

```

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
10 for
11 permission to use this code. To see the original version, please visit
12 cs231n.stanford.edu.
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     out = None
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     X = x.reshape(x.shape[0], -1)
41     out = X.dot(w) + b
42
43     # ===== #
44     # END YOUR CODE HERE
45     # ===== #
46
47     cache = (x, w, b)
48     return out, cache
49
50
51 def affine_backward(dout, cache):
52     """
53     Computes the backward pass for an affine layer.
54
55     Inputs:
56     - dout: Upstream derivative, of shape (N, M)
57     - cache: Tuple of:
58       - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
59       - w: A numpy array of weights, of shape (D, M)

```

```

60     - b: A numpy array of biases, of shape (M,)
61
62     Returns a tuple of:
63     - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
64     - dw: Gradient with respect to w, of shape (D, M)
65     - db: Gradient with respect to b, of shape (M,)
66     """
67     x, w, b = cache
68     dx, dw, db = None, None, None
69
70     # ===== #
71     # YOUR CODE HERE:
72     # Calculate the gradients for the backward pass.
73     # Notice:
74     # dout is N x M
75     # dx should be N x d1 x ... x dk; it relates to dout through
multiplication with w, which is D x M
76     # dw should be D x M; it relates to dout through multiplication with x,
which is N x D after reshaping
77     # db should be M; it is just the sum over dout examples
78     # ===== #
79
80     dx = dout.dot(w.T).reshape(x.shape)
81     dw = x.reshape(x.shape[0], -1).T.dot(dout)
82     db = dout.sum(axis=0)
83
84     # ===== #
85     # END YOUR CODE HERE
86     # ===== #
87
88     return dx, dw, db
89
90 def relu_forward(x):
91     """
92     Computes the forward pass for a layer of rectified linear units (ReLU).
93
94     Input:
95     - x: Inputs, of any shape
96
97     Returns a tuple of:
98     - out: Output, of the same shape as x
99     - cache: x
100    """
101    # ===== #
102    # YOUR CODE HERE:
103    # Implement the ReLU forward pass.
104    # ===== #
105
106    out = x * (x > 0)
107
108    # ===== #
109    # END YOUR CODE HERE
110    # ===== #
111
112    cache = x
113    return out, cache
114
115
116 def relu_backward(dout, cache):
117     """

```

```

118     Computes the backward pass for a layer of rectified linear units (ReLUs).
119
120     Input:
121     - dout: Upstream derivatives, of any shape
122     - cache: Input x, of same shape as dout
123
124     Returns:
125     - dx: Gradient with respect to x
126     """
127     x = cache
128
129     # ===== #
130     # YOUR CODE HERE:
131     #     Implement the ReLU backward pass
132     # ===== #
133
134     dx = dout * (cache > 0)
135
136     # ===== #
137     # END YOUR CODE HERE
138     # ===== #
139
140     return dx
141
142 def batchnorm_forward(x, gamma, beta, bn_param):
143     """
144     Forward pass for batch normalization.
145
146     During training the sample mean and (uncorrected) sample variance are
147     computed from minibatch statistics and used to normalize the incoming
148     data.
149     During training we also keep an exponentially decaying running mean of
150     the mean
151     and variance of each feature, and these averages are used to normalize
152     data
153     at test-time.
154
155     At each timestep we update the running averages for mean and variance
156     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
167     implementation
168     of batch normalization also uses running averages.
169
170     Input:
171     - x: Data of shape (N, D)
172     - gamma: Scale parameter of shape (D,)
173     - beta: Shift parameter of shape (D,)
174     - bn_param: Dictionary with the following keys:
175         - mode: 'train' or 'test'; required
176         - eps: Constant for numeric stability
177         - momentum: Constant for running mean / variance.

```

```

173     - running_mean: Array of shape (D,) giving running mean of features
174     - running_var: Array of shape (D,) giving running variance of features
175
176 Returns a tuple of:
177 - out: of shape (N, D)
178 - cache: A tuple of values needed in the backward pass
179 """
180 mode = bn_param['mode']
181 eps = bn_param.get('eps', 1e-5)
182 momentum = bn_param.get('momentum', 0.9)
183
184 N, D = x.shape
185 running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
186 running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
187
188 out, cache = None, None
189 if mode == 'train':
190
191     # ===== #
192     # YOUR CODE HERE:
193     #   A few steps here:
194     #   (1) Calculate the running mean and variance of the minibatch.
195     #   (2) Normalize the activations with the running mean and
196 variance.
197     #   (3) Scale and shift the normalized activations. Store this
198     #       as the variable 'out'
199     #   (4) Store any variables you may need for the backward pass in
200     #       the 'cache' variable.
201     # ===== #
202
203     mean = np.mean(x, axis=0)
204     var = np.sum((x - mean)**2, axis=0) / N
205
206     running_mean = momentum * running_mean + (1 - momentum) * mean
207     running_var = momentum * running_var + (1 - momentum) * var
208
209     var_eps_sum_inv = 1 / np.sqrt(var + eps)
210     x_mean_diff = (x - mean)
211     x_n = var_eps_sum_inv * x_mean_diff
212     out = gamma * x_n + beta
213
214     cache = (x_n, x, gamma, var_eps_sum_inv, x_mean_diff)
215
216     # ===== #
217     # END YOUR CODE HERE
218     # ===== #
219 elif mode == 'test':
220     # ===== #
221     # YOUR CODE HERE:
222     #   Calculate the testing time normalized activation. Normalize
223 using
224     #   the running mean and variance, and then scale and shift
225 appropriately.
226     #   Store the output as 'out'.
227     # ===== #
228
229     x_n = (x - running_mean) / np.sqrt(running_var + eps)
230     out = gamma * x_n + beta
231
232     # ===== #

```

```

230         # END YOUR CODE HERE
231         # ===== #
232     else:
233         raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
234
235     # Store the updated running means back into bn_param
236     bn_param['running_mean'] = running_mean
237     bn_param['running_var'] = running_var
238
239     return out, cache
240
241 def batchnorm_backward(dout, cache):
242     """
243     Backward pass for batch normalization.
244
245     For this implementation, you should write out a computation graph for
246     batch normalization on paper and propagate gradients backward through
247     intermediate nodes.
248
249     Inputs:
250     - dout: Upstream derivatives, of shape (N, D)
251     - cache: Variable of intermediates from batchnorm_forward.
252
253     Returns a tuple of:
254     - dx: Gradient with respect to inputs x, of shape (N, D)
255     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
256     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
257     """
258     dx, dgamma, dbeta = None, None, None
259
260     # ===== #
261     # YOUR CODE HERE:
262     # Implement the batchnorm backward pass, calculating dx, dgamma, and
263     # ===== #
264     dbeta.
265
266     x_n, x, gamma, var_eps_sum_inv, x_mean_diff = cache
267     N = x.shape[0]
268     dgamma = np.sum(dout * x_n, axis=0)
269     dbeta = np.sum(dout, axis=0)
270
271     dxhat = dout * gamma
272     da = var_eps_sum_inv * dxhat
273     db = x_mean_diff * dxhat
274     dc = -(var_eps_sum_inv**2) * db
275     de = 0.5 * var_eps_sum_inv * dc
276     dsig = np.sum(de, axis=0)
277     dmu = -var_eps_sum_inv * np.sum(dxhat, axis=0) - dsig * (2/N) *
278     np.sum(x_mean_diff, axis=0)
279
280     dx = da + (2*x_mean_diff/N) * dsig + (dmu/N)
281
282     # ===== #
283     # END YOUR CODE HERE
284     # ===== #
285
286     return dx, dgamma, dbeta
287
288 def dropout_forward(x, dropout_param):
289     """

```

```

288     Performs the forward pass for (inverted) dropout.
289
290     Inputs:
291     - x: Input data, of any shape
292     - dropout_param: A dictionary with the following keys:
293         - p: Dropout parameter. We keep each neuron output with probability p.
294         - mode: 'test' or 'train'. If the mode is train, then perform dropout;
295           if the mode is test, then just return the input.
296         - seed: Seed for the random number generator. Passing seed makes this
297           function deterministic, which is needed for gradient checking but not
in
298         real networks.
299
300     Outputs:
301     - out: Array of the same shape as x.
302     - cache: A tuple (dropout_param, mask). In training mode, mask is the
dropout
303     mask that was used to multiply the input; in test mode, mask is None.
304     """
305     p, mode = dropout_param['p'], dropout_param['mode']
306     if 'seed' in dropout_param:
307         np.random.seed(dropout_param['seed'])
308
309     mask = None
310     out = None
311
312     if mode == 'train':
313         # ===== #
314         # YOUR CODE HERE:
315         #     Implement the inverted dropout forward pass during training time.
316
317         #     Store the masked and scaled activations in out, and store the
318         #     dropout mask as the variable mask.
319         # ===== #
320         mask = (np.random.rand(*x.shape) < p) / p
321         out = x * mask
322
323         # ===== #
324         # END YOUR CODE HERE
325         # ===== #
326
327     elif mode == 'test':
328
329         # ===== #
330         # YOUR CODE HERE:
331         #     Implement the inverted dropout forward pass during test time.
332         # ===== #
333
334         out = x
335
336         # ===== #
337         # END YOUR CODE HERE
338         # ===== #
339
340     cache = (dropout_param, mask)
341     out = out.astype(x.dtype, copy=False)
342
343     return out, cache
344

```

```

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

```

```

403     dx[np.arange(N), y] -= num_pos
404     dx /= N
405     return loss, dx
406
407
408 def softmax_loss(x, y):
409     """
410     Computes the loss and gradient for softmax classification.
411
412     Inputs:
413     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
class
414       for the ith input.
415     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
416       0 <= y[i] < C
417
418     Returns a tuple of:
419     - loss: Scalar giving the loss
420     - dx: Gradient of the loss with respect to x
421     """
422
423     probs = np.exp(x - np.max(x, axis=1, keepdims=True))
424     probs /= np.sum(probs, axis=1, keepdims=True)
425     N = x.shape[0]
426     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
427     dx = probs.copy()
428     dx[np.arange(N), y] -= 1
429     dx /= N
430     return loss, dx
431

```