

layers.py

```

1  import numpy as np
2  import pdb
3
4  """
5  This code was originally written for CS 231n at Stanford University
6  (cs231n.stanford.edu). It has been modified in various areas for use in the
7  ECE 239AS class at UCLA. This includes the descriptions of what code to
8  implement as well as some slight potential changes in variable names to be
9  consistent with class nomenclature. We thank Justin Johnson & Serena Yeung for
10 permission to use this code. To see the original version, please visit
11 cs231n.stanford.edu.
12 """
13
14
15 def affine_forward(x, w, b):
16     """
17     Computes the forward pass for an affine (fully-connected) layer.
18
19     The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
20     examples, where each example x[i] has shape (d_1, ..., d_k). We will
21     reshape each input into a vector of dimension D = d_1 * ... * d_k, and
22     then transform it to an output vector of dimension M.
23
24     Inputs:
25     - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
26     - w: A numpy array of weights, of shape (D, M)
27     - b: A numpy array of biases, of shape (M,)
28
29     Returns a tuple of:
30     - out: output, of shape (N, M)
31     - cache: (x, w, b)
32     """
33
34     # ===== #
35     # YOUR CODE HERE:
36     # Calculate the output of the forward pass. Notice the dimensions
37     # of w are D x M, which is the transpose of what we did in earlier
38     # assignments.
39     # ===== #
40     out = x.reshape(x.shape[0], -1).dot(w) + b
41
42     # ===== #
43     # END YOUR CODE HERE
44     # ===== #
45
46     cache = (x, w, b)
47     return out, cache
48
49
50 def affine_backward(dout, cache):
51     """
52     Computes the backward pass for an affine layer.
53
54     Inputs:
55     - dout: Upstream derivative, of shape (N, M)
56     - cache: Tuple of:
57       - x: Input data, of shape (N, d_1, ... d_k)
58       - w: Weights, of shape (D, M)
59
60     Returns a tuple of:
61     - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
62     - dw: Gradient with respect to w, of shape (D, M)
63     - db: Gradient with respect to b, of shape (M,)
64     """
65     x, w, b = cache
66     dx, dw, db = None, None, None
67
68     # ===== #
69     # YOUR CODE HERE:
70     # Calculate the gradients for the backward pass.
71     # ===== #
72
73     # dout is N x M
74     # dx should be N x d1 x ... x dk; it relates to dout through multiplication with w, which is D x M

```

```

75 # dw should be D x M; it relates to dout through multiplication with x, which is N x D after reshaping
76 # db should be M; it is just the sum over dout examples
77
78 db = np.sum(dout, axis=0)
79 dx = np.array(dout).dot(w.T).reshape(x.shape)
80 dw = x.reshape(x.shape[0], -1).T.dot(dout)
81 # ===== #
82 # END YOUR CODE HERE
83 # ===== #
84
85 return dx, dw, db
86
87
88 def relu_forward(x):
89     """
90     Computes the forward pass for a layer of rectified linear units (ReLU).
91
92     Input:
93     - x: Inputs, of any shape
94
95     Returns a tuple of:
96     - out: Output, of the same shape as x
97     - cache: x
98     """
99     # ===== #
100    # YOUR CODE HERE:
101    # Implement the ReLU forward pass.
102    # ===== #
103
104    out = x * (x > 0)
105    # ===== #
106    # END YOUR CODE HERE
107    # ===== #
108
109    cache = x
110    return out, cache
111
112
113 def relu_backward(dout, cache):
114     """
115     Computes the backward pass for a layer of rectified linear units (ReLU).
116
117     Input:
118     - dout: Upstream derivatives, of any shape
119     - cache: Input x, of same shape as dout
120
121     Returns:
122     - dx: Gradient with respect to x
123     """
124     x = cache
125
126     # ===== #
127     # YOUR CODE HERE:
128     # Implement the ReLU backward pass
129     # ===== #
130
131     # ReLU directs linearly to those > 0
132     dx = dout * (x > 0)
133
134     # ===== #
135     # END YOUR CODE HERE
136     # ===== #
137
138     return dx
139
140 def batchnorm_forward(x, gamma, beta, bn_param):
141     """
142     Forward pass for batch normalization.
143
144     During training the sample mean and (uncorrected) sample variance are
145     computed from minibatch statistics and used to normalize the incoming data.
146     During training we also keep an exponentially decaying running mean of the mean
147     and variance of each feature, and these averages are used to normalize data
148     at test-time.
149
150     At each timestep we update the running averages for mean and variance using
151     an exponential decay based on the momentum parameter:

```

```

152
153 running_mean = momentum * running_mean + (1 - momentum) * sample_mean
154 running_var = momentum * running_var + (1 - momentum) * sample_var
155
156 Note that the batch normalization paper suggests a different test-time
157 behavior: they compute sample mean and variance for each feature using a
158 large number of training images rather than using a running average. For
159 this implementation we have chosen to use running averages instead since
160 they do not require an additional estimation step; the torch7 implementation
161 of batch normalization also uses running averages.
162

```

```

163 Input:

```

```

164 - x: Data of shape (N, D)
165 - gamma: Scale parameter of shape (D,)
166 - beta: Shift parameter of shape (D,)
167 - bn_param: Dictionary with the following keys:
168   - mode: 'train' or 'test'; required
169   - eps: Constant for numeric stability
170   - momentum: Constant for running mean / variance.
171   - running_mean: Array of shape (D,) giving running mean of features
172   - running_var: Array of shape (D,) giving running variance of features
173

```

```

174 Returns a tuple of:

```

```

175 - out: of shape (N, D)
176 - cache: A tuple of values needed in the backward pass
177 """

```

```

178 mode = bn_param['mode']
179 eps = bn_param.get('eps', 1e-5)
180 momentum = bn_param.get('momentum', 0.9)
181

```

```

182 N, D = x.shape

```

```

183 running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
184 running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
185

```

```

186 out, cache = None, None

```

```

187 if mode == 'train':

```

```

188     # ===== #
189     # YOUR CODE HERE:
190     #   A few steps here:
191     #   (1) Calculate the running mean and variance of the minibatch.
192     #   (2) Normalize the activations with the running mean and variance.
193     #   (3) Scale and shift the normalized activations. Store this
194     #       as the variable 'out'
195     #   (4) Store any variables you may need for the backward pass in
196     #       the 'cache' variable.
197     # ===== #

```

```

198
199
200 running_mean = np.mean(x, axis=0)
201 running_var = np.var(x, axis=0)
202 x_hat = (x - running_mean) / np.sqrt(eps + running_var)
203 out = gamma * x_hat + beta
204 cache = (x_hat, x, running_mean, running_var, eps, gamma)
205

```

```

206     # ===== #
207     # END YOUR CODE HERE
208     # ===== #

```

```

209 elif mode == 'test':

```

```

210     # ===== #
211     # YOUR CODE HERE:
212     #   Calculate the testing time normalized activation. Normalize using
213     #   the running mean and variance, and then scale and shift appropriately.
214     #   Store the output as 'out'.
215     # ===== #

```

```

216
217 x_hat = (x - running_mean) / np.sqrt(eps + running_var)
218 out = gamma * x_hat + beta
219

```

```

220     # ===== #
221     # END YOUR CODE HERE
222     # ===== #

```

```

223 else:

```

```

224     raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
225

```

```

226 # Store the updated running means back into bn_param

```

```

227 bn_param['running_mean'] = running_mean

```

```

228 bn_param['running_var'] = running_var

```

```

229
230     return out, cache
231
232 def batchnorm_backward(dout, cache):
233     """
234     Backward pass for batch normalization.
235
236     For this implementation, you should write out a computation graph for
237     batch normalization on paper and propagate gradients backward through
238     intermediate nodes.
239
240     Inputs:
241     - dout: Upstream derivatives, of shape (N, D)
242     - cache: Variable of intermediates from batchnorm_forward.
243
244     Returns a tuple of:
245     - dx: Gradient with respect to inputs x, of shape (N, D)
246     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
247     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
248     """
249     dx, dgamma, dbeta = None, None, None
250
251     # ===== #
252     # YOUR CODE HERE:
253     # Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
254     # ===== #
255     x_hat, x, mean, var, eps, gamma = cache
256     N, D = x.shape
257     dbeta = np.sum(dout, axis=0)
258     dgamma = np.sum(dout * x_hat, axis=0)
259     dl_dxhat = dout * gamma
260     dl_dvar = (-1/2) * np.sum((1/((var + eps) ** (3/2))) * (x - mean) * dl_dxhat, axis=0)
261     dl_dmean = (-1/(np.sqrt(var+eps))) * np.sum(dl_dxhat, axis=0)
262     dx = np.array((1/np.sqrt(var + eps)) * dl_dxhat + (2 * (x - mean) / N) * dl_dvar + (1/N) * dl_dmean)
263     # ===== #
264     # END YOUR CODE HERE
265     # ===== #
266
267     return dx, dgamma, dbeta
268
269 def dropout_forward(x, dropout_param):
270     """
271     Performs the forward pass for (inverted) dropout.
272
273     Inputs:
274     - x: Input data, of any shape
275     - dropout_param: A dictionary with the following keys:
276       - p: Dropout parameter. We keep each neuron output with probability p.
277       - mode: 'test' or 'train'. If the mode is train, then perform dropout;
278         if the mode is test, then just return the input.
279       - seed: Seed for the random number generator. Passing seed makes this
280         function deterministic, which is needed for gradient checking but not in
281         real networks.
282
283     Outputs:
284     - out: Array of the same shape as x.
285     - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
286       mask that was used to multiply the input; in test mode, mask is None.
287     """
288     p, mode = dropout_param['p'], dropout_param['mode']
289     if 'seed' in dropout_param:
290         np.random.seed(dropout_param['seed'])
291
292     mask = None
293     out = None
294
295     if mode == 'train':
296         # ===== #
297         # YOUR CODE HERE:
298         # Implement the inverted dropout forward pass during training time.
299         # Store the masked and scaled activations in out, and store the
300         # dropout mask as the variable mask.
301         # ===== #
302
303         mask = np.random.rand(*x.shape) < p
304         out = x * mask / p
305

```

```

306 # ===== #
307 # END YOUR CODE HERE
308 # ===== #
309
310 elif mode == 'test':
311
312 # ===== #
313 # YOUR CODE HERE:
314 # Implement the inverted dropout forward pass during test time.
315 # ===== #
316
317
318 out = x
319 # ===== #
320 # END YOUR CODE HERE
321 # ===== #
322
323 cache = (dropout_param, mask)
324 out = out.astype(x.dtype, copy=False)
325
326 return out, cache
327
328 def dropout_backward(dout, cache):
329 """
330 Perform the backward pass for (inverted) dropout.
331
332 Inputs:
333 - dout: Upstream derivatives, of any shape
334 - cache: (dropout_param, mask) from dropout_forward.
335 """
336 dropout_param, mask = cache
337 mode = dropout_param['mode']
338
339 dx = None
340 if mode == 'train':
341 # ===== #
342 # YOUR CODE HERE:
343 # Implement the inverted dropout backward pass during training time.
344 # ===== #
345
346
347 dx = dout * mask / dropout_param['p']
348 # ===== #
349 # END YOUR CODE HERE
350 # ===== #
351 elif mode == 'test':
352 # ===== #
353 # YOUR CODE HERE:
354 # Implement the inverted dropout backward pass during test time.
355 # ===== #
356
357
358 dx = dout
359 # ===== #
360 # END YOUR CODE HERE
361 # ===== #
362 return dx
363
364 def svm_loss(x, y):
365 """
366 Computes the loss and gradient using for multiclass SVM classification.
367
368 Inputs:
369 - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
370 for the ith input.
371 - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
372 0 <= y[i] < C
373
374 Returns a tuple of:
375 - loss: Scalar giving the loss
376 - dx: Gradient of the loss with respect to x
377 """
378 N = x.shape[0]
379 correct_class_scores = x[np.arange(N), y]
380 margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
381 margins[np.arange(N), y] = 0
382 loss = np.sum(margins) / N

```

```
383     num_pos = np.sum(margins > 0, axis=1)
384     dx = np.zeros_like(x)
385     dx[margins > 0] = 1
386     dx[np.arange(N), y] -= num_pos
387     dx /= N
388     return loss, dx
389
390
391 def softmax_loss(x, y):
392     """
393     Computes the loss and gradient for softmax classification.
394
395     Inputs:
396     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
397       for the ith input.
398     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
399       0 <= y[i] < C
400
401     Returns a tuple of:
402     - loss: Scalar giving the loss
403     - dx: Gradient of the loss with respect to x
404     """
405
406     probs = np.exp(x - np.max(x, axis=1, keepdims=True))
407     probs /= np.sum(probs, axis=1, keepdims=True)
408     N = x.shape[0]
409     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
410     dx = probs.copy()
411     dx[np.arange(N), y] -= 1
412     dx /= N
413     return loss, dx
414
```