

```

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 * pad - HH) / stride
35       W' = 1 + (W + 2 * pad - WW) / stride
36     - cache: (x, w, b, conv_param)
37     """
38
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     (N, C, H, W) = x.shape
49     (F, C, HH, WW) = w.shape
50     H_new = 1 + (H + 2 * pad - HH) / stride
51     W_new = 1 + (W + 2 * pad - WW) / stride
52     xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
53     out = np.zeros((N,F,H_new, W_new))
54
55     for n in range(N):
56         for f in range(F):
57             for h in range(H_new):
58                 for wd in range(W_new):
59                     h1 = h*stride
60                     h2 = h1+HH
61                     w1 = wd*stride
62                     w2 = w1+WW
63                     window = xpad[n,:,h1:h2, w1:w2] * w[f,:,:,:]
64                     sumWindow = np.sum(window)
65                     out[n,f,h,wd] = sumWindow + b[f]
66
67
68
69     # ===== #
70     # END YOUR CODE HERE
71     # ===== #
72
73     cache = (x, w, b, conv_param)
74     return out, cache

```

```

75
76
77 def conv_backward_naive(dout, cache):
78     """
79     A naive implementation of the backward pass for a convolutional layer.
80
81     Inputs:
82     - dout: Upstream derivatives.
83     - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
84
85     Returns a tuple of:
86     - dx: Gradient with respect to x
87     - dw: Gradient with respect to w
88     - db: Gradient with respect to b
89     """
90     N, F, out_height, out_width = dout.shape
91     x, w, b, conv_param = cache
92     dx, dw, db = None, None, None
93     dx = np.zeros_like(x)
94     dw = np.zeros_like(w)
95     db = np.zeros_like(b)
96
97     N, C, H, W = x.shape
98     F, _, HH, WW = w.shape
99
100    stride, pad = [conv_param['stride'], conv_param['pad']]
101    xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
102    dxpad = np.pad(dx, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
103    num_filts, _, f_height, f_width = w.shape
104
105    for n in range(N):
106        for f in range(F):
107            for ht in range(out_height):
108                for wd in range(out_width):
109                    h1 = ht*stride
110                    h2 = h1+HH
111                    w1 = wd*stride
112                    w2 = w1+WW
113                    dxpad[n,:,h1:h2, w1:w2] += w[f,:,:,] * dout[n,f,ht,wd]
114                    dw[f,:,:,] += xpad[n,:,h1:h2, w1:w2] * dout[n,f,ht,wd]
115                    db[f] += dout[n,f,ht,wd]
116
117    dx[n,:,:,] = dxpad[n,:, pad:-pad, pad:-pad]
118
119
120    # ===== #
121    # YOUR CODE HERE:
122    # Implement the backward pass of a convolutional neural network.
123    # Calculate the gradients: dx, dw, and db.
124    # ===== #
125
126
127    # ===== #
128    # END YOUR CODE HERE
129    # ===== #
130
131    return dx, dw, db
132
133
134 def max_pool_forward_naive(x, pool_param):
135     """
136     A naive implementation of the forward pass for a max pooling layer.
137
138     Inputs:
139     - x: Input data, of shape (N, C, H, W)
140     - pool_param: dictionary with the following keys:
141         - 'pool_height': The height of each pooling region
142         - 'pool_width': The width of each pooling region
143         - 'stride': The distance between adjacent pooling regions
144
145     Returns a tuple of:
146     - out: Output data
147     - cache: (x, pool_param)
148     """
149     out = None

```

```

150 N, C, H, W = x.shape
151 pool_height = pool_param['pool_height']
152 pool_width = pool_param['pool_width']
153 stride = pool_param['stride']
154 H_new = 1 + (H - pool_height) / stride
155 W_new = 1 + (W - pool_width) / stride
156 out = np.zeros((N,C,H_new, W_new))
157 # ===== #
158 # YOUR CODE HERE:
159 # Implement the max pooling forward pass.
160 # ===== #
161 for n in range(N):
162     for c in range(C):
163         for h in range(H_new):
164             for wd in range(W_new):
165                 h1 = h*stride
166                 h2 = h1+pool_height
167                 w1 = wd*stride
168                 w2 = w1+pool_width
169                 window = np.max(x[n,c,h1:h2, w1:w2])
170                 out[n,c,h,wd] = window
171
172 # ===== #
173 # END YOUR CODE HERE
174 # ===== #
175 cache = (x, pool_param)
176 return out, cache
177
178 def max_pool_backward_naive(dout, cache):
179     """
180     A naive implementation of the backward pass for a max pooling layer.
181
182     Inputs:
183     - dout: Upstream derivatives
184     - cache: A tuple of (x, pool_param) as in the forward pass.
185
186     Returns:
187     - dx: Gradient with respect to x
188     """
189     dx = None
190     x, pool_param = cache
191     pool_height, pool_width, stride = pool_param['pool_height'], pool_param['pool_width'], pool_param['stride']
192
193     # ===== #
194     # YOUR CODE HERE:
195     # Implement the max pooling backward pass.
196     # ===== #
197     N, F, out_height, out_width = dout.shape
198     dx = np.zeros_like(x)
199
200     N, C, H, W = x.shape
201
202
203     for n in range(N):
204         for c in range(C):
205             for ht in range(out_height):
206                 for wd in range(out_width):
207                     h1 = ht*stride
208                     h2 = h1+pool_height
209                     w1 = wd*stride
210                     w2 = w1+pool_width
211                     window = x[n,c,h1:h2, w1:w2]
212                     window2 = np.reshape(window, (pool_height*pool_width))
213                     window3 = np.zeros_like(window2)
214                     window3[np.argmax(window2)] = 1
215
216                     dx[n,c,h1:h2, w1:w2] = np.reshape(window3, (pool_height, pool_width)) * dout[n,c,ht,wd]
217
218
219
220 # ===== #
221 # END YOUR CODE HERE
222 # ===== #
223
224 return dx

```

```

225
226 def spatial_batchnorm_forward(x, gamma, beta, bn_param):
227     """
228     Computes the forward pass for spatial batch normalization.
229
230     Inputs:
231     - x: Input data of shape (N, C, H, W)
232     - gamma: Scale parameter, of shape (C,)
233     - beta: Shift parameter, of shape (C,)
234     - bn_param: Dictionary with the following keys:
235       - mode: 'train' or 'test'; required
236       - eps: Constant for numeric stability
237       - momentum: Constant for running mean / variance. momentum=0 means that
238         old information is discarded completely at every time step, while
239         momentum=1 means that new information is never incorporated. The
240         default of momentum=0.9 should work well in most situations.
241       - running_mean: Array of shape (D,) giving running mean of features
242       - running_var: Array of shape (D,) giving running variance of features
243
244     Returns a tuple of:
245     - out: Output data, of shape (N, C, H, W)
246     - cache: Values needed for the backward pass
247     """
248     out, cache = None, None
249     N, C, H, W = x.shape
250     XTranspose = x.transpose(0,2,3,1)
251     x_reshape = np.reshape(XTranspose, (N*H*W, C))
252     # ===== #
253     # YOUR CODE HERE:
254     # Implement the spatial batchnorm forward pass.
255     #
256     # You may find it useful to use the batchnorm forward pass you
257     # implemented in HW #4.
258     # ===== #
259     out1, cache = batchnorm_forward(x_reshape, gamma, beta, bn_param)
260
261     out = out1.reshape((N,H,W,C)).transpose(0,3,1,2)
262
263     # ===== #
264     # END YOUR CODE HERE
265     # ===== #
266
267     return out, cache
268
269
270 def spatial_batchnorm_backward(dout, cache):
271     """
272     Computes the backward pass for spatial batch normalization.
273
274     Inputs:
275     - dout: Upstream derivatives, of shape (N, C, H, W)
276     - cache: Values from the forward pass
277
278     Returns a tuple of:
279     - dx: Gradient with respect to inputs, of shape (N, C, H, W)
280     - dgamma: Gradient with respect to scale parameter, of shape (C,)
281     - dbeta: Gradient with respect to shift parameter, of shape (C,)
282     """
283     dx, dgamma, dbeta = None, None, None
284
285     # ===== #
286     # YOUR CODE HERE:
287     # Implement the spatial batchnorm backward pass.
288     #
289     # You may find it useful to use the batchnorm forward pass you
290     # implemented in HW #4.
291     # ===== #
292     dx = np.zeros_like(dout)
293     N, C, H, W = dout.shape
294     doutTranspose = dout.transpose((0, 2, 3, 1))
295     dout_reshape = np.reshape(doutTranspose, (-1, C))
296     dx_flat, dgamma, dbeta = batchnorm_backward(dout_reshape, cache)
297     dx = dx_flat.reshape((N, H, W, C)).transpose(0, 3, 1, 2)
298
299     # ===== #

```

```
300  # END YOUR CODE HERE
301  # ===== #
302
303  return dx, dgamma, dbeta
```