

```

1  import numpy as np
2
3  def affine_forward(x, w, b):
4      """
5      Computes the forward pass for an affine (fully-connected) layer.
6
7      The input x has shape (N, d_1, ..., d_k) where x[i] is the ith input.
8      We multiply this against a weight matrix of shape (D, M) where
9      D = \prod_i d_i
10
11      Inputs:
12      x - Input data, of shape (N, d_1, ..., d_k)
13      w - Weights, of shape (D, M)
14      b - Biases, of shape (M,)
15
16      Returns a tuple of:
17      - out: output, of shape (N, M)
18      - cache: (x, w, b)
19      """
20      out = None
21      #####
22      # TODO: Implement the affine forward pass. Store the result in out. You      #
23      # will need to reshape the input into rows.                                #
24      #####
25      row_dim = x.shape[0]
26      col_dim = np.prod(x.shape[1:])
27      x_reshape = x.reshape(row_dim, col_dim)
28      out = np.dot(x_reshape, w) + b
29      #####
30      #                                     END OF YOUR CODE                        #
31      #####
32      cache = (x, w, b)
33      return out, cache
34
35
36 def affine_backward(dout, cache):
37     """
38     Computes the backward pass for an affine layer.
39
40     Inputs:
41     - dout: Upstream derivative, of shape (N, M)
42     - cache: Tuple of:
43       - x: Input data, of shape (N, d_1, ... d_k)
44       - w: Weights, of shape (D, M)
45
46     Returns a tuple of:
47     - dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
48     - dw: Gradient with respect to w, of shape (D, M)
49     - db: Gradient with respect to b, of shape (M,)
50     """
51     x, w, b = cache
52     dx, dw, db = None, None, None
53     #####
54     # TODO: Implement the affine backward pass.                                #
55     #####
56     row_dim = x.shape[0]
57     col_dim = np.prod(x.shape[1:])
58     x2 = np.reshape(x, (row_dim, col_dim))
59
60     dx2 = np.dot(dout, w.T) # row_dim x col_dim
61     dw = np.dot(x2.T, dout) # col_dim x M
62     db = np.dot(dout.T, np.ones(row_dim)) # M x 1
63
64     dx = np.reshape(dx2, x.shape)
65     #####
66     #                                     END OF YOUR CODE                        #
67     #####

```

```

68     return dx, dw, db
69
70
71 def relu_forward(x):
72     """
73     Computes the forward pass for a layer of rectified linear units (ReLU).
74
75     Input:
76     - x: Inputs, of any shape
77
78     Returns a tuple of:
79     - out: Output, of the same shape as x
80     - cache: x
81     """
82     out = None
83     #####
84     # TODO: Implement the ReLU forward pass. #
85     #####
86     # This secures that ReLU never goes below zero
87     out = np.maximum(0, x)
88     #####
89     #                               END OF YOUR CODE #
90     #####
91     cache = x
92     return out, cache
93
94
95 def relu_backward(dout, cache):
96     """
97     Computes the backward pass for a layer of rectified linear units (ReLU).
98
99     Input:
100    - dout: Upstream derivatives, of any shape
101    - cache: Input x, of same shape as dout
102
103    Returns:
104    - dx: Gradient with respect to x
105    """
106    dx, x = None, cache
107    #####
108    # TODO: Implement the ReLU backward pass. #
109    #####
110    # This secures that ReLU never goes below zero
111    out = np.maximum(0, x)
112    out[out > 0] = 1
113    dx = out * dout
114    #####
115    #                               END OF YOUR CODE #
116    #####
117    return dx
118
119 def dropout_forward(x, dropout_param):
120     """
121     Performs the forward pass for (inverted) dropout.
122
123     Inputs:
124     - x: Input data, of any shape
125     - dropout_param: A dictionary with the following keys:
126       - p: Dropout parameter. We keep each neuron output with probability p.
127       - mode: 'test' or 'train'. If the mode is train, then perform dropout;
128         if the mode is test, then just return the input.
129       - seed: Seed for the random number generator. Passing seed makes this
130         function deterministic, which is needed for gradient checking but not in
131         real networks.
132
133     Outputs:
134     - out: Array of the same shape as x.

```

```

135 - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
136 mask that was used to multiply the input; in test mode, mask is None.
137 """
138 p, mode = dropout_param['p'], dropout_param['mode']
139 if 'seed' in dropout_param:
140     np.random.seed(dropout_param['seed'])
141
142 mask = None
143 out = None
144
145 if mode == 'train':
146     #####
147     # TODO: Implement the training phase forward pass for inverted dropout. #
148     # Store the dropout mask in the mask variable. #
149     #####
150     preMask = (np.random.rand(*x.shape) < p)
151     mask = preMask / p
152     out = x * mask
153     #####
154     #                               END OF YOUR CODE                               #
155     #####
156 elif mode == 'test':
157     #####
158     # TODO: Implement the test phase forward pass for inverted dropout. #
159     #####
160     out = x
161     #####
162     #                               END OF YOUR CODE                               #
163     #####
164
165 cache = (dropout_param, mask)
166 out = out.astype(x.dtype, copy=False)
167
168 return out, cache
169
170
171 def dropout_backward(dout, cache):
172     """
173     Perform the backward pass for (inverted) dropout.
174
175     Inputs:
176     - dout: Upstream derivatives, of any shape
177     - cache: (dropout_param, mask) from dropout_forward.
178     """
179     dropout_param, mask = cache
180     mode = dropout_param['mode']
181     if mode == 'train':
182         #####
183         # TODO: Implement the training phase forward pass for inverted dropout. #
184         # Store the dropout mask in the mask variable. #
185         #####
186         dx = dout * mask
187         #####
188         #                               END OF YOUR CODE                               #
189         #####
190     elif mode == 'test':
191         dx = dout
192     return dx
193
194
195 def conv_forward_naive(x, w, b, conv_param):
196     """
197     A naive implementation of the forward pass for a convolutional layer.
198
199     The input consists of N data points, each with C channels, height H and width
200     W. We convolve each input with F different filters, where each filter spans
201     all C channels and has height HH and width HH.

```

```

202
203 Input:
204 - x: Input data of shape (N, C, H, W)
205 - w: Filter weights of shape (F, C, HH, WW)
206 - b: Biases, of shape (F,)
207 - conv_param: A dictionary with the following keys:
208   - 'stride': The number of pixels between adjacent receptive fields in the
209     horizontal and vertical directions.
210   - 'pad': The number of pixels that will be used to zero-pad the input.
211
212 Returns a tuple of:
213 - out: Output data, of shape (N, F, H', W') where H' and W' are given by
214   H' = 1 + (H + 2 * pad - HH) / stride
215   W' = 1 + (W + 2 * pad - WW) / stride
216 - cache: (x, w, b, conv_param)
217 """
218 out = None
219 #####
220 # TODO: Implement the convolutional forward pass. #
221 # Hint: you can use the function np.pad for padding. #
222 #####
223 N,C1,H,W = x.shape
224 F,C2,HH,WW = w.shape
225 stride = conv_param['stride']
226 pad = conv_param['pad']
227
228 H_mark = 1 + (H + 2 * pad - HH) / stride
229 W_mark = 1 + (W + 2 * pad - WW) / stride
230 out = np.zeros((N,F,H_mark,W_mark))
231
232 for n in range(N):
233     x_pad = np.pad(x[n,:,:,:], ((0,0),(pad,pad),(pad,pad)), 'constant')
234     for f in range(F):
235         for h_mark in range(H_mark):
236             for w_mark in range(W_mark):
237                 h1 = h_mark * stride
238                 h2 = h_mark * stride + HH
239                 w1 = w_mark * stride
240                 w2 = w_mark * stride + WW
241                 window = x_pad[:, h1:h2, w1:w2]
242                 out[n, f, h_mark, w_mark] = np.sum(window * w[f,:,:,:]) + b[f]
243 #####
244 #                                     END OF YOUR CODE                                     #
245 #####
246 cache = (x, w, b, conv_param)
247 return out, cache
248
249
250 def conv_backward_naive(dout, cache):
251     """
252     A naive implementation of the backward pass for a convolutional layer.
253
254     Inputs:
255     - dout: Upstream derivatives.
256     - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
257
258     Returns a tuple of:
259     - dx: Gradient with respect to x
260     - dw: Gradient with respect to w
261     - db: Gradient with respect to b
262     """
263     dx, dw, db = None, None, None
264     #####
265     # TODO: Implement the convolutional backward pass. #
266     #####
267     x,w,b,conv_param = cache
268     N,C1,H,W = x.shape

```

```

269 F,C2,HH,WW = w.shape
270 N,F,H_mark,W_mark = dout.shape
271
272 stride = conv_param['stride']
273 pad = conv_param['pad']
274
275 dx = np.zeros(x.shape)
276 dw = np.zeros(w.shape)
277 db = np.zeros(b.shape)
278
279 for n in xrange(N):
280     dx_pad = np.pad(dx[n,:,:,:], ((0,0),(pad,pad),(pad,pad)), 'constant')
281     x_pad = np.pad(x[n,:,:,:], ((0,0),(pad,pad),(pad,pad)), 'constant')
282     for f in xrange(F):
283         for h_mark in xrange(H_mark):
284             for w_mark in xrange(W_mark):
285                 h1 = h_mark * stride
286                 h2 = h_mark * stride + HH
287                 w1 = w_mark * stride
288                 w2 = w_mark * stride + WW
289                 dx_pad[:, h1:h2, w1:w2] += w[f,:,:,:] * dout[n,f,h_mark,w_mark]
290                 dw[f,:,:,:] += x_pad[:, h1:h2, w1:w2] * dout[n,f,h_mark,w_mark]
291                 db[f] += dout[n,f,h_mark,w_mark]
292     dx[n,:,:,:] = dx_pad[:,1:-1,1:-1]
293 #####
294 #                                     END OF YOUR CODE                                     #
295 #####
296 return dx, dw, db
297
298
299 def max_pool_forward_naive(x, pool_param):
300     """
301     A naive implementation of the forward pass for a max pooling layer.
302
303     Inputs:
304     - x: Input data, of shape (N, C, H, W)
305     - pool_param: dictionary with the following keys:
306       - 'pool_height': The height of each pooling region
307       - 'pool_width': The width of each pooling region
308       - 'stride': The distance between adjacent pooling regions
309
310     Returns a tuple of:
311     - out: Output data
312     - cache: (x, pool_param)
313     """
314     out = None
315     #####
316     # TODO: Implement the max pooling forward pass                                     #
317     #####
318     N, C, H, W = x.shape
319     pool_height = pool_param['pool_height']
320     pool_width = pool_param['pool_width']
321     stride = pool_param['stride']
322
323     H_mark = 1 + (H - pool_height) / stride
324     W_mark = 1 + (W - pool_width) / stride
325
326     out = np.zeros((N, C, H_mark, W_mark))
327
328     for n in xrange(N):
329         for h in xrange(H_mark):
330             for w in xrange(W_mark):
331                 h1 = h * stride
332                 h2 = h * stride + pool_height
333                 w1 = w * stride
334                 w2 = w * stride + pool_width
335                 window = x[n, :, h1:h2, w1:w2]

```

```

336         out[n,:,h,w] = np.max(window.reshape((C, pool_height*pool_width)), axis=1)
337     #####
338     #                                     END OF YOUR CODE                                     #
339     #####
340     cache = (x, pool_param)
341     return out, cache
342
343
344 def max_pool_backward_naive(dout, cache):
345     """
346     A naive implementation of the backward pass for a max pooling layer.
347
348     Inputs:
349     - dout: Upstream derivatives
350     - cache: A tuple of (x, pool_param) as in the forward pass.
351
352     Returns:
353     - dx: Gradient with respect to x
354     """
355     dx = None
356     #####
357     # TODO: Implement the max pooling backward pass                                     #
358     #####
359     x, pool_param = cache
360     N, C, H, W = x.shape
361
362     pool_height = pool_param['pool_height']
363     pool_width = pool_param['pool_width']
364     stride = pool_param['stride']
365
366     H_mark = 1 + (H - pool_height) / stride
367     W_mark = 1 + (W - pool_width) / stride
368
369     dx = np.zeros(x.shape)
370
371     for n in xrange(N):
372         for c in xrange(C):
373             for h in xrange(H_mark):
374                 for w in xrange(W_mark):
375                     h1 = h * stride
376                     h2 = h * stride + pool_height
377                     w1 = w * stride
378                     w2 = w * stride + pool_width
379
380                     window = x[n, c, h1:h2, w1:w2]
381                     window2 = np.reshape(window, (pool_height*pool_width))
382                     window3 = np.zeros_like(window2)
383                     window3[np.argmax(window2)] = 1
384
385                     dx[n,c,h1:h2,w1:w2] = np.reshape(window3, (pool_height, pool_width)) *
386                     dout[n,c,h,w]
387     #####
388     #                                     END OF YOUR CODE                                     #
389     #####
390     return dx
391
392 def svm_loss(x, y):
393     """
394     Computes the loss and gradient using for multiclass SVM classification.
395
396     Inputs:
397     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
398         for the ith input.
399     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
400         0 <= y[i] < C
401

```

```

402     Returns a tuple of:
403     - loss: Scalar giving the loss
404     - dx: Gradient of the loss with respect to x
405     """
406     N = x.shape[0]
407     correct_class_scores = x[np.arange(N), y]
408     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
409     margins[np.arange(N), y] = 0
410     loss = np.sum(margins) / N
411     num_pos = np.sum(margins > 0, axis=1)
412     dx = np.zeros_like(x)
413     dx[margins > 0] = 1
414     dx[np.arange(N), y] -= num_pos
415     dx /= N
416     return loss, dx
417
418
419 def softmax_loss(x, y):
420     """
421     Computes the loss and gradient for softmax classification.
422
423     Inputs:
424     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
425         for the ith input.
426     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
427         0 <= y[i] < C
428
429     Returns a tuple of:
430     - loss: Scalar giving the loss
431     - dx: Gradient of the loss with respect to x
432     """
433     probs = np.exp(x - np.max(x, axis=1, keepdims=True))
434     probs /= np.sum(probs, axis=1, keepdims=True)
435     N = x.shape[0]
436     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
437     dx = probs.copy()
438     dx[np.arange(N), y] -= 1
439     dx /= N
440     return loss, dx
441
442

```