Freie Universität Berlin

Prof. R. Rojas

Mustererkennung, WS17/18

Übungsblatt 9

Boyan Hristov, Nedeltscho Petrov

18. Dezember 2017

Link zum Git Repository: https://github.com/BoyanH/FU-MachineLearning-17-18/tree/master/Solutions/Homework9

Neural Networks

Wir haben zwei Implementierungen - eine mit streng nur 1 hidden Layer und eine, die als Argument ein Tupel, dass die Anzahl von Neuronen pro Layer beschreibt und damit auch die Anzahl von layers. Die komplexere Implementierung ist aber deutlich weniger lesbar und viel zu schwierig einzustellen. Deswegen werden wir uns hier auf die Beschreibung unserer simpleren Implementierung konzentrieren. Beide Implementierungen sind am Ende des Blattes zu sehen.

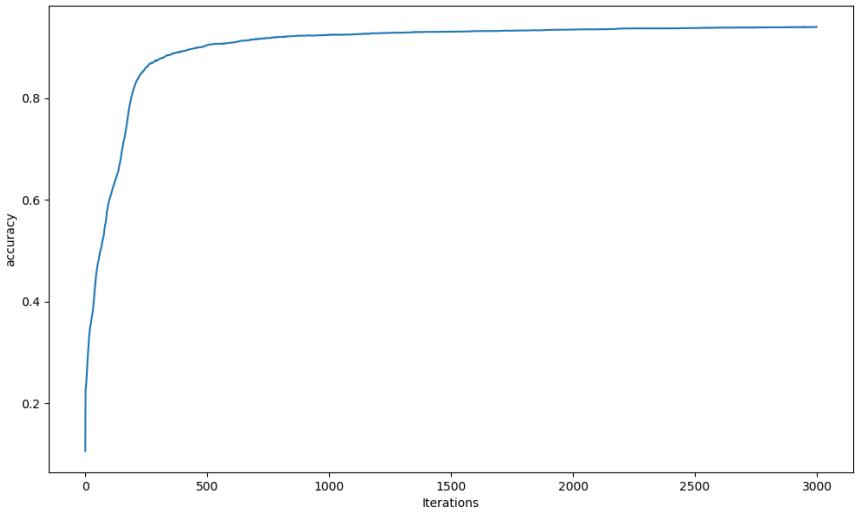
Evaluation

Die Methode ist, wie man später anhand der booleschen Funktionen sieht, viel mächtiger, aber auch viel schwieriger einzustellen. Wir haben eine Fehlerrate von knapp über 11% bekommen, was eigentlich sehr schlecht ist. Wir denken aber, dass das eher an der Lernrate und Anzahl von Neuronen / Schichten im Netz hängt. Die Methode ist auch sehr aufwändig im Sinne von Rechnerressourcen, wie aber im Tutorial beschrieben, könnte diese effizient in Hardware implementiert werden. Wenn wir aber uns entschieden sollen für diesen Datensatz (Ziffererkennung) wurden wir eher eine lineare Methode nehmen, die viel schneller ist und leichter zu interpretieren. Wir haben aber immer noch Hoffnung, dass wir bessere neuronale Netze implementieren werden.

Wie wir auf der offizieler Seite des Datensatz gesehen haben, benutzt man echt viele hidden Layers um die Daten gut zu klassifizieren. Leider sind beide unsere Implementierung und Rechnern nicht so perfekt für solche Ansätze.

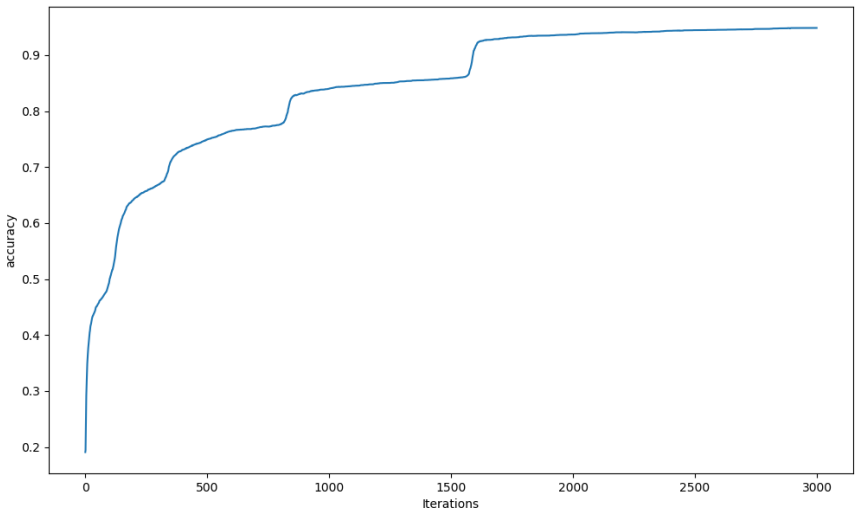
Fehler über die Iterationen

1 Hidden Layer mit 15 Neuronen



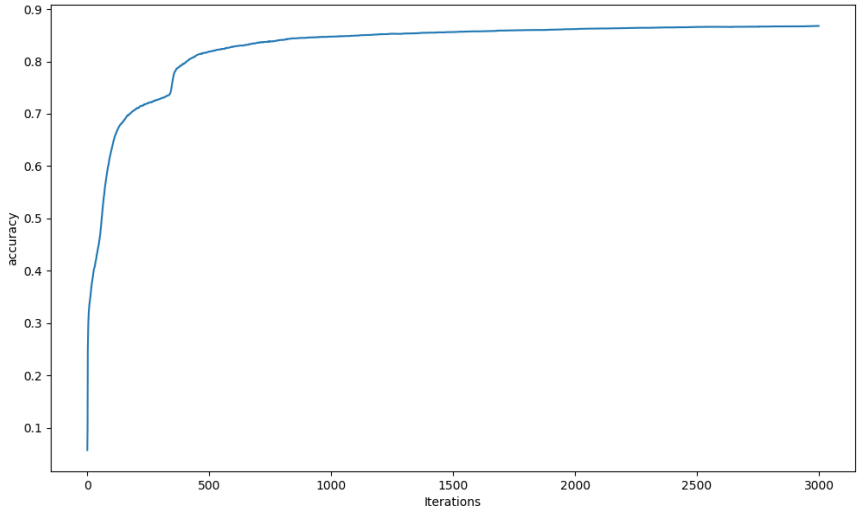
Output:
Score: 0.88458781362

1 Hidden Layer mit 20 Neuronen



Output:
Score: 0.8863799283154122
Score train: 0.9512907191149355

1 Hidden Layer mit 25 Neuronen



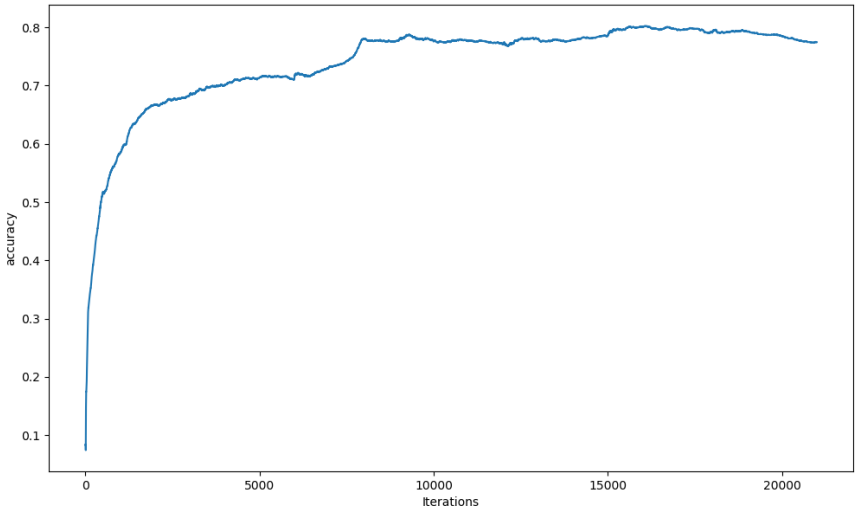
Output:
Score: 0.8182795698924731
Score train: 0.8678549477566072

2 Hidden Layers mit 20 und 15 Neuronen, circa 7 Batches (max. 1000 Beispiele pro Batch)

Wir wurden gerne noch mehr mit mehreren Schichten spielen, das war aber viel zu rechenaufwändig...

Output

```
Score: 0.7548387096774194
Score train: 0.7744314689612785
```



1 Hidden Layer, 20 Neuronen, großen Datensatz

Leider müssten wir den Code von NN.py noch etwas ändern, damit wir eine vernünftige Laufzeit kriegen. Wir haben batches eingefügt und auch einstellbare Anzahl von Beispiele pro Batch. Dazu müssten wir leider nur den Score innerhalb des Batches plotten lassen, sonst dauert eine Prediction innerhalb der Iterationsschleife viel zu lange. Unsere Methode ist für den großen Datensatz gar nicht optimal, da es lange Zeit braucht zum Ausführen könnten wir es nicht viel optimieren.

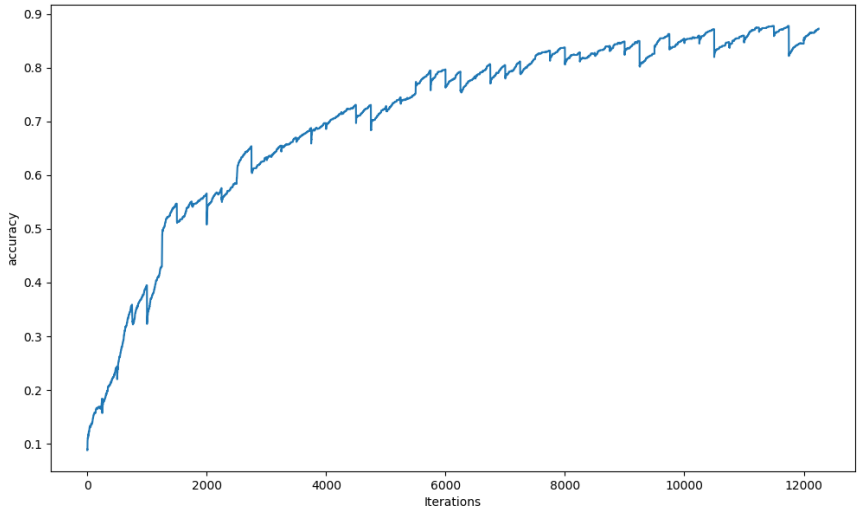
Da das kleine Datensatz nur etwas über 6 tausend Beispiele hat, ist das Plotting da immer noch korrekt mit der ganzen Fehler bei der Trainingsdatensatz.

Hier wurden auch nur 250 Iterationen pro Batch gemacht, also das ganze ist weit weg von präzise, wir wollten aber schauen was für Ergebnisse man bekommt mit dem großen Datensatz, wenn man mehrere unterschiedliche Beispiele sieht und weniger dabei korrigiert.

Output

```
fetching data-set
MNIST original fetched
#Batches: 49
#Iterations per batch: 250
Batch #1 completed!
Batch #2 completed!
Batch #3 completed!
Batch #4 completed!
Batch #5 completed!
Batch #6 completed!
Batch #7 completed!
Batch #8 completed!
Batch #9 completed!
Batch #10 completed!
Batch #11 completed!
Batch #12 completed!
Batch #13 completed!
Batch #14 completed!
```

```
Batch #15 completed!
Batch #16 completed!
Batch #17 completed!
Batch #18 completed!
Batch #19 completed!
Batch #20 completed!
Batch #21 completed!
Batch #22 completed!
Batch #23 completed!
Batch #24 completed!
Batch #25 completed!
Batch #26 completed!
Batch #27 completed!
Batch #28 completed!
Batch #29 completed!
Batch #30 completed!
Batch #31 completed!
Batch #32 completed!
Batch #33 completed!
Batch #34 completed!
Batch #35 completed!
Batch #36 completed!
Batch #37 completed!
Batch #38 completed!
Batch #39 completed!
Batch #40 completed!
Batch #41 completed!
Batch #42 completed!
Batch #43 completed!
Batch #44 completed!
Batch #45 completed!
Batch #46 completed!
Batch #47 completed!
Batch #48 completed!
Batch #49 completed!
Score train: 0.8551020408163266
Score: 0.8544761904761905
```



Wie man auf dem Plot sieht, senkt das Score wenn ein neues Batch eingefügt wird, da wir diese Beispiele noch nie gesehen haben. Dann wird die Korrektur bei den Gewichtsmatritzen gemacht und den Score steigt wieder. Ein Score von 85% finden wir eigentlich ganz toll für nur 250 Iterationen pro Batch und

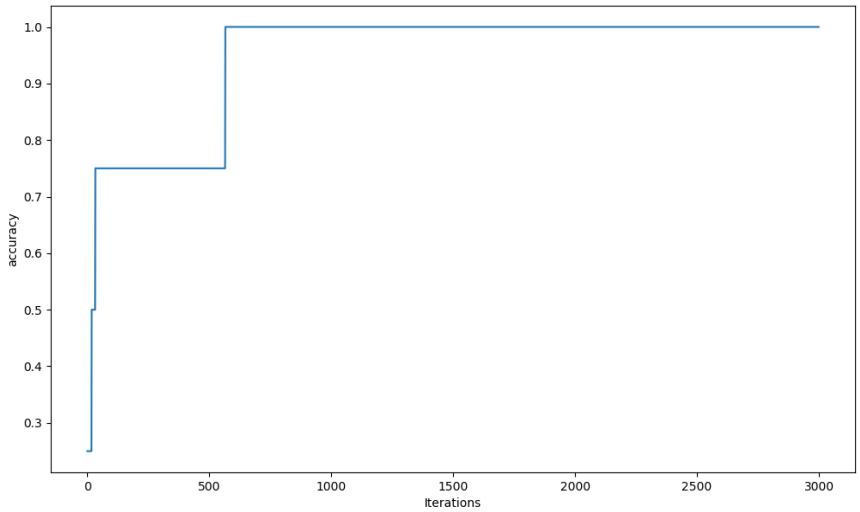
keine perfekte Anpassung des Netzes. Das zeigt, dass die Unterteilung des Datensatzes in Batches keine schlechte Idee ist.

Boolesche Funktionen

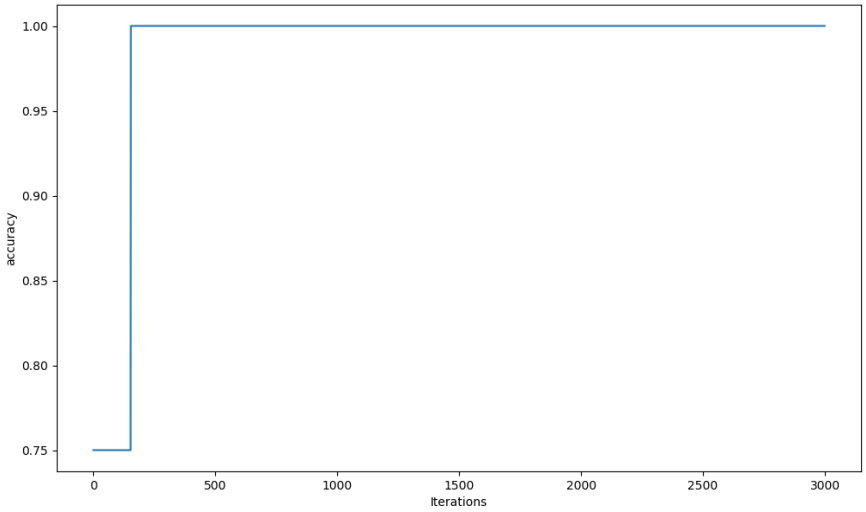
Output

```
Score and: 1.0
Score or: 1.0
Score xor: 1.0
```

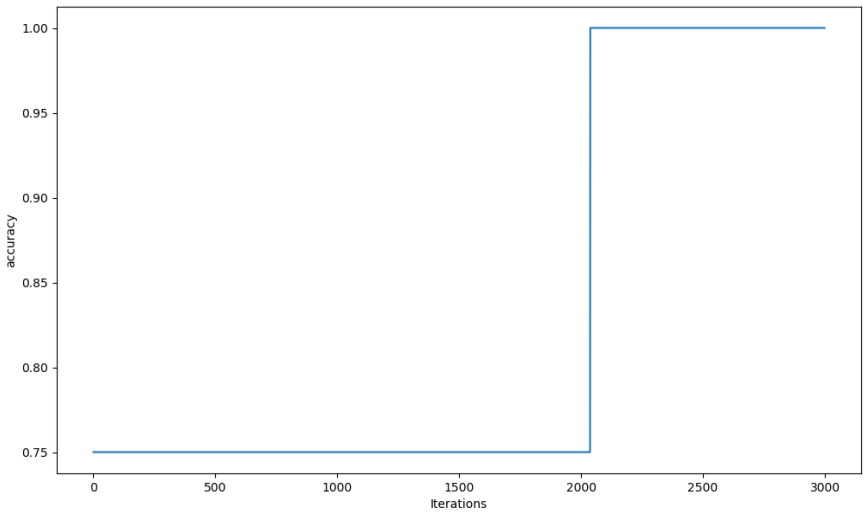
AND



OR



XOR



Erklärung des Codes

Diesmal gibt es extrem wenig zu erklären, da das meiste Code von der Vorlesung / Tutorial 1 zu 1 genommen wurde und die Matritzenmultiplikationen in python geschrieben wurden. Nur eine Transponierung hat gefehlt in den Formeln, dazu kommen wir nocht.

Fit Methode

Interessant hier wäre vielleicht, dass wir den Datensatz in Batches mit je 7000 Beispiele unterteilen, sonst können wir von den großen Datensatz gar nicht lernen. Adaptive Lernrate hat uns in dem Beispiel nicht geholfen, wahrscheinlich war die Fehler kleiner wird und damit auch ihre Ableitung, wir korrigieren mit der Zeit immer weniger. Weiter plotten wir nur den Score innerhalb des Batches und nicht das globale auf dem ganzen Trainingssatz. Deswegen sieht man auch, dass den Score nach je Batch etwas senkt.

Es gibt nur zwei Unterschiede von den gegebenen Formeln. Erstens haben wir keine Diagonalmatrix benutzt, aber das selbe mit Skalarprodukt erreicht. Der Grund war, dass wir für je Output eine Matrix bekommen, und diese man in den Diagonalen einer weiteren Matrtitze nicht darstellen kann.

Die zweite Unterschied ist, dass wir für die Berechnung der Delta (Backpropagated error) die transponierte Gewichtsmatrix benutzt haben.

```
        def fit(self, X, y):
        self.history = []
        self.unique_labels = np.unique(y)
        X_ = self.data_normalizer.fit_transform(X)
        y_ = self.transform_y(y)
        self.W1 = np.vstack((
            np.random.randn(len(X_[0]), self.size_hidden),
            np.ones(self.size_hidden)))
        self.W2 = np.vstack((
            np.random.randn(self.size_hidden, self.size_output),
            np.ones(self.size_output)))

        batch_size = self.batch_size
        print('#Batches: {}'.format(math.ceil(len(X_) / batch_size)))
        print('#Iterations per batch: {}'.format(self.max_iterations))
        for batch_start in range(0, len(X_), batch_size):
            Xb = X_[batch_start:batch_start + batch_size]
            yb = y_[batch_start:batch_start + batch_size]
            for i in range(self.max_iterations):
                o_, o1, o1_, o2, o2_ = self.feed_forward(Xb)
                W2_ = self.W2[:-1]
                d1 = NN.sigmoid_derived(o1) # not diagonal matrix as in lecture, because
    sigmoid_derived(o1) is a vector
                d2 = NN.sigmoid_derived(o2)
                e = o2 - yb
                delta2 = d2 * e
                # transposing of the weights matrix missing in formula in lecture/tutorial
    of professor
                delta1 = d1 * (delta2.dot(W2_.T))
                deltaW2 = (-self.learning_rate * (delta2.T.dot(o1_))).T
                deltaW1 = (-self.learning_rate * delta1.T.dot(o_)).T
                self.W1 += deltaW1
                self.W2 += deltaW2

                # self.learning_rate = self.learning_rate * 1 / (1 + 0.0001 * i)

                if self.print_score_per_bach:
                    self.history.append(np.mean(self.unique_labels[o2.argmax(1)] == self.
    unique_labels[yb.argmax(1)]))
                else:
                    self.history.append(self.score(X, y))

            print('Batch #{} completed!'.format(math.floor(batch_start / batch_size) + 1))
```

Labels zu gewünschten Output

Interessant ist vielleicht, dass wir von den Netz 10 Outputs haben für je Ziffer und diese müssen wir mit den Labels vergleichen können. Dafür haben wir die folgende Methoden benutzt und damit auch die folgende Predict Methode. Dabei nehmen wir das Output der letzten Schicht in dem Netz und wählen den Neuron mit der größten Wert. Danach sollen wir dass wieder zu ein Label mappen, deswegen haben wir alle Mögliche Labels gespeichert. In der transform_y mappen wir z.B. 4 zu

$$0, 0, 0, 1.0, 0, 0, 0, 0$$

```
ef transform_y(self, y):
        y_ = np.zeros((len(y), len(self.unique_labels)))
        y_in_unique = np.vectorize(lambda x: list(self.unique_labels).index(x))(y)
        y_[range(len(y)), y_in_unique] = 1
        return y_

    def predict(self, X):
        X = self.data_normalizer.transform(X)
        return self.predict_(X)

    def predict_(self, X):
        o2 = self.feed_forward(X)[3]
        return self.unique_labels[o2.argmax(1)]
```

Sigmoid-Funktion und ihre Ableitung

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

    @staticmethod
    def sigmoid_derived(x):
        return x*(1-x)
```

Vollständiges Code boolean_functions_nn.py

```
import numpy as np
from NN import NN


X = np.array([
    [0,0],
    [0,1],
    [1,0],
    [1,1]
])
y_and = np.array([a & b for a,b in X])
y_or  = np.array([a | b for a,b in X])
y_xor = np.array([a ^ b for a,b in X])

# AND
nn = NN(max_iterations=3000, size_hidden=10, size_output=2, learning_rate=0.01)
nn.fit(X, y_and)
nn.plot_accuracies('./nn_and.png')
print('Score and: {}'.format(nn.score(X, y_and)))

# OR
```

```
nn = NN(max_iterations=3000, size_hidden=10, size_output=2, learning_rate=0.01)
nn.fit(X, y_or)
nn.plot_accuracies('./nn_or.png')
print('Score or: {}'.format(nn.score(X, y_or)))

# XOR
nn = NN(max_iterations=3000, size_hidden=10, size_output=2, learning_rate=0.01)
nn.fit(X, y_and)
nn.plot_accuracies('./nn_xor.png')
print('Score xor: {}'.format(nn.score(X, y_and)))
```

Vollständiges Code digits_small_nn.py

```
from Parser import get_data_set
from NN import NN


X_train, X_test, y_train, y_test = get_data_set('digits.data')
nn = NN()
nn.fit(X_train, y_train)
nn.plot_accuracies()
print('Score train: {}'.format(nn.score(X_train, y_train)))
print('Score: {}'.format(nn.score(X_test, y_test)))
```

Vollständiges Code digits_big_nn.py

```
from sklearn.datasets import fetch_mldata
from sklearn.model_selection import train_test_split
import numpy as np
from NN import NN

print('fetching data-set')
mnist = fetch_mldata('mnist-original', data_home='./Dataset/mnist.pkl/mnist_dataset/')
print('MNIST original fetched')
X = np.array(mnist.data, dtype=np.float64)
y = mnist.target
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, test_size=0.3,
                                                    random_state=1)


nn = NN(max_iterations=250, print_score_per_bach=True, batch_size=1000)
nn.fit(X_train, y_train)
nn.plot_accuracies('./20_inner_big.png')
print('Score train: {}'.format(nn.score(X_train, y_train)))
print('Score: {}'.format(nn.score(X_test, y_test)))
```

Vollständiges Code DataNormalizer.py (von Musterlösung genommen und etwas angepasst)

```
import numpy as np

class DataNormalizer:
    def fit(self, X):
        self.mean = np.mean(X, axis=0)
        self.var = np.var(X, axis=0) + np.nextafter(0, 1)
```

```
    def transform(self, X):
        return (X - self.mean) / self.var

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)
```

Vollständiges Code NN.py

```
from Classifier import Classifier
from DataNormalizer import DataNormalizer
import numpy as np
from matplotlib import pyplot as plt
import math


class NN(Classifier):
    def __init__(self, max_iterations=3000, learning_rate=0.0020,
                 size_hidden=20, size_output=10, print_score_per_bach=False, batch_size
    =7000):
        self.data_normalizer = DataNormalizer()
        self.size_hidden = size_hidden
        self.size_output = size_output
        self.max_iterations = max_iterations
        self.learning_rate = learning_rate
        self.print_score_per_bach = print_score_per_bach
        self.W1 = None
        self.W2 = None
        self.unique_labels = None
        self.batch_size = batch_size

    def fit(self, X, y):
        self.history = []
        self.unique_labels = np.unique(y)
        X_ = self.data_normalizer.fit_transform(X)
        y_ = self.transform_y(y)
        self.W1 = np.vstack((
            np.random.randn(len(X_[0]), self.size_hidden),
            np.ones(self.size_hidden)))
        self.W2 = np.vstack((
            np.random.randn(self.size_hidden, self.size_output),
            np.ones(self.size_output)))

        batch_size = self.batch_size
        print('#Batches: {}'.format(math.ceil(len(X_) / batch_size)))
        print('#Iterations per batch: {}'.format(self.max_iterations))
        for batch_start in range(0, len(X_), batch_size):
            Xb = X_[batch_start:batch_start + batch_size]
            yb = y_[batch_start:batch_start + batch_size]
            for i in range(self.max_iterations):
                o_, o1, o1_, o2, o2_ = self.feed_forward(Xb)
                W2_ = self.W2[:-1]
                d1 = NN.sigmoid_derived(o1) # not diagonal matrix as in lecture, because
    sigmoid_derived(o1) is a vector
                d2 = NN.sigmoid_derived(o2)
                e = o2 - yb
                delta2 = d2 * e
                # transposing of the weights matrix missing in formula in lecture/tutorial
    of professor
                delta1 = d1 * (delta2.dot(W2_.T))
                deltaW2 = (-self.learning_rate * (delta2.T.dot(o1_))).T
                deltaW1 = (-self.learning_rate * delta1.T.dot(o_)).T
                self.W1 += deltaW1
```

```
                self.W2 += deltaW2

                # self.learning_rate = self.learning_rate * 1 / (1 + 0.0001 * i)

                if self.print_score_per_bach:
                    self.history.append(np.mean(self.unique_labels[o2.argmax(1)] == self.
    unique_labels[yb.argmax(1)]))
                else:
                    self.history.append(self.score(X, y))

            print('Batch #{} completed!'.format(math.floor(batch_start / batch_size) + 1))

    def feed_forward(self, X):
        o_ = np.c_[X, np.ones(len(X))]
        o1 = NN.sigmoid(o_.dot(self.W1))
        o1_ = np.c_[o1, np.ones(len(o1))]
        o2 = NN.sigmoid(o1_.dot(self.W2))
        o2_ = np.c_[o2, np.ones(len(o2))]

        return o_, o1, o1_, o2, o2_

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

    @staticmethod
    def sigmoid_derived(x):
        return x*(1-x)

    def transform_y(self, y):
        y_ = np.zeros((len(y), len(self.unique_labels)))
        y_in_unique = np.vectorize(lambda x: list(self.unique_labels).index(x))(y)
        y_[range(len(y)), y_in_unique] = 1
        return y_

    def predict(self, X):
        X = self.data_normalizer.transform(X)
        return self.predict_(X)

    def predict_(self, X):
        o2 = self.feed_forward(X)[3]
        return self.unique_labels[o2.argmax(1)]

    def plot_accuracies(self, file_name=None):
        plt.figure(figsize=(12, 7))
        plt.plot(self.history)
        plt.xlabel("Iterations")
        plt.ylabel("accuracy")

        if file_name is None:
            plt.show()
        else:
            plt.savefig(file_name)
```

Vollständiges Code NeuralNetwork.py

```
from Classifier import Classifier
from DataNormalizer import DataNormalizer
import numpy as np
from matplotlib import pyplot as plt
from Parser import get_data_set


class NeuralNetwork(Classifier):
```

```
    def __init__(self, layers=None, max_iterations=3000, learning_rate=0.0025):
        """

        :param layers: tuple defining the amount of neurons to be used pro layers, thereby defining the network
        :param max_iterations:
        :param learning_rate:
        """
        self.data_normalizer = DataNormalizer()
        self.W_ext = None
        self.max_iterations = max_iterations
        self.learning_rate = learning_rate
        self.layers = layers
        self.history = []

        if self.layers is None:
            self.k_layers = 3
        else:
            self.k_layers = len(self.layers)

    def transform_y(self, y):
        y_ = np.zeros((len(y), len(self.unique_labels)))
        y_in_unique = np.vectorize(lambda x: list(self.unique_labels).index(x))(y)
        y_[range(len(y)), y_in_unique] = 1
        return y_

    def fit(self, X, y):
        self.unique_labels = np.unique(y)
        X_ = self.data_normalizer.fit_transform(X)
        y_ = self.transform_y(y)

        if self.layers is not None:
            self.W_ext = np.array([np.random.randn(len(X_[0] + 1, self.layers[i])) if i == 0 else
                                   np.random.randn(self.layers[i - 1] + 1, self.layers[i])
                                   for i in self.layers])
        else:
            self.W_ext = np.array([np.random.randn(len(X_[0]) + 1, 20), np.random.randn(20 + 1, 15),
                                   np.random.randn(15+1, 10)])

        batch_size = 1000
        for batch_start in range(0, len(X_), batch_size):
            Xb = X_[batch_start:batch_start + batch_size + 1]
            yb = y_[batch_start:batch_start + batch_size + 1]
            for it in range(self.max_iterations):
                O_s = self.get_O_s(Xb)
                D_s = [(o * (1.0 - o)) for o in O_s[2::2]]
                e = (O_s[-1] - yb)
                der_e_s = self.get_der_e_s(D_s, e)
                delta_W_ext = self.get_delta_W_ext(der_e_s, O_s[::2])

                for i, delta in enumerate(delta_W_ext):
                    self.W_ext[i] += delta

                self.history.append(self.score(X, y))

    @staticmethod
    def add_ones(X):
        return np.c_[np.ones(len(X)), X]

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

    def get_O_s(self, X):
        O_s = [X]
        for i in range(self.k_layers):
```

```
74              O_i_minus_1 = np.c_[O_s[-1], np.ones(len(X))]  # extend to get o(i-1)^
75              O_s.append(O_i_minus_1)
76              O_i = NeuralNetwork.sigmoid(O_i_minus_1.dot(self.W_ext[i]))
77              O_s.append(O_i)
78
79          return O_s
80
81      def get_der_e_s(self, D_s, e):
82          W = [w[1:] for w in self.W_ext]
83          der_e_l = e * D_s[-1]
84          der_e_s = [der_e_l]
85
86          for i in range(self.k_layers - 1):
87              der_e_i = D_s[-i - 2] * (der_e_s[0]).dot(W[-i - 1].T)
88              der_e_s = [der_e_i] + der_e_s
89
90          return der_e_s
91
92      def get_delta_W_ext(self, der_e_s, O_s):
93          delta_W_ext = []
94
95          for i in range(self.k_layers):
96              o_i_ext = np.c_[O_s[i], np.ones(len(O_s[i]))]
97              delta_W_i = der_e_s[i].T.dot(o_i_ext)
98              delta_W_ext.append(delta_W_i.T)
99
100         return np.array(delta_W_ext) * -self.learning_rate
101
102     def predict(self, X):
103         X = self.data_normalizer.transform(X)
104         return self.predict_(X)
105
106     def predict_(self, X):
107         O_s = self.get_O_s(X)
108         results = O_s[-1]
109         return self.unique_labels[results.argmax(1)]
110
111     def plot_accuracies(self, file_name=None):
112         plt.figure(figsize=(12, 7))
113         plt.plot(self.history)
114         plt.xlabel("Iterations")
115         plt.ylabel("accuracy")
116
117         if file_name is None:
118             plt.show()
119         else:
120             plt.savefig(file_name)
121
122
123 X_train, X_test, y_train, y_test = get_data_set('digits.data')
124 nn = NeuralNetwork()
125 nn.fit(X_train, y_train)
126 nn.plot_accuracies('./2_hidden_layers_20_15_with_7_batches.png')
127 print('Score train: {}'.format(nn.score(X_train, y_train)))
128 print('Score: {}'.format(nn.score(X_test, y_test)))
```