dem wahren Label des Bildes. SGD wird versuchen, diesen Fehler zu minimieren, um möglichst präzise Vorhersagen von dem Netz zu bekommen.

Nachdem wir unsere ganzen Daten durch unser Netz gejagt haben werden wir dieselben Daten immer wieder durchjagen. Dieser Prozess des

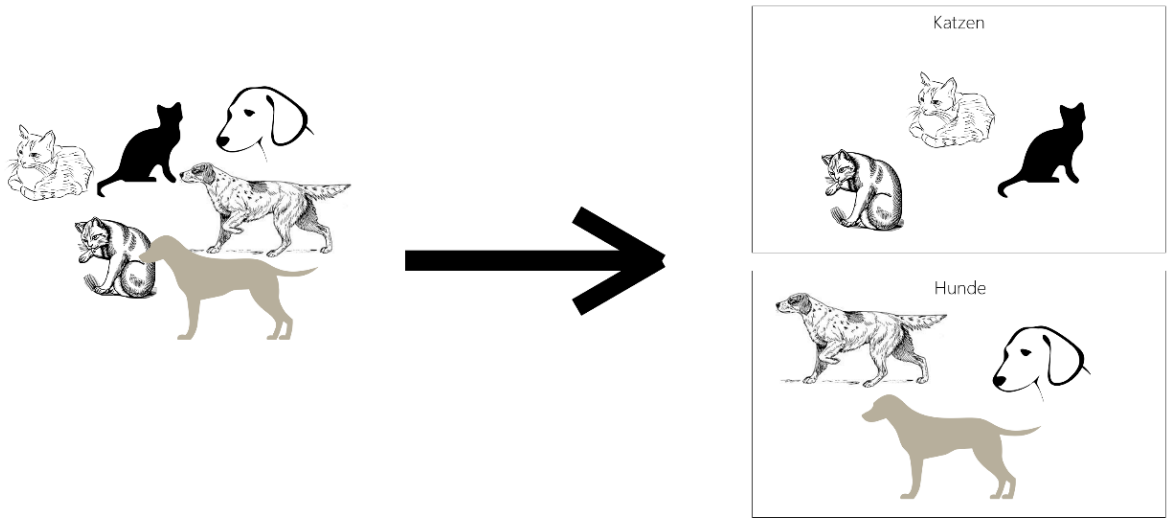
fortlaufenden senden der gleichen durch das Netz ist das Training. Während diesem Prozess wird das Netz lernen.

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

Untrainiertes Netz



Trainiertes Netz



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

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

durch das Netz gejagt haben, sprechen wir von einer abgeschlossenen Epoche.

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

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

rate (= Lernrate). Eine Lernrate ist 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([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1
,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 (12.2.4) 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.

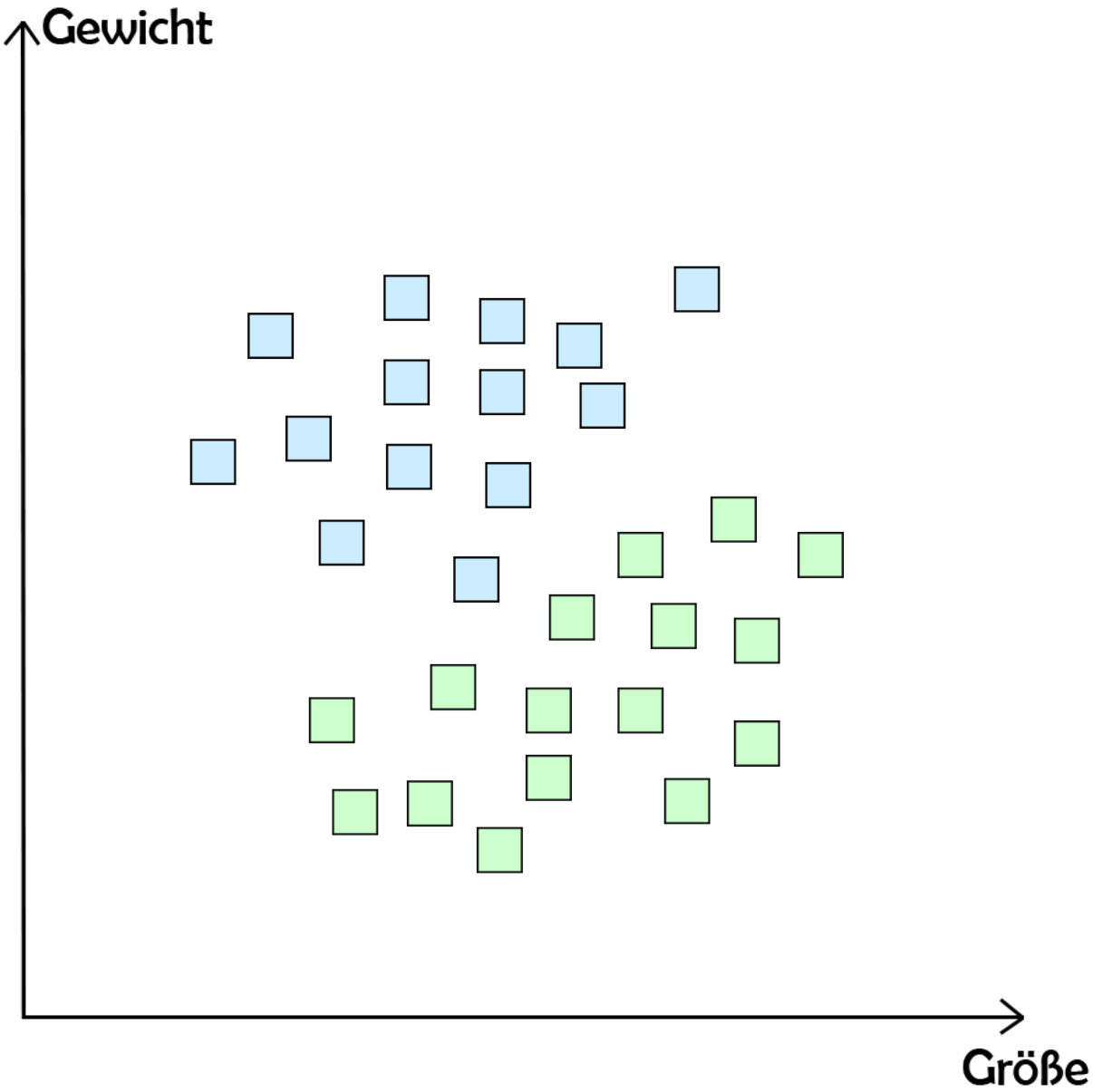
15.2.2 Beispiel

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:



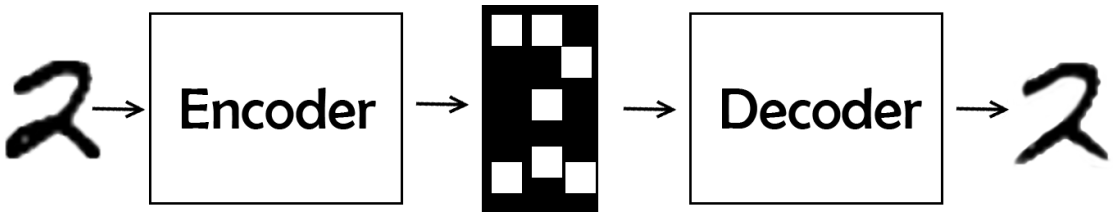
Nichts sagt uns hier explizit die Labels für diese Daten, aber wir können sehen, dass es hier zwei unterschiedliche Cluster gibt, und so können wir daraus schließen, dass diese zwei Cluster Männer und Frauen darstellt

15.2.2 Autoencoders

Beim unüberwachten Lernen werden auch Autoencoders eingesetzt.

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

Basierend auf allem, was wir bisher über neuronale Netze gelernt haben,

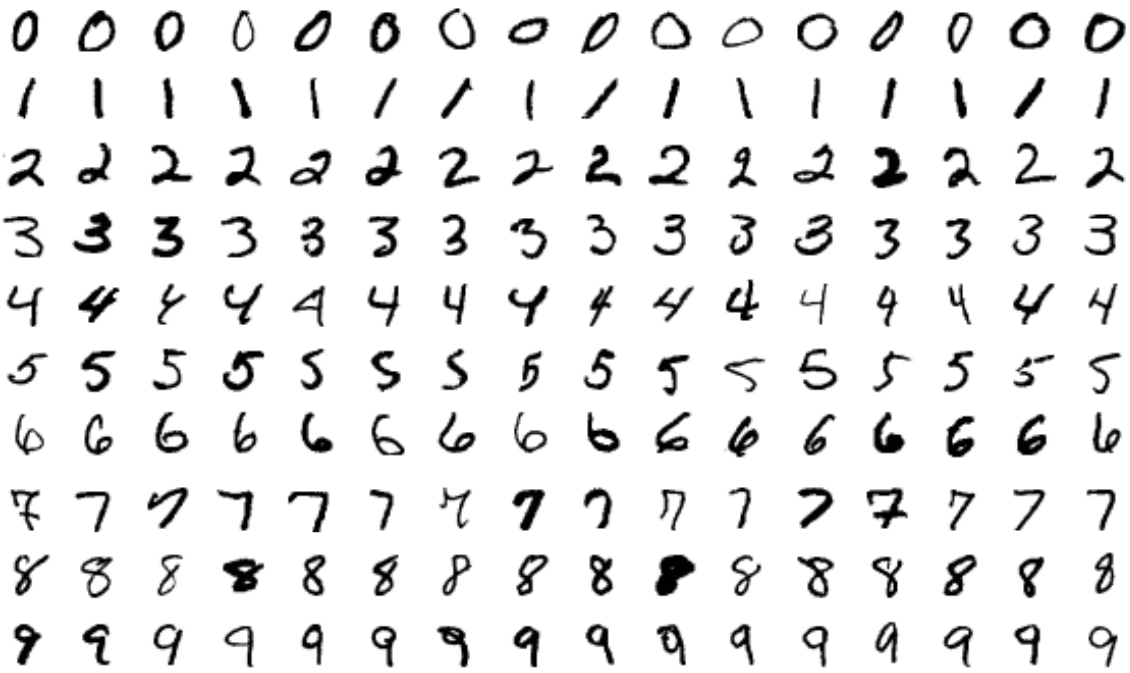


erscheint dies seltsam, aber lass mich das an einem Beispiel näher erklären.

(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.

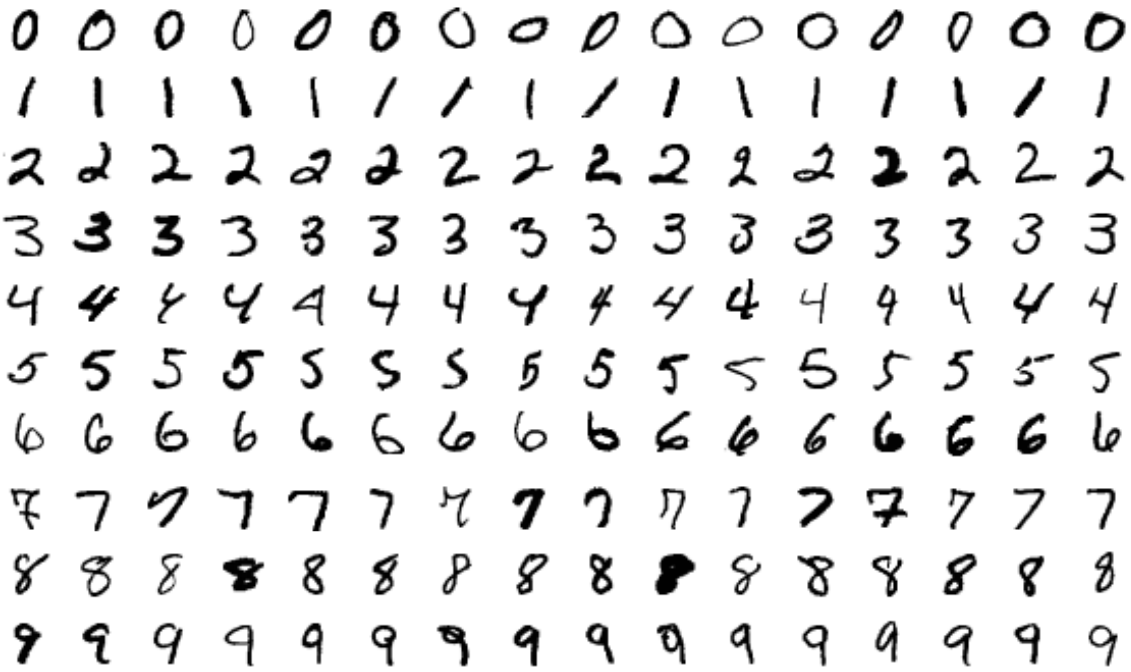
Je tiefer das Netzwerk ist, desto ausgefeilter werden die Filter. In späteren Layers können Filter statt einfachen Objekten komplexe Formen wie Ohren, Augen, Haare, Federn, Schuppen… erkennen.

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.

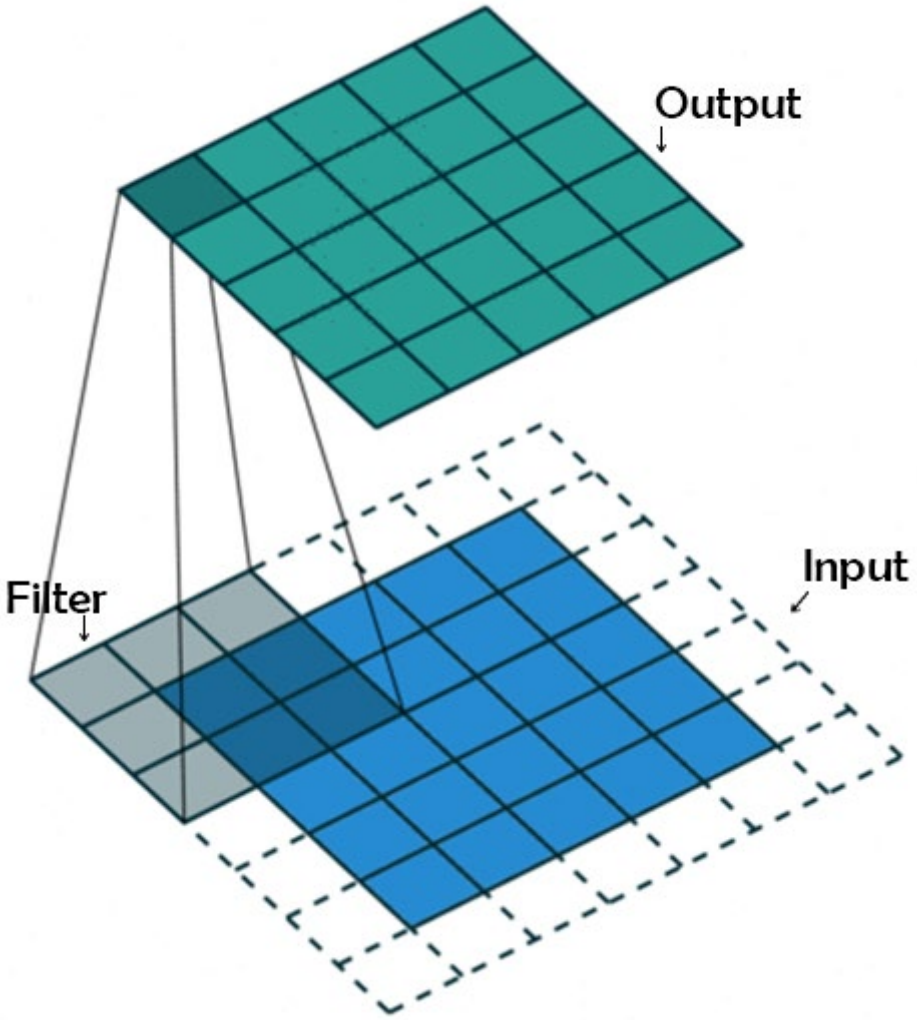
Die Anzahl der Filter bestimmt die Anzahl der Output channels

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 convoled 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:

$$a_{1,1}*b_{1,1}+a_{1,2}*b_{1,2}+\cdots+a_{3,3}*b_{3,3}$$

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.

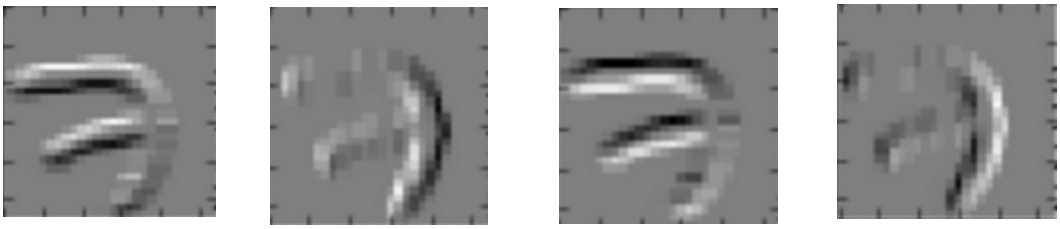
19.3 Input und Output channels

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

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

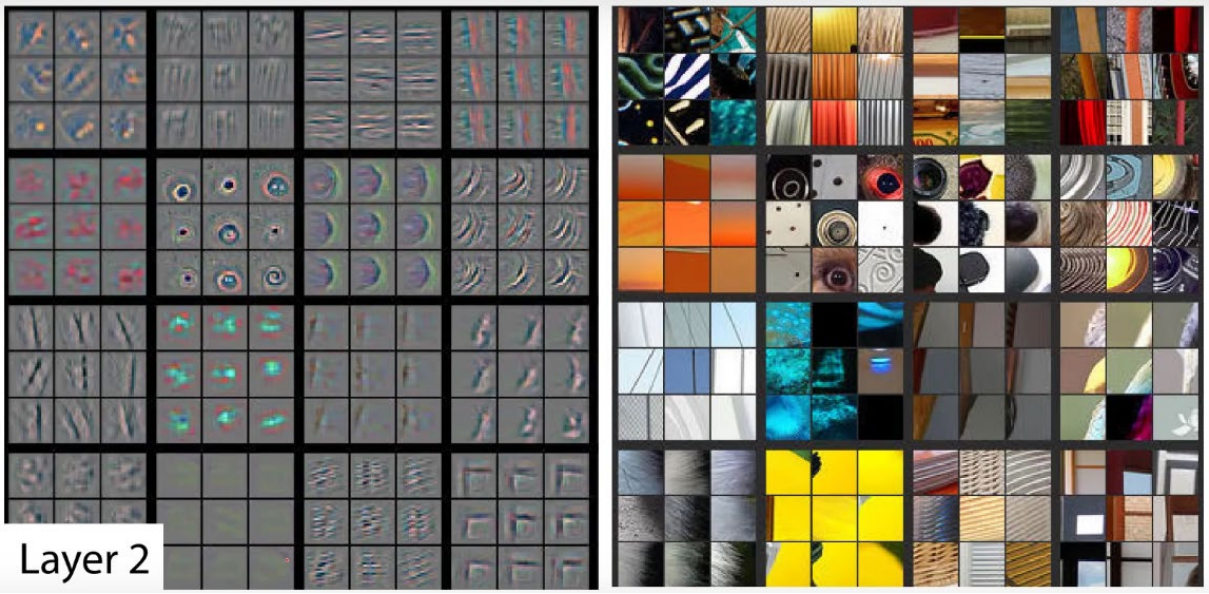
Wenn wir unser originales Bild der Sieben konvolieren, ist das in etwas das, was für jeden Filter als Ergebnis kommt:



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 Convolutions reduzieren die channel dimensions

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

1	0.3	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	0.0
0.7	0.1	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
0.6	0.3	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	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
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:

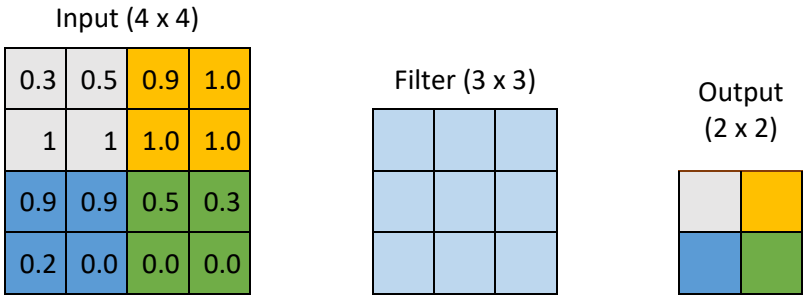
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	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:

$$(n - f + 1) \times (n-f + 1)$$

Wenn wir dies an unserem Beispiel anwenden kommen wir auf

$$(n-f+1) =$$

$$(4-3+2)=$$

$$2$$

➔ 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

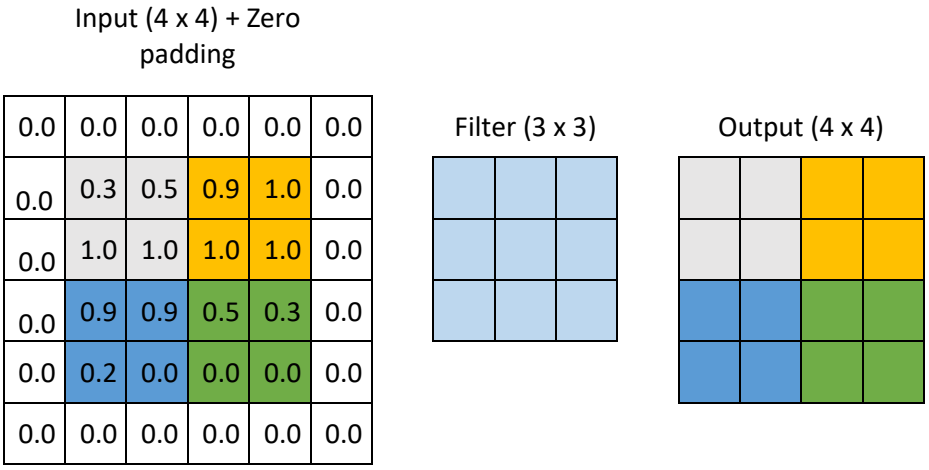
20.3 Zero padding

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