Lernverfahren autonomer Roboter - Übung 10

G10
Andre Osse
Waldemar Stockmann
Markus Strehling
Tobias Hahn

January 25, 2017

1 Multilayer Neural Network and Backpropagation

1.1 Implementation

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

../code/multi.py

```python
1   """Multilayer Neural Network."""
2   import numpy as np
3   from tools import linear, linear_derivative, relu, relu_derivative
4
5
6   class FullyConnectedLayer(object):
7       """Represents a trainable fully connected layer.
8
9       Parameters
10      ----------
11      I : int
12          inputs (without bias)
13
14      J : int; outputs
15
16      g : function: array-like -> array-like
17          activation function y = g(a)
18
19      gd : function: array-like -> array-like
20          derivative g'(a) = gd(y)
21
22      std_dev : float
23          standard deviation of the normal distribution that we use to draw
24          the initial weights
25
26      verbose : int, optional
27          verbosity level
28      """
29      def __init__(self, I, J, g, gd, std_dev, verbose=0):
30          self.I = np.prod(I) + 1   # Add bias component
31          self.J = J
32          self.g = g
33          self.gd = gd
34          self.std_dev = std_dev
35
36          self.W = np.empty((self.J, self.I))
37
38          if verbose:
39              print("Fully connected layer (%d nodes, %d x %d weights)"
40                    % (self.J, self.J, self.I))
41
42      def initialize_weights(self, random_state):
43          """Initialize weights randomly.
44
45          Parameters
46          ----------
47          random_state : RandomState or int
48              random number generator or seed
49          """
50          #if type is int it is a seed, else we can use it as a RandomState
51          if (type(random_state) is int):
52              random_state = np.random.RandomState(random_state)
53          self.W = random_state.normal(0, self.std_dev, self.W.shape)
54
55      def get_output_shape(self):
56          """Get shape of the output.
57
58          Returns
59          -------
60          shape : tuple
61              shape of the output
62          """
63          return (self.J,)
```

```python
64
65        def forward(self, X):
66            """Forward propagate the output of the previous layer.
67
68            Parameters
69            ----------
70            X : array-like, shape = [N, I or self.I-1]
71                input
72
73            Returns
74            -------
75            Y : array-like, shape = [N, J]
76                output
77            """
78            self.X = X
79            N = X.shape[0]
80            D = np.prod(X.shape[1:])
81            if D != self.I - 1:
82                raise ValueError("shape = " + str(X.shape))
83            self.Z = np.ones((X.shape[0], X.shape[1]+1))
84            self.Z[:,1:] = X
85            self.Y = self.g(self.Z.dot(self.W.T))
86
87            return self.Y
88
89        def backpropagation(self, dEdY):
90            """Backpropagate errors of the next layer.
91
92            Parameters
93            ----------
94            dEdY : array-like, shape = [N, J]
95                errors from the next layer
96
97            Returns
98            -------
99            dEdX : array-like, shape = [N, I+1 or self.I]
100                errors from this layer
101
102            Wd : array-like, shape = [J, self.I]
103                derivatives of the weights
104            """
105            if dEdY.shape[1] != self.J:
106                raise ValueError("%r != %r" % (len(dEdY), self.J))
107
108            dEdY = dEdY * self.gd(self.Y)
109            dEdX = dEdY.dot(self.W[:,1:])
110            Wd = dEdY.T.dot(self.Z)
111
112            return dEdX, Wd
113
114        def get_weights(self):
115            """Get current weights.
116
117            Returns
118            -------
119            W : array-like, shape = [J * I + 1 or self.I]
120                weight matrix
121            """
122            return self.W.flat
123
124        def set_weights(self, W):
125            """Set new weights.
126
127            Parameters
128            ----------
129            W : array-like, shape = [J * I + 1 or self.I]
130                weight matrix
131            """
132            self.W = W.reshape((self.J, self.I))
133
134        def num_weights(self):
```

```python
135            """Get number of weights.
136
137            Returns
138            -------
139            K : int
140                number of weights
141            """
142            return self.W.size
143
144        def __getstate__(self):
145            # This will be called by pickle.dump, so we remove everything that
146            # requires too much memory
147            d = dict(self.__dict__)
148            if "X" in d:
149                del d["X"]
150            if "Y" in d:
151                del d["Y"]
152            return d
153
154
155    class MultilayerNeuralNetwork(object):
156        """Multilayer Neural Network (MLNN).
157
158        Parameters
159        ----------
160        D : int
161            number of inputs
162
163        F : int
164            number of outputs
165
166        layers : list of dicts
167            layer definitions
168
169        training : string
170            must be either classification or regression and defines the
171            activation function of the last layer as well as the error function
172
173        std_dev : float
174            standard deviation of the normal distribution that we use to draw
175            the initial weights
176
177        verbose : int, optional
178            verbosity level
179        """
180
181        def __init__(self, D, F, layers, training="classification", std_dev=0.05,
182                     verbose=0):
183            self.D = D
184            self.F = F
185
186            # Initialize layers
187            self.layers = []
188            I = self.D  # Dimensions of the input layer
189            for layer in layers:
190                l = None
191                if layer["type"] == "fully_connected":
192                    l = FullyConnectedLayer(
193                        I, layer["num_nodes"], relu, relu_derivative, std_dev,
194                        verbose)
195                    I = l.get_output_shape()
196                else:
197                    raise NotImplementedException("Layer type '%s' is not "
198                                                  "implemented." % layer["type"])
199                self.layers.append(l)
200            if training == "regression":
201                self.layers.append(FullyConnectedLayer(
202                    I, self.F, linear, linear_derivative, std_dev, verbose))
203                self.error_function = "sse"
204            else:
205                raise ValueError("Unknown 'training': %s" % training)
```

```
206
207        def initialize_weights(self, random_state):
208            """Initialize weights randomly.
209
210            Parameters
211            ----------
212            random_state : RandomState or int
213                random number generator or seed
214            """
215            for layer in self.layers:
216                layer.initialize_weights(random_state)
217
218        def error(self, X, T):
219            """Calculate the Cross Entropy (CE).
220
221            .. math::
222
223                E = -\sum_n \sum_f ln(y^n_f) t^n_f,
224
225            where n is the index of the instance, f is the index of the output
226            component, y is the prediction and t is the target.
227
228            Parameters
229            ----------
230            X : array-like, shape = [N, D]
231                each row represents an instance
232
233            T : array-like, shape = [N, F]
234                each row represents a target
235
236            Returns
237            -------
238            E : float
239                error: SSE for regression, cross entropy for classification
240            """
241            if len(X) != len(T):
242                raise ValueError("Number of samples and targets must match")
243
244            # Compute error of the dataset
245
246            if (self.error_function == "sse"):
247                return 1/2 * np.sum(np.power(self.predict(X) - T, 2))
248            else:
249                return np.sum(np.log(self.predict(X)) * T)
250
251        def numerical_gradient(self, X, T, eps=1e-5):
252            """Compute the derivatives of the weights with finite differences.
253
254            This function can be used to check the analytical gradient
255            numerically. The partial derivative of E with respect to w is
256            approximated through
257
258            .. math::
259
260                \partial E / \partial w = (E(w+\epsilon) - E(w-\epsilon)) /
261                                            (2 \epsilon) + O(\epsilon^2),
262
263            where :math:`\epsilon` is a small number.
264
265            Parameters
266            ----------
267            X : array-like, shape = [N, D]
268                input
269
270            T : array-like, shape = [N, F]
271                desired output (target)
272
273            eps : float, optional
274                small number, you can make eps smaller to increase the accuracy
275                of the differentiation until roundoff errors occur
276
```

```
277            Returns
278            -------
279            wd : array-like, shape = [K,]
280                weight vector derivative
281            """
282            w = self.get_weights()
283            w_original = w.copy()
284            wd = np.empty_like(w)
285            for k in range(len(w)):
286                w[k] = w_original[k] + eps
287                self.set_weights(w)
288                Ep = self.error(X, T)
289                w[k] = w_original[k] - eps
290                self.set_weights(w)
291                Em = self.error(X, T)
292                w[k] = w_original[k]
293                wd[k] = (Ep - Em) / (2.0 * eps)
294            self.set_weights(w_original)
295            return wd
296
297        def gradient(self, X, T, get_error=False):
298            """Calculate the derivatives of the weights.
299
300            Parameters
301            ----------
302            X : array-like, shape = [N, D]
303                input
304
305            T : array-like, shape = [N, F]
306                desired output (target)
307
308            get_error : bool, optional (default: False)
309                Return a tuple (g, e), otherwise only g will be returned
310
311            Returns
312            -------
313            g : array-like, shape = [K,]
314                gradient of weight vector
315
316            e : float, optional
317                error
318            """
319            e = self.error(X, T)
320            Y = self.predict(X)
321            D = Y - T
322
323            part_grad = []
324            for l in reversed(self.layers):
325                D, g = l.backpropagation(D)
326                part_grad.insert(0, g.flat)
327            if not get_error:
328                return np.concatenate(part_grad)
329            else:
330                return (np.concatenate(part_grad), e)
331
332        def get_weights(self):
333            """Get current weight vector.
334
335            Returns
336            -------
337            w : array-like, shape (K,)
338                weight vector
339            """
340            return np.concatenate([self.layers[l].get_weights()
341                                   for l in range(len(self.layers))])
342
343        def set_weights(self, w):
344            """Set new weight vector.
345
346            Parameters
347            ----------
```

```
348         w :  a r r a y −l i k e ,   shape=[K, ]
349             weight  v e c t o r
350         """
351         i  = 0
352         for  l  in  range ( l e n ( s e l f . l a y e r s ) ) :
353             k =  s e l f . l a y e r s [ l ] . num_weights ( )
354             s e l f . l a y e r s [ l ] . s e t _ w e i g h t s ( w [ i : i  + k ] )
355             i  += k
356
357     def  p r e d i c t ( s e l f ,  X ) :
358         """ P r e d i c t  v a l u e s .
359
360         Parameters
361         −−−−−−−−−−
362         X :  a r r a y −l i k e ,  shape =  [N, D]
363             each  row  r e p r e s e n t s  an  i n s t a n c e
364
365         Returns
366         −−−−−−−
367         Y:  a r r a y −l i k e ,  shape =  [N, F]
368             each  row  r e p r e s e n t s  a  p r e d i c t i o n
369         """
370         # Forward  propagation
371         for  l  in  s e l f . l a y e r s :
372             l . X = X
373             X = l . forward (X)
374         return  X
```

Außerdem wurde der Test ausgeführt, mit folgendem Ergebnis:

```
1  F u l l y  connected  l a y e r  (20  nodes ,  20 x 11  weights )
2  F u l l y  connected  l a y e r  (10  nodes ,  10 x 21  weights )
3  F u l l y  connected  l a y e r  (3  nodes ,  3 x 11  weights )
4  Checking  g r a d i e n t s  up  to  5  p o s i t i o n s  a f t e r  decimal  point ...OK
```

1.2   Sinus lernen

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