

```

1 import numpy as np
2 import pdb
3
4 def affine_forward(x, w, b):
5     """
6     Computes the forward pass for an affine (fully-connected) layer.
7
8     The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
9     examples, where each example x[i] has shape (d_1, ..., d_k). We will
10    reshape each input into a vector of dimension D = d_1 * ... * d_k, and
11    then transform it to an output vector of dimension M.
12
13    Inputs:
14    - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
15    - w: A numpy array of weights, of shape (D, M)
16    - b: A numpy array of biases, of shape (M,)
17
18    Returns a tuple of:
19    - out: output, of shape (N, M)
20    - cache: (x, w, b)
21    """
22
23    # ===== #
24    # YOUR CODE HERE:
25    # Calculate the output of the forward pass. Notice the dimensions
26    # of w are D x M, which is the transpose of what we did in earlier
27    # assignments.
28    # ===== #
29
30    x_reshape = x.reshape((x.shape[0], w.shape[0])) #N x D
31    out = np.dot(x_reshape, w) + b.reshape((1, b.shape[0])) #N x M
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: Input data, of shape (N, d_1, ... d_k)
49       - w: Weights, of shape (D, M)
50
51     Returns a tuple of:
52     - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
53     - dw: Gradient with respect to w, of shape (D, M)
54     - db: Gradient with respect to b, of shape (M,)
55     """
56     x, w, b = cache
57     dx, dw, db = None, None, None
58
59    # ===== #

```

```

60 # YOUR CODE HERE:
61 #   Calculate the gradients for the backward pass.
62 # ===== #
63
64 x_reshape = np.reshape(x, (x.shape[0], w.shape[0])) #N x D
65 dx_reshape = np.dot(dout, w.T)
66
67 dx = np.reshape(dx_reshape, x.shape) #N x D
68 dw = np.dot(x_reshape.T, dout) #D x M
69 db = np.dot(dout.T, np.ones(x.shape[0])) #M x 1
70
71 # ===== #
72 # END YOUR CODE HERE
73 # ===== #
74
75 return dx, dw, db
76
77 def relu_forward(x):
78     """
79     Computes the forward pass for a layer of rectified linear units (ReLUs).
80
81     Input:
82     - x: Inputs, of any shape
83
84     Returns a tuple of:
85     - out: Output, of the same shape as x
86     - cache: x
87     """
88     # ===== #
89     # YOUR CODE HERE:
90     #   Implement the ReLU forward pass.
91     # ===== #
92
93     out = np.maximum(x, 0)
94
95     # ===== #
96     # END YOUR CODE HERE
97     # ===== #
98
99     cache = x
100    return out, cache
101
102
103 def relu_backward(dout, cache):
104     """
105     Computes the backward pass for a layer of rectified linear units (ReLUs).
106
107     Input:
108     - dout: Upstream derivatives, of any shape
109     - cache: Input x, of same shape as dout
110
111     Returns:
112     - dx: Gradient with respect to x
113     """
114     x = cache
115
116     # ===== #
117     # YOUR CODE HERE:
118     #   Implement the ReLU backward pass
119     # ===== #

```

```

120
121     dx = dout * (x > 0)
122
123     # ===== #
124     # END YOUR CODE HERE
125     # ===== #
126
127     return dx
128
129 def batchnorm_forward(x, gamma, beta, bn_param):
130     """
131     Forward pass for batch normalization.
132
133     During training the sample mean and (uncorrected) sample variance are
134     computed from minibatch statistics and used to normalize the incoming data.
135     During training we also keep an exponentially decaying running mean of the
136     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
149     this implementation we have chosen to use running averages instead since
150     they do not require an additional estimation step; the torch7
151     implementation
152     of batch normalization also uses running averages.
153
154     Input:
155     - x: Data of shape (N, D)
156     - gamma: Scale parameter of shape (D,)
157     - beta: Shift parameter of shape (D,)
158     - bn_param: Dictionary with the following keys:
159       - mode: 'train' or 'test'; required
160       - eps: Constant for numeric stability
161       - momentum: Constant for running mean / variance.
162       - running_mean: Array of shape (D,) giving running mean of features
163       - running_var: Array of shape (D,) giving running variance of features
164
165     Returns a tuple of:
166     - out: of shape (N, D)
167     - cache: A tuple of values needed in the backward pass
168     """
169     mode = bn_param['mode']
170     eps = bn_param.get('eps', 1e-5)
171     momentum = bn_param.get('momentum', 0.9)
172
173     N, D = x.shape
174     running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
175     running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
176
177     out, cache = None, None
178     if mode == 'train':

```

```

178 # ===== #
179 # YOUR CODE HERE:
180 #   A few steps here:
181 #   (1) Calculate the running mean and variance of the minibatch.
182 #   (2) Normalize the activations with the running mean and variance.
183 #   (3) Scale and shift the normalized activations. Store this
184 #       as the variable 'out'
185 #   (4) Store any variables you may need for the backward pass in
186 #       the 'cache' variable.
187 # ===== #
188
189 mean = np.mean(x, axis = 0)
190 var = np.var(x, axis = 0)
191 normalize_x = (x - mean) / np.sqrt(var + eps)
192
193 running_mean = momentum * running_mean + (1 - momentum) * mean
194 running_var = momentum * running_var + (1 - momentum) * var
195
196 out = gamma * normalize_x + beta
197
198 cache = {'normalize_x': normalize_x,
199         'x_minus_mean': (x - mean),
200         'sqrt_var_eps': np.sqrt(var + eps),
201         'gamma': gamma
202         }
203
204 # ===== #
205 # END YOUR CODE HERE
206 # ===== #
207
208 elif mode == 'test':
209
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
214     #   appropriately.
215     #   Store the output as 'out'.
216     # ===== #
217
218     normalize_x = (x - running_mean) / np.sqrt(running_var + eps)
219     out = gamma * normalize_x + beta
220
221     # ===== #
222     # END YOUR CODE HERE
223     # ===== #
224
225 else:
226     raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
227
228 # Store the updated running means back into bn_param
229 bn_param['running_mean'] = running_mean
230 bn_param['running_var'] = running_var
231
232 return out, cache
233
234 def batchnorm_backward(dout, cache):
235     """
236     Backward pass for batch normalization.

```

```

237 For this implementation, you should write out a computation graph for
238 batch normalization on paper and propagate gradients backward through
239 intermediate nodes.
240
241 Inputs:
242 - dout: Upstream derivatives, of shape (N, D)
243 - cache: Variable of intermediates from batchnorm_forward.
244
245 Returns a tuple of:
246 - dx: Gradient with respect to inputs x, of shape (N, D)
247 - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
248 - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
249 """
250 dx, dgamma, dbeta = None, None, None
251
252 # ===== #
253 # YOUR CODE HERE:
254 # Implement the batchnorm backward pass, calculating dx, dgamma, and
255 # dbeta.
256 # ===== #
257
258 normalize_x = cache.get('normalize_x')
259 x_minus_mean = cache.get('x_minus_mean')
260 sqrt_var_eps = cache.get('sqrt_var_eps')
261 gamma = cache.get('gamma')
262 N = dout.shape[0]
263
264 dbeta = np.sum(dout, axis = 0)
265 dgamma = np.sum(dout * normalize_x, axis = 0)
266
267 dnormalize_x = dout * gamma
268
269 db = x_minus_mean * dnormalize_x # b = 1 / sqrt_var_eps
270 dc = (-1 / (sqrt_var_eps * sqrt_var_eps)) * db # c = sqrt_var_eps
271 de = (1 / (2 * sqrt_var_eps)) * dc # e = sqrt_var_eps * sqrt_var_eps
272 dvar = np.sum(de, axis = 0)
273
274 da = dnormalize_x / sqrt_var_eps # a = x - mu
275 dm_u = -np.sum(da, axis = 0) - 2 * np.sum(x_minus_mean, axis = 0) * dvar / N
276
277 dx = da + 2 * x_minus_mean * dvar / N + dm_u / N
278
279 # ===== #
280 # END YOUR CODE HERE
281 # ===== #
282
283 return dx, dgamma, dbeta
284
285 def dropout_forward(x, dropout_param):
286     """
287     Performs the forward pass for (inverted) dropout.
288
289     Inputs:
290     - x: Input data, of any shape
291     - dropout_param: A dictionary with the following keys:
292         - p: Dropout parameter. We drop each neuron output with probability p.
293         - mode: 'test' or 'train'. If the mode is train, then perform dropout;
294           if the mode is test, then just return the input.
295         - seed: Seed for the random number generator. Passing seed makes this

```

```

295     function deterministic, which is needed for gradient checking but not
in
296     real networks.
297
298     Outputs:
299     - out: Array of the same shape as x.
300     - cache: A tuple (dropout_param, mask). In training mode, mask is the
dropout
301     mask that was used to multiply the input; in test mode, mask is None.
302     """
303     p, mode = dropout_param['p'], dropout_param['mode']
304     if 'seed' in dropout_param:
305         np.random.seed(dropout_param['seed'])
306
307     mask = None
308     out = None
309
310     if mode == 'train':
311         # ===== #
312         # YOUR CODE HERE:
313         #     Implement the inverted dropout forward pass during training time.
314         #     Store the masked and scaled activations in out, and store the
315         #     dropout mask as the variable mask.
316         # ===== #
317
318         mask = (np.random.rand(*x.shape) < (1 - p)) / (1 - p)
319         out = x * mask
320
321         # ===== #
322         # END YOUR CODE HERE
323         # ===== #
324
325     elif mode == 'test':
326
327         # ===== #
328         # YOUR CODE HERE:
329         #     Implement the inverted dropout forward pass during test time.
330         # ===== #
331
332         out = x
333
334         # ===== #
335         # END YOUR CODE HERE
336         # ===== #
337
338     cache = (dropout_param, mask)
339     out = out.astype(x.dtype, copy=False)
340
341     return out, cache
342
343 def dropout_backward(dout, cache):
344     """
345     Perform the backward pass for (inverted) dropout.
346
347     Inputs:
348     - dout: Upstream derivatives, of any shape
349     - cache: (dropout_param, mask) from dropout_forward.
350     """
351     dropout_param, mask = cache
352     mode = dropout_param['mode']

```

```

353
354 dx = None
355 if mode == 'train':
356     # ===== #
357     # YOUR CODE HERE:
358     # Implement the inverted dropout backward pass during training time.
359     # ===== #
360
361     dx = dout * mask
362
363     # ===== #
364     # END YOUR CODE HERE
365     # ===== #
366 elif mode == 'test':
367     # ===== #
368     # YOUR CODE HERE:
369     # Implement the inverted dropout backward pass during test time.
370     # ===== #
371
372     dx = dout
373
374     # ===== #
375     # END YOUR CODE HERE
376     # ===== #
377 return dx
378
379 def svm_loss(x, y):
380     """
381     Computes the loss and gradient using for multiclass SVM classification.
382
383     Inputs:
384     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
385       class
386       for the ith input.
387     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
388       0 <= y[i] < C
389
390     Returns a tuple of:
391     - loss: Scalar giving the loss
392     - dx: Gradient of the loss with respect to x
393     """
394     N = x.shape[0]
395     correct_class_scores = x[np.arange(N), y]
396     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
397     margins[np.arange(N), y] = 0
398     loss = np.sum(margins) / N
399     num_pos = np.sum(margins > 0, axis=1)
400     dx = np.zeros_like(x)
401     dx[margins > 0] = 1
402     dx[np.arange(N), y] -= num_pos
403     dx /= N
404     return loss, dx
405
406 def softmax_loss(x, y):
407     """
408     Computes the loss and gradient for softmax classification.
409
410     Inputs:

```

```
411     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
class
412     for the ith input.
413     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
414         0 <= y[i] < C
415
416     Returns a tuple of:
417     - loss: Scalar giving the loss
418     - dx: Gradient of the loss with respect to x
419     """
420
421     probs = np.exp(x - np.max(x, axis=1, keepdims=True))
422     probs /= np.sum(probs, axis=1, keepdims=True)
423     N = x.shape[0]
424     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
425     dx = probs.copy()
426     dx[np.arange(N), y] -= 1
427     dx /= N
428     return loss, dx
429
```