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 uns hier auf die Beschreibung unserer simpleren Implementierung konzentrieren. Beide Implementierungen sind am Ende des Blattes zu sehen.

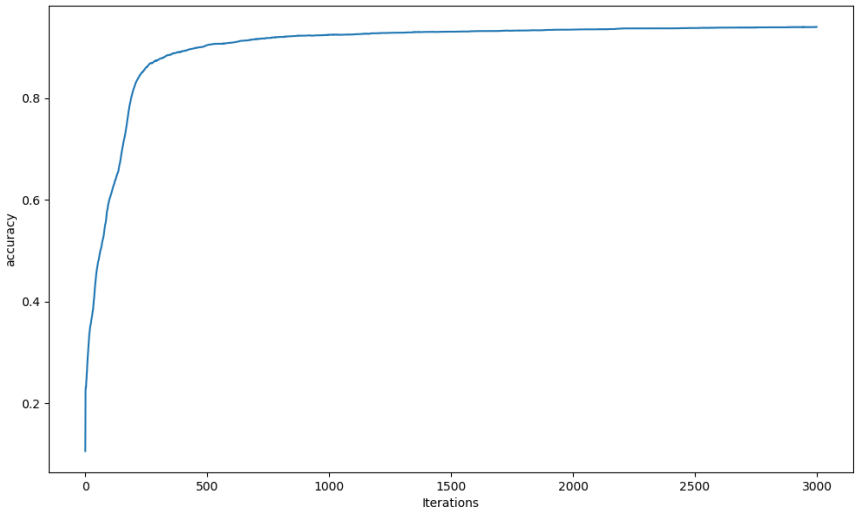
Evaluation

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 11dass 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, benutzen die Leute echt viele hidden Layers um die Daten gut zu klassifizieren. Leider sind beide unsere Implementierung und Rechnern nicht so perfekt für solche Ansätze.

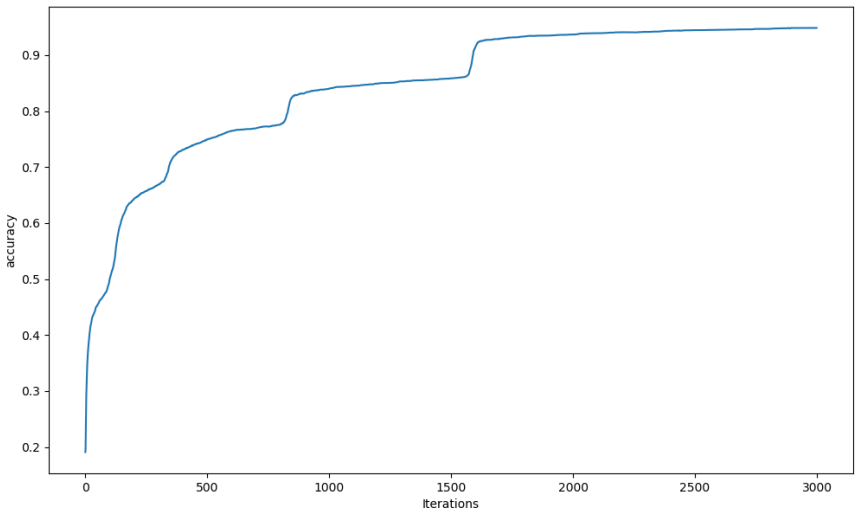
Fehler über die Iterationen

1 Hidden Layer mit 15 Neuronen



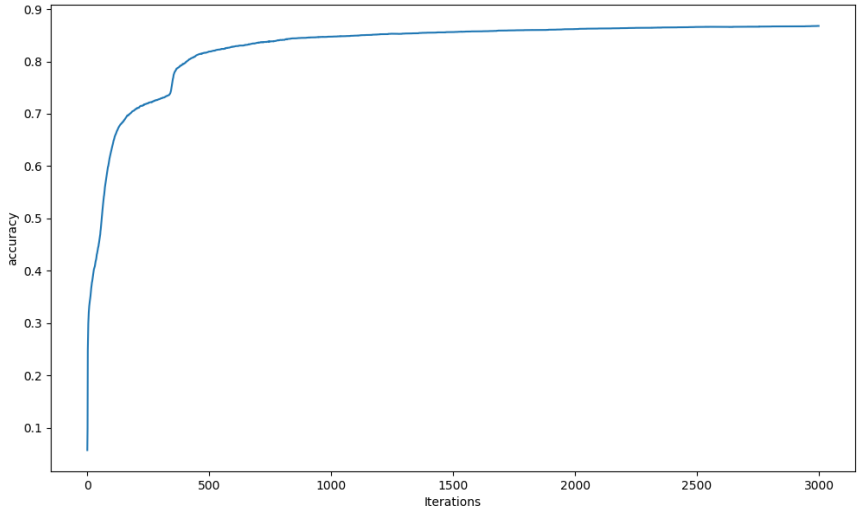
Output:
Score: 0.88458781362

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 lernenïst. Yey M

Output

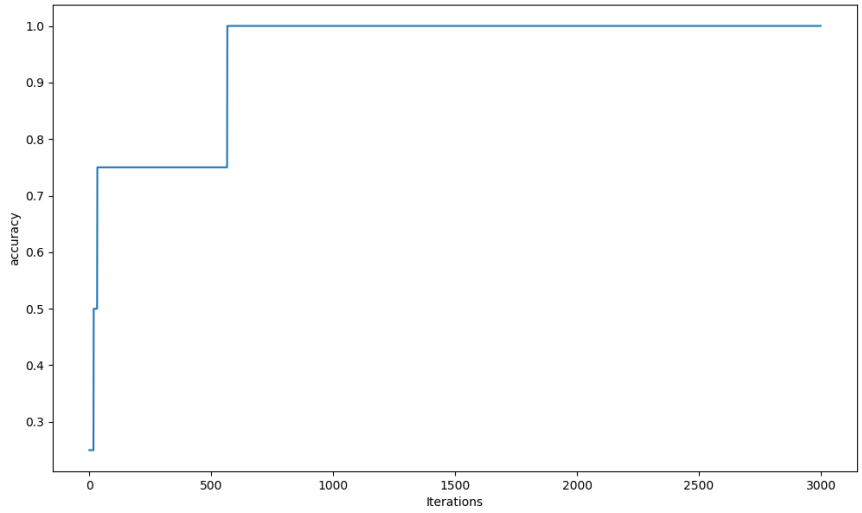
```
Score train: 0.7744314689612785
Score: 0.7548387096774194

\includegraphics[height=8cm]{./2_hidden_layers_20_15_with_7_batches.png}
```

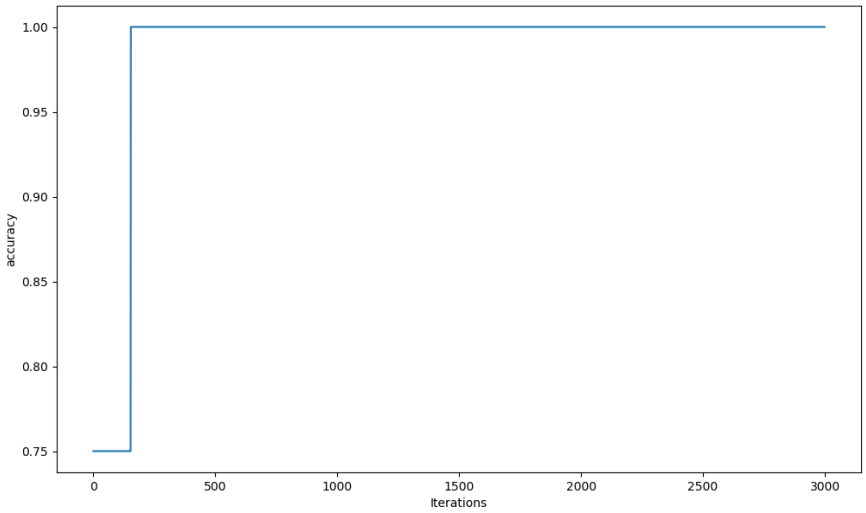
1 Hidden Layer, 20 Neuronen, großen Datensatz

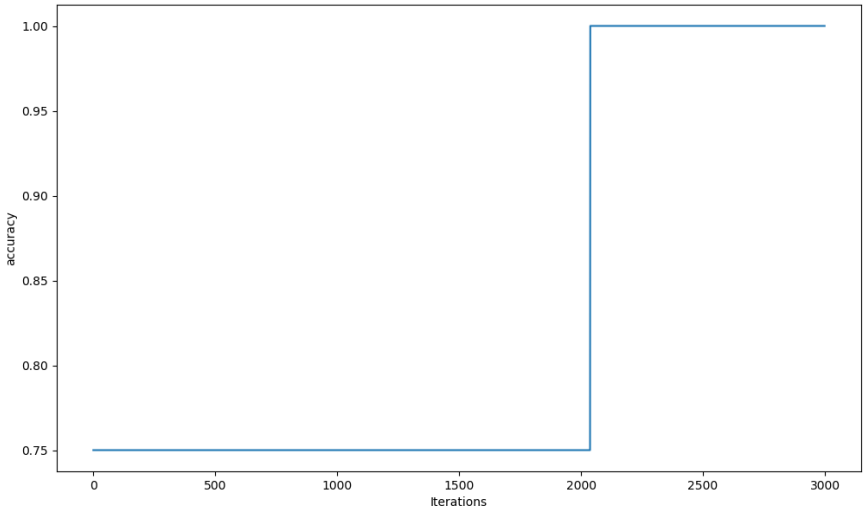
Boolesche Funktionen

Output

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


1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
accuracy
0
500
1000
1500
2000
2500
3000
Iterations


1.00
0.95
0.90
0.85
0.80
0.75
accuracy
0
500
1000
1500
2000
2500
3000
Iterations



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 5000 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.

```
    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 = 5000
        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)
                self.history.append(self.score(X, y))
```

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 digits_big_nn.py analog

```
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 NN.py

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


class NN(Classifier):
    def __init__(self, max_iterations=3000, learning_rate=0.0020, size_hidden=20,
    size_output=10):
        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.W1 = None
```

```
        self.W2 = None
        self.unique_labels = None

    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 = 5000
        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)
                self.history.append(self.score(X, y))

    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):
            O_i_minus_1 = np.c_[O_s[-1], np.ones(len(X))]  # extend to get o(i-1)^
            O_s.append(O_i_minus_1)
            O_i = NeuralNetwork.sigmoid(O_i_minus_1.dot(self.W_ext[i]))
            O_s.append(O_i)

        return O_s

    def get_der_e_s(self, D_s, e):
        W = [w[1:] for w in self.W_ext]
        der_e_l = e * D_s[-1]
        der_e_s = [der_e_l]

        for i in range(self.k_layers - 1):
            der_e_i = D_s[-i - 2] * (der_e_s[0]).dot(W[-i - 1].T)
            der_e_s = [der_e_i] + der_e_s

        return der_e_s

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

        for i in range(self.k_layers):
            o_i_ext = np.c_[O_s[i], np.ones(len(O_s[i]))]
            delta_W_i = der_e_s[i].T.dot(o_i_ext)
            delta_W_ext.append(delta_W_i.T)

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

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

    def predict_(self, X):
        O_s = self.get_O_s(X)
        results = O_s[-1]
        return self.unique_labels[results.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)


X_train, X_test, y_train, y_test = get_data_set('digits.data')
nn = NeuralNetwork()
nn.fit(X_train, y_train)
nn.plot_accuracies('./2_hidden_layers_20_15_with_7_batches.png')
print('Score train: {}'.format(nn.score(X_train, y_train)))
print('Score: {}'.format(nn.score(X_test, y_test)))
```