```
1
    import numpy as np
2
    import pdb
3
4
5
    def affine forward(x, w, b):
6
7
       Computes the forward pass for an affine (fully-connected) layer.
8
9
       The input x has shape (N, d 1, ..., d k) and contains a minibatch of N
       examples, where each example x[i] has shape (d_1, \ldots, d_k). We will
10
11
       reshape each input into a vector of dimension D = d 1 * ... * d k, and
12
       then transform it to an output vector of dimension M.
13
14
       Inputs:
15
       - x: A numpy array containing input data, of shape (N, d 1, ..., d k)
16
       - w: A numpy array of weights, of shape (D, M)
17
       - b: A numpy array of biases, of shape (M,)
18
19
       Returns a tuple of:
20
       - out: output, of shape (N, M)
21
       - cache: (x, w, b)
22
23
       out = None
24
       25
       # YOUR CODE HERE:
       # Calculate the output of the forward pass. Notice the dimensions
26
27
         of w are D x M, which is the transpose of what we did in earlier
28
       # assignments.
29
       # ----- #
30
       out = np.dot(x.reshape(x.shape[0],-1),w) + b
31
32
33
       34
       # END YOUR CODE HERE
35
       # ----- #
36
37
       cache = (x, w, b)
38
       return out, cache
39
40
41
   def affine backward(dout, cache):
42
43
       Computes the backward pass for an affine layer.
44
45
       Inputs:
46
       - dout: Upstream derivative, of shape (N, M)
47
       - cache: Tuple of:
48
         - x: A numpy array containing input data, of shape (N, d_1, \ldots, d_k)
49
         - w: A numpy array of weights, of shape (D, M)
50
        - b: A numpy array of biases, of shape (M,)
51
52
       Returns a tuple of:
53
       - dx: Gradient with respect to x, of shape (N, d1, ..., d k)
54
       - dw: Gradient with respect to w, of shape (D, M)
55
       - db: Gradient with respect to b, of shape (M,)
       .....
56
57
       x, w, b = cache
58
       dx, dw, db = None, None, None
59
60
       # ========= #
       # YOUR CODE HERE:
61
62
          Calculate the gradients for the backward pass.
       # Notice:
63
64
       # dout is N x M
65
       # dx should be N x d1 x ... x dk; it relates to dout through multiplication with
       w, which is D x M
66
         dw should be D x M; it relates to dout through multiplication with x, which is N
```

```
x D after reshaping
67
         db should be M; it is just the sum over dout examples
68
       # ========= #
69
       flattened x = x.reshape(x.shape[0],-1)
       dx = np.dot(dout, w.T).reshape(x.shape)
71
       dw = np.dot(flattened x.T,dout)
72
       db = np.sum(dout, axis = 0)
73
74
       # ====================== #
75
       # END YOUR CODE HERE
76
       77
78
       return dx, dw, db
79
80
   def relu forward(x):
81
       Computes the forward pass for a layer of rectified linear units (ReLUs).
82
83
84
       Input:
85
       - x: Inputs, of any shape
86
87
       Returns a tuple of:
88
       - out: Output, of the same shape as x
       - cache: x
89
       ......
90
91
       # ============ #
92
       # YOUR CODE HERE:
93
       # Implement the ReLU forward pass.
94
      # ----- #
95
      relu = lambda x: x * (x > 0)
96
       out = relu(x)
97
       # ============= #
98
       # END YOUR CODE HERE
99
       # ----- #
100
101
       cache = x
102
       return out, cache
103
104
105
    def relu backward(dout, cache):
       11 11 11
106
107
       Computes the backward pass for a layer of rectified linear units (ReLUs).
108
109
110
       - dout: Upstream derivatives, of any shape
111
       - cache: Input x, of same shape as dout
112
113
      Returns:
114
       - dx: Gradient with respect to x
       11 11 11
115
116
       x = cache
117
       # ----- #
118
119
       # YOUR CODE HERE:
120
         Implement the ReLU backward pass
121
       # ----- #
122
       dx = dout * (x.reshape(x.shape[0],-1) > 0)
123
124
       # =========== #
125
       # END YOUR CODE HERE
126
       # =========== #
127
128
       return dx
129
130
   def batchnorm forward(x, gamma, beta, bn param):
131
132
       Forward pass for batch normalization.
```

```
133
134
         During training the sample mean and (uncorrected) sample variance are
135
         computed from minibatch statistics and used to normalize the incoming data.
136
         During training we also keep an exponentially decaying running mean of the mean
137
         and variance of each feature, and these averages are used to normalize data
138
         at test-time.
139
140
         At each timestep we update the running averages for mean and variance using
141
         an exponential decay based on the momentum parameter:
142
143
         running mean = momentum * running mean + (1 - momentum) * sample mean
144
         running var = momentum * running var + (1 - momentum) * sample var
145
146
         Note that the batch normalization paper suggests a different test-time
147
         behavior: they compute sample mean and variance for each feature using a
148
         large number of training images rather than using a running average. For
         this implementation we have chosen to use running averages instead since
149
150
         they do not require an additional estimation step; the torch7 implementation
         of batch normalization also uses running averages.
151
152
153
         Input:
154
         - x: Data of shape (N, D)
         - gamma: Scale parameter of shape (D,)
155
156
         - beta: Shift paremeter of shape (D,)
157
         - bn param: Dictionary with the following keys:
158
           - mode: 'train' or 'test'; required
159
           - eps: Constant for numeric stability
160
          - momentum: Constant for running mean / variance.
161
          - running mean: Array of shape (D,) giving running mean of features
162
          - running_var Array of shape (D,) giving running variance of features
163
164
         Returns a tuple of:
165
         - out: of shape (N, D)
166
         - cache: A tuple of values needed in the backward pass
167
168
         mode = bn param['mode']
169
         eps = bn param.get('eps', 1e-5)
170
         momentum = bn param.get('momentum', 0.9)
171
172
         N, D = x.shape
173
         running mean = bn param.get('running mean', np.zeros(D, dtype=x.dtype))
174
         running var = bn param.get('running var', np.zeros(D, dtype=x.dtype))
175
176
         out, cache = None, None
177
         if mode == 'train':
178
             179
             # YOUR CODE HERE:
180
181
               A few steps here:
182
                   (1) Calculate the running mean and variance of the minibatch.
183
                   (2) Normalize the activations with the running mean and variance.
184
                   (3) Scale and shift the normalized activations. Store this
                      as the variable 'out'
185
186
             #
                   (4) Store any variables you may need for the backward pass in
187
                      the 'cache' variable.
188
             # ----- #
189
            mean x = np.mean(x,axis = 0)
190
            var x = np.var(x, axis = 0)
191
192
            running mean = momentum * running mean + ( 1 - momentum ) * mean x
             running var = momentum * running var + ( 1 - momentum ) * var x
193
194
195
            standard x = (x - mean x) / (np.sqrt(var x + eps))
196
197
             out = gamma * standard x + beta
             cache = (mean_x, var_x, standard x, gamma, x, eps)
198
199
```

```
201
           # END YOUR CODE HERE
202
           203
        elif mode == 'test':
204
           # ----- #
205
           # YOUR CODE HERE:
206
           # Calculate the testing time normalized activation. Normalize using
207
           # the running mean and variance, and then scale and shift appropriately.
           # Store the output as 'out'.
208
           # ----- #
209
210
211
           standard x = (x - running mean) / (np.sqrt(running var))
212
           out = gamma * standard x + beta
213
           \#cache = []
           # ----- #
214
215
           # END YOUR CODE HERE
216
           217
       else:
           raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
218
219
220
        # Store the updated running means back into bn param
221
        bn param['running mean'] = running mean
222
        bn param['running var'] = running var
223
224
        return out, cache
225
226
   def batchnorm backward(dout, cache):
227
228
        Backward pass for batch normalization.
229
230
        For this implementation, you should write out a computation graph for
231
        batch normalization on paper and propagate gradients backward through
232
        intermediate nodes.
233
234
       Inputs:
235
        - dout: Upstream derivatives, of shape (N, D)
        - cache: Variable of intermediates from batchnorm forward.
236
237
238
       Returns a tuple of:
239
        - dx: Gradient with respect to inputs x, of shape (N, D)
        - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
240
241
        - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
242
243
        dx, dgamma, dbeta = None, None, None
244
245
        # ================== #
246
        # YOUR CODE HERE:
247
        # Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
248
        # ============ #
249
        (mean x, var x, standard x, gamma, x, eps) = cache
250
        sample size = x.shape[0]
251
        sigma x = np.sqrt(var x + eps)
252
253
        dgamma = np.sum(standard x * dout,axis = 0)
254
        dbeta = np.sum(dout, axis = 0)
255
256
        dL dx st = dout * gamma
257
258
        dx st da = 1 / sigma x
259
        dL da = dx st da * dL dx st
260
        da dx = 1
261
        dx_st_de = -0.5 * (dx_st_da ** 3) * (x - mean x)
262
        dL de = dx st de * dL dx st
263
264
265
        dL dvar = np.sum(dL de, axis = 0)
266
        dvar dx = (2 * (x - mean x)) / sample size
```

```
268
        dL dmean = np.sum(-dL da, axis = 0)
269
       dmean dx = 1 / sample size
270
271
       dx = da dx * dL da + dvar dx * dL dvar + dmean dx * dL dmean
272
        # ______ #
273
        # END YOUR CODE HERE
274
        # ----- #
275
276
       return dx, dgamma, dbeta
277
278
    def dropout forward(x, dropout param):
279
280
       Performs the forward pass for (inverted) dropout.
281
282
       Inputs:
283
        - x: Input data, of any shape
284
       - dropout param: A dictionary with the following keys:
         - p: Dropout parameter. We keep each neuron output with probability p.
285
         - mode: 'test' or 'train'. If the mode is train, then perform dropout;
286
287
          if the mode is test, then just return the input.
288
         - seed: Seed for the random number generator. Passing seed makes this
289
          function deterministic, which is needed for gradient checking but not in
290
           real networks.
291
292
       Outputs:
293
       - out: Array of the same shape as x.
294
       - cache: A tuple (dropout param, mask). In training mode, mask is the dropout
295
        mask that was used to multiply the input; in test mode, mask is None.
296
297
       p, mode = dropout_param['p'], dropout_param['mode']
       assert (0<p<=1), "Dropout probability is not in (0,1]"</pre>
298
299
       if 'seed' in dropout param:
300
           np.random.seed(dropout param['seed'])
301
302
       mask = None
303
       out = None
304
305
       if mode == 'train':
306
           # ================= #
307
           # YOUR CODE HERE:
308
             Implement the inverted dropout forward pass during training time.
309
             Store the masked and scaled activations in out, and store the
310
            dropout mask as the variable mask.
311
           # ================= #
312
313
          mask = (np.random.rand(x.shape[0],x.shape[1]) < p) / p
314
          out = x * mask
315
           # ----- #
316
           # END YOUR CODE HERE
317
           318
319
       elif mode == 'test':
320
321
           # ----- #
322
           # YOUR CODE HERE:
323
            Implement the inverted dropout forward pass during test time.
           # ----- #
324
325
326
          out = x
327
328
           329
           # END YOUR CODE HERE
           # ----- #
330
3.31
332
       cache = (dropout param, mask)
       out = out.astype(x.dtype, copy=False)
333
```

```
335
        return out, cache
336
    def dropout backward(dout, cache):
337
338
339
        Perform the backward pass for (inverted) dropout.
340
341
        Inputs:
342
        - dout: Upstream derivatives, of any shape
343
        - cache: (dropout param, mask) from dropout forward.
        ......
344
345
        dropout param, mask = cache
346
       mode = dropout param['mode']
347
348
       dx = None
349
       if mode == 'train':
350
           # ------ #
351
           # YOUR CODE HERE:
352
             Implement the inverted dropout backward pass during training time.
           # ----- #
353
354
355
           dx = dout * mask
356
357
           358
           # END YOUR CODE HERE
359
           360
        elif mode == 'test':
361
           # ================== #
362
           # YOUR CODE HERE:
363
             Implement the inverted dropout backward pass during test time.
364
           # =================== #
365
366
           dx = dout
367
368
           # ================ #
369
           # END YOUR CODE HERE
370
           371
        return dx
372
373
    def svm loss(x, y):
        .....
374
375
        Computes the loss and gradient using for multiclass SVM classification.
376
377
378
       - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
379
         for the ith input.
380
       - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
381
         0 <= y[i] < C
382
383
       Returns a tuple of:
384
        - loss: Scalar giving the loss
385
        - dx: Gradient of the loss with respect to x
       11 11 11
386
387
       N = x.shape[0]
388
        correct class scores = x[np.arange(N), y]
389
       margins = np.maximum(0, x - correct class scores[:, np.newaxis] + 1.0)
390
       margins[np.arange(N), y] = 0
391
        loss = np.sum(margins) / N
392
       num pos = np.sum(margins > 0, axis=1)
393
        dx = np.zeros like(x)
394
        dx[margins > 0] = 1
        dx[np.arange(N), y] -= num_pos
395
396
        dx /= N
397
        return loss, dx
398
399
400
    def softmax loss(x, y):
```

```
11 11 11
401
402
         Computes the loss and gradient for softmax classification.
403
         Inputs:
404
405
         - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
406
           for the ith input.
407
         - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
408
           0 \le y[i] < C
409
410
         Returns a tuple of:
411
          - loss: Scalar giving the loss
412
          - dx: Gradient of the loss with respect to x
413
         11 11 11
414
415
         probs = np.exp(x - np.max(x, axis=1, keepdims=True))
416
         probs /= np.sum(probs, axis=1, keepdims=True)
417
         N = x.shape[0]
418
         loss = -np.sum(np.log(np.maximum(probs[np.arange(N), y], 1e-8))) / N
419
         dx = probs.copy()
         dx[np.arange(N), y] -= 1
420
421
         dx /= N
422
         return loss, dx
```