Lernverfahren autonomer Roboter - Übung 10

G10
Andre Osse
Waldemar Stockmann
Markus Strehling
Tobias Hahn

January 26, 2017

1 Multilayer Neural Network and Backpropagation

1.1 Implementation

Das Netzwerk wurde wie gewünscht implementiert, der Code ist wie folgt:

../code/multi.py

```
"""Multilayer Neural Network."""
import numpy as np
from tools import linear, linear_derivative, relu, relu_derivative


class FullyConnectedLayer(object):
    """Represents a trainable fully connected layer.

    Parameters
    ----------
    I : int
        inputs (without bias)

    J : int; outputs

    g : function: array-like -> array-like
        activation function y = g(a)

    gd : function: array-like -> array-like
        derivative g'(a) = gd(y)

    std_dev : float
        standard deviation of the normal distribution that we use to draw
        the initial weights

    verbose : int, optional
        verbosity level
    """
    def __init__(self, I, J, g, gd, std_dev, verbose=0):
        self.I = np.prod(I) + 1  # Add bias component
        self.J = J
        self.g = g
        self.gd = gd
        self.std_dev = std_dev

        self.W = np.empty((self.J, self.I))

        if verbose:
            print("Fully connected layer (%d nodes, %d x %d weights)"
                  % (self.J, self.J, self.I))

    def initialize_weights(self, random_state):
        """Initialize weights randomly.

        Parameters
        ----------
        random_state : RandomState or int
            random number generator or seed
        """
        #if type is int it is a seed, else we can use it as a RandomState
        if (type(random_state) is int):
            random_state = np.random.RandomState(random_state)
        self.W = random_state.normal(0, self.std_dev, self.W.shape)

    def get_output_shape(self):
        """Get shape of the output.

        Returns
        -------
        shape : tuple
            shape of the output
        """
        return (self.J,)
```

```
    def forward(self, X):
        """Forward propagate the output of the previous layer.

        Parameters
        ----------
        X : array-like, shape = [N, I or self.I-1]
            input

        Returns
        -------
        Y : array-like, shape = [N, J]
            output
        """
        self.X = X
        N = X.shape[0]
        D = np.prod(X.shape[1:])
        if D != self.I - 1:
            raise ValueError("shape = " + str(X.shape))
        self.Z = np.ones((X.shape[0], X.shape[1]+1))
        self.Z[:,1:] = X
        self.Y = self.g(self.Z.dot(self.W.T))

        return self.Y

    def backpropagation(self, dEdY):
        """Backpropagate errors of the next layer.

        Parameters
        ----------
        dEdY : array-like, shape = [N, J]
            errors from the next layer

        Returns
        -------
        dEdX : array-like, shape = [N, I+1 or self.I]
            errors from this layer

        Wd : array-like, shape = [J, self.I]
            derivatives of the weights
        """
        if dEdY.shape[1] != self.J:
            raise ValueError("%r != %r" % (len(dEdY), self.J))

        dEdY = dEdY * self.gd(self.Y)
        dEdX = dEdY.dot(self.W[:,1:])
        Wd = dEdY.T.dot(self.Z)

        return dEdX, Wd

    def get_weights(self):
        """Get current weights.

        Returns
        -------
        W : array-like, shape = [J * I + 1 or self.I]
            weight matrix
        """
        return self.W.flat

    def set_weights(self, W):
        """Set new weights.

        Parameters
        ----------
        W : array-like, shape = [J * I + 1 or self.I]
            weight matrix
        """
        self.W = W.reshape((self.J, self.I))

    def num_weights(self):
```

```
        """Get number of weights.

        Returns
        -------
        K : int
            number of weights
        """
        return self.W.size

    def __getstate__(self):
        # This will be called by pickle.dump, so we remove everything that
        # requires too much memory
        d = dict(self.__dict__)
        if "X" in d:
            del d["X"]
        if "Y" in d:
            del d["Y"]
        return d


class MultilayerNeuralNetwork(object):
    """Multilayer Neural Network (MLNN).

    Parameters
    ----------
    D : int
        number of inputs

    F : int
        number of outputs

    layers : list of dicts
        layer definitions

    training : string
        must be either classification or regression and defines the
        activation function of the last layer as well as the error function

    std_dev : float
        standard deviation of the normal distribution that we use to draw
        the initial weights

    verbose : int, optional
        verbosity level
    """

    def __init__(self, D, F, layers, training="classification", std_dev=0.05,
                 verbose=0):
        self.D = D
        self.F = F

        # Initialize layers
        self.layers = []
        I = self.D  # Dimensions of the input layer
        for layer in layers:
            l = None
            if layer["type"] == "fully_connected":
                l = FullyConnectedLayer(
                    I, layer["num_nodes"], relu, relu_derivative, std_dev,
                    verbose)
                I = l.get_output_shape()
            else:
                raise NotImplementedException("Layer type '%s' is not "
                                              "implemented." % layer["type"])
            self.layers.append(l)
        if training == "regression":
            self.layers.append(FullyConnectedLayer(
                I, self.F, linear, linear_derivative, std_dev, verbose))
            self.error_function = "sse"
        else:
            raise ValueError("Unknown 'training': %s" % training)
```

```
    def initialize_weights(self, random_state):
        """Initialize weights randomly.

        Parameters
        ----------
        random_state : RandomState or int
            random number generator or seed
        """
        for layer in self.layers:
            layer.initialize_weights(random_state)

    def error(self, X, T):
        """Calculate the Cross Entropy (CE).

        .. math::

            E = -\sum_n \sum_f ln(y^n_f) t^n_f,

        where n is the index of the instance, f is the index of the output
        component, y is the prediction and t is the target.

        Parameters
        ----------
        X : array-like, shape = [N, D]
            each row represents an instance

        T : array-like, shape = [N, F]
            each row represents a target

        Returns
        -------
        E : float
            error: SSE for regression, cross entropy for classification
        """
        if len(X) != len(T):
            raise ValueError("Number of samples and targets must match")

        # Compute error of the dataset

        if (self.error_function == "sse"):
            return 1/2 * np.sum(np.power(self.predict(X) - T, 2))
        else:
            return np.sum(np.log(self.predict(X)) * T)

    def numerical_gradient(self, X, T, eps=1e-5):
        """Compute the derivatives of the weights with finite differences.

        This function can be used to check the analytical gradient
        numerically. The partial derivative of E with respect to w is
        approximated through

        .. math::

            \partial E / \partial w = (E(w+\epsilon) - E(w-\epsilon)) /
                                      (2 \epsilon) + O(\epsilon^2),

        where :math:`\epsilon` is a small number.

        Parameters
        ----------
        X : array-like, shape = [N, D]
            input

        T : array-like, shape = [N, F]
            desired output (target)

        eps : float, optional
            small number, you can make eps smaller to increase the accuracy
            of the differentiation until roundoff errors occur

```

```
        Returns
        -------
        wd : array-like, shape = [K,]
            weight vector derivative
        """
        w = self.get_weights()
        w_original = w.copy()
        wd = np.empty_like(w)
        for k in range(len(w)):
            w[k] = w_original[k] + eps
            self.set_weights(w)
            Ep = self.error(X, T)
            w[k] = w_original[k] - eps
            self.set_weights(w)
            Em = self.error(X, T)
            w[k] = w_original[k]
            wd[k] = (Ep - Em) / (2.0 * eps)
        self.set_weights(w_original)
        return wd

    def gradient(self, X, T, get_error=False):
        """Calculate the derivatives of the weights.

        Parameters
        ----------
        X : array-like, shape = [N, D]
            input

        T : array-like, shape = [N, F]
            desired output (target)

        get_error : bool, optional (default: False)
            Return a tuple (g, e), otherwise only g will be returned

        Returns
        -------
        g : array-like, shape = [K,]
            gradient of weight vector

        e : float, optional
            error
        """
        e = self.error(X, T)
        Y = self.predict(X)
        D = Y - T

        part_grad = []
        for l in reversed(self.layers):
            D, g = l.backpropagation(D)
            part_grad.insert(0, g.flat)
        if not get_error:
            return np.concatenate(part_grad)
        else:
            return (np.concatenate(part_grad), e)

    def get_weights(self):
        """Get current weight vector.

        Returns
        -------
        w : array-like, shape (K,)
            weight vector
        """
        return np.concatenate([self.layers[l].get_weights()
                               for l in range(len(self.layers))])

    def set_weights(self, w):
        """Set new weight vector.

        Parameters
        ----------
```

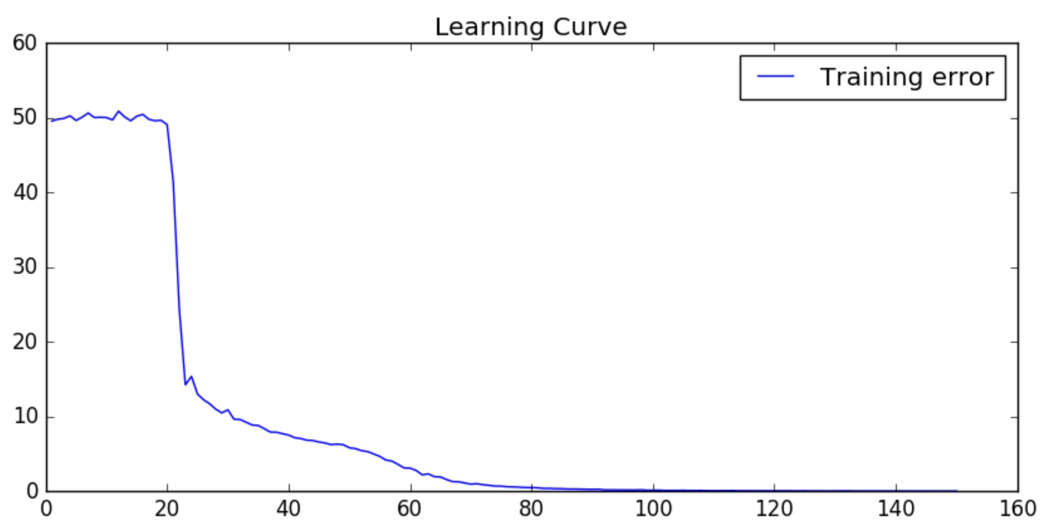
```
        w : array-like, shape=[K,]
            weight vector
        """
        i = 0
        for l in range(len(self.layers)):
            k = self.layers[l].num_weights()
            self.layers[l].set_weights(w[i:i + k])
            i += k

    def predict(self, X):
        """Predict values.

        Parameters
        ----------
        X : array-like, shape = [N, D]
            each row represents an instance

        Returns
        -------
        Y: array-like, shape = [N, F]
            each row represents a prediction
        """
        # Forward propagation
        for l in self.layers:
            l.X = X
            X = l.forward(X)
        return X
```

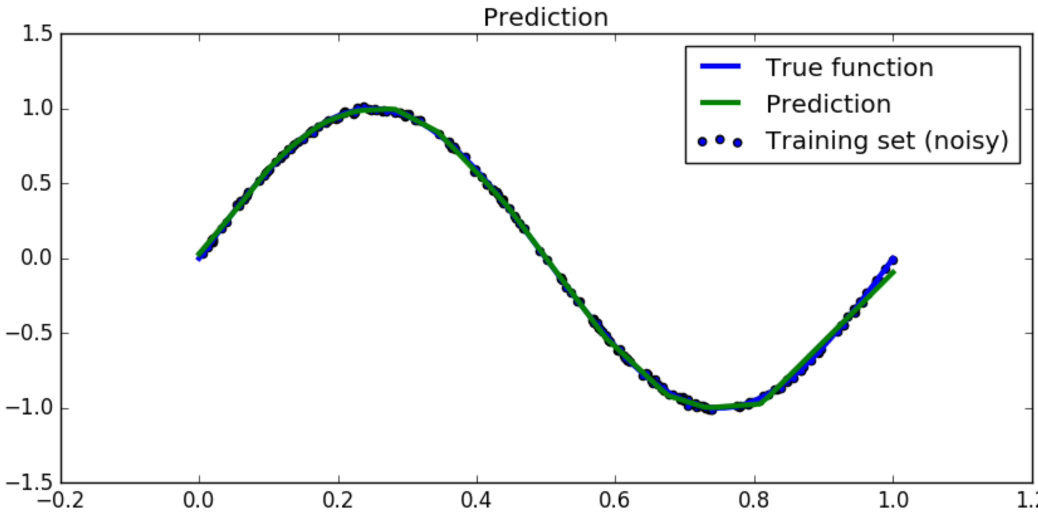
Außerdem wurde der Test ausgeführt, mit folgendem Ergebnis:

```
Fully connected layer (20 nodes, 20 x 11 weights)
Fully connected layer (10 nodes, 10 x 21 weights)
Fully connected layer (3 nodes, 3 x 11 weights)
Checking gradients up to 5 positions after decimal point...OK
```

1.2 Sinus lernen

Der Sinus wurde gelernt, folgende Ausgaben wurden mit dem Testprogramm erzielt:





2 Roboter lernen

2.1 Code

../code/sac.py

```
"""Train multilayer neural network with MBSGD on Sarcos data set."""
import numpy as np
import pickle
from sarcos import download_sarcos
from sarcos import load_sarcos
from sarcos import nMSE
from sklearn.preprocessing import StandardScaler
from multilayer_neural_network import MultilayerNeuralNetwork
from minibatch_sgd import MiniBatchSGD
from sklearn.linear_model import LinearRegression

def printnMSE(model, X, Y, name):
    print(name)
    # Print nMSE on test set
    Y_pred = model.predict(X)
    for f in range(Y_pred.shape[1]):
        print("Dimension %d: nMSE = %.2f %%"
              % (f + 1, 100 * nMSE(Y_pred[:, f], Y[:, f])))

def LinReg(X, Y, X_test, Y_test):
    model = LinearRegression()
    model.fit(X, Y)
    print("## Linear Regression ##")
    printnMSE(model, X, Y, "Train data")
    printnMSE(model, X_test, Y_test, "Test data")
    return model

def NeuralNet(X, Y, X_test, Y_test):
    layers = [{"type":"fully_connected", "num_nodes": 50}]
    mlnn = MultilayerNeuralNetwork(D=21, F=7, layers=layers, training="regression", std_dev=0.01)
    model = MiniBatchSGD(net=mlnn, epochs=100, batch_size=32, alpha=0.005, eta=0.5, random_state=0, verbose=0)
    model.fit(X,Y)
    print("## Neural Net ##")
    printnMSE(model, X, Y, "Train data")
    printnMSE(model, X_test, Y_test, "Test data")
    return model

if __name__ == "__main__":
```

```
    np.random.seed(0)

    # Download Sarcos dataset if this is required
    #download_sarcos()

    # Load training set and test set
    X, Y = load_sarcos("train")
    X_test, Y_test = load_sarcos("test")
    # Scale targets
    target_scaler = StandardScaler()
    Y = target_scaler.fit_transform(Y)
    Y_test = target_scaler.transform(Y_test)

    # Train model (code for exercise 10.2 1/2/3)
    model = LinReg(X, Y, X_test, Y_test)
    model = NeuralNet(X, Y, X_test, Y_test)
    # Store learned model, you can restore it with
    # model = pickle.load(open("sarcos_model.pickle", "rb"))
    # and use it in your evaluation script
    pickle.dump(model, open("sarcos_model.pickle", "wb"))
```

2.2 Output

```
## Linear Regression ##
Train data
Dimension 1: nMSE = 7.36 %
Dimension 2: nMSE = 10.26 %
Dimension 3: nMSE = 9.11 %
Dimension 4: nMSE = 5.12 %
Dimension 5: nMSE = 14.56 %
Dimension 6: nMSE = 27.44 %
Dimension 7: nMSE = 6.52 %
Test data
Dimension 1: nMSE = 7.42 %
Dimension 2: nMSE = 10.10 %
Dimension 3: nMSE = 9.18 %
Dimension 4: nMSE = 5.13 %
Dimension 5: nMSE = 14.13 %
Dimension 6: nMSE = 28.24 %
Dimension 7: nMSE = 6.46 %
## Neural Net ##
Train data
Dimension 1: nMSE = 5.96 %
Dimension 2: nMSE = 4.94 %
Dimension 3: nMSE = 4.13 %
Dimension 4: nMSE = 2.23 %
Dimension 5: nMSE = 6.49 %
Dimension 6: nMSE = 7.95 %
Dimension 7: nMSE = 2.82 %
Test data
Dimension 1: nMSE = 5.93 %
Dimension 2: nMSE = 4.88 %
Dimension 3: nMSE = 4.20 %
Dimension 4: nMSE = 2.22 %
Dimension 5: nMSE = 6.26 %
Dimension 6: nMSE = 8.02 %
Dimension 7: nMSE = 2.95 %
```