Deep Learning

DVA-Seminar 2018

Michael Schwab
Fakultät für Informatik
Hochschule für angewandte Wissenschaften Augsburg
Augsburg, Deutschland
michael.schwab@hs-augsburg.de

***Zusammenfassung*—**

Lizens

I. Einleitung

II. Konventionen

KNN Künstliches Neuronales Netz
DL Deep Learning
IDS Iris-Datensatz
ML Machine Learning
IT Informationstechnik
CSV Coma Seperated Values
ReLU Rectified Linear Units

III. Grundlagen

A. Grundbegriffe

B. Datensätze

Eine essentielle Kompenete des Deep Learning (DL) sind Daten. Ein Datensatz ist eine Sammlung von Daten in einem gleichen oder ähnlichen Format. Möchte man DL erlernen, steht man oft vor dem Problem keine passenden Daten parat zu haben. Aus diesem Grund gibt es verschiedene bereits gut klassifizierte Datensätze im Internet zu finden, die den Einstieg in das DL erleichtern. Ein Beispiel für einen dieser Datensätze ist der Iris-Datensatz (IDS)[1].

Der IDS wurde 1936 von dem Statisker und Biologen Richard Fischer vorgestellt. Der Datensatz besteht aus 150 Einträgen und umfasst drei verschiedene Gattungen der Iris Blume. Jeder Eintrag einer einzelnen Blume wird jeweils durch sowohl durch die Blütenkelch Länge und Breite als auch durch die Blütenblatt Länge und Breite beschrieben. Die drei verschiedenen Gattungen der Iris Blume heißen Iris setosa, Iris virginica und Iris versitosa [vgl. 2].

[1] https://archive.ics.uci.edu/ml/datasets/iris

Tabelle I
Einträge aus dem IDS

	Feature 1	Feature 2	Feature 3	Feature 4	Label
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
140	6.7	3.1	5.6	2.4	Iris-virginica
141	6.9	3.1	5.1	2.3	Iris-virginica
142	5.8	2.7	5.1	1.9	Iris-virginica
143	6.8	3.2	5.9	2.3	Iris-virginica
144	6.7	3.3	5.7	2.5	Iris-virginica

Tabelle I zeigt einen Auszug aus dem IDS. Anhand diesem Beispiels sollen drei Begriffe erklärt werden, die sehr häufig im Bereich des Machine Learning (ML) und DL vorkommen.

Sample Ein Sample ist ein komplettes Beispiel aus einem Datensatz. In den meißten Fällen ist ein Sample mit einer Reihe in einem Datensatz gleichzusetzen. In dem Fall des IDS entspricht ein Sample einer Blume mit ihren Eigenschaften und ihrer Gattung.

Feature Ein Feature ist beschreibt die Eigenschaften eines Samples. Im Falle des IDS hat jedes Sample vier Features.

Label Das Label drückt die Art des Samples aus, eine Zugehörigkeit zu einer Klasse. Im Falle des IDS besitzt jedes Sample ein Label. Im ganzen Datensatz gibt es drei Labels, und zwar die drei Blumengattungen.

C. Python

Laut Tiobe Index Juni 2018 ist Python die viert häufigst verwendete Programmiersprache [vgl. 4]. Python gilt als sehr einfach zu lernen und zu schreibende Programmiersprache. Deshalb ist die Python in sehr vielen Disziplienen der Informationstechnik (IT) vertreten, besonders auch im Bereich des Scientific Computing. Das die meißten großen DL-Frameworks Python als erste Programmiersprache anbieten

zeigt, dass Python sich auch hier durchsetzen konnte. Alle Code-Beispiele in dieser Arbeit sind in Python geschrieben.

D. Python Bibliotheken

Im Bereich des DL gibt es eine Reihe von Python Bibliotheken, deren Einsatz stark zu empfehlen ist:

numpy Numpy ist die Standart Python Bibiliothek für Matrizen Operationen. Numpy ist hoch optimiert und extrem schnell.

pandas Pandas ist eine Python Bibliothek Dataframe Objekte zur Verfügung stellt, mit denen Daten leicht gelesen, bearbeitet und gespeichert werden können. Unter anderem können Coma Seperated Values (CSV) Dateien gelesen werden.

sklearn Sklearn ist eine ML Python Bibliothek die viele nützliche Funktionen zu dem Vorbereiten und Verarbeiten der Datensätze mitbringt.

matplotlib Matplotlib orintiert sich stark an der Programmiersprache Matlab. Mit der Bibliothek Matplotlib können Daten visualisiert werden.

E. Deep Learning Frameworks

TODO

IV. Das Neuron

TODO

A. Das Perzeptron

Das Perzeptron bassiert auf der Arbeit von Frank Rosenblatt aus dem Jahr 1958. Es ist eine einfach Form des Künstlichen Neurons. Mathematisch lässt sich das Perzeptron wie folgt ausdrücken:

$$f(x) = \begin{cases} 1 & wenn \sum_{i=1}^{m} w_i * x_i + b \\ 0 & andernfalls \end{cases}$$

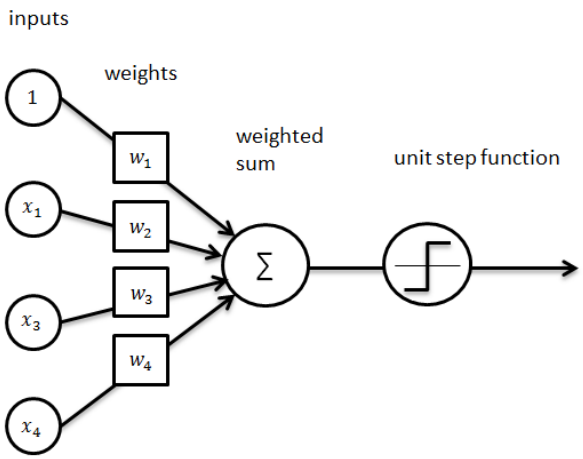

Abbildung 1. Schema eines Perzeptrons[3]

Das Konzept des Perzeptrons wird im Bild 1 veranschaulicht. Eine Reihe von Eingabe Werten werden jeweils mit einem Weight multipliziert. Alle Muliplikationen werden aufsummiert, die Bias Variable addiert. Ist das Ergebnis größer als Null wird eine Eins zurückgegeben, ist es kleiner Null wird eine Null zurückgegeben. Damit kann das Perzeptron zwischen zwei verschiedenen Labels unterschieden, die als Nullen und Einsen Codiert sind.

Die Fähigkeit zu lernen erhält das Perzeptron durch Rosenblatts Perzeptron Lern Algorithmus. Dieser sagt folgendes aus:

- Falls die Ausgabe dem erwarteden Wert enstpricht, tue nichts
- andernfalls:
  - Falls die Ausgabe eine Eins ist, aber eine Null sein sollte: dekrementiere die Weights um eine vorher deffinierte Lernrathe.
  - Falls die Ausgabe eine Null ist, aber eine Eins sein sollte: inkrementiere die Weights um eine vorher definierte Lernrathe.

Das Perzeptron kann in Python wie folgt umgesetzt werden:

```
import numpy as np

class Perceptron():
    def __init__(self, input_shape,
        ↪ learning_rate=0.1, epochs=10):
        self.weights = np.zeros(input_shape
            ↪ + 1)
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation_function(self, x):
        return binary_step(x)
```

```
    def predict(self, features):
        #Insert Bias of 1 at Postion 0
        x = np.insert(features, 0, 1)
        x = self.weights.T.dot(x)
        x = self.activation_function(x)
        return x

    def fit(self, features, labels):
        for _ in range(self.epochs):
            for i in range(len(labels)):
                predicted_label = self.predict
                    ↪ (features[i])
                error = labels[i] -
                    ↪ predicted_label
                if error != 0:
                    self.update_weights(
                        ↪ features[i], error)

    def update_weights(self, features,
        ↪ error):
        self.weights = self.weights + self.
            ↪ learning_rate * error * np.
            ↪ insert(features, 0, 1)

    def binary_step(x):
        return np.where(x >= 0.0, 1, 0)
```

Das *Perceptron* Objekt wird mit einer Lern-Rate, einer Input Form und einer Epoch Zahl initialisert. Die Input Form ist wichtig um zu wissen wie viele Weights erzeugt werden müssen, da jeder Input(jedes Feature) mit einer eignen Weight-Variable multipliziert wird. Der Weights Liste in dem Perceptron Objekt wird ein Wert mehr hinzugefügt um die Bias Variable zu simulieren. Die Epochs definieren wie oft über das Datenset iteriert wird. Die *activation_function* Methode liefert das Ergebnis der *binary_step* Methode zurück, diese beiden Methoden übernehmen den Aufgabe, die Ausgabe des Neurons in eine Eins oder Null zu quantizieren. In der *predict* Methode werden die Eingabewerte mit den Weights multipliziert. Zuvor wird eine Eins in die Eingabewerte eingefügt, um diese mit der Bias Variable multiplizieren zu können, was einer Addition der Bias nach der Muliplikation der Weights entspricht.

In der *fit* Methode wird das Perzeptron nun trainiert. Zunächst wird über mehrere Epochs hinweg der gesamte Datensatz durchlaufen. Für jedes Sample wird die Ausgabe des Perzeptrons und das richtige Label verglichen. Falls das Ergebnis von dem Label abweicht, werden die Weights entsprechend der Pezeptron Lern Regel angepasst. Die *learning_rate* entscheidet hierbei wie stark die Weights angepasst werden. Bei einer sehr kleinen Lern-Rate braucht das Perzeptron länger um die Weights optimal anzupassen. Bei einer sehr großen Lern-Rate kann es passieren, dass das Pezeptron über das Ziel hinausschießt.

B. Das ADALINE

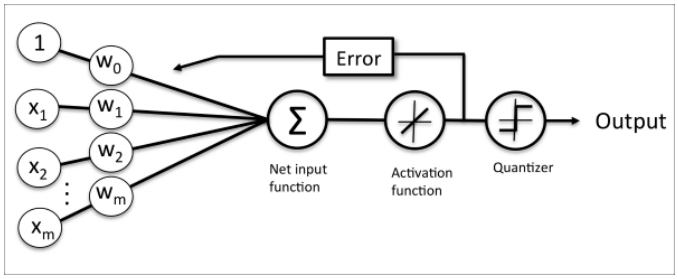

Abbildung 2. Schema eines ADALINE-Neuron [3]

Das ADALINE oder auch Adaptive Linear Neuron ist eine Weiterentwicklung des Perzeptrons. Abbildung 2 beschreibt den Aufbau eines ADALINE. Das ADALINE bietet im Gegensatz zu dem Perzeptron einige Unterschiede wie verschieden Aktivierungsfunktionen, Fehlerfunktionen und Gradient Descent.

Das ADALINE kann in Python wie folgt umgesetzt werden:

```
class Adaline():
    def __init__(self, input_shape,
        ↪ learning_rate=0.1, epochs=50):
        self.weights = np.zeros(input_shape
            ↪ + 1)
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation_function(self, X):
        return sigmoid(self.net_input(X))

    def predict(self, features):
        x = np.insert(features, 0, 1)
        x = self.weights.T.dot(x)
        x = self.activation_function(x)

    def fit(self, features, labels):
        self.cost_ = []
        for i in range(self.epochs):
            output = self.activation_function
                ↪ (features)
            errors = (y - output)
            errors = (erros**2).sum() / 2.0
            self.weights[1:] += self.
                ↪ learning_rate * features.T.
                ↪ dot(errors)
            self.weights[0] += self.
                ↪ learning_rate * errors
            cost = errors
            self.cost_.append(cost)
        return self

    def net_input(self, X):
```

```
        return np.dot(X, self.weights[1:]) +
            ↪ self.weights[0]
```

C. Aktivierungsfunktionen

Eine Aktivierungsfunktion verarbeitet den Ausgangswert der Weights Multiplikation und der Bias Addition. Bisher wurde in dem Perzeptron die Binary-Step oder auch Threshold Aktivierungsfunktion genutzt. Bei der Binary-Step Aktivierungsfunktion wird jeder Wert über Null zu einer Eins und jeder Wert unter Null zu einer Null. Die Binary-Step Funktion ist in Abbildung 3 dargestellt.

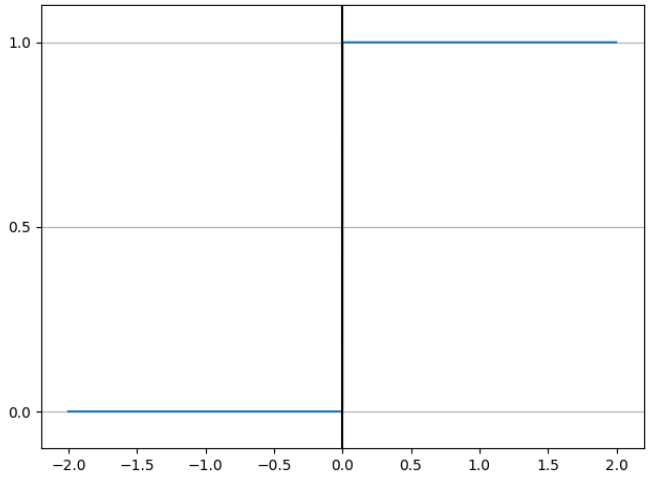

Abbildung 3. Binary-Step Aktievierungsfunktion

Doch gibt es noch viele weitere Aktivierungsfunktionen. Die wichtigsten sind Sigmoid, Rectified Linear Units (ReLU) und Tanh.

Die Sigmoid Aktivierungsfunktion ist in Abildung 4 dargestellt. Mit ihr lassen sich Wahrscheinlichkeiten zwischen zwei Labeln angeben.

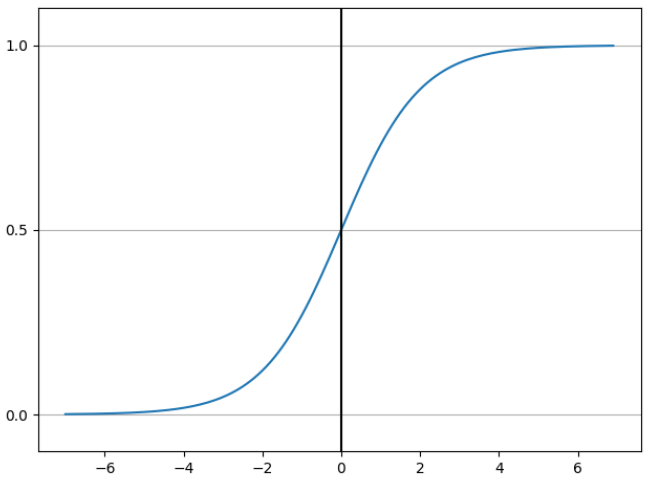

Abbildung 4. Sigmoid Aktievierungsfunktion

Ähnlich wie Sigmoid ist die Tanh Aktivierungsfunktion, ihr Ausgabe Wert liegt anders als bei Sigmoid zwischen minus Eins und Eins. Die Tanh Funktion ist in Abbildung 5 abgebildet.

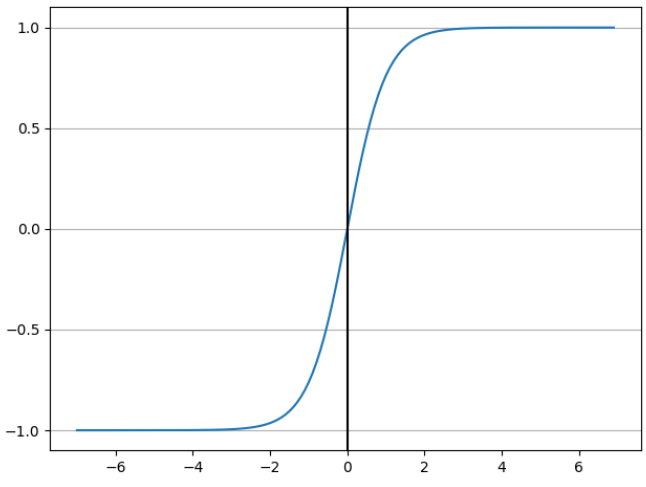

Abbildung 5. Tanh Aktievierungsfunktion

ReLU ist eine sehr wichtige Aktivierungsfunktion innerhalb von mehrschichtigen Neuronalen Netzen, sie erlaubt es gewisse Verbindungen on Neuronalen Netzen du deaktivieren. Die ReLU Aktivierungsfunktion ist in Abbildung 6 dargestellt.

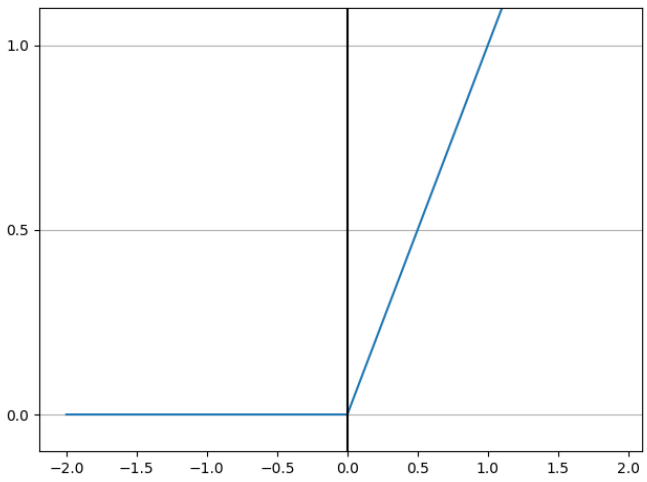

Abbildung 6. ReLU Aktievierungsfunktion

D. Fehlerfunktionen

E. Gradient Descent

F. Stochastic und Batch Gradient Descent

V. Künstliche Neuronale Netze

A. Definition

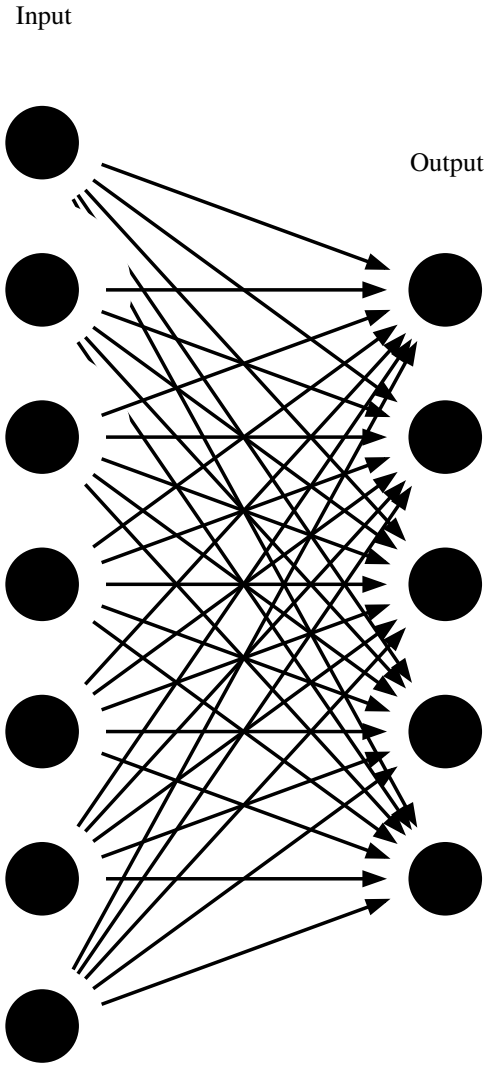

Abbildung 7. Künstliches Neuronales Netz (KNN)

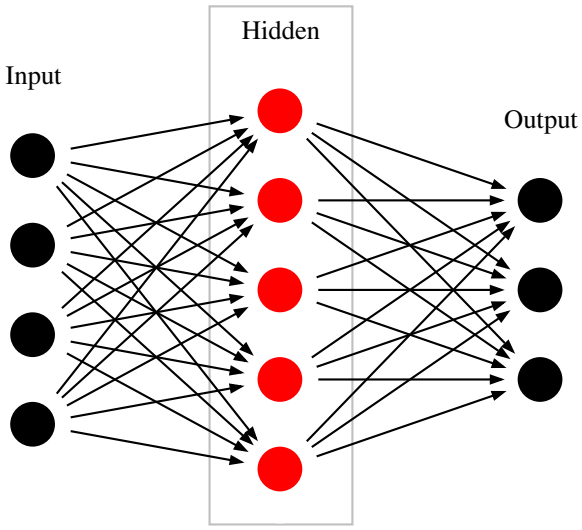

Abbildung 8. Tiefes KNN

B. Arten des Deep Learning

Im DL wird zwischen drei verschiedenen Arten des Lernens unterschieden:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

C. Vom Neuron zum Deep Learning

```
import numpy as np
import sys

X=np.array([[1,0,1,0], [0,0,0,0],
    ↪ [1,1,1,1], [1,0,0,0]])

y=np.array([[1], [0], [1], [0]])

def sigmoid (x):
   return 1/(1 + np.exp(-x))

def derivatives_sigmoid(x):
   return x * (1 - x)

epoch=5000
lr=0.1
inputlayer_neurons = X.shape[1]
hiddenlayer_neurons = 3
output_neurons = 1

wh=np.random.uniform(size=(
    ↪ inputlayer_neurons,
    ↪ hiddenlayer_neurons))
bh=np.random.uniform(size=(1,
    ↪ hiddenlayer_neurons))
wout=np.random.uniform(size=(
    ↪ hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,
    ↪ output_neurons))

for i in range(epoch):
   hidden_layer_input1=np.dot(X,wh)
   hidden_layer_input=hidden_layer_input1
       ↪ + bh
   hiddenlayer_activations = sigmoid(
       ↪ hidden_layer_input)
   output_layer_input1=np.dot(
       ↪ hiddenlayer_activations,wout)
   output_layer_input= output_layer_input1
       ↪ + bout
   output = sigmoid(output_layer_input)

   E = y-output
   slope_output_layer =
       ↪ derivatives_sigmoid(output)
   slope_hidden_layer =
       ↪ derivatives_sigmoid(
       ↪ hiddenlayer_activations)
   d_output = E * slope_output_layer
   Error_at_hidden_layer = d_output.dot(
       ↪ wout.T)
   d_hiddenlayer = Error_at_hidden_layer *
       ↪  slope_hidden_layer
   wout += hiddenlayer_activations.T.dot(
       ↪ d_output) *lr
   bout += np.sum(d_output, axis=0,
       ↪ keepdims=True) *lr
   wh += X.T.dot(d_hiddenlayer) *lr
   bh += np.sum(d_hiddenlayer, axis=0,
       ↪ keepdims=True) *lr

print(output)
```

D. Backpropagation

E. Das Densenet

F. Das Convolutional Neural Network

G. Das Recurrent Neural Network

Literatur

[1] Zahid Hasad. *Perceptron from scratch in Python*. URL: https://zahidhasan.github.io/2017-09-10-perceptron/ (besucht am 29.06.2018).

[2] *Iris flower data set*. URL: https://en.wikipedia.org/wiki/Iris_flower_data_set (besucht am 29.06.2018).

[3] Sebastian Raschka. *Python Machine Learning*. Puckt Puplishing, 2016.

[4] *TIOBE Index June 2018*. URL: https://www.tiobe.com/tiobe-index/ (besucht am 29.06.2018).