

```

1 import numpy as np
2 from numpy.core.defchararray import add
3 from nndl.layers import *
4 import pdb
5
6 """
7 This code was originally written for CS 231n at Stanford University
8 (cs231n.stanford.edu). It has been modified in various areas for use in the
9 ECE 239AS class at UCLA. This includes the descriptions of what code to
10 implement as well as some slight potential changes in variable names to be
11 consistent with class nomenclature. We thank Justin Johnson & Serena Yeung
12 for
13 permission to use this code. To see the original version, please visit
14 cs231n.stanford.edu.
15 """
16
17 def conv_forward_naive(x, w, b, conv_param):
18     """
19     A naive implementation of the forward pass for a convolutional layer.
20
21     The input consists of N data points, each with C channels, height H and
22     width W. We convolve each input with F different filters, where each filter
23     spans
24     all C channels and has height HH and width HH.
25
26     Input:
27     - x: Input data of shape (N, C, H, W)
28     - w: Filter weights of shape (F, C, HH, WW)
29     - b: Biases, of shape (F,)
30     - conv_param: A dictionary with the following keys:
31         - 'stride': The number of pixels between adjacent receptive fields in
32           the horizontal and vertical directions.
33         - 'pad': The number of pixels that will be used to zero-pad the input.
34
35     Returns a tuple of:
36     - out: Output data, of shape (N, F, H', W') where H' and W' are given by
37        $H' = 1 + (H + 2 * pad - HH) / stride$ 
38        $W' = 1 + (W + 2 * pad - WW) / stride$ 
39     - cache: (x, w, b, conv_param)
40     """
41     out = None
42     pad = conv_param['pad']
43     stride = conv_param['stride']
44
45     # ===== #
46     # YOUR CODE HERE:
47     # Implement the forward pass of a convolutional neural network.
48     # Store the output as 'out'.
49     # Hint: to pad the array, you can use the function np.pad.
50     # ===== #
51     N, C, H, W = x.shape
52     F, C, HH, WW = w.shape
53     Hout = 1 + (H + 2 * pad - HH) // stride
54     Wout = 1 + (W + 2 * pad - WW) // stride
55     x_padded = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)))
56     out = np.zeros((N, F, Hout, Wout))

```

```

57     for n in range(N):
58         for f in range(F):
59             for h in range(Hout):
60                 for j in range(Wout):
61                     cur_x = x_padded[n, :, h * stride:h *
62                                     stride + HH, j * stride:j * stride + WW]
63                     out[n, f, h, j] = np.sum(cur_x * w[f]) + b[f]
64
65     # ===== #
66     # END YOUR CODE HERE
67     # ===== #
68
69     cache = (x, w, b, conv_param)
70     return out, cache
71
72
73 def conv_backward_naive(dout, cache):
74     """
75     A naive implementation of the backward pass for a convolutional layer.
76
77     Inputs:
78     - dout: Upstream derivatives.
79     - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
80
81     Returns a tuple of:
82     - dx: Gradient with respect to x
83     - dw: Gradient with respect to w
84     - db: Gradient with respect to b
85     """
86     dx, dw, db = None, None, None
87
88     N, F, out_height, out_width = dout.shape
89     x, w, b, conv_param = cache
90
91     stride, pad = [conv_param['stride'], conv_param['pad']]
92     xpad = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)),
mode='constant')
93     num_filts, _, f_height, f_width = w.shape
94
95     # ===== #
96     # YOUR CODE HERE:
97     # Implement the backward pass of a convolutional neural network.
98     # Calculate the gradients: dx, dw, and db.
99     # ===== #
100    N, C, H, W = x.shape
101    H_out = 1 + (H + 2 * pad - f_height) // stride
102    W_out = 1 + (W + 2 * pad - f_width) // stride
103    dxpad = np.zeros_like(xpad)
104    dx = np.zeros(x.shape)
105    dw = np.zeros(w.shape)
106    db = np.zeros(b.shape)
107
108    for n in range(N):
109        for f in range(num_filts):
110            db[f] += np.sum(dout[n, f])
111            for h in range(H_out):
112                hs = h*stride
113                for j in range(W_out):
114                    ws = j * stride
115                    dw[f] += xpad[n, :, hs:hs + f_height,

```

```

116         ws:ws + f_width] * dout[n, f, h, j]
117         dxpad[n, :, hs:hs + f_height, ws:ws +
118             f_width] += w[f] * dout[n, f, h, j]
119     dx = dxpad[:, :, pad:pad+H, pad:pad+W]
120     # ===== #
121     # END YOUR CODE HERE
122     # ===== #
123
124     return dx, dw, db
125
126
127 def max_pool_forward_naive(x, pool_param):
128     """
129     A naive implementation of the forward pass for a max pooling layer.
130
131     Inputs:
132     - x: Input data, of shape (N, C, H, W)
133     - pool_param: dictionary with the following keys:
134         - 'pool_height': The height of each pooling region
135         - 'pool_width': The width of each pooling region
136         - 'stride': The distance between adjacent pooling regions
137
138     Returns a tuple of:
139     - out: Output data
140     - cache: (x, pool_param)
141     """
142     out = None
143
144     # ===== #
145     # YOUR CODE HERE:
146     #   Implement the max pooling forward pass.
147     # ===== #
148     N, C, H, W = x.shape
149     pool_height = pool_param['pool_height']
150     pool_width = pool_param['pool_width']
151     stride = pool_param['stride']
152     Hout = 1 + (H - pool_height) // stride
153     Wout = 1 + (W - pool_width) // stride
154     out = np.zeros((N, C, Hout, Wout))
155
156     for n in range(N):
157         for c in range(C):
158             for j in range(Wout):
159                 for m in range(Hout):
160                     mstride = m * stride
161                     ws = j * stride
162                     window = x[n, c, mstride:mstride +
163                             pool_height, ws:ws+pool_width]
164                     out[n, c, m, j] = np.max(window)
165     # ===== #
166     # END YOUR CODE HERE
167     # ===== #
168     cache = (x, pool_param)
169     return out, cache
170
171
172 def max_pool_backward_naive(dout, cache):
173     """
174     A naive implementation of the backward pass for a max pooling layer.
175

```

```

176 Inputs:
177 - dout: Upstream derivatives
178 - cache: A tuple of (x, pool_param) as in the forward pass.
179
180 Returns:
181 - dx: Gradient with respect to x
182 """
183 dx = None
184 x, pool_param = cache
185 pool_height, pool_width, stride = pool_param['pool_height'],
pool_param['pool_width'], pool_param['stride']
186
187 # ===== #
188 # YOUR CODE HERE:
189 # Implement the max pooling backward pass.
190 # ===== #
191 N, C, H, W = x.shape
192 H_out = 1 + (H - pool_height) // stride
193 W_out = 1 + (W - pool_width) // stride
194 dx = np.zeros(x.shape)
195
196 for n in range(N):
197     for c in range(C):
198         for h in range(H_out):
199             hstride = h * stride
200             for j in range(W_out):
201                 wstride = j * stride
202                 window = x[n, c, hstride:hstride +
203                             pool_height, wstride:wstride+pool_width]
204                 m = np.max(window)
205                 dx[n, c, hstride:hstride+pool_height, wstride:wstride +
206                     pool_width] += (window == m) * dout[n, c, h, j]
207 # ===== #
208 # END YOUR CODE HERE
209 # ===== #
210
211 return dx
212
213
214 def spatial_batchnorm_forward(x, gamma, beta, bn_param):
215     """
216     Computes the forward pass for spatial batch normalization.
217
218     Inputs:
219     - x: Input data of shape (N, C, H, W)
220     - gamma: Scale parameter, of shape (C,)
221     - beta: Shift parameter, of shape (C,)
222     - bn_param: Dictionary with the following keys:
223         - mode: 'train' or 'test'; required
224         - eps: Constant for numeric stability
225         - momentum: Constant for running mean / variance. momentum=0 means that
226           old information is discarded completely at every time step, while
227           momentum=1 means that new information is never incorporated. The
228           default of momentum=0.9 should work well in most situations.
229         - running_mean: Array of shape (D,) giving running mean of features
230         - running_var: Array of shape (D,) giving running variance of features
231
232     Returns a tuple of:
233     - out: Output data, of shape (N, C, H, W)
234     - cache: Values needed for the backward pass

```

```

235     """
236     out, cache = None, None
237
238     # ===== #
239     # YOUR CODE HERE:
240     #     Implement the spatial batchnorm forward pass.
241     #
242     #     You may find it useful to use the batchnorm forward pass you
243     #     implemented in HW #4.
244     # ===== #
245     N, C, H, W = x.shape
246     x_new = x.transpose(0, 3, 2, 1).reshape((N*H*W, C))
247
248     out, cache = batchnorm_forward(x_new, gamma, beta, bn_param)
249     out = out.reshape(N, W, H, C).transpose(0, 3, 2, 1)
250
251     # ===== #
252     # END YOUR CODE HERE
253     # ===== #
254
255     return out, cache
256
257
258 def spatial_batchnorm_backward(dout, cache):
259     """
260     Computes the backward pass for spatial batch normalization.
261
262     Inputs:
263     - dout: Upstream derivatives, of shape (N, C, H, W)
264     - cache: Values from the forward pass
265
266     Returns a tuple of:
267     - dx: Gradient with respect to inputs, of shape (N, C, H, W)
268     - dgamma: Gradient with respect to scale parameter, of shape (C,)
269     - dbeta: Gradient with respect to shift parameter, of shape (C,)
270     """
271     dx, dgamma, dbeta = None, None, None
272
273     # ===== #
274     # YOUR CODE HERE:
275     #     Implement the spatial batchnorm backward pass.
276     #
277     #     You may find it useful to use the batchnorm forward pass you
278     #     implemented in HW #4.
279     # ===== #
280     N, C, H, W = dout.shape
281     dout_new = dout.transpose(0,3,2,1).reshape((N*H*W, C))
282     dx, dgamma, dbeta = batchnorm_backward(dout_new, cache)
283     dx = dx.reshape(N, W, H, C).transpose(0, 3, 2, 1)
284     # ===== #
285     # END YOUR CODE HERE
286     # ===== #
287
288     return dx, dgamma, dbeta
289

```