

```

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) @ 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: A numpy array containing input data, of shape (N, d_1, ..., d_k)
58       - w: A numpy array of weights, of shape (D, M)
59       - b: A numpy array of biases, of shape (M,)
60

```

```

61     Returns a tuple of:
62     - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
63     - dw: Gradient with respect to w, of shape (D, M)
64     - db: Gradient with respect to b, of shape (M,)
65     """
66     x, w, b = cache
67     dx, dw, db = None, None, None
68
69     # ===== #
70     # YOUR CODE HERE:
71     #   Calculate the gradients for the backward pass.
72     # Notice:
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     x_ND = x.reshape(x.shape[0], -1) # (N, D)
79     dx = (dout @ w.T).reshape(x.shape)
80     dw = x_ND.T @ dout
81     db = np.sum(dout, axis=0)
82     # ===== #
83     # END YOUR CODE HERE
84     # ===== #
85
86     return dx, dw, db
87
88
89 def relu_forward(x):
90     """
91     Computes the forward pass for a layer of rectified linear units (ReLU).
92
93     Input:
94     - x: Inputs, of any shape
95
96     Returns a tuple of:
97     - out: Output, of the same shape as x
98     - cache: x
99     """
100    # ===== #
101    # YOUR CODE HERE:
102    #   Implement the ReLU forward pass.
103    # ===== #
104    def relu(x): return x * (x > 0)
105    out = relu(x)
106    # ===== #
107    # END YOUR CODE HERE
108    # ===== #
109
110    cache = x
111    return out, cache
112
113
114 def relu_backward(dout, cache):
115     """
116     Computes the backward pass for a layer of rectified linear units (ReLU).
117
118     Input:

```

```

119 - dout: Upstream derivatives, of any shape
120 - cache: Input x, of same shape as dout
121
122 Returns:
123 - dx: Gradient with respect to x
124 """
125 x = cache
126
127 # ===== #
128 # YOUR CODE HERE:
129 #   Implement the ReLU backward pass
130 # ===== #
131 dx = dout * (x > 0)
132 # ===== #
133 # END YOUR CODE HERE
134 # ===== #
135
136 return dx
137
138
139 def batchnorm_forward(x, gamma, beta, bn_param):
140     """
141     Forward pass for batch normalization.
142
143     During training the sample mean and (uncorrected) sample variance are
144     computed from minibatch statistics and used to normalize the incoming data.
145     During training we also keep an exponentially decaying running mean of the mean
146     and variance of each feature, and these averages are used to normalize data
147     at test-time.
148
149     At each timestep we update the running averages for mean and variance using
150     an exponential decay based on the momentum parameter:
151
152     running_mean = momentum * running_mean + (1 - momentum) * sample_mean
153     running_var = momentum * running_var + (1 - momentum) * sample_var
154
155     Note that the batch normalization paper suggests a different test-time
156     behavior: they compute sample mean and variance for each feature using a
157     large number of training images rather than using a running average. For
158     this implementation we have chosen to use running averages instead since
159     they do not require an additional estimation step; the torch7 implementation
160     of batch normalization also uses running averages.
161
162     Input:
163     - x: Data of shape (N, D)
164     - gamma: Scale parameter of shape (D,)
165     - beta: Shift parameter of shape (D,)
166     - bn_param: Dictionary with the following keys:
167       - mode: 'train' or 'test'; required
168       - eps: Constant for numeric stability
169       - momentum: Constant for running mean / variance.
170       - running_mean: Array of shape (D,) giving running mean of features
171       - running_var: Array of shape (D,) giving running variance of features
172
173     Returns a tuple of:
174     - out: of shape (N, D)
175     - cache: A tuple of values needed in the backward pass
176     """
177     mode = bn_param['mode']
178     eps = bn_param.get('eps', 1e-5)

```

```

179 momentum = bn_param.get('momentum', 0.9)
180
181 N, D = x.shape
182 running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
183 running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
184
185 out, cache = None, None
186 if mode == 'train':
187
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     sample_mean = np.mean(x, axis=0)
199     sample_var = np.var(x, axis=0)
200     xhat = (x - sample_mean) / np.sqrt(sample_var + eps)
201     out = gamma * xhat + beta
202
203     # update running mean and var
204     running_mean = momentum * running_mean + (1 - momentum) * sample_mean
205     running_var = momentum * running_var + (1 - momentum) * sample_var
206
207     cache = (x, xhat, sample_var, gamma, beta, eps)
208     # ===== #
209     # END YOUR CODE HERE
210     # ===== #
211
212 elif mode == 'test':
213
214     # ===== #
215     # YOUR CODE HERE:
216     #   Calculate the testing time normalized activation. Normalize using
217     #   the running mean and variance, and then scale and shift appropriately.
218     #   Store the output as 'out'.
219     # ===== #
220     x_hat = (x - running_mean) / np.sqrt(running_var + eps)
221     out = gamma * x_hat + beta
222     # ===== #
223     # END YOUR CODE HERE
224     # ===== #
225
226 else:
227     raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
228
229 # Store the updated running means back into bn_param
230 bn_param['running_mean'] = running_mean
231 bn_param['running_var'] = running_var
232
233 return out, cache
234
235
236 def batchnorm_backward(dout, cache):
237     """
238     Backward pass for batch normalization.

```

```

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

```

```

299 if mode == 'train':
300     # ===== #
301     # YOUR CODE HERE:
302     # Implement the inverted dropout forward pass during training time.
303     # Store the masked and scaled activations in out, and store the
304     # dropout mask as the variable mask.
305     # ===== #
306     mask = np.random.uniform(low=0, high=1, size=x.shape) < p
307     out = x * mask
308     # ===== #
309     # END YOUR CODE HERE
310     # ===== #
311
312 elif mode == 'test':
313     # ===== #
314     # YOUR CODE HERE:
315     # Implement the inverted dropout forward pass during test time.
316     # ===== #
317     out = x
318     # ===== #
319     # END YOUR CODE HERE
320     # ===== #
321
322
323 cache = (dropout_param, mask)
324 out = out.astype(x.dtype, copy=False)
325
326 return out, cache
327
328
329 def dropout_backward(dout, cache):
330     """
331     Perform the backward pass for (inverted) dropout.
332
333     Inputs:
334     - dout: Upstream derivatives, of any shape
335     - cache: (dropout_param, mask) from dropout_forward.
336     """
337     dropout_param, mask = cache
338     mode = dropout_param['mode']
339
340     dx = None
341     if mode == 'train':
342         # ===== #
343         # YOUR CODE HERE:
344         # Implement the inverted dropout backward pass during training time.
345         # ===== #
346         dx = dout * mask
347         # ===== #
348         # END YOUR CODE HERE
349         # ===== #
350     elif mode == 'test':
351         # ===== #
352         # YOUR CODE HERE:
353         # Implement the inverted dropout backward pass during test time.
354         # ===== #
355         dx = dout
356         # ===== #
357         # END YOUR CODE HERE
358         # ===== #

```

```

359     return dx
360
361
362 def svm_loss(x, y):
363     """
364     Computes the loss and gradient using for multiclass SVM classification.
365
366     Inputs:
367     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
368         for the ith input.
369     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
370         0 <= y[i] < C
371
372     Returns a tuple of:
373     - loss: Scalar giving the loss
374     - dx: Gradient of the loss with respect to x
375     """
376     N = x.shape[0]
377     correct_class_scores = x[np.arange(N), y]
378     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
379     margins[np.arange(N), y] = 0
380     loss = np.sum(margins) / N
381     num_pos = np.sum(margins > 0, axis=1)
382     dx = np.zeros_like(x)
383     dx[margins > 0] = 1
384     dx[np.arange(N), y] -= num_pos
385     dx /= N
386     return loss, dx
387
388
389 def softmax_loss(x, y):
390     """
391     Computes the loss and gradient for softmax classification.
392
393     Inputs:
394     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
395         for the ith input.
396     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
397         0 <= y[i] < C
398
399     Returns a tuple of:
400     - loss: Scalar giving the loss
401     - dx: Gradient of the loss with respect to x
402     """
403
404     probs = np.exp(x - np.max(x, axis=1, keepdims=True)) + 1e-8
405     probs /= np.sum(probs, axis=1, keepdims=True)
406     N = x.shape[0]
407     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
408     dx = probs.copy()
409     dx[np.arange(N), y] -= 1
410     dx /= N
411     return loss, dx

```