

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.

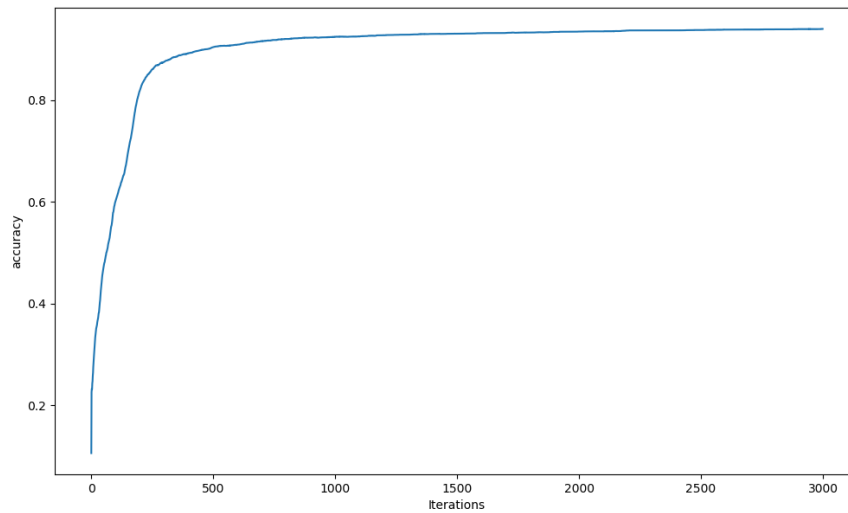
Evaluation

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 offizieller 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

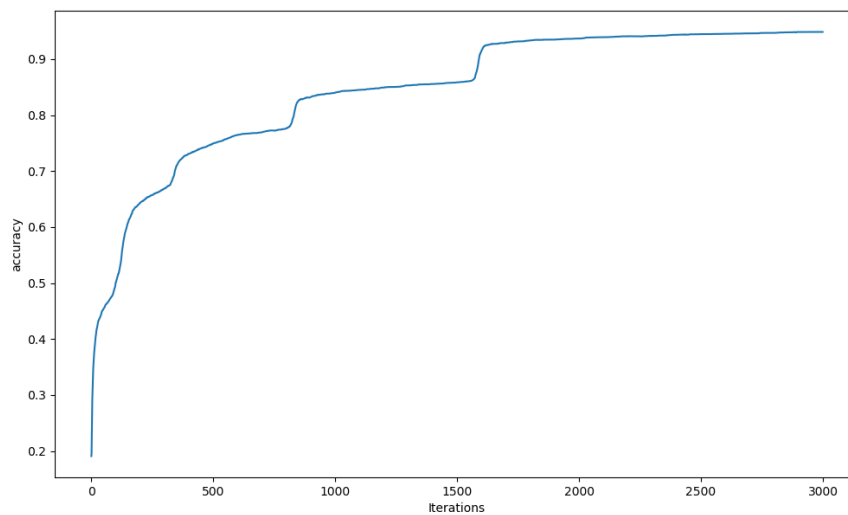
1 Hidden Layer mit 15 Neuronen



Output:

Score: 0.88458781362

1 Hidden Layer mit 20 Neuronen

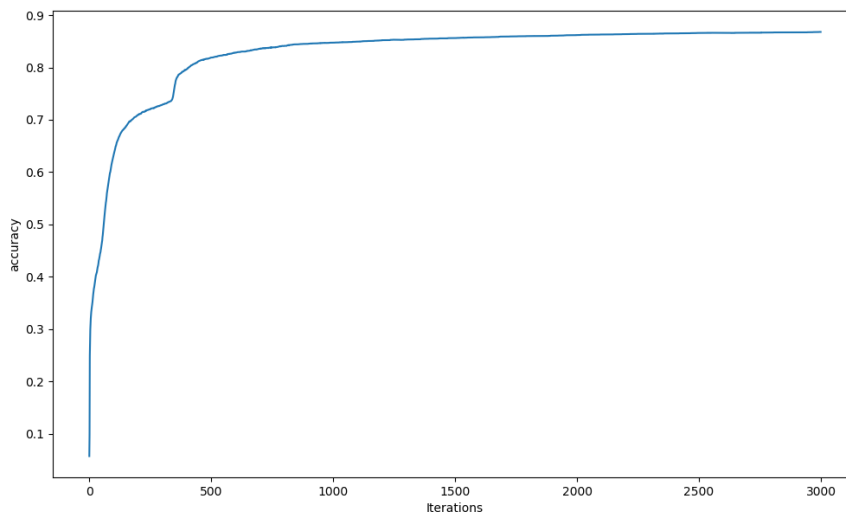


Output:

Score: 0.8863799283154122

Score train: 0.9512907191149355

1 Hidden Layer mit 25 Neuronen



Output:

Score train: 0.8678549477566072

Score: 0.8182795698924731

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...

Wie man aber hier sieht, haben wir bei den Testdatensatz besseres Score als bei den Trainingsdatesatz, was ein sehr gutes Zeichen in Richtung "nicht auswendig lernen" ist. Yey :)

Output

```
1 Score train: 0.7744314689612785
2 Score: 0.7548387096774194
4 \includegraphics[height=8cm]{./2_hidden_layers_20_15_with_7_batches.png}
```

1 Hidden Layer, 20 Neuronen, großen Datensatz

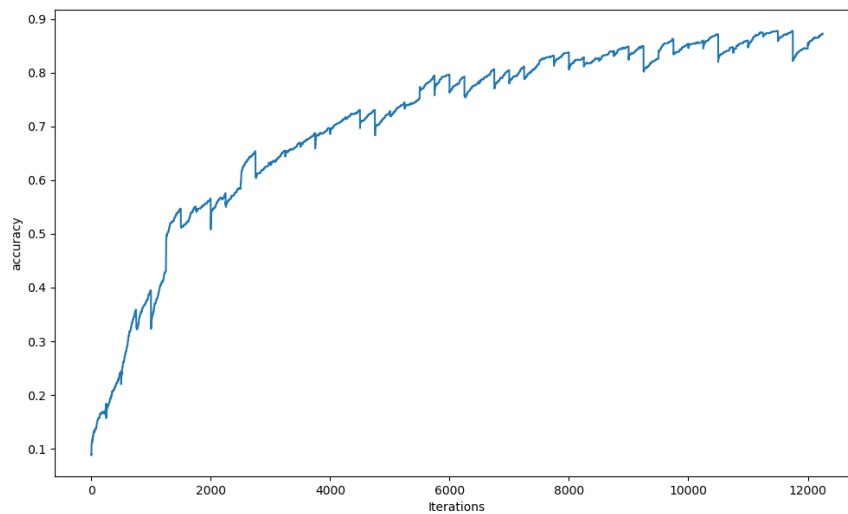
Leider mussten wir den Code von NN.py noch etwas ändern, damit wir eine vernünftige Laufzeit kriegen. Wir haben batches eingefügt und auch einstellbare Anzahl von Beispiele pro Batch. Dazu mussten wir leider nur den Score innerhalb des Batches plotten lassen, sonst dauert eine Prediction innerhalb der 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

```
1 fetching data-set
2 MNIST original fetched
3 #Batches: 49
4 #Iterations per batch: 250
5 Batch #1 completed!
6 Batch #2 completed!
7 Batch #3 completed!
8 Batch #4 completed!
9 Batch #5 completed!
10 Batch #6 completed!
11 Batch #7 completed!
12 Batch #8 completed!
13 Batch #9 completed!
14 Batch #10 completed!
15 Batch #11 completed!
16 Batch #12 completed!
17 Batch #13 completed!
18 Batch #14 completed!
19 Batch #15 completed!
20 Batch #16 completed!
21 Batch #17 completed!
22 Batch #18 completed!
23 Batch #19 completed!
24 Batch #20 completed!
25 Batch #21 completed!
26 Batch #22 completed!
27 Batch #23 completed!
28 Batch #24 completed!
29 Batch #25 completed!
30 Batch #26 completed!
31 Batch #27 completed!
32 Batch #28 completed!
33 Batch #29 completed!
34 Batch #30 completed!
35 Batch #31 completed!
36 Batch #32 completed!
37 Batch #33 completed!
38 Batch #34 completed!
39 Batch #35 completed!
40 Batch #36 completed!
41 Batch #37 completed!
42 Batch #38 completed!
43 Batch #39 completed!
44 Batch #40 completed!
45 Batch #41 completed!
46 Batch #42 completed!
47 Batch #43 completed!
48 Batch #44 completed!
49 Batch #45 completed!
50 Batch #46 completed!
51 Batch #47 completed!
52 Batch #48 completed!
53 Batch #49 completed!
54 Score train: 0.8551020408163266
55 Score: 0.8544761904761905
```



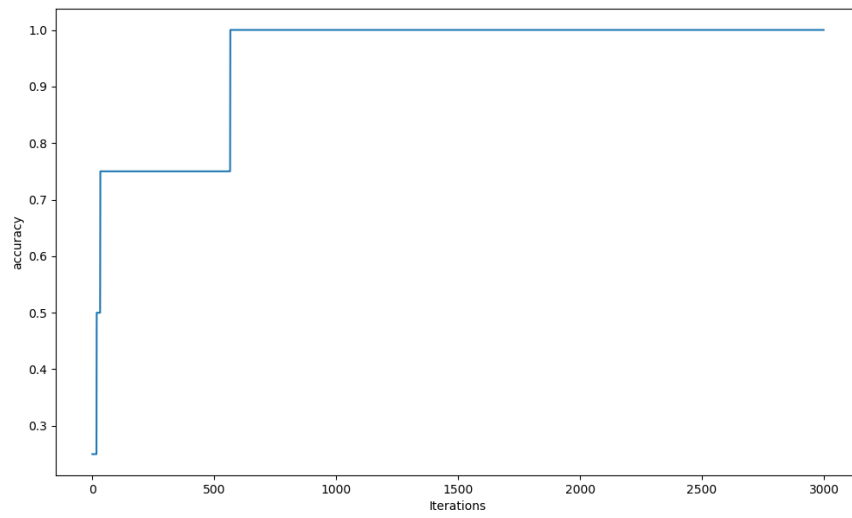
Wie man auf dem Plot sieht, senkt das Score wenn ein neues Batch eingefügt wird, da wir diese Beispiele noch nie gesehen haben. Dann wird die Korrektur bei den Gewichtsmatrizen gemacht und den Score steigt wieder. Ein Score von 85% finden wir eigentlich ganz toll für nur 250 Iterationen pro Batch und keine perfekte Anpassung des Netzes. Das zeigt, dass die Unterteilung des Datensatzes in Batches keine schlechte Idee ist.

Boolesche Funktionen

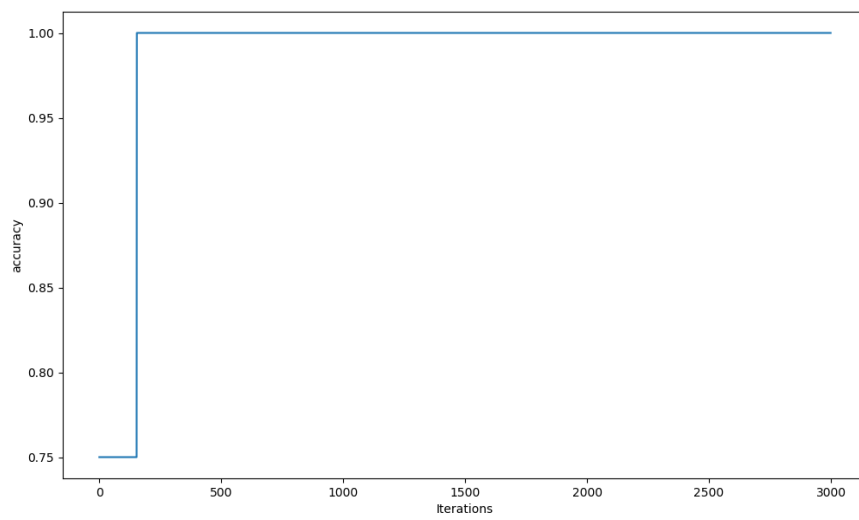
Output

```
1 Score and: 1.0
2 Score or: 1.0
3 Score xor: 1.0
```

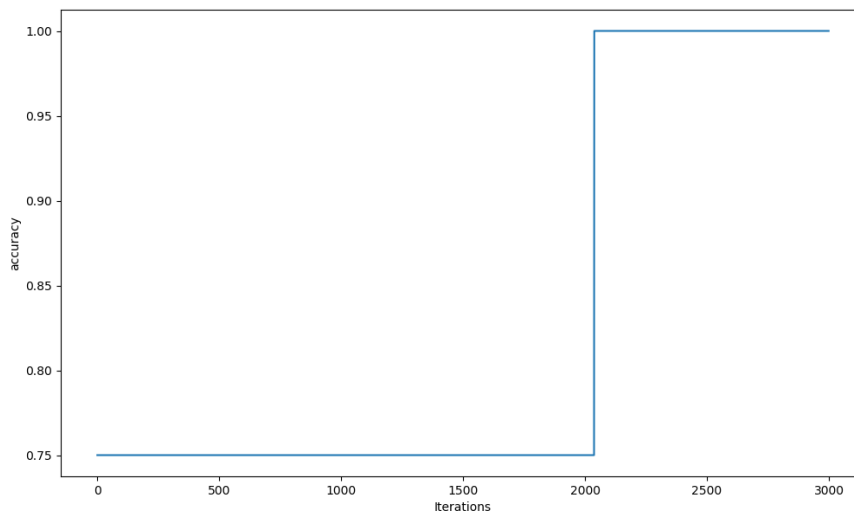
AND



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 Matrizenmultiplikationen in python geschrieben wurden. Nur eine Transponierung hat gefehlt in den Formeln, dazu kommen wir noch.

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 Matritze nicht darstellen kann.

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

```
1 def fit(self, X, y):
2     self.history = []
3     self.unique_labels = np.unique(y)
4     X_ = self.data_normalizer.fit_transform(X)
5     y_ = self.transform_y(y)
6     self.W1 = np.vstack((
7         np.random.randn(len(X_[0]), self.size_hidden),
8         np.ones(self.size_hidden)))
9     self.W2 = np.vstack((
```

```

10         np.random.randn(self.size_hidden, self.size_output),
11         np.ones(self.size_output)))

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

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

35             if self.print_score_per_batch:
36                 self.history.append(np.mean(self.unique_labels[o2.argmax(1)] == self.
unique_labels[yb.argmax(1)]))
37             else:
38                 self.history.append(self.score(X, y))

40         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

```

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

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

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

```


Sigmoid-Funktion und ihre Ableitung

```
1 @staticmethod
2     def sigmoid(x):
3         return 1 / (1 + np.exp(-x))
4
5     @staticmethod
6     def sigmoid_derived(x):
7         return x*(1-x)
```

Vollständiges Code boolean_functions_nn.py

```
1 import numpy as np
2 from NN import NN
3
4
5 X = np.array([
6     [0,0],
7     [0,1],
8     [1,0],
9     [1,1]
10 ])
11 y_and = np.array([a & b for a,b in X])
12 y_or = np.array([a | b for a,b in X])
13 y_xor = np.array([a ^ b for a,b in X])
14
15 # AND
16 nn = NN(max_iterations=3000, size_hidden=10, size_output=2, learning_rate=0.01)
17 nn.fit(X, y_and)
18 nn.plot_accuracies('./nn_and.png')
19 print('Score and: {}'.format(nn.score(X, y_and)))
20
21 # OR
22 nn = NN(max_iterations=3000, size_hidden=10, size_output=2, learning_rate=0.01)
23 nn.fit(X, y_or)
24 nn.plot_accuracies('./nn_or.png')
25 print('Score or: {}'.format(nn.score(X, y_or)))
26
27 # XOR
28 nn = NN(max_iterations=3000, size_hidden=10, size_output=2, learning_rate=0.01)
29 nn.fit(X, y_xor)
30 nn.plot_accuracies('./nn_xor.png')
31 print('Score xor: {}'.format(nn.score(X, y_xor)))
```

Vollständiges Code digits_small_nn.py

```
1 from Parser import get_data_set
2 from NN import NN
3
4
5 X_train, X_test, y_train, y_test = get_data_set('digits.data')
6 nn = NN()
7 nn.fit(X_train, y_train)
8 nn.plot_accuracies()
9 print('Score train: {}'.format(nn.score(X_train, y_train)))
10 print('Score: {}'.format(nn.score(X_test, y_test)))
```

Vollständiges Code digits_big_nn.py

```
1 from sklearn.datasets import fetch_mldata
2 from sklearn.model_selection import train_test_split
3 import numpy as np
4 from NN import NN
5
6 print('fetching data-set')
7 mnist = fetch_mldata('mnist-original', data_home='./Dataset/mnist.pkl/mnist_dataset/')
8 print('MNIST original fetched')
9 X = np.array(mnist.data, dtype=np.float64)
10 y = mnist.target
11 X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, test_size=0.3,
12                                                    random_state=1)
13
14
15 nn = NN(max_iterations=250, print_score_per_batch=True, batch_size=1000)
16 nn.fit(X_train, y_train)
17 nn.plot accuracies('./20_inner_big.png')
18 print('Score train: {}'.format(nn.score(X_train, y_train)))
19 print('Score: {}'.format(nn.score(X_test, y_test)))
```

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

```
1 import numpy as np
2
3 class DataNormalizer:
4     def fit(self, X):
5         self.mean = np.mean(X, axis=0)
6         self.var = np.var(X, axis=0) + np.nextafter(0, 1)
7
8     def transform(self, X):
9         return (X - self.mean) / self.var
10
11     def fit_transform(self, X):
12         self.fit(X)
13         return self.transform(X)
```

Vollständiges Code NN.py

```
1 from Classifier import Classifier
2 from DataNormalizer import DataNormalizer
3 import numpy as np
4 from matplotlib import pyplot as plt
5 import math
6
7
8 class NN(Classifier):
9     def __init__(self, max_iterations=3000, learning_rate=0.0020,
10                 size_hidden=20, size_output=10, print_score_per_batch=False, batch_size
11                 =7000):
12         self.data_normalizer = DataNormalizer()
13         self.size_hidden = size_hidden
14         self.size_output = size_output
15         self.max_iterations = max_iterations
16         self.learning_rate = learning_rate
17         self.print_score_per_batch = print_score_per_batch
```

```

17     self.W1 = None
18     self.W2 = None
19     self.unique_labels = None
20     self.batch_size = batch_size

22     def fit(self, X, y):
23         self.history = []
24         self.unique_labels = np.unique(y)
25         X_ = self.data_normalizer.fit_transform(X)
26         y_ = self.transform_y(y)
27         self.W1 = np.vstack((
28             np.random.randn(len(X_[0]), self.size_hidden),
29             np.ones(self.size_hidden)))
30         self.W2 = np.vstack((
31             np.random.randn(self.size_hidden, self.size_output),
32             np.ones(self.size_output)))

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

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

56                 if self.print_score_per_batch:
57                     self.history.append(np.mean(self.unique_labels[o2.argmax(1)] == self.
unique_labels[yb.argmax(1)]))
58                 else:
59                     self.history.append(self.score(X, y))

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

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

70         return o_, o1, o1_, o2, o2_

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

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

80     def transform_y(self, y):
81         y_ = np.zeros((len(y), len(self.unique_labels)))

```

```

82     y_in_unique = np.vectorize(lambda x: list(self.unique_labels).index(x))(y)
83     y_[range(len(y)), y_in_unique] = 1
84     return y_

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

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

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

100         if file_name is None:
101             plt.show()
102         else:
103             plt.savefig(file_name)

```

Vollständiges Code NeuralNetwork.py

```

1  from Classifier import Classifier
2  from DataNormalizer import DataNormalizer
3  import numpy as np
4  from matplotlib import pyplot as plt
5  from Parser import get_data_set

8  class NeuralNetwork(Classifier):
9      def __init__(self, layers=None, max_iterations=3000, learning_rate=0.0025):
10         """
11
12         :param layers: tuple defining the amount of neurons to be used pro layers, thereby
13         defining the network
14         :param max_iterations:
15         :param learning_rate:
16         """
17         self.data_normalizer = DataNormalizer()
18         self.W_ext = None
19         self.max_iterations = max_iterations
20         self.learning_rate = learning_rate
21         self.layers = layers
22         self.history = []

23         if self.layers is None:
24             self.k_layers = 3
25         else:
26             self.k_layers = len(self.layers)

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

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

```

```

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

47         batch_size = 1000
48         for batch_start in range(0, len(X_), batch_size):
49             Xb = X_[batch_start:batch_start + batch_size + 1]
50             yb = y_[batch_start:batch_start + batch_size + 1]
51             for it in range(self.max_iterations):
52                 O_s = self.get_O_s(Xb)
53                 D_s = [(o * (1.0 - o)) for o in O_s[2::2]]
54                 e = (O_s[-1] - yb)
55                 der_e_s = self.get_der_e_s(D_s, e)
56                 delta_W_ext = self.get_delta_W_ext(der_e_s, O_s[2::2])

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

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

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

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

71     def get_O_s(self, X):
72         O_s = [X]
73         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)

79         return O_s

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]

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

90         return der_e_s

92     def get_delta_W_ext(self, der_e_s, O_s):
93         delta_W_ext = []

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)

100         return np.array(delta_W_ext) * -self.learning_rate

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

```

```

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)]

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")

117         if file_name is None:
118             plt.show()
119         else:
120             plt.savefig(file_name)

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)))

```