

```

1 import numpy as np
2 from nndl.layers import *
3 import pdb
4
5 """
6 This code was originally written for CS 231n at Stanford University
7 (cs231n.stanford.edu). It has been modified in various areas for use in the
8 ECE 239AS class at UCLA. This includes the descriptions of what code to
9 implement as well as some slight potential changes in variable names to be
10 consistent with class nomenclature. We thank Justin Johnson & Serena Yeung for
11 permission to use this code. To see the original version, please visit
12 cs231n.stanford.edu.
13 """
14
15 def conv_forward_naive(x, w, b, conv_param):
16     """
17     A naive implementation of the forward pass for a convolutional layer.
18
19     The input consists of N data points, each with C channels, height H and width
20     W. We convolve each input with F different filters, where each filter spans
21     all C channels and has height HH and width WW.
22
23     Input:
24     - x: Input data of shape (N, C, H, W)
25     - w: Filter weights of shape (F, C, HH, WW)
26     - b: Biases, of shape (F,)
27     - conv_param: A dictionary with the following keys:
28       - 'stride': The number of pixels between adjacent receptive fields in the
29         horizontal and vertical directions.
30       - 'pad': The number of pixels that will be used to zero-pad the input.
31
32     Returns a tuple of:
33     - out: Output data, of shape (N, F, H', W') where H' and W' are given by
34        $H' = 1 + (H + 2 * \text{pad} - \text{HH}) / \text{stride}$ 
35        $W' = 1 + (W + 2 * \text{pad} - \text{WW}) / \text{stride}$ 
36     - cache: (x, w, b, conv_param)
37     """
38     out = None
39     pad = conv_param['pad']
40     stride = conv_param['stride']
41
42     # ===== #
43     # YOUR CODE HERE:
44     # Implement the forward pass of a convolutional neural network.
45     # Store the output as 'out'.
46     # Hint: to pad the array, you can use the function np.pad.
47     # ===== #
48
49     N, C, H, W = x.shape
50     F, _, HH, WW = w.shape
51     npad = ((0,0), (0,0), (pad,pad), (pad,pad))
52     x_padded = np.pad(x, pad_width=npad, mode='constant', constant_values=0)
53     H_prime = 1 + (H + 2*pad - HH)/stride
54     W_prime = 1 + (W + 2*pad - WW)/stride
55
56     out = np.empty((N,F,H_prime, W_prime))
57
58     for n in np.arange(0,N):
59         for i in np.arange(0, F):
60             for j in np.arange(0, H_prime):
61                 for k in np.arange(0, W_prime):
62                     out[n,i,j,k] = np.sum(w[i, :, :, :] * x_padded[n, :, j*stride:j*stride+HH, k*stride:k*stride+WW]) + b[i]
63
64
65     # ===== #
66     # END YOUR CODE HERE
67     # ===== #
68
69     cache = (x, w, b, conv_param)
70     return out, cache
71
72
73
74 def conv_backward_naive(dout, cache):
75     """
76     A naive implementation of the backward pass for a convolutional layer.
77
78     Inputs:
79     - dout: Upstream derivatives.
80     - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
81

```

```

82     Returns a tuple of:
83     - dx: Gradient with respect to x
84     - dw: Gradient with respect to w
85     - db: Gradient with respect to b
86     """
87     dx, dw, db = None, None, None
88
89     N, F, out_height, out_width = dout.shape
90     x, w, b, conv_param = cache
91
92     stride, pad = [conv_param['stride'], conv_param['pad']]
93     xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
94     num_filts, _, f_height, f_width = w.shape
95
96     # ===== #
97     # YOUR CODE HERE:
98     # Implement the backward pass of a convolutional neural network.
99     # Calculate the gradients: dx, dw, and db.
100    # ===== #
101
102    dw = np.zeros_like(w)
103    db = np.zeros_like(b)
104    dx = np.zeros_like(x)
105    dxpad = np.zeros_like(xpad)
106    F, _, HH, WW = w.shape
107
108    for n in np.arange(0,N):
109        for f in np.arange(0, F):
110            for j in np.arange(0, out_height):
111                for k in np.arange(0, out_width):
112                    dw[f] += xpad[n, :, j*stride:j*stride+HH, k*stride:k*stride+WW]*dout[n, f, j, k]
113                    db[f] += dout[n,f,j,k]
114                    dxpad[n, :, j*stride:j*stride+HH, k*stride:k*stride+WW] += w[f]*dout[n,f,j,k]
115
116    dx[:] = dxpad[:, :, pad:-pad, pad:-pad]
117
118
119
120
121
122    # ===== #
123    # END YOUR CODE HERE
124    # ===== #
125
126    return dx, dw, db
127
128
129 def max_pool_forward_naive(x, pool_param):
130     """
131     A naive implementation of the forward pass for a max pooling layer.
132
133     Inputs:
134     - x: Input data, of shape (N, C, H, W)
135     - pool_param: dictionary with the following keys:
136       - 'pool_height': The height of each pooling region
137       - 'pool_width': The width of each pooling region
138       - 'stride': The distance between adjacent pooling regions
139
140     Returns a tuple of:
141     - out: Output data
142     - cache: (x, pool_param)
143     """
144     out = None
145
146     # ===== #
147     # YOUR CODE HERE:
148     # Implement the max pooling forward pass.
149     # ===== #
150
151     N, C, H, W = x.shape
152
153     ph = pool_param['pool_height']
154     pw = pool_param['pool_width']
155     stride = pool_param['stride']
156     h_prime = (H - ph)/stride + 1
157     w_prime = (W - pw)/stride + 1
158
159     out = np.empty((N,C,h_prime, w_prime))
160
161     for n in np.arange(0,N):
162         for c in np.arange(0,C):
163             for h in np.arange(0,h_prime):
164                 for w in np.arange(0,w_prime):

```

```

164         out[n,c,h,w] = np.max(x[n,c,h*stride:h*stride+ph,w*stride:w*stride+pw])
165
166
167         # ===== #
168         # END YOUR CODE HERE
169         # ===== #
170     cache = (x, pool_param)
171     return out, cache
172
173 def max_pool_backward_naive(dout, cache):
174     """
175     A naive implementation of the backward pass for a max pooling layer.
176
177     Inputs:
178     - dout: Upstream derivatives
179     - cache: A tuple of (x, pool_param) as in the forward pass.
180
181     Returns:
182     - dx: Gradient with respect to x
183     """
184     dx = None
185     x, pool_param = cache
186     pool_height, pool_width, stride = pool_param['pool_height'], pool_param['pool_width'], pool_param['stride']
187
188     # ===== #
189     # YOUR CODE HERE:
190     # Implement the max pooling backward pass.
191     # ===== #
192     N, C, H, W = x.shape
193     h_prime, w_prime = dout.shape[-2:]
194     dx = np.zeros_like(x)
195
196     for n in np.arange(0,N):
197         for c in np.arange(0,C):
198             for h in np.arange(0,h_prime):
199                 for w in np.arange(0,w_prime):
200                     idx = np.unravel_index( np.argmax( x[n, c, h*stride:h*stride+pool_height, w*stride:w*stride+pool_width]),
201                     (pool_height, pool_width))
202                     dx[n,c,h*stride+idx[0],w*stride+idx[1]] = dout[n,c,h,w]
203
204
205
206     # ===== #
207     # END YOUR CODE HERE
208     # ===== #
209
210     return dx
211
212 def spatial_batchnorm_forward(x, gamma, beta, bn_param):
213     """
214     Computes the forward pass for spatial batch normalization.
215
216     Inputs:
217     - x: Input data of shape (N, C, H, W)
218     - gamma: Scale parameter, of shape (C,)
219     - beta: Shift parameter, of shape (C,)
220     - bn_param: Dictionary with the following keys:
221       - mode: 'train' or 'test'; required
222       - eps: Constant for numeric stability
223       - momentum: Constant for running mean / variance. momentum=0 means that
224         old information is discarded completely at every time step, while
225         momentum=1 means that new information is never incorporated. The
226         default of momentum=0.9 should work well in most situations.
227       - running_mean: Array of shape (D,) giving running mean of features
228       - running_var: Array of shape (D,) giving running variance of features
229
230     Returns a tuple of:
231     - out: Output data, of shape (N, C, H, W)
232     - cache: Values needed for the backward pass
233     """
234     out, cache = None, None
235
236     # ===== #
237     # YOUR CODE HERE:
238     # Implement the spatial batchnorm forward pass.
239     #
240     # You may find it useful to use the batchnorm forward pass you
241     # implemented in HW #4.
242     # ===== #
243
244     x_transpose = x.transpose((0,2,3,1))
245     x_reshaped = x_transpose.reshape((-1,x.shape[1]))

```

```

246
247 out, cache = batchnorm_forward(x_resaped, gamma, beta, bn_param)
248
249 out = out.reshape(*x_transpose.shape).transpose((0,3,1,2))
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
281     dout_t = dout.transpose(0,2,3,1)
282     dout_r = dout_t.reshape(-1, dout.shape[1])
283
284     dx, dgamma, dbeta = batchnorm_backward(dout_r, cache)
285
286     dx = dx.reshape(*dout_t.shape).transpose((0,3,1,2))
287
288     # ===== #
289     # END YOUR CODE HERE
290     # ===== #
291
292     return dx, dgamma, dbeta

```