Maschinelles Lernen

AB5

AB5.1 Neuronale Netze und Ziffern

Implementieren Sie ein neuronales Netz und wenden Sie dieses auf alle Klassen des Digit-Datensatzes an. Um eine bessere Genauigkeit zu erreichen, können Sie anschließend den kompletten Trainingsdatensatz benutzen. Hier müssen ggf. die Merkmale normalisiert werden. Variieren Sie die Anzahl hidden-Layers bzw. Anzahl Neuronen und vergleichen Sie anschließend die Ergebnisse. Plotten Sie wie sich die Klassifikationsgenauigkeit über die Iterationen verändert.

Bei der Implementierung (siehe unten) gibt scheinbar einen Fehler, da die Konfusionsmatrizen fehlerhaft aussehen. Egal wie das Netz konfiguriert und trainiert wird, alle Testdaten werden immer auf nur *ein Label* abgebildet, d. h. es ist immer nur eine Spalte der Konfusionsmatrizen gefüllt. Von den rund 7000 Testdaten ergeben ca. höchstens zehn Stück ein anderes Label. Dadurch ist die Genauigkeit natürlich nur sehr klein (unter 10 %, siehe Plots).
Beim Debuggen bin ich darauf gestoßen, dass ab einer bestimmten Gewichtsmatrix $\overline{W}_i$ in der Kette von Multiplikationen die Output-Vektoren $\overrightarrow{O}_i$ für alle $x_{test} \in X_{test}$ gleich sind. D. h. es gibt folgende Beobachtung:

$$\hat{O}_{ix_p} \neq \hat{O}_{ix_q} \quad x_p, x_q \in X_{test}$$

aber

$$\hat{O}_{ix_p} \cdot \overline{W}_i = \hat{O}_{ix_q} \cdot \overline{W}_i$$

Somit sind ab diesem einen i alle Ouputs $\{\overrightarrow{O}_j \,|\, i \leq j \leq L\}$ [1] für alle x_{test} gleich, d. h. auch der finale Output-Vektor.
Ich habe auch versucht die *RELU*- statt der Sigmoidalfunktion benutzen, weil ich dachte, es gibt vielleicht zu kleine Werte, sodass Parameter zu Null werden, aber das hat nichts verändert.

Aus diesem Grund wurde der größere Testdatensatz nicht verwendet.

[1] L ist die Anzahl aller Layer.

Für die Plots der Klassifikationsgenauigkeiten wurden einmal zwei *hidden layers* mit 40 bzw. 50 Elementen (siehe `main.py`) und ein zweites Mal drei *hidden layers* mit 180, 100 bzw. 50 Elementen verwendet.

Beim zweiten Versuch, war die Genauigkeit zwar etwas größer, aber ist trotzdem nicht konvergiert.

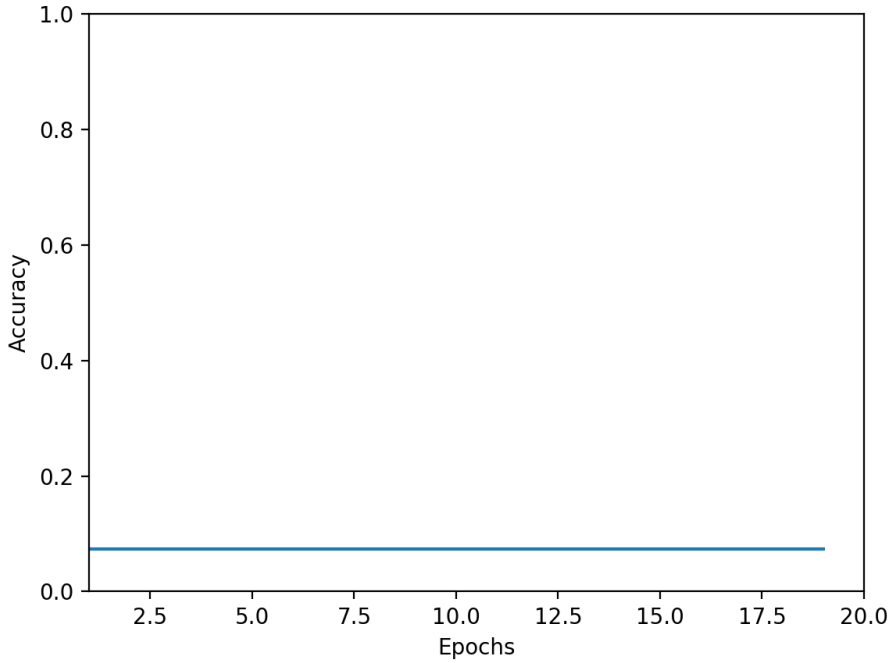

Abbildung 1: Klassifikationsgenauigkeit pro Anzahl von Epochen für $l = (40, 50)$ und $\gamma = 10^{-3}$

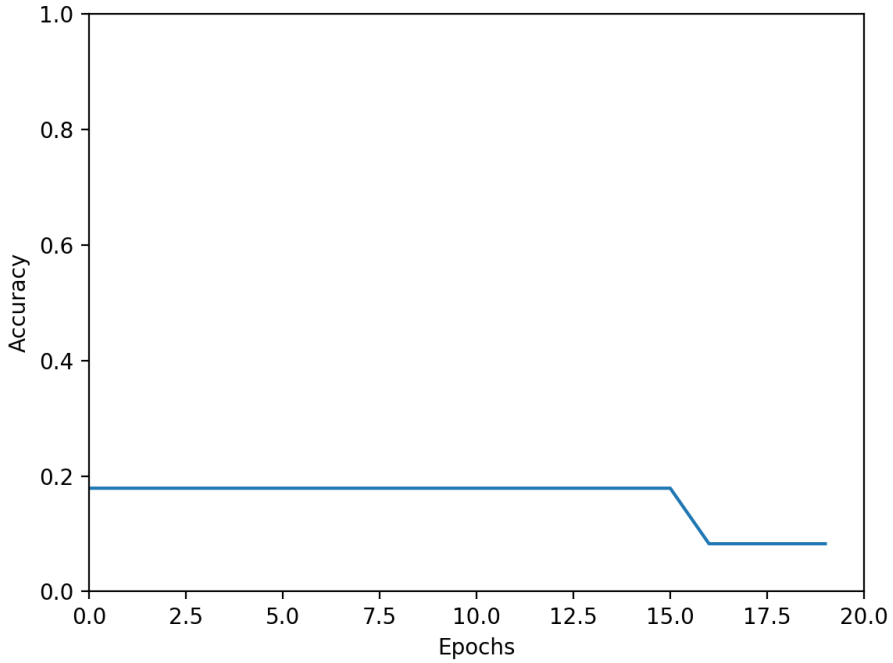

Abbildung 2: Klassifikationsgenauigkeit pro Anzahl von Epochen für $l = (180, 100, 50)$ und $\gamma = 10^{-5}$

Das neuronale Netz wurde wie folgt implementiert.

Listing 1: src/main.py

```
"""
Run from parent directory, e.g. 'pipenv run python ./src/main.py'
"""

import matplotlib.pyplot as plt
# import numpy as np
import pandas as pd

from neuralnetwork import NeuralNetwork, BatchMethod


def main():
    data_frame = pd.read_csv('res/zip.train', header=None, sep=' ')
    X_train = data_frame.iloc[:, 1:-1].values
    y_train = data_frame.iloc[:, 0].values
    data_frame = pd.read_csv('res/zip.test', header=None, sep=' ')
    X_test = data_frame.iloc[:, 1:].values
    y_test = data_frame.iloc[:, 0].values

    accuracies = []
    NUM_EPOCHS = 20
    neural_network = NeuralNetwork(X_train, y_train, 16*16, 10, [180, 100,
       50])
    neural_network.train(
        X_train, y_train,
        batch_size=32,
        learning_constant=1e-5,
        num_epochs=NUM_EPOCHS,
        callback=lambda weights: accuracies.append(
            neural_network.accuracy(X_test, y_test, weights=weights)
        )
    )

    plt.plot(accuracies)
    plt.axis([0, NUM_EPOCHS, 0, 1])
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.show()


if __name__ == '__main__':
    main()
```

Listing 2: src/classifier.py

```
from abc import ABC, abstractmethod

import numpy as np


class Classifier(ABC):
    """
    Abstract superclass for all classifiers
    """

    def __init__(self, X, y, num_classes=None):
        self.X = X
        self.y = y
        self.num_classes = num_classes or len(set(y))

    @classmethod
    def trained(cls, X, y):
        instance = cls(X, y)
        instance.train(X, y)
        return instance

    @abstractmethod
    def train(self, X, y):
        pass

    @abstractmethod
    def predict_label(self, x_test):
        pass

    def get_confusion_matrix(self, X_test, y_test, shape=None, **kwargs):
        if shape is None:
            shape = (self.num_classes, self.num_classes)
        matrix = np.zeros(shape=shape)
        for i, x in enumerate(X_test):
            true_label = y_test[i]
            predicted_label = self.predict_label(x, **kwargs)
            matrix[true_label][predicted_label] += 1
        return matrix

    def print_confusion_matrix(self, X_test, y_test):
        matrix = self.get_confusion_matrix(X_test, y_test)
        print(matrix)
        print('accuracy: {}'.format(self.accuracy(X_test, y_test, matrix)))
        return matrix

    def accuracy(self, X_test, y_test, matrix=None, **kwargs):
        if matrix is None:
            matrix = self.get_confusion_matrix(X_test, y_test, **kwargs)
        return np.sum(np.diag(matrix)) / len(X_test)
```

Listing 3: src/neuralnetwork.py

```
import math
from typing import Any, Callable, List, Tuple

import numpy as np

from classifier import Classifier
import utils
from utils import hot_one_encode_ints as hoe_ints, hot_one_decode_int as
   hod_int


class BatchMethod:
    BATCH = 0
    MINI_BATCH = 1
    ONLINE_BATCH = 2


class NeuralNetwork(Classifier):
    learning_constant = 1e-3

    def __init__(self, X, y,
                 size_in: int, size_out: int,
                 hidden_layers: List[int],
                 hot_one_encode_y: Callable[[int, Any], np.ndarray] =
                    hoe_ints,
                 hot_one_decode: Callable[[np.ndarray], int] = hod_int):
        """
        'size_in' Number of features.
        'size_out' Number of classes.
        'hidden_layers' Defines how many nodes each layer has.
        'hot_one_encode_y' Hot-one encodes a label.
        """
        super().__init__(X, y, num_classes=size_out)
        assert len(hidden_layers) > 0, 'Need at least 1 hidden layer.'

        self.size_in = size_in
        self.size_out = size_out
        self.hidden_layers = hidden_layers
        self.hot_one_encode_y = hot_one_encode_y
        self.hot_one_decode = hot_one_decode

    def train(self, X, y, *,
              num_epochs=10,
              batch_method=BatchMethod.MINI_BATCH,
              batch_size=None,
              learning_constant=1e-3,
              callback=None):
        N = len(X)
        if batch_method == BatchMethod.MINI_BATCH:
            if batch_size is None:
                batch_size = N // 20
        elif batch_method == BatchMethod.BATCH:
            batch_size = N
            if batch_size is not None:
                print('WARNING: batch_size given but ignored.')
        elif batch_method == BatchMethod.ONLINE_BATCH:
            batch_size = 1
            if batch_size is not None:
                print('WARNING: batch_size given but ignored.')
        else:
            raise ValueError('Invalid batch method.')
```

```
        X_shuffled = X[:]
        np.random.shuffle(X_shuffled)

        weights, augmented_weights = self._initialize_weight_matrices()
        num_batches = math.ceil(N / batch_size)
        for epoch in range(num_epochs):
            for batch_index in range(num_batches):
                batch = X_shuffled[batch_index:(batch_index + batch_size)]
                corrections = [
                    np.zeros(matrix.shape)
                    for matrix in augmented_weights
                ]
                for i, x in enumerate(batch):
                    new_corrections = self.backpropagation(
                        weights,
                        *self.feed_forward(augmented_weights, x, y[i]),
                        learning_constant=learning_constant,
                    )
                    corrections = self._sum_matrix_lists(
                        corrections,
                        new_corrections
                    )

                weights, augmented_weights = self._apply_weight_corrections(
                    augmented_weights,
                    corrections
                )
            if callable(callback):
                callback(augmented_weights)

        self.weights = augmented_weights

    def feed_forward(self, weights, x, y_i):
        outputs = [self._O_hat(x)]
        diagonals = []
        s = utils.sigmoid
        sd = utils.sigmoid_d

        # Start at 1 to match math notation.
        for i, augmented_matrix in enumerate(weights, start=1):
            O_hat_prev = outputs[i - 1]
            W = augmented_matrix
            O = s(O_hat_prev.T @ W)
            D = np.diag(sd(O))

            outputs.append(self._O_hat(O))
            diagonals.append(D)
        try:
            t = self.hot_one_encode_y(self.num_classes, int(y_i))
        except IndexError as e:
            raise ValueError((
                'Cannot hot one encode "{}" because too few outputs '
                '(change "size_out" argument for "__init__")'
            ).format(int(y_i))) from e

        # O is the final (unaugmented) output.
        error = O - t
        return outputs, diagonals, error

    def backpropagation(self, weights, outputs, diagonals, error,
                        *,
```

```
                    learning_constant):
        """
        'weights' Weight matrices W_i.
        'outputs' Augmented output vectors.
        'diagonals' Diagonal matrices D_i containing derivates
        'error' Error derivate vector e
        """
        N = len(diagonals)
        deltas = []
        i_max = N - 1
        for i in range(i_max, -1, -1):
            D = diagonals[i]
            if i == i_max:
                delta = D @ error
            else:
                W = weights[i + 1]
                delta = D @ W @ prev_delta
            # Prepend delta to keep order equal to the other variables.
            deltas.insert(0, delta)
            prev_delta = delta

        # The corrections' indices must be ascending
        # to match the order of weight matrices.
        return [
            -learning_constant * np.outer(delta, outputs[i]).T
            for i, delta in enumerate(deltas)
        ]

    def predict_label(self, x_test, weights=None):
        """
        'weights' Override self.weights, used for accuracy measurement.
        """
        if weights is None:
            weights = self.weights

        O_hat_prev = self._O_hat(x_test)
        s = utils.sigmoid

        # Start at 1 to match math notation.
        for i, augmented_matrix in enumerate(weights, start=1):
            W = augmented_matrix
            O = s(O_hat_prev.T @ W)
            O_hat_prev = self._O_hat(O)

        return self.hot_one_decode(O)

    def _initialize_weight_matrices(self) -> Tuple[List[np.ndarray]]:
        matrices = []
        prev_dim_size = self.size_in
        for layer_size in self.hidden_layers:
            shape = (prev_dim_size, layer_size)
            matrix = np.random.uniform(0, 1, shape)
            matrices.append(matrix)
            prev_dim_size = layer_size

        shape = (prev_dim_size, self.size_out)
        matrix = np.random.uniform(0, 1, shape)
        matrices.append(matrix)
        return matrices, [utils.augmented(matrix) for matrix in matrices]

    def _apply_weight_corrections(self, augmented_weights, corrections):
        corrected_weights = self._sum_matrix_lists(
```

```
            augmented_weights,
            corrections
        )
        return (
            [utils.unaugmented(matrix) for matrix in corrected_weights],
            corrected_weights,
        )

    def _sum_matrix_lists(self, a, b):
        # TODO: Use zip
        if len(a) != len(b):
            raise ValueError('Unequally long lists.')
        return [a_i + b[i] for i, a_i in enumerate(a)]

    def _O_hat(self, O):
        return utils.augmented(O)
```

Listing 4: src/utils.py

```
from collections import Counter
import math

import numpy as np
import numpy.linalg
import numpy.matlib


def hot_one_encode_ints(num_classes, ints):
    """
    'ints' May also be a single int.
    """
    # See https://stackoverflow.com/a/42874726/6928824
    targets = np.array(ints).reshape(-1)
    one_hot_targets = np.eye(num_classes)[targets]
    return one_hot_targets.reshape(-1)


def hot_one_decode_int(encoded):
    return np.argmax(encoded, axis=0)


def sigmoid(x):
    return 1.0 / (1 + np.exp(-x))


def sigmoid_d(x):
    return sigmoid(x) * (1 - sigmoid(x))


def relu(x):
    return np.clip(x, 0, np.inf)


def relu_d(x):
    return (x >= 0).astype(float)


def augmented(array, append=True):
    """Add ones to 0-axis."""
    shape = array.shape
    ones = np.ones((1, *shape[1:]))
    if append:
        items = (array, ones)
    else:
        items = (ones, array)
    return np.concatenate(items, axis=0)


def unaugmented(array, appended=True):
    """Inverse operation to 'augmented'."""
    if appended:
        s = np.s_[:-1]
    else:
        s = np.s_[1:]
    return array[s]
```