

```

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 def affine_forward(x, w, b):
15     """
16     Computes the forward pass for an affine (fully-connected) layer.
17
18     The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
19     examples, where each example x[i] has shape (d_1, ..., d_k). We will
20     reshape each input into a vector of dimension D = d_1 * ... * d_k, and
21     then transform it to an output vector of dimension M.
22
23     Inputs:
24     - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
25     - w: A numpy array of weights, of shape (D, M)
26     - b: A numpy array of biases, of shape (M,)
27
28     Returns a tuple of:
29     - out: output, of shape (N, M)
30     - cache: (x, w, b)
31     """
32
33     # ===== #
34     # YOUR CODE HERE:
35     # Calculate the output of the forward pass. Notice the dimensions
36     # of w are D x M, which is the transpose of what we did in earlier
37     # assignments.
38     # ===== #
39
40     shape = x.shape
41     N = shape[0]
42     D = np.prod(shape[1:])
43     reshaped_x = np.reshape(x, (N,D))
44
45     out = reshaped_x.dot(w) + b[:, np.newaxis].T
46
47     # ===== #
48     # END YOUR CODE HERE
49     # ===== #
50
51     cache = (x, w, b)
52     return out, cache
53
54
55 def affine_backward(dout, cache):
56     """
57     Computes the backward pass for an affine layer.
58
59     Inputs:
60     - dout: Upstream derivative, of shape (N, M)
61     - cache: Tuple of:
62       - x: Input data, of shape (N, d_1, ... d_k)
63       - w: Weights, of shape (D, M)
64
65     Returns a tuple of:
66     - dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
67     - dw: Gradient with respect to w, of shape (D, M)
68     - db: Gradient with respect to b, of shape (M,)
69     """

```

```

70 x, w, b = cache
71 dx, dw, db = None, None, None
72
73 # ===== #
74 # YOUR CODE HERE:
75 #   Calculate the gradients for the backward pass.
76 # ===== #
77
78 N, M = dout.shape
79 D = w.shape[0]
80 reshaped_x = np.reshape(x, (N,D))
81
82 db = np.sum(dout, axis=0)
83 dw = reshaped_x.T.dot(dout)
84 dx = np.reshape(dout.dot(w.T), x.shape)
85
86 # ===== #
87 # END YOUR CODE HERE
88 # ===== #
89
90 return dx, dw, db
91
92 def relu_forward(x):
93     """
94     Computes the forward pass for a layer of rectified linear units (ReLU).
95
96     Input:
97     - x: Inputs, of any shape
98
99     Returns a tuple of:
100     - out: Output, of the same shape as x
101     - cache: x
102     """
103     # ===== #
104     # YOUR CODE HERE:
105     #   Implement the ReLU forward pass.
106     # ===== #
107     out = np.empty_like(x)
108     out[:] = x
109     out[out<0] = 0
110     # ===== #
111     # END YOUR CODE HERE
112     # ===== #
113
114     cache = x
115     return out, cache
116
117
118 def relu_backward(dout, cache):
119     """
120     Computes the backward pass for a layer of rectified linear units (ReLU).
121
122     Input:
123     - dout: Upstream derivatives, of any shape
124     - cache: Input x, of same shape as dout
125
126     Returns:
127     - dx: Gradient with respect to x
128     """
129     x = cache
130
131     # ===== #
132     # YOUR CODE HERE:
133     #   Implement the ReLU backward pass
134     # ===== #
135
136     dx = np.empty_like(dout)
137     dx[:] = dout
138     dx[x<0] = 0

```

```

139
140 # ===== #
141 # END YOUR CODE HERE
142 # ===== #
143
144 return dx
145
146 def batchnorm_forward(x, gamma, beta, bn_param):
147     """
148     Forward pass for batch normalization.
149
150     During training the sample mean and (uncorrected) sample variance are
151     computed from minibatch statistics and used to normalize the incoming data.
152     During training we also keep an exponentially decaying running mean of the mean
153     and variance of each feature, and these averages are used to normalize data
154     at test-time.
155
156     At each timestep we update the running averages for mean and variance using
157     an exponential decay based on the momentum parameter:
158
159     running_mean = momentum * running_mean + (1 - momentum) * sample_mean
160     running_var = momentum * running_var + (1 - momentum) * sample_var
161
162     Note that the batch normalization paper suggests a different test-time
163     behavior: they compute sample mean and variance for each feature using a
164     large number of training images rather than using a running average. For
165     this implementation we have chosen to use running averages instead since
166     they do not require an additional estimation step; the torch7 implementation
167     of batch normalization also uses running averages.
168
169     Input:
170     - x: Data of shape (N, D)
171     - gamma: Scale parameter of shape (D,)
172     - beta: Shift parameter of shape (D,)
173     - bn_param: Dictionary with the following keys:
174       - mode: 'train' or 'test'; required
175       - eps: Constant for numeric stability
176       - momentum: Constant for running mean / variance.
177       - running_mean: Array of shape (D,) giving running mean of features
178       - running_var: Array of shape (D,) giving running variance of features
179
180     Returns a tuple of:
181     - out: of shape (N, D)
182     - cache: A tuple of values needed in the backward pass
183     """
184     mode = bn_param['mode']
185     eps = bn_param.get('eps', 1e-5)
186     momentum = bn_param.get('momentum', 0.9)
187
188     N, D = x.shape
189     running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
190     running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
191
192     out, cache = None, None
193     if mode == 'train':
194
195         # ===== #
196         # YOUR CODE HERE:
197         #   A few steps here:
198         #   (1) Calculate the running mean and variance of the minibatch.
199         #   (2) Normalize the activations with the running mean and variance.
200         #   (3) Scale and shift the normalized activations. Store this
201         #       as the variable 'out'
202         #   (4) Store any variables you may need for the backward pass in
203         #       the 'cache' variable.
204         # ===== #
205
206     sample_mean = x.mean(axis=0)
207     sample_var = x.var(axis=0)

```

```

208
209     running_mean = momentum * running_mean + (1 - momentum) * sample_mean
210     running_var = momentum * running_var + (1 - momentum) * sample_var
211
212     x_hat = x - sample_mean[:, np.newaxis].T
213     x_hat /= np.sqrt((sample_var[:, np.newaxis].T + eps))
214
215     out = gamma*x_hat + beta
216
217     cache = (x, gamma, x_hat, sample_var, sample_mean, eps)
218
219     # ===== #
220     # END YOUR CODE HERE
221     # ===== #
222
223     elif mode == 'test':
224
225         # ===== #
226         # YOUR CODE HERE:
227         #   Calculate the testing time normalized activations. Normalize using
228         #   the running mean and variance, and then scale and shift appropriately.
229         #   Store the output as 'out'.
230         # ===== #
231
232         out = x - running_mean[:, np.newaxis].T
233         out /= np.sqrt((running_var[:, np.newaxis].T + eps))
234
235         out = gamma*out + beta
236
237
238         # ===== #
239         # END YOUR CODE HERE
240         # ===== #
241
242     else:
243         raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
244
245     # Store the updated running means back into bn_param
246     bn_param['running_mean'] = running_mean
247     bn_param['running_var'] = running_var
248
249     return out, cache
250
251 def batchnorm_backward(dout, cache):
252     """
253     Backward pass for batch normalization.
254
255     For this implementation, you should write out a computation graph for
256     batch normalization on paper and propagate gradients backward through
257     intermediate nodes.
258
259     Inputs:
260     - dout: Upstream derivatives, of shape (N, D)
261     - cache: Variable of intermediates from batchnorm_forward.
262
263     Returns a tuple of:
264     - dx: Gradient with respect to inputs x, of shape (N, D)
265     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
266     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
267     """
268     dx, dgamma, dbeta = None, None, None
269
270     # ===== #
271     # YOUR CODE HERE:
272     #   Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
273     # ===== #
274     x, gamma, x_hat, var, mu, eps = cache
275
276     N = x.shape[0]

```

```

277
278 dgamma = np.sum(np.multiply(x_hat, dout), axis=0)
279 dbeta = dout.sum(axis=0)
280
281 dx_hat = np.multiply(gamma, dout)
282
283 dmu = -1.0/np.sqrt(var + eps)*np.sum(dx_hat, axis=0)
284 da = np.multiply(1/np.sqrt(var+eps), dx_hat)
285 de = -.5*1.0/np.power(var+eps, 1.5)*(x - mu)*dx_hat
286 dvar = np.sum(de, axis=0)
287
288 dx = da + 2.0*(x-mu)/N*dvar + 1.0/N*dmu
289
290
291
292
293 # ===== #
294 # END YOUR CODE HERE
295 # ===== #
296
297 return dx, dgamma, dbeta
298
299 def dropout_forward(x, dropout_param):
300     """
301     Performs the forward pass for (inverted) dropout.
302
303     Inputs:
304     - x: Input data, of any shape
305     - dropout_param: A dictionary with the following keys:
306       - p: Dropout parameter. We drop each neuron output with probability p.
307       - mode: 'test' or 'train'. If the mode is train, then perform dropout;
308         if the mode is test, then just return the input.
309       - seed: Seed for the random number generator. Passing seed makes this
310         function deterministic, which is needed for gradient checking but not in
311         real networks.
312
313     Outputs:
314     - out: Array of the same shape as x.
315     - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
316       mask that was used to multiply the input; in test mode, mask is None.
317     """
318     p, mode = dropout_param['p'], dropout_param['mode']
319     if 'seed' in dropout_param:
320         np.random.seed(dropout_param['seed'])
321
322     mask = None
323     out = None
324
325     if mode == 'train':
326         # ===== #
327         # YOUR CODE HERE:
328         # Implement the inverted dropout forward pass during training time.
329         # Store the masked and scaled activations in out, and store the
330         # dropout mask as the variable mask.
331         # ===== #
332
333         mask = (np.random.rand(*x.shape) > p)/(1-p)
334         out = x*mask
335
336         # ===== #
337         # END YOUR CODE HERE
338         # ===== #
339
340     elif mode == 'test':
341
342         # ===== #
343         # YOUR CODE HERE:
344         # Implement the inverted dropout forward pass during test time.
345         # ===== #

```

```

346
347     out = np.empty_like(x)
348     out[:] = x
349
350
351     # ===== #
352     # END YOUR CODE HERE
353     # ===== #
354
355     cache = (dropout_param, mask)
356     out = out.astype(x.dtype, copy=False)
357
358     return out, cache
359
360 def dropout_backward(dout, cache):
361     """
362     Perform the backward pass for (inverted) dropout.
363
364     Inputs:
365     - dout: Upstream derivatives, of any shape
366     - cache: (dropout_param, mask) from dropout_forward.
367     """
368     dropout_param, mask = cache
369     mode = dropout_param['mode']
370
371     dx = None
372     if mode == 'train':
373         # ===== #
374         # YOUR CODE HERE:
375         #   Implement the inverted dropout backward pass during training time.
376         # ===== #
377         dx = dout*mask
378         # ===== #
379         # END YOUR CODE HERE
380         # ===== #
381     elif mode == 'test':
382         # ===== #
383         # YOUR CODE HERE:
384         #   Implement the inverted dropout backward pass during test time.
385         # ===== #
386         dx = np.empty_like(dout)
387         dx[:] = dout
388         # ===== #
389         # END YOUR CODE HERE
390         # ===== #
391     return dx
392
393 def svm_loss(x, y):
394     """
395     Computes the loss and gradient using for multiclass SVM classification.
396
397     Inputs:
398     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
399         for the ith input.
400     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
401         0 <= y[i] < C
402
403     Returns a tuple of:
404     - loss: Scalar giving the loss
405     - dx: Gradient of the loss with respect to x
406     """
407     N = x.shape[0]
408     correct_class_scores = x[np.arange(N), y]
409     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
410     margins[np.arange(N), y] = 0
411     loss = np.sum(margins) / N
412     num_pos = np.sum(margins > 0, axis=1)
413     dx = np.zeros_like(x)
414     dx[margins > 0] = 1

```

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