Neuronale Netze - Übung 7

Tobias Hahn
3073375

December 4, 2016

1 Neuronales Netz

1.1 Implementierung

Ich habe das Netz als untereinander verknotete Sammlung von Schichten implementiert. Der Vorwärts-, Rückwärts und Gewichtsanpassungsschritt werden von den Schichten ganz von selbst erledigt, man muss ihnen am Anfang nur die Daten und die Labels bereitstellen. Daneben gibt es eine Predict Methode, mit der man den Forwärtsschritt ohne Rückwärtsschritt und Gewichtsanpassung machen kann, wobei man entweder die vorhergesagten Ergebnisse oder die Fehler berechnen lassen kann - je nachdem ob man Labels mitgibt oder nicht.

1.2 Code

Der Code des Netzes ist hier:

../code/NeuralNet.py

```
import numpy as np

def sigmoid(X, Y=None, deriv=False):
    if not deriv:
        return 1 / (1 + np.exp(-X))
    else:
        return sigmoid(X)*(1 - sigmoid(X))

def quadratic_error(X, Y):
    return 0.5 * (X - Y)**2

def learn_rate(last_grad, this_grad, last_l, up=1.2, down=0.5, l_min=1E-06, l_max=50.0):
    if (last_grad * this_grad > 0):
        return min(last_l*up, l_max)
    elif (last_grad * this_grad < 0):
        return max(last_l*down, l_min)
    return last_l

def learn_rate_line(last_grad, this_grad, last_l):
    return map(learn_rate, last_grad, this_grad, last_l)

class InputLayer:
    def __init__(self, size):
        #this is needed for initialization of later layers
        self.nodeNumber = size[0]
        #weight matrix
        self.W = np.random.normal(size=[size[0]+1, size[1]], scale=1E-4)
        #matrix of learn rates
        self.L = np.zeros((size[0]+1, size[1]))
        self.L.fill(0.1)
        #matrix for last gradient, important for calculating next learn rate
        self.lastGradient = np.ones((size[0]+1, size[1]))

    def forwardPropagate(self, X, Y):
        #this is important if only one example vector is supplied
        X = np.atleast_2d(X)
        #bias unit
        self.X = np.append(X, np.ones((X.shape[0], 1)), axis=1)
        #sets the true labels for error calculation afterwards
        self.errorLayer.setTrueLabels(np.atleast_2d(Y))
        #forward propagate activation values to next layer
        return self.nextLayer.forwardPropagate(self.X.dot(self.W))

    def backwardPropagate(self, D):
        #calculates gradient and updates according to last learnrate
        gradient = (D.dot(self.X)).T
        self.W -= self.L * np.sign(gradient)

        #update learnrate
        self.L = np.array(map(learn_rate_line, self.lastGradient, gradient, self.L))
```

```
        self.lastGradient = gradient

        #update weights in next layer
        self.nextLayer.updateWeights()

    def predict(self, X, Y=None):
        #this is important if only one example vector is supplied
        X = np.atleast_2d(X)
        #bias unit
        self.X = np.append(X, np.ones((X.shape[0], 1)), axis=1)

        #set true labels in error layer only if supplied, for error calculation
        if (Y != None):
            self.errorLayer.setTrueLabels(np.atleast_2d(Y))

        #propagate activation and information about error calculation to next layer
        return self.nextLayer.predict(self.X.dot(self.W), Y!=None)

    def setErrorLayer(self, errorLayer):
        #saves a reference to the error layer in order to set true labels
        self.errorLayer = errorLayer

class HiddenLayer:
    def __init__(self, size, activation=sigmoid):
        #this is needed for initialization of later layers
        self.nodeNumber = size[0]
        #weight matrix
        self.W = np.random.normal(size=[size[0]+1, size[1]], scale=1E-4)
        #matrix of learn rates
        self.L = np.zeros((size[0]+1, size[1]))
        self.L.fill(0.1)
        #matrix for last gradient, important for calculating next learn rate
        self.lastGradient = np.ones((size[0]+1, size[1]))

    def forwardPropagate(self, S):
        #calculate activation and derivatives
        self.Z = self.activation(S)
        self.Fp = self.activation(S, deriv=True).T
        #bias unit
        self.Z = np.append(self.Z, np.ones((self.Z.shape[0], 1)), axis=1)
        #propagate activation to next layer
        return self.nextLayer.forwardPropagate(self.Z.dot(self.W))

    def backwardPropagate(self, D):
        #caculate deltas for this layer according to dalte from last layer (not for bias!)
        self.D = self.W[0:-1, :].dot(D) * self.Fp
        #backpropagate D to last layer
        self.lastLayer.backwardPropagate(self.D)

    def updateWeights(self):
        #calculates gradient and updates weight according to last learnrates
        gradient = (self.nextLayer.D.dot(self.Z)).T
        self.W -= self.L * np.sign(gradient)
        #updates learnrates
        self.L = np.array(map(learn_rate_line, self.lastGradient, gradient, self.L))
        #saves gradient for next step
        self.lastGradient = gradient
        #weight update for next layer
        self.nextLayer.updateWeights()

    def predict(self, X, calcError=False):
        #calculates activation and adds bias unit only, no derivation
        self.Z = self.activation(X)
        self.Z = np.append(self.Z, np.ones((self.Z.shape[0], 1)), axis=1)
        #propagates activation and information about error calculation
        return self.nextLayer.predict(self.Z.dot(self.W), calcError)

class OutputLayer:
    def __init__(self, activation=sigmoid):
        self.activation = activation

```

```
    def forwardPropagate(self, S):
        #this layer has no weights, only propagate activation
        return self.nextLayer.forwardPropagate(self.activation(S))

    def backwardPropagate(self, D):
        #need to transpose deltas coming from error layer
        self.D = D.T
        self.lastLayer.backwardPropagate(self.D)

    def updateWeights(self):
        #the last hidden layer doesn't know it is the last, so it tries to call updateWeights on the output layer, therefore this stub is needed.
        return

    def predict(self, X, calcError=False):
        #calculates our prediction of the label or propagates it in order to get error sum.
        if not calcError:
            return self.activation(X)
        else:
            return self.nextLayer.forwardPropagate(self.activation(X), calcError)

    def setLastLayer(self, lastLayer):
        #method to set last layer for backward propagation
        self.lastLayer = lastLayer

class ErrorLayer:
    def __init__(self, error=quadratic_error):
        self.error = error

    def forwardPropagate(self, S, justError=False):
        #starts backward propagation by setting first D if justError is False, in both cases propagates error to next layer
        if not justError:
            self.lastLayer.backwardPropagate(S - self.Y)
        return self.nextLayer.forwardPropagate(self.error(S, self.Y))

    def setNextLayer(self, nextLayer):
        #sets the next layer for forward propagation
        self.nextLayer = nextLayer

    def setTrueLabels(self, Y):
        #sets true labels for error calculation. called by input layer
        self.Y = Y

class ErrorSum:
    def forwardPropagate(self, S):
        #returns the sum of squared errors, the end of forward propagation
        return np.sum(S)

def connectLayers(first, second):
    #connects two layers so they can forward and backward propagate each other
    first.nextLayer = second
    second.lastLayer = first

class NeuralNet:
    def __init__(self, inputSize, outputSize, hiddenLayerConfig):
        #adds the input layer
        self.inputLayer = InputLayer([inputSize, hiddenLayerConfig[0]])
        self.layers = [self.inputLayer]
        self.numberHidden = sum(hiddenLayerConfig)

        #adds the hidden layers up to the last
        for i,nodeNumber in enumerate(hiddenLayerConfig[:-2]):
            self.layers.append(HiddenLayer([nodeNumber, hiddenLayerConfig[i+1]]))
            connectLayers(self.layers[-2], self.layers[-1])

        #adds the last hidden layer
        self.layers.append(HiddenLayer([hiddenLayerConfig[-1], outputSize]))
        connectLayers(self.layers[-2], self.layers[-1])

        #adds the output layer
```

```
        self.layers.append(OutputLayer())
        connectLayers(self.layers[-2], self.layers[-1])

        #adds the error layer, connects the input layer with it, and adds the error sum layer to it
        self.layers.append(ErrorLayer())
        connectLayers(self.layers[-2], self.layers[-1])
        self.inputLayer.setErrorLayer(self.layers[-1])
        self.layers[-1].setNextLayer(ErrorSum())

    def train(self, train_data, train_labels, test_data, test_labels, difference=0.001):
        #trains our network until convergence (which is determined by small difference of error before and after training)
        last_error = float("Inf")
        error = 0

        while True:
            error = self.inputLayer.forwardPropagate(train_data, train_labels)
            if (error < last_error and last_error - error < difference):
                break
            last_error = error

        #predicts error for test set on trained network and gives us some benchmarks
        print("# hidden nodes: {0} //Training error: {1} // Test error: {2}".format(self.numberHidden, last_error, self.inputLayer.predict(test_data, test_labels)))
```

../code/train.py

```
import numpy as np
from NeuralNet import NeuralNet

def extractDigits(filename, expected_num):
    data_count = 0
    digit_count = 0
    data_points_per_digit = 192
    data_points_per_line = 12

    digits = np.zeros(expected_num, dtype=[('data', 'f', data_points_per_digit), ('value', 'f', 10)])

    with open(filename) as f:
        lines = f.readlines()

    for i,line in enumerate(lines):
        digits_line = line.split()
        if (len(digits_line) == data_points_per_line):
            for num in digits_line:
                digits['data'][digit_count][data_count] = float(num)
                data_count += 1
        elif (len(digits_line) == 10):
            digits['value'][digit_count] = np.zeros(10)
            for i,num in enumerate(digits_line):
                if (num == "1.0"):
                    digits['value'][digit_count][i] = float(num)
                    break
        else:
            if (data_count == data_points_per_digit and digit_count < expected_num):
                digit_count += 1
                data_count = 0
            else:
                print("Exited because of wrong data")
                raise SystemExit

    if (digit_count == expected_num):
        return digits
    else:
        print("Exited because of few digits")
        raise SystemExit

if __name__ == "__main__":
    train_name = "../data/digits.trn"
```

```
    train_number = 1000
    train_digits = extractDigits(train_name, train_number)

    test_name = "../data/digits.tst"
    test_number = 200
    test_digits = extractDigits(test_name, test_number)

    myNet = NeuralNet(192,10,[20,20,20])
    myNet.train(train_digits['data'], train_digits['value'], test_digits['data'], test_digits
        ['value'])
```

1.3 Ergebnisse

```
# hidden nodes: 10 // Training error: 1.5951366864 // Test error: 25.2602785299
# hidden nodes: 20 // Training error: 0.00345100731381 // Test error: 20.2978443278
# hidden nodes: 30 // Training error: 0.0188063340479 // Test error: 23.1903266333
# hidden nodes: 40 // Training error: 0.047092620751 // Test error: 19.6124136129
# hidden nodes: 50 // Training error: 0.0295850736737 // Test error: 15.2657572425
# hidden nodes: 60 // Training error: 0.00615232677917 // Test error: 20.9680180911
# hidden nodes: 70 // Training error: 0.00389029399399 // Test error: 18.184484134
# hidden nodes: 80 // Training error: 0.00449535821462 // Test error: 16.6341676022
# hidden nodes: 90 // Training error: 0.0834084331233 // Test error: 19.677263485
# hidden nodes: 100 // Training error: 0.00195384190943 // Test error: 19.9208690295
# hidden nodes: 110 // Training error: 0.00440194332911 // Test error: 19.9072392755
# hidden nodes: 120 // Training error: 0.016182981848 // Test error: 16.8904571922
# hidden nodes: 130 // Training error: 0.00728166997087 // Test error: 17.6028740996
# hidden nodes: 140 // Training error: 0.00204702287436 // Test error: 14.677020486
# hidden nodes: 150 // Training error: 0.00935833008499 // Test error: 19.3108718565
# hidden nodes: 160 // Training error: 0.00299756302837 // Test error: 19.9936050608
# hidden nodes: 170 // Training error: 0.971229311966 // Test error: 16.5626160393
# hidden nodes: 180 // Training error: 0.00279814105182 // Test error: 13.9867308023
# hidden nodes: 190 // Training error: 0.00342643109315 // Test error: 16.936395778
```

1.4 Interpretation

Wie zu sehen ist änder sich der Error für eine ansteigende Anzahl an Knoten kaum. Auffallend ist, dass 10 Knoten wohl etwas zu wenig sind - hier ist der Fehler am Trainingset noch über 1. Anschließend aber schwankt der Fehler am Trainingset zwischen 1 und 0.002 hin und her, ohne irgendeine Tendenz, während der Fehler am Testset zwischen 23-15 hin und herschwankt, auch recht unbeeindruckt von der Anzahl der verdeckten Knoten. Dies deutet darauf hin dass zwischen 10 und 20 verdeckten Knoten genügen, um alle interessanten Features die man mit einer verdeckten Schicht entdecken kann abzudecken. Versuche mit mehreren verdeckten Schichten ergaben dass die das Ergebnis auch nicht merkbar verbesserten.