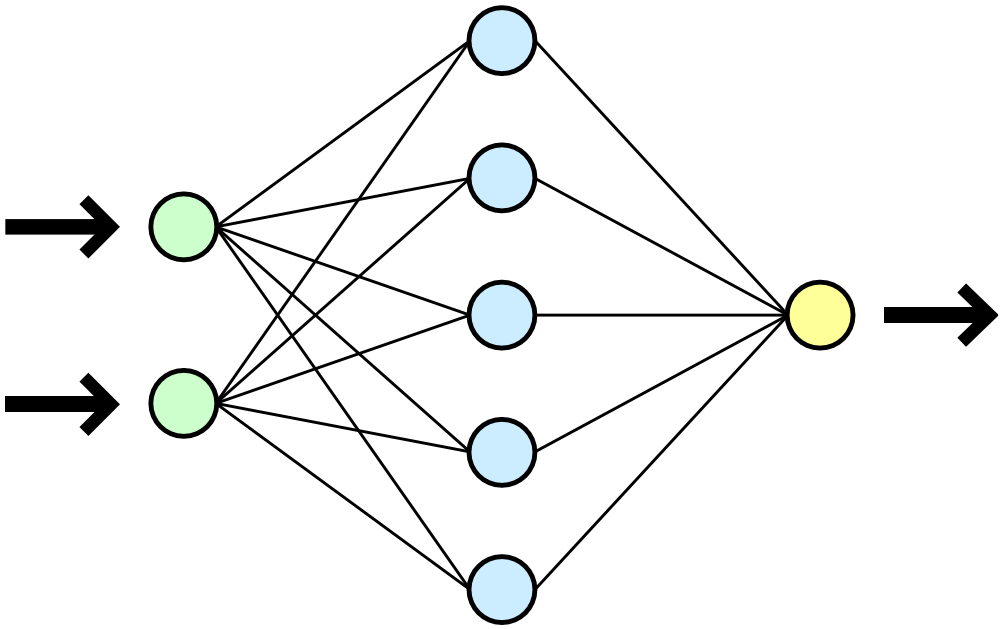
Machine learning

oder

„Des Gehirn Dings-do“

Allgemeines



Inhaltsverzeichnis

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 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 Emotionen 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.

Bsp:

```
//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 er weiß als positiv oder negativ klassifiziert werden.

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 positiv oder negativen Wörtern 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, dass 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.

Bsp:

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

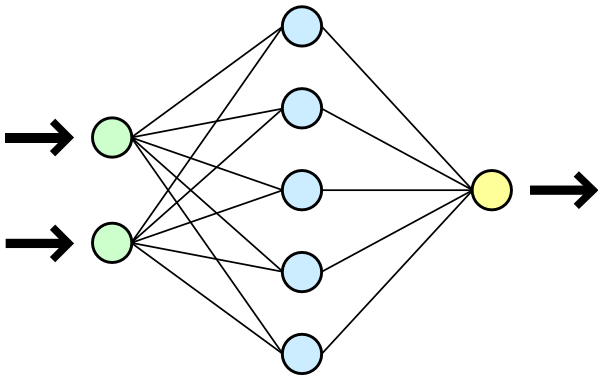
2 Deep Learning erklärt

2.1 Definition

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

2.2 Konzept

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

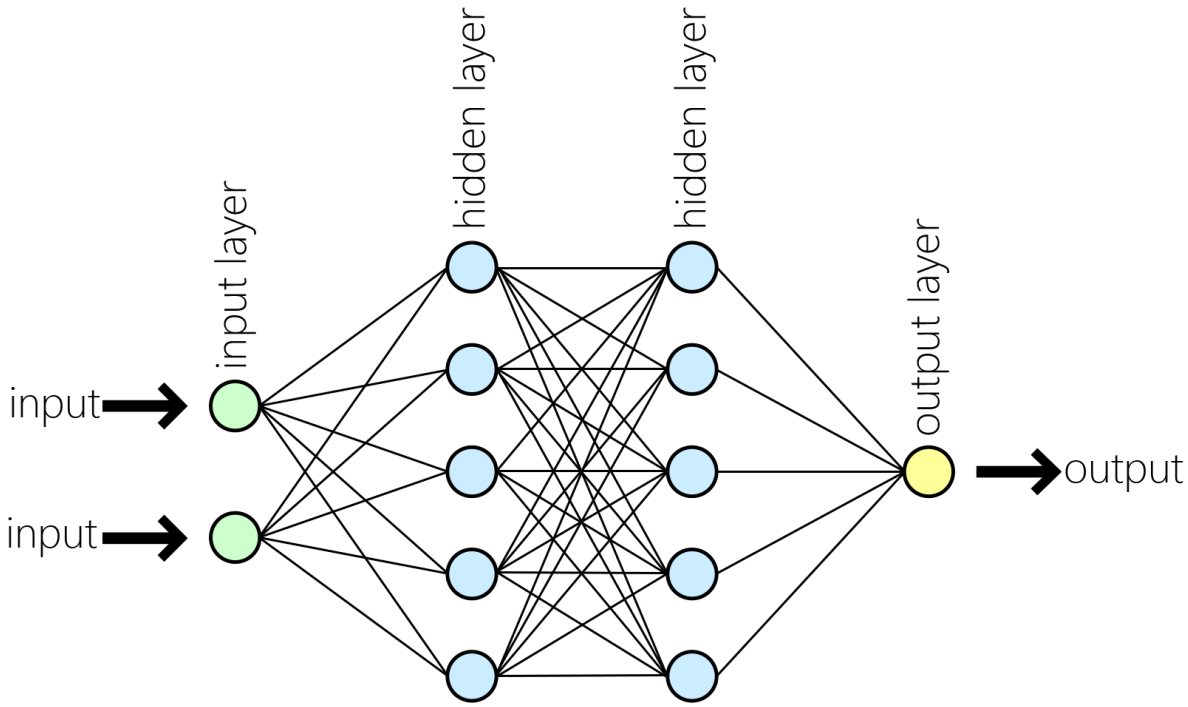


2.3 Das „Deep“ in Deep Learning

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

Vorweg:

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



3 Artificial Neural Network (ANN) erklärt

3.1 Definition

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

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 Neuronen wird auch als Nodes bezeichnet.

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

1. Input Layer
2. Hidden Layer(s)
3. Output Layer

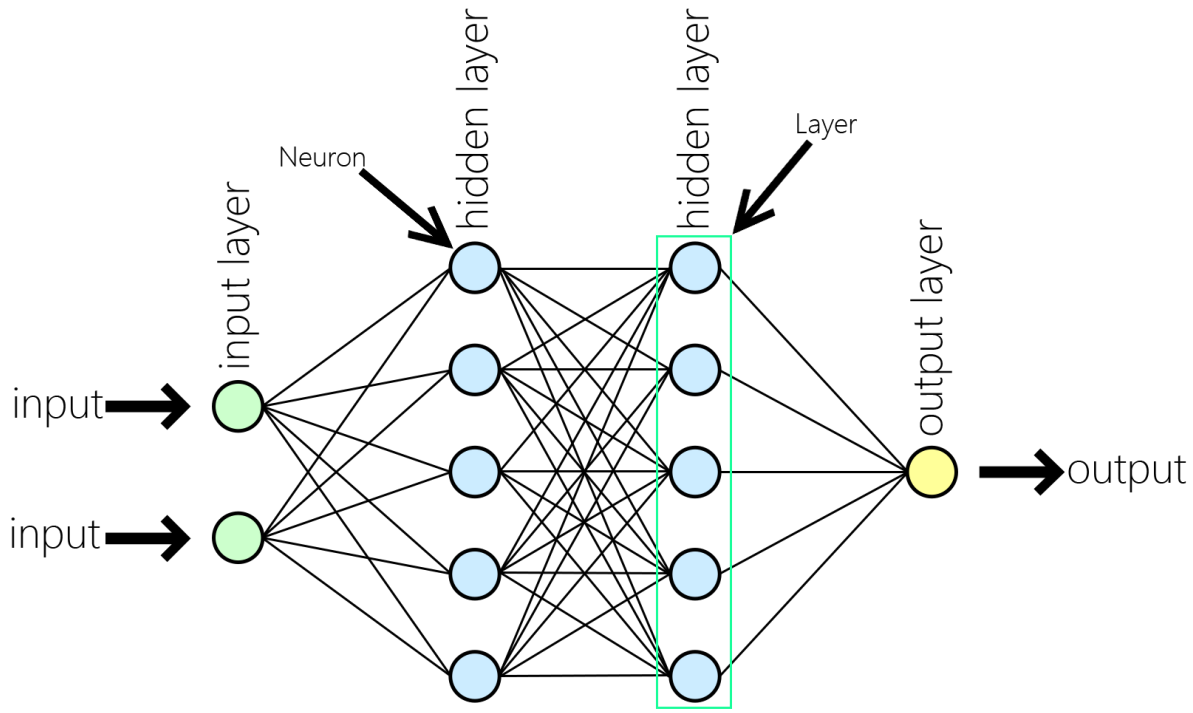
Verschiedene Layer führen verschiedene Arten von Transformationen an ihren Eingängen durch. Daten fließen durch das Netzwerk, beginnend bei dem Input Layer, durch die Hidden Layers und kommt schließlich zum Output Layer. Das nennt man „forward pass“ durch ein Netzwerk. Alle Layer zwischen Input und Output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

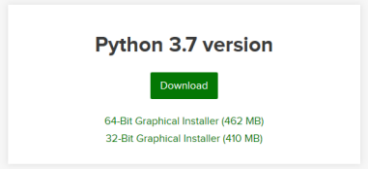
3.4.1 Keras Introduction

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:



   Installieren 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. Downgrade Python zu einer Keras & Tensorflow kompatiblen Version. Um Python auf 3.6 zu downgraden verwende
   `conda install python=3.6`
   2.3. Erstelle ein neues conda Environment wo wir unser Modules installieren:
   `conda create --name PythonCPU`
   2.4. Aktiviere das conda Environment mit:
   `activate PythonCPU`
   2.5. (zum Deaktivieren: `conda deactivate`)
   2.6. Installiere Keras & Tensorflow
   `conda install -c anaconda keras`
   2.7. Installiere Spyder (eine IDE)
   `conda install spyer`
   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 ModuleImport 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 names 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 von einem 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:

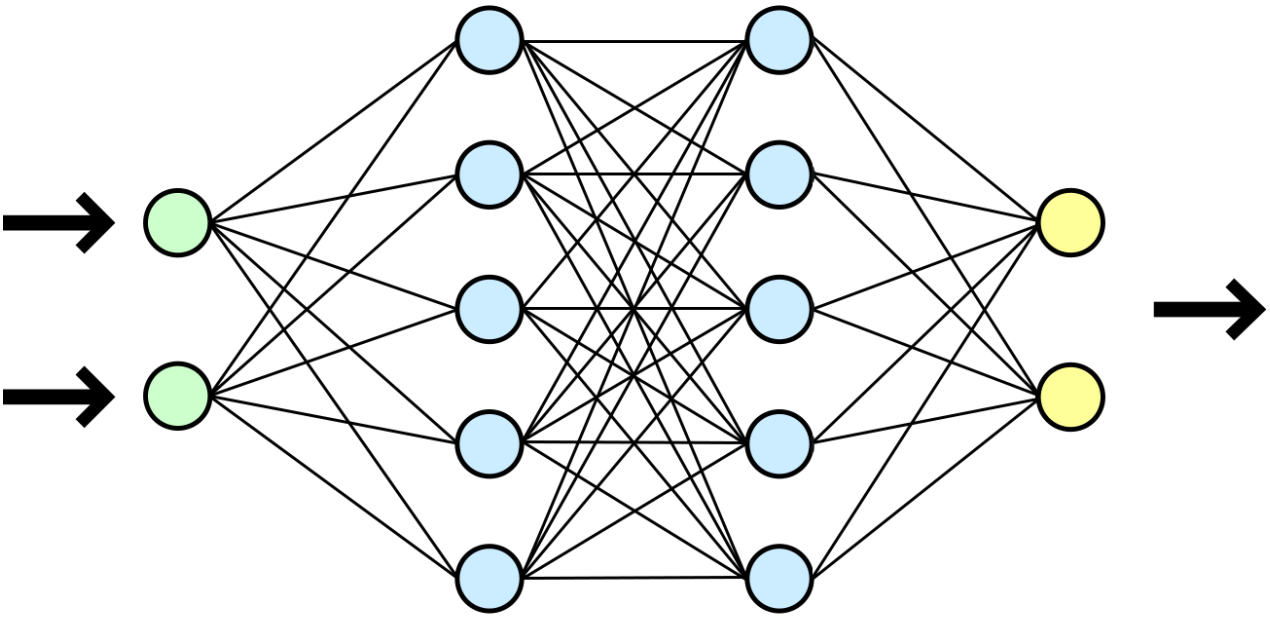
- Dense (fully connected) Layer
- Convolution Layer
- Pooling Layer
- Recurrent Layer
- Normalization Layer

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 si 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 Path 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/Predection 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.

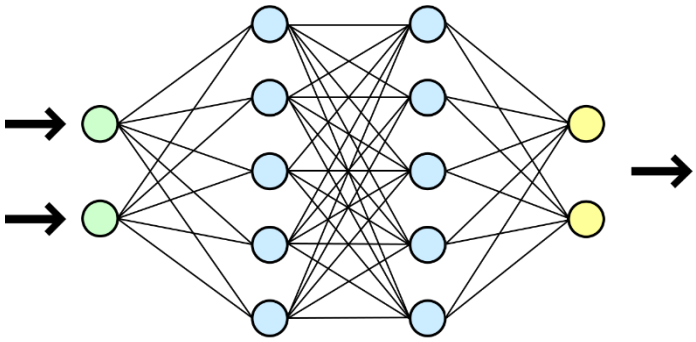
Für ein Beispiel aus dem Datensatz wird der gesamte Prozess von der Eingangsschicht bis zur Ausgangsschicht als Forward Path durch das Netzwerk bezeichnet.

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 Layern. Dieses Array wird dann an den Konstruktor des sequential models 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, das wir eine Aktivierungsfunktion names relu für beide unserer Hidden Layers und für den Output Layer eine Aktivierungsfunktion names softmax.
`activation='relu'`;`activation='softmax';`

Unser finales Produkt schaut folgendermaßen aus:

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

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

model = Sequential(layers)
```

So wird ein ANN mit Keras geschrieben.

5. Aktivierungsfunktionen

5.1 Was ist eine Aktivierungsfunktion?

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

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

Node output = Aktivierungsfunktion(gewichtet Summe der eingänge)

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

5.2 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.

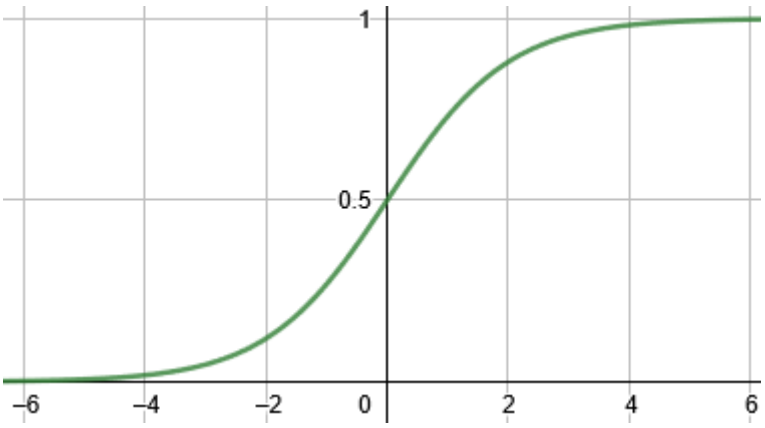
5.2.1 Sigmoid Aktivierungsfunktion

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.

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

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

5.2.3 Relu Aktivierungsfunktion

Es ist aber nicht immer der Fall, dass unserer Aktivierungsfunktion den Input in eine Nummer zwischen 0 und 1 transformiert. Tatsächlich tut eine der gebräuchlichsten

Aktivierungsfunktionen namens Relu (= Rectified Linear Unit) genau das nicht. Relu transformiert den input zu einem maximum von entweder 0 oder in den Input selbst.

$$\text{relu}(x) = \max(0,x)$$

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

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

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

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

5.3 Warum benutzen wir Aktivierungsfunktionen?

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

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

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

Stell dir vor, dass x eine reale Zahl ist.

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

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

und

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

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

Typischerweise sind die Mappings, die wir mit unseren deep neural networks lernen wollen, komplexer als einfache lineare mappings.

5.3.1 Beweis, dass ReLu nicht linear ist

Für jede reale Nummer x definieren wir eine Funktion f:

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

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

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

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

Und das

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

Dass lässt uns zu dem Schluss kommen:

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

➔ Die Funktion f ist nicht linear

5.4 Aktivierungsfunktionen in Code mit Keras

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

Zuerst müssen unsere benötigten Klassen importiert werden mit:

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

Dann gibt es 2 verschiedene Wege

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

6. Trainieren eines NN

6.1 Was ist Trainieren in einem ANN?

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

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

werden diese Gewichte iterativ geupdated und deren Optimalwert gebracht.

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 path fertig ist und die Bilddaten durch das Netz gejagt wurde wird uns unser Netz uns unser Netz vorhersagen, ob das Bild eine Katze oder ein Hund ist.

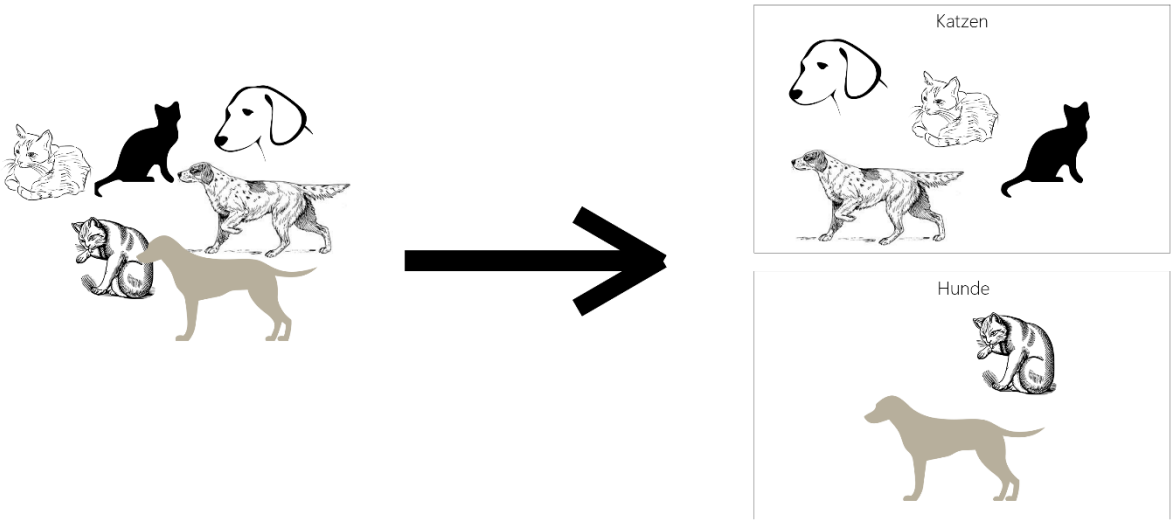
Der Output besteht dabei aus wie wahrscheinlich es ist, dass auf dem Bild eine Katze bzw ein Hund ist. Es wird zum Beispiel sagen, dass das Bild zu 75% eine Katze und zu 25% ein Hund ist. In diesem Fall weißt das Netz dem Bild eine höhere Wahrscheinlichkeit zu, dass es eine Katze ist und kein Hund. Diese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

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

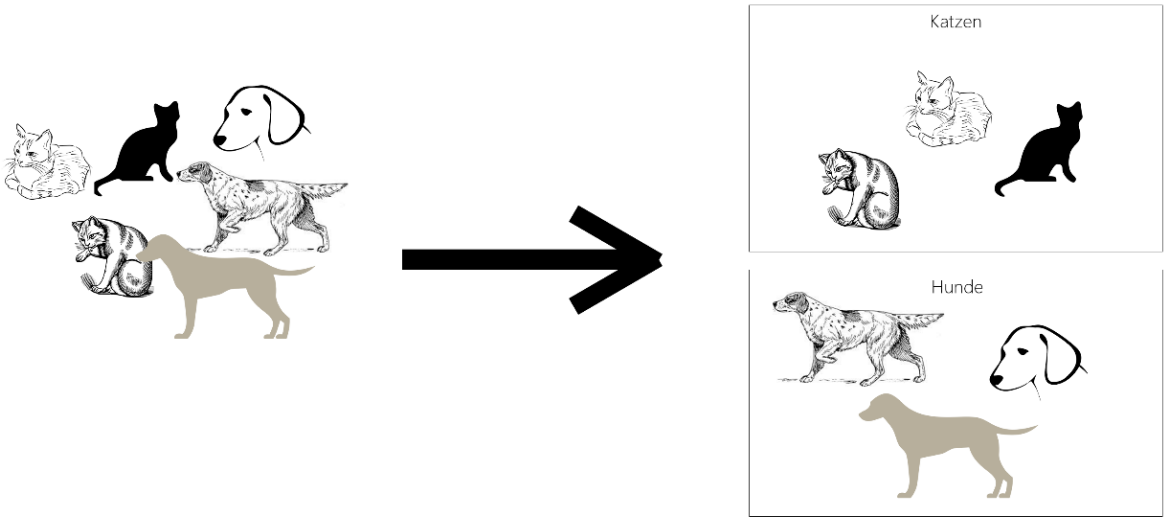
Nachdem wir unsere ganzen Daten durch unser Netz gejagt haben werden wir dieselben Daten immer wieder durchjagen. Dieser Prozess des fortlaufenden senden der gleichen durch das Netz ist das Training. Während diesem Prozess wird das Netz lernen.

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

Untrainiertes Netz



Trainiertes Netz



7. Wie lernt ein ANN

7.1 Was ist eine Epoche?

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

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

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 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 names learning rate (= Lernrate).

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

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.

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

$$\text{MSE(input)} = (\text{output} - \text{label})(\text{output} - \text{label})$$

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 with 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 called sparse cateforical 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 lerning rate für den jeweiligen Optimizer. Die default lerning 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 werden 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, das 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