

```

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     N,C,H,W = x.shape
40     F,C,HH,WW = w.shape
41     H_out_shape = 1 + (H + 2 * pad - HH) // stride
42     W_out_shape = 1 + (W + 2 * pad - WW) // stride
43
44     out = np.zeros((N,F,H_out_shape,W_out_shape))
45
46     x = np.pad(x, pad_width = ((0,0),(0,0),(pad,pad),(pad,pad)), mode = 'constant')
47     for i in range(N):
48         for j in range(F):
49             for k in range(H_out_shape):
50                 for l in range(W_out_shape):
51                     x_selected = x[i,:,k * stride:(k*stride + HH), l * stride : (l * stride +
52                                     WW)]
53                     w_selected = w[j,:,:,:]
54                     out[i,j,k,l] = np.sum(x_selected * w_selected) + b[j]
55
56     # ===== #
57     # END YOUR CODE HERE
58     # ===== #
59
60     cache = (x, w, b, conv_param)
61     return out, cache
62
63 def conv_backward_naive(dout, cache):
64     """
65     A naive implementation of the backward pass for a convolutional layer.
66

```

```

67     Inputs:
68     - dout: Upstream derivatives.
69     - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
70
71     Returns a tuple of:
72     - dx: Gradient with respect to x
73     - dw: Gradient with respect to w
74     - db: Gradient with respect to b
75     """
76     dx, dw, db = None, None, None
77
78     N, F, out_height, out_width = dout.shape
79     x, w, b, conv_param = cache
80
81     stride, pad = [conv_param['stride'], conv_param['pad']]
82     xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
83     num_filts, _, f_height, f_width = w.shape
84
85     # ===== #
86     # YOUR CODE HERE:
87     # Implement the backward pass of a convolutional neural network.
88     # Calculate the gradients: dx, dw, and db.
89     # ===== #
90     N,F,H,W = x.shape
91
92     H_out_shape = 1 + (H - f_height) // stride
93     W_out_shape = 1 + (W - f_width) // stride
94
95     dx = np.zeros(x.shape)
96     dw = np.zeros(w.shape)
97     db = np.zeros(b.shape)
98
99     for i in range(N):
100         for j in range(num_filts):
101             if i == 0:
102                 db[j] += np.sum(dout[:,j,:,:])
103             for k in range(H_out_shape):
104                 for l in range(W_out_shape):
105                     k_tmp = k * stride
106                     l_tmp = l * stride
107                     dout_tmp = dout[i,j,k,l]
108                     dx[i,:, k_tmp:(k_tmp + f_height), l_tmp:(l_tmp + f_width)] += w[j,:,:, :]
109                     * dout_tmp
110                     dw[j,:,:, :] += x[i,:, k_tmp:(k_tmp + f_height), l_tmp:(l_tmp + f_width)]
111                     * dout_tmp
112                 dx = dx[:, :, pad:-pad, pad:-pad]
113             # ===== #
114             # END YOUR CODE HERE
115             # ===== #
116
117     return dx, dw, db
118
119 def max_pool_forward_naive(x, pool_param):
120     """
121     A naive implementation of the forward pass for a max pooling layer.
122
123     Inputs:
124     - x: Input data, of shape (N, C, H, W)
125     - pool_param: dictionary with the following keys:
126         - 'pool_height': The height of each pooling region
127         - 'pool_width': The width of each pooling region
128         - 'stride': The distance between adjacent pooling regions
129
130     Returns a tuple of:
131     - out: Output data
132     - cache: (x, pool_param)

```

```

132 """
133 out = None
134
135 # ===== #
136 # YOUR CODE HERE:
137 #   Implement the max pooling forward pass.
138 # ===== #
139
140 pool_height, pool_width, stride = pool_param['pool_height'], pool_param['pool_width'],
141 pool_param['stride']
142 N,C,H,W = x.shape
143
144 H_out_shape = 1 + (H - pool_height) // stride
145 W_out_shape = 1 + (W - pool_width) // stride
146 out = np.zeros((N,C, H_out_shape, W_out_shape))
147 for i in range(N):
148     for j in range(C):
149         for k in range(H_out_shape):
150             for l in range(W_out_shape):
151                 k_tmp = k * stride
152                 l_tmp = l * stride
153                 x_tmp = x[i,j,k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)]
154                 out[i,j,k,l] = np.max(x_tmp)
155
156 # ===== #
157 # END YOUR CODE HERE
158 # ===== #
159 cache = (x, pool_param)
160 return out, cache
161
162 def max_pool_backward_naive(dout, cache):
163     """
164     A naive implementation of the backward pass for a max pooling layer.
165
166     Inputs:
167     - dout: Upstream derivatives
168     - cache: A tuple of (x, pool_param) as in the forward pass.
169
170     Returns:
171     - dx: Gradient with respect to x
172     """
173     dx = None
174     x, pool_param = cache
175     pool_height, pool_width, stride = pool_param['pool_height'], pool_param['pool_width'],
176     pool_param['stride']
177
178 # ===== #
179 # YOUR CODE HERE:
180 #   Implement the max pooling backward pass.
181 # ===== #
182
183 N,C,H,W = x.shape
184 H_out_shape = 1 + (H - pool_height) // stride
185 W_out_shape = 1 + (W - pool_width) // stride
186
187 dx = np.zeros((N,C,H,W))
188 for i in range(N):
189     for j in range(C):
190         for k in range(H_out_shape):
191             for l in range(W_out_shape):
192                 k_tmp = k * stride
193                 l_tmp = l * stride
194                 x_tmp = x[i,j,k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)]
195                 dout_tmp = dout[i,j,k,l]
196                 din_mask = x_tmp == np.max(x_tmp)
197                 dx[i,j, k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)] +=
198                 din_mask * dout_tmp

```

```

196 # ===== #
197 # END YOUR CODE HERE
198 # ===== #
199
200 return dx
201
202 def spatial_batchnorm_forward(x, gamma, beta, bn_param):
203     """
204     Computes the forward pass for spatial batch normalization.
205
206     Inputs:
207     - x: Input data of shape (N, C, H, W)
208     - gamma: Scale parameter, of shape (C,)
209     - beta: Shift parameter, of shape (C,)
210     - bn_param: Dictionary with the following keys:
211       - mode: 'train' or 'test'; required
212       - eps: Constant for numeric stability
213       - momentum: Constant for running mean / variance. momentum=0 means that
214         old information is discarded completely at every time step, while
215         momentum=1 means that new information is never incorporated. The
216         default of momentum=0.9 should work well in most situations.
217       - running_mean: Array of shape (D,) giving running mean of features
218       - running_var: Array of shape (D,) giving running variance of features
219
220     Returns a tuple of:
221     - out: Output data, of shape (N, C, H, W)
222     - cache: Values needed for the backward pass
223     """
224     out, cache = None, None
225
226     # ===== #
227     # YOUR CODE HERE:
228     #   Implement the spatial batchnorm forward pass.
229     #
230     #   You may find it useful to use the batchnorm forward pass you
231     #   implemented in HW #4.
232     # ===== #
233     N,C,H,W = x.shape
234     x_flattened = (x.reshape((N,H,W,C))).reshape((N*W*H,C))
235     out_bn, cache = batchnorm_forward(x_flattened, gamma,beta, bn_param = bn_param)
236     out = (out_bn.reshape((N,W,H,C))).swapaxes(1,3)
237     # ===== #
238     # END YOUR CODE HERE
239     # ===== #
240
241     return out, cache
242
243
244 def spatial_batchnorm_backward(dout, cache):
245     """
246     Computes the backward pass for spatial batch normalization.
247
248     Inputs:
249     - dout: Upstream derivatives, of shape (N, C, H, W)
250     - cache: Values from the forward pass
251
252     Returns a tuple of:
253     - dx: Gradient with respect to inputs, of shape (N, C, H, W)
254     - dgamma: Gradient with respect to scale parameter, of shape (C,)
255     - dbeta: Gradient with respect to shift parameter, of shape (C,)
256     """
257     dx, dgamma, dbeta = None, None, None
258
259     # ===== #
260     # YOUR CODE HERE:
261     #   Implement the spatial batchnorm backward pass.
262     #

```

```

263 # You may find it useful to use the batchnorm forward pass you
264 # implemented in HW #4.
265 # ===== #
266 N,C,H,W = dout.shape
267 dout_bn = dout.swapaxes(1,3).reshape((N*W*H,C))
268 dx_bn, dgamma_bn, dbeta_bn = batchnorm_backward(dout_bn,cache)
269 dx = dx_bn.reshape((N,C,H,W))
270 dgamma = dgamma_bn.reshape((C,))
271 dbeta = dbeta_bn.reshape((C,))
272 # ===== #
273 # END YOUR CODE HERE
274 # ===== #
275
276 return dx, dgamma, dbeta

```