

```

1 import numpy as np
2 from nn.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
11 for
12 permission to use this code. To see the original version, please visit
13 cs231n.stanford.edu.
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
20 width
21 W. We convolve each input with F different filters, where each filter spans
22 all C channels and has height HH and width WW.
23
24 Input:
25 - x: Input data of shape (N, C, H, W)
26 - w: Filter weights of shape (F, C, HH, WW)
27 - b: Biases, of shape (F,)
28 - conv_param: A dictionary with the following keys:
29     - 'stride': The number of pixels between adjacent receptive fields in the
30       horizontal and vertical directions.
31     - 'pad': The number of pixels that will be used to zero-pad the input.
32
33 Returns a tuple of:
34 - out: Output data, of shape (N, F, H', W') where H' and W' are given by
35      $H' = 1 + (H + 2 * pad - HH) / stride$ 
36      $W' = 1 + (W + 2 * pad - WW) / stride$ 
37 - cache: (x, w, b, conv_param)
38 """
39 out = None
40 pad = conv_param['pad']
41 stride = conv_param['stride']
42
43 # ===== #
44 # YOUR CODE HERE:
45 # Implement the forward pass of a convolutional neural network.
46 # Store the output as 'out'.
47 # Hint: to pad the array, you can use the function np.pad.
48 # ===== #
49 xpad = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), 'constant',
50 constant_values=0)
51 N, _, H, W = xpad.shape
52 F, _, HH, WW = w.shape
53 H_new = int(1 + (H - HH) / stride)
54 W_new = int(1 + (W - WW) / stride)
55 out = np.zeros((N, F, H_new, W_new))
56 for n in range(N):
57     for f in range(F):
58         for i in range(0, H_new):

```

```

58         for j in range(0, W_new):
59             i_start = i * stride
60             j_start = j * stride
61             x_patch = xpad[n, :, i_start:i_start+HH, j_start:j_start+WW]
62             conv = np.sum(np.multiply(x_patch, w[f])) + b[f]
63             out[n, f, i, j] = conv
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
101     dx = np.zeros_like(xpad)
102     dw = np.zeros_like(w)
103     db = np.zeros_like(b)
104     for n in range(N):
105         for f in range(F):
106             db[f] += np.sum(dout[n, f])
107             for i in range(out_height):
108                 for j in range(out_width):
109                     i_start = i * stride
110                     j_start = j * stride
111                     i_end = i_start + f_height
112                     j_end = j_start + f_width
113                     x_patch = xpad[n, :, i_start:i_end, j_start:j_end]
114
115                     dx[n, :, i_start:i_end, j_start:j_end] += w[f] * dout[n, f, i, j]
116                     dw[f] += dout[n, f, i, j] * x_patch
117

```

```

118 dx = dx[:, :, pad:pad+x.shape[2], pad:pad+x.shape[3]]
119
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
149     N, C, H, W = x.shape
150     pool_h = pool_param['pool_height']
151     pool_w = pool_param['pool_width']
152     stride = pool_param['stride']
153     H_new = int(1 + (H-pool_h) / stride)
154     W_new = int(1 + (W-pool_w) / stride)
155     out = np.zeros((N, C, H_new, W_new))
156
157     for n in range(N):
158         for c in range(C):
159             for i in range(H_new):
160                 for j in range(W_new):
161                     i_start = i * stride
162                     j_start = j * stride
163                     i_end = i_start + pool_h
164                     j_end = j_start + pool_w
165                     out[n, c, i, j] = np.max(x[n, c, i_start:i_end, j_start:j_end])
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 = dout.shape
193
194 dx = np.zeros_like(x)
195
196 for n in range(N):
197     for c in range(C):
198         for i in range(H):
199             for j in range(W):
200                 i_start = i * stride
201                 j_start = j * stride
202                 i_end = i_start + pool_height
203                 j_end = j_start + pool_width
204                 x_patch = x[n, c, i_start:i_end, j_start:j_end]
205                 mask = x_patch == np.max(x_patch)
206                 dx[n, c, i_start:i_end, j_start:j_end] += mask * dout[n, c, i, j]
207
208 # ===== #
209 # END YOUR CODE HERE
210 # ===== #
211
212 return dx
213 def spatial_batchnorm_forward(x, gamma, beta, bn_param):
214     """
215     Computes the forward pass for spatial batch normalization.
216
217     Inputs:
218     - x: Input data of shape (N, C, H, W)
219     - gamma: Scale parameter, of shape (C,)
220     - beta: Shift parameter, of shape (C,)
221     - bn_param: Dictionary with the following keys:
222         - mode: 'train' or 'test'; required
223         - eps: Constant for numeric stability
224         - momentum: Constant for running mean / variance. momentum=0 means that
225           old information is discarded completely at every time step, while
226           momentum=1 means that new information is never incorporated. The
227           default of momentum=0.9 should work well in most situations.
228         - running_mean: Array of shape (D,) giving running mean of features
229         - running_var: Array of shape (D,) giving running variance of features
230
231     Returns a tuple of:
232     - out: Output data, of shape (N, C, H, W)
233     - cache: Values needed for the backward pass
234     """
235     out, cache = None, None
236

```

```

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

```