Neuronale Netze - Übung 8

Tobias Hahn
3073375

January 9, 2017

1 Neuronales Netz

1.1 Implementierung

Meine Implementierung hat wohl einen Fehler, ich bin aber leider nicht dahintergekommen wo. Die Deltas werden wie gewohnt berechnet, nur werden, wie gefordert, am Ende für das Update nicht die Inputs als fix betrachtet sondern die Gewichte, um so den Gradienten für die Inputs zu bekommen. Das Bild ändert sich jedoch nicht viel, und vor allem nicht zwischen den Durchgängen. Woran das liegt kann ich nicht sagen. Ich habe versucht für die Berechnung der Delta auch die Signale statt der Gewichte zu benutzen, dies führte jedoch zum selben Ergebnis.

1.2 Code

Der Code des Netzes ist hier:

../code/NeuralNet.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def sigmoid(X, Y=None, deriv=False):
5          if not deriv:
6                  return 1 / (1 + np.exp(-X))
7          else:
8                  return sigmoid(X)*(1 - sigmoid(X))
9
10 def quadratic_error(X, Y):
11         return 0.5 * (X - Y)**2
12
13 def learn_rate(last_grad, this_grad, last_l, up=1.2, down=0.5, l_min=1E-06, l_max=50.0):
14         if (last_grad * this_grad > 0):
15                 return min(last_l*up, l_max)
16         elif (last_grad * this_grad < 0):
17                 return max(last_l*down, l_min)
18         return last_l
19
20 def learn_rate_line(last_grad, this_grad, last_l):
21         return map(learn_rate, last_grad, this_grad, last_l)
22
23 class InputLayer:
24         def __init__(self, size):
25                 #this is needed for initialization of later layers
26                 self.nodeNumber = size[0]
27                 #weight matrix
28                 self.W = np.random.normal(size=[size[0]+1, size[1]], scale=1E-4)
29                 #matrix of learn rates
30                 self.L = np.zeros((size[0]+1, size[1]))
31                 self.L.fill(0.1)
32                 #matrix for last gradient, important for calculating next learn rate
33                 self.lastGradient = np.ones((size[0]+1, size[1]))
34
35         def forwardPropagate(self, X, Y):
36                 #this is important if only one example vector is supplied
37                 X = np.atleast_2d(X)
38                 #bias unit
39                 self.X = np.append(X, np.ones((X.shape[0], 1)), axis=1)
40                 #sets the true labels for error calculation afterwards
41                 self.errorLayer.setTrueLabels(np.atleast_2d(Y))
42                 #forward propagate activation values to next layer
43                 return self.nextLayer.forwardPropagate(self.X.dot(self.W))
44
45         def forwardDream(self):
46                 self.errorLayer.setDream()
47                 return self.nextLayer.forwardPropagate(self.X.dot(self.W))
48
49         def backwardPropagate(self, D):
50                 #calculates gradient and updates according to last learnrate
```

```
51                  gradient = (D.dot(self.X)).T
52                  self.W -= self.L * np.sign(gradient)
53
54                  #update learnrate
55                  self.L = np.array(map(learn_rate_line, self.lastGradient, gradient, self.L))
56                  self.lastGradient = gradient
57
58                  #update weights in next layer
59                  self.nextLayer.updateWeights()
60
61          def dream(self, D):
62                  gradient = self.W.dot(D).T
63                  self.X -= 0.01 * gradient
64
65          def predict(self, X, Y=None):
66                  #this is important if only one example vector is supplied
67                  X = np.atleast_2d(X)
68                  #bias unit
69                  self.X = np.append(X, np.ones((X.shape[0], 1)), axis=1)
70
71                  #set true labels in error layer only if supplied, for error calculation
72                  if (Y != None):
73                          self.errorLayer.setTrueLabels(np.atleast_2d(Y))
74
75                  #propagate activation and information about error calculation to next layer
76                  return self.nextLayer.predict(self.X.dot(self.W), Y!=None)
77
78          def setErrorLayer(self, errorLayer):
79                  #saves a reference to the error layer in order to set true labels
80                  self.errorLayer = errorLayer
81
82  class HiddenLayer:
83          def __init__(self, size, activation=sigmoid):
84                  #this is needed for initialization of later layers
85                  self.nodeNumber = size[0]
86                  #weight matrix
87                  self.W = np.random.normal(size=[size[0]+1, size[1]], scale=1E-4)
88                  #matrix of learn rates
89                  self.L = np.zeros((size[0]+1, size[1]))
90                  self.L.fill(0.1)
91                  #matrix for last gradient, important for calculating next learn rate
92                  self.lastGradient = np.ones((size[0]+1, size[1]))
93                  self.activation = activation
94
95          def forwardPropagate(self, S):
96                  #calculate activation and derivatives
97                  self.Z = self.activation(S)
98                  self.Fp = self.activation(S, deriv=True).T
99                  #bias unit
100                 self.Z = np.append(self.Z, np.ones((self.Z.shape[0], 1)), axis=1)
101                 #propagate activation to next layer
102                 return self.nextLayer.forwardPropagate(self.Z.dot(self.W))
103
104         def backwardPropagate(self, D):
105                 #caculate deltas for this layer according to dalte from last layer (not for
                        bias!)
106                 self.D = self.W[0:-1, :].dot(D) * self.Fp
107                 #backpropagate D to last layer
108                 self.lastLayer.backwardPropagate(self.D)
109
110         def dream(self, D):
111                 #weight_z = self.Z[:, 0:-1].T
112                 #weight_z = np.repeat(weight_z, self.W.shape[1], axis=1)
113                 #self.D = weight_z.dot(D) * self.Fp
114                 self.D = self.W[0:-1, :].dot(D) * self.Fp
115                 self.lastLayer.dream(self.D)
116
117         def updateWeights(self):
118                 #calculates gradient and updates weight according to last learnrates
119                 gradient = (self.nextLayer.D.dot(self.Z)).T
120                 self.W -= self.L * np.sign(gradient)
```

```
121                 #updates learnrates
122                 self.L = np.array(map(learn_rate_line, self.lastGradient, gradient, self.L))
123                 #saves gradient for next step
124                 self.lastGradient = gradient
125                 #weight update for next layer
126                 self.nextLayer.updateWeights()
127 
128         def predict(self, X, calcError=False):
129                 #calculates activation and adds bias unit only, no derivation
130                 self.Z = self.activation(X)
131                 self.Z = np.append(self.Z, np.ones((self.Z.shape[0], 1)), axis=1)
132                 #propagates activation and information about error calculation
133                 return self.nextLayer.predict(self.Z.dot(self.W), calcError)
134 
135 class OutputLayer:
136         def __init__(self, activation=sigmoid):
137                 self.activation = activation
138 
139         def forwardPropagate(self, S):
140                 #this layer has no weights, only propagate activation
141                 return self.nextLayer.forwardPropagate(self.activation(S))
142 
143         def backwardPropagate(self, D):
144                 #need to transpose deltas coming from error layer
145                 self.D = D.T
146                 self.lastLayer.backwardPropagate(self.D)
147 
148         def dream(self, D):
149                 self.D = D.T
150                 self.lastLayer.dream(self.D)
151 
152         def updateWeights(self):
153                 #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.
154                 return
155 
156         def predict(self, X, calcError=False):
157                 #calculates our prediction of the label or propagates it in order to get
                        error sum.
158                 if not calcError:
159                         return self.activation(X)
160                 else:
161                         return self.nextLayer.forwardPropagate(self.activation(X), calcError
                                )
162 
163         def setLastLayer(self, lastLayer):
164                 #method to set last layer for backward propagation
165                 self.lastLayer = lastLayer
166 
167 class ErrorLayer:
168         def __init__(self, error=quadratic_error):
169                 self.error = error
170                 self.dream = False
171 
172         def forwardPropagate(self, S, justError=False):
173                 #starts backward propagation by setting first D if justError is False, in
                        both cases propagates error to next layer
174                 if self.dream:
175                         newS = [0] * S.shape[1]
176                         newS[np.argmax(S)] = 1
177                         self.lastLayer.dream(S - newS)
178                         return self.nextLayer.forwardPropagate(self.error(S, newS))
179                 elif not justError:
180                         self.lastLayer.backwardPropagate(S - self.Y)
181                 return self.nextLayer.forwardPropagate(self.error(S, self.Y))
182 
183         def setNextLayer(self, nextLayer):
184                 #sets the next layer for forward propagation
185                 self.nextLayer = nextLayer
186 
187         def setTrueLabels(self, Y):
```

```
188                 #sets true labels for error calculation. called by input layer
189                 self.Y = Y
190 
191         def setDream(self):
192                 self.dream = True
193 
194 class ErrorSum:
195         def forwardPropagate(self, S):
196                 #returns the sum of squared errors, the end of forward propagation
197                 return np.sum(S)
198 
199 def connectLayers(first, second):
200         #connects two layers so they can forward and backward propagate each other
201         first.nextLayer = second
202         second.lastLayer = first
203 
204 class NeuralNet:
205         def __init__(self, inputSize, outputSize, hiddenLayerConfig):
206                 #adds the input layer
207                 self.inputLayer = InputLayer([inputSize, hiddenLayerConfig[0]])
208                 self.layers = [self.inputLayer]
209                 self.numberHidden = sum(hiddenLayerConfig)
210 
211                 #adds the hidden layers up to the last
212                 for i,nodeNumber in enumerate(hiddenLayerConfig[:-2]):
213                         self.layers.append(HiddenLayer([nodeNumber, hiddenLayerConfig[i+1]])
                                )
214                         connectLayers(self.layers[-2], self.layers[-1])
215 
216                 #adds the last hidden layer
217                 self.layers.append(HiddenLayer([hiddenLayerConfig[-1], outputSize]))
218                 connectLayers(self.layers[-2], self.layers[-1])
219 
220                 #adds the output layer
221                 self.layers.append(OutputLayer())
222                 connectLayers(self.layers[-2], self.layers[-1])
223 
224                 #adds the error layer, connects the input layer with it, and adds the error
                        sum layer to it
225                 self.layers.append(ErrorLayer())
226                 connectLayers(self.layers[-2], self.layers[-1])
227                 self.inputLayer.setErrorLayer(self.layers[-1])
228                 self.layers[-1].setNextLayer(ErrorSum())
229 
230         def train(self, train_data, train_labels, test_data, test_labels, difference=0.001):
231                 #trains our network until convergence (which is determined by small
                        difference of error before and after training)
232                 last_error = float("Inf")
233                 error = 0
234 
235                 while True:
236                         error = self.inputLayer.forwardPropagate(train_data, train_labels)
237                         if (error < last_error and last_error - error < difference):
238                                 break
239                         last_error = error
240 
241                 #predicts error for test set on trained network and gives us some benchmarks
242                 print("#␣hidden␣nodes:␣{0}␣//␣Training␣error:␣{1}␣//␣Test␣error:␣{2}".format
                        (self.numberHidden, last_error, self.inputLayer.predict(test_data,
                        test_labels)))
243 
244         def dream(self, dream_digit, difference=0.001):
245                 #dreams a digit until convergence (saving picture at intervals)
246                 last_error = float("Inf")
247                 error = 0
248                 counter = 0
249 
250                 dream_digit = np.atleast_2d(dream_digit)
251                 self.inputLayer.X = np.append(dream_digit, np.ones((dream_digit.shape[0], 1)
                        ), axis=1)
252 
```

```
253                 for i in range(1,100):
254                         last_error = self.inputLayer.forwardDream()
255                         picture = self.inputLayer.X
256 
257 
258                 print("Dreaming error: {0}\nDream: {1}".format(last_error, picture))
259                 plt.gray()
260                 plt.imshow(np.reshape(picture[:, 1:], (16,12)))
261                 plt.show()
```

../code/train.py

```
1  import numpy as np
2  from NeuralNet import NeuralNet
3 
4  def extractDigits(filename, expected_num):
5          data_count = 0
6          digit_count = 0
7          data_points_per_digit = 192
8          data_points_per_line = 12
9 
10         digits = np.zeros(expected_num, dtype=[('data', 'f', data_points_per_digit), ('value
               ', 'f', 10)])
11 
12         with open(filename) as f:
13                 lines = f.readlines()
14 
15         for i,line in enumerate(lines):
16                 digits_line = line.split()
17                 if (len(digits_line) == data_points_per_line):
18                         for num in digits_line:
19                                 digits['data'][digit_count][data_count] = float(num)
20                                 data_count += 1
21                 elif (len(digits_line) == 10):
22                         digits['value'][digit_count] = np.zeros(10)
23                         for i,num in enumerate(digits_line):
24                                 if (num == "1.0"):
25                                         digits['value'][digit_count][i] = float(num)
26                                         break
27                 else:
28                         if (data_count == data_points_per_digit and digit_count <
                               expected_num):
29                                 digit_count += 1
30                                 data_count = 0
31                         else:
32                                 print("Exited because of wrong data")
33                                 raise SystemExit
34 
35         if (digit_count == expected_num):
36                 return digits
37         else:
38                 print("Exited because of few digits")
39                 raise SystemExit
40 
41 if __name__ == "__main__":
42         dream_digit = [0.0, 0.5, 0.2, 0.8, 0.2, 1.0, 0.3, 0.7, 0.2, 0.8, 0.6, 1.0, 0.7, 1.0,
                   0.4, 0.3, 0.2, 0.3, 0.2, 0.8, 0.9, 0.8, 0.6, 0.4, 0.2, 0.7, 0.2, 0.1, 0.8, 0.4,
                   0.1, 0.7, 0.6, 0.0, 0.9, 1.0, 0.9, 0.2, 0.8, 0.0, 0.2, 0.3, 0.4, 0.3, 0.3, 0.9,
                   0.5, 0.5, 0.9, 0.0, 0.2, 0.2, 0.8, 0.2, 0.8, 0.9, 0.4, 0.0, 0.9, 0.3, 0.0, 0.4,
                   0.8, 0.2, 0.9, 0.4, 0.4, 1.0, 0.9, 0.7, 0.8, 0.6, 0.7, 0.9, 0.3, 0.6, 0.5, 0.4,
                   0.0, 0.8, 0.1, 0.1, 0.3, 0.1, 0.7, 1.0, 0.7, 0.8, 0.5, 0.4, 0.6, 0.8, 0.4, 0.2,
                   0.8, 0.2, 0.2, 0.4, 0.6, 0.8, 0.1, 0.7, 0.9, 0.7, 0.8, 0.1, 0.1, 0.7, 0.9, 0.5,
                   0.2, 0.0, 0.6, 0.9, 0.9, 1.0, 0.3, 0.5, 0.9, 0.4, 0.4, 0.7, 0.0, 0.7, 0.7, 0.0,
                   0.6, 0.1, 0.7, 0.4, 0.2, 0.5, 0.8, 0.9, 0.6, 0.4, 0.0, 0.2, 0.6, 0.0, 0.9, 0.9,
                   0.5, 0.0, 1.0, 0.4, 0.6, 0.1, 0.5, 1.0, 0.2, 0.1, 0.6, 0.6, 0.6, 0.4, 0.1, 0.3,
                   1.0, 0.2, 0.1, 0.2, 0.3, 0.5, 0.4, 0.6, 0.1, 0.7, 0.2, 0.3, 0.3, 0.1, 0.1, 0.2,
                   0.3, 0.4, 0.2, 1.0, 0.7, 0.7, 0.4, 1.0, 0.2, 1.0, 0.8, 0.2, 1.0, 0.7, 0.3, 0.2,
                   0.0, 0.9]
43 
44         train_name = "../data/digits.trn"
45         train_number = 1000
```

```
46          train_digits = extractDigits(train_name, train_number)
47
48          test_name = "../data/digits.tst"
49          test_number = 200
50          test_digits = extractDigits(test_name, test_number)
51
52          myNet = NeuralNet(192,10,[20,20,20])
53          myNet.train(train_digits['data'], train_digits['value'], test_digits['data'],
                test_digits['value'])
54
55          myNet.dream(dream_digit);
```

1.3 Ergebnisse