

```

1 import numpy as np
2 from nndl.layers import *
3 import pdb
4
5
6 def conv_forward_naive(x, w, b, conv_param):
7     """
8     A naive implementation of the forward pass for a convolutional layer.
9
10    The input consists of N data points, each with C channels, height H and
width
11    W. We convolve each input with F different filters, where each filter spans
12    all C channels and has height HH and width WW.
13
14    Input:
15    - x: Input data of shape (N, C, H, W)
16    - w: Filter weights of shape (F, C, HH, WW)
17    - b: Biases, of shape (F,)
18    - conv_param: A dictionary with the following keys:
19        - 'stride': The number of pixels between adjacent receptive fields in the
20          horizontal and vertical directions.
21        - 'pad': The number of pixels that will be used to zero-pad the input.
22
23    Returns a tuple of:
24    - out: Output data, of shape (N, F, H', W') where H' and W' are given by
25        H' = 1 + (H + 2 * pad - HH) / stride
26        W' = 1 + (W + 2 * pad - WW) / stride
27    - cache: (x, w, b, conv_param)
28    """
29    out = None
30    pad = conv_param['pad']
31    stride = conv_param['stride']
32
33    # ===== #
34    # YOUR CODE HERE:
35    # Implement the forward pass of a convolutional neural network.
36    # Store the output as 'out'.
37    # Hint: to pad the array, you can use the function np.pad.
38    # ===== #
39
40    x_pad = np.pad(x, [(0, 0), (0, 0), (pad, pad), (pad, pad)],
mode='constant')
41
42    N, C, H, W = x.shape
43    F, C, HH, WW = w.shape
44
45    H2 = int(1 + (H + 2 * pad - HH) / stride)
46    W2 = int(1 + (W + 2 * pad - WW) / stride)
47
48    out = np.zeros([N, F, H2, W2])
49
50    for n in np.arange(N):
51        for f in np.arange(F):
52            for row in np.arange(H2):
53                for col in np.arange(W2):
54                    out[n, f, row, col] = np.sum(w[f, :, :, :] * x_pad[n, :, row*stride
: row*stride+HH, col*stride : col*stride+WW]) + b[f]
55
56    # ===== #

```

```

57 # END YOUR CODE HERE
58 # ===== #
59
60 cache = (x, w, b, conv_param)
61 return out, cache
62
63
64 def conv_backward_naive(dout, cache):
65     """
66     A naive implementation of the backward pass for a convolutional layer.
67
68     Inputs:
69     - dout: Upstream derivatives.
70     - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
71
72     Returns a tuple of:
73     - dx: Gradient with respect to x
74     - dw: Gradient with respect to w
75     - db: Gradient with respect to b
76     """
77     dx, dw, db = None, None, None
78
79     N, F, out_height, out_width = dout.shape
80     x, w, b, conv_param = cache
81
82     stride, pad = [conv_param['stride'], conv_param['pad']]
83     xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
84     num_filts, _, f_height, f_width = w.shape
85
86     # ===== #
87     # YOUR CODE HERE:
88     #   Implement the backward pass of a convolutional neural network.
89     #   Calculate the gradients: dx, dw, and db.
90     # ===== #
91
92     H = x.shape[2]
93     W = x.shape[3]
94
95     dx_pad = np.zeros_like(xpad)
96     dw = np.zeros_like(w)
97     db = np.zeros_like(b)
98
99     # dx
100     for n in np.arange(N):
101         for f in np.arange(F):
102             for row in np.arange(out_height):
103                 for col in np.arange(out_width):
104                     dx_pad[n, :, row*stride : row*stride+f_height, col*stride :
col*stride+f_width] += dout[n, f, row, col] * w[f, :, :, :]
105     dx = dx_pad[:, :, pad : pad+H, pad : pad+W]
106
107     # dw
108     for n in np.arange(N):
109         for f in np.arange(F):
110             for row in np.arange(out_height):
111                 for col in np.arange(out_width):
112                     dw[f, :, :, :] += dout[n, f, row, col] * xpad[n, :, row*stride :
row*stride+f_height, col*stride : col*stride+f_width]
113
114     # db

```

```

115     for f in np.arange(F):
116         db[f] += np.sum(dout[:, f, :, :])
117
118     # ===== #
119     # END YOUR CODE HERE
120     # ===== #
121
122     return dx, dw, db
123
124
125 def max_pool_forward_naive(x, pool_param):
126     """
127     A naive implementation of the forward pass for a max pooling layer.
128
129     Inputs:
130     - x: Input data, of shape (N, C, H, W)
131     - pool_param: dictionary with the following keys:
132         - 'pool_height': The height of each pooling region
133         - 'pool_width': The width of each pooling region
134         - 'stride': The distance between adjacent pooling regions
135
136     Returns a tuple of:
137     - out: Output data
138     - cache: (x, pool_param)
139     """
140     out = None
141
142     # ===== #
143     # YOUR CODE HERE:
144     #   Implement the max pooling forward pass.
145     # ===== #
146
147     pool_height = pool_param['pool_height']
148     pool_width = pool_param['pool_width']
149     stride = pool_param['stride']
150
151     N, C, H, W = x.shape
152
153     H2 = int(1 + (H - pool_height) / stride)
154     W2 = int(1 + (W - pool_width) / stride)
155
156     out = np.zeros([N, C, H2, W2])
157
158     for n in np.arange(N):
159         for c in np.arange(C):
160             for row in np.arange(H2):
161                 for col in np.arange(W2):
162                     out[n, c, row, col] = np.max(x[n, c, row*stride :
163 row*stride+pool_height, col*stride : col*stride+pool_width])
164
165     # ===== #
166     # END YOUR CODE HERE
167     # ===== #
168     cache = (x, pool_param)
169     return out, cache
170
171 def max_pool_backward_naive(dout, cache):
172     """
173     A naive implementation of the backward pass for a max pooling layer.

```

```

174     Inputs:
175     - dout: Upstream derivatives
176     - cache: A tuple of (x, pool_param) as in the forward pass.
177
178     Returns:
179     - dx: Gradient with respect to x
180     """
181     dx = None
182     x, pool_param = cache
183     pool_height, pool_width, stride = pool_param['pool_height'],
pool_param['pool_width'], pool_param['stride']
184
185     # ===== #
186     # YOUR CODE HERE:
187     #   Implement the max pooling backward pass.
188     # ===== #
189
190     N, C, H, W = x.shape
191     out_height = dout.shape[2]
192     out_width = dout.shape[3]
193
194     dx = np.zeros_like(x)
195
196     for n in np.arange(N):
197         for c in np.arange(C):
198             for row in np.arange(out_height):
199                 for col in np.arange(out_width):
200                     max_idx = np.unravel_index(np.argmax(x[n, c, row*stride :
row*stride+pool_height, col*stride : col*stride+pool_width]), [pool_height,
pool_width])
201                     dx[n, c, row*stride+max_idx[0], col*stride+max_idx[1]] = dout[n, c,
row, col]
202
203     # ===== #
204     # END YOUR CODE HERE
205     # ===== #
206
207     return dx
208
209 def spatial_batchnorm_forward(x, gamma, beta, bn_param):
210     """
211     Computes the forward pass for spatial batch normalization.
212
213     Inputs:
214     - x: Input data of shape (N, C, H, W)
215     - gamma: Scale parameter, of shape (C,)
216     - beta: Shift parameter, of shape (C,)
217     - bn_param: Dictionary with the following keys:
218         - mode: 'train' or 'test'; required
219         - eps: Constant for numeric stability
220         - momentum: Constant for running mean / variance. momentum=0 means that
221           old information is discarded completely at every time step, while
222           momentum=1 means that new information is never incorporated. The
223           default of momentum=0.9 should work well in most situations.
224         - running_mean: Array of shape (D,) giving running mean of features
225         - running_var: Array of shape (D,) giving running variance of features
226
227     Returns a tuple of:
228     - out: Output data, of shape (N, C, H, W)
229     - cache: Values needed for the backward pass

```

```

230     """
231     out, cache = None, None
232
233     # ===== #
234     # YOUR CODE HERE:
235     #   Implement the spatial batchnorm forward pass.
236     #
237     #   You may find it useful to use the batchnorm forward pass you
238     #   implemented in HW #4.
239     # ===== #
240
241     N, C, H, W = x.shape
242
243     x_reshape = np.reshape(np.transpose(x, (0, 2, 3, 1)), (N*H*W, C))
244     out_2D, cache = batchnorm_forward(x_reshape, gamma, beta, bn_param)
245
246     out = np.transpose(np.reshape(out_2D, (N, H, W, C)), (0, 3, 1, 2))
247
248     # ===== #
249     # END YOUR CODE HERE
250     # ===== #
251
252     return out, cache
253
254
255 def spatial_batchnorm_backward(dout, cache):
256     """
257     Computes the backward pass for spatial batch normalization.
258
259     Inputs:
260     - dout: Upstream derivatives, of shape (N, C, H, W)
261     - cache: Values from the forward pass
262
263     Returns a tuple of:
264     - dx: Gradient with respect to inputs, of shape (N, C, H, W)
265     - dgamma: Gradient with respect to scale parameter, of shape (C,)
266     - dbeta: Gradient with respect to shift parameter, of shape (C,)
267     """
268     dx, dgamma, dbeta = None, None, None
269
270     # ===== #
271     # YOUR CODE HERE:
272     #   Implement the spatial batchnorm backward pass.
273     #
274     #   You may find it useful to use the batchnorm forward pass you
275     #   implemented in HW #4.
276     # ===== #
277
278     N, C, H, W = dout.shape
279
280     dout_reshape = np.reshape(np.transpose(dout, (0, 2, 3, 1)), (N*H*W, C))
281     dx_2D, dgamma, dbeta = batchnorm_backward(dout_reshape, cache)
282
283     dx = np.transpose(np.reshape(dx_2D, (N, H, W, C)), (0, 3, 1, 2))
284
285     # ===== #
286     # END YOUR CODE HERE
287     # ===== #
288
289     return dx, dgamma, dbeta

```