

```

1 import numpy as np
2
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: A numpy array containing input data, of shape (N, d_1, ..., 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
58    x, w, b = cache
59    dx, dw, db = None, None, None

```

```

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

```

```

118 - dx: Gradient with respect to x
119 """
120 x = cache
121
122 # ===== #
123 # YOUR CODE HERE:
124 #   Implement the ReLU backward pass
125 # ===== #
126
127 dx = dout * (x > 0)
128
129 # ===== #
130 # END YOUR CODE HERE
131 # ===== #
132
133 return dx
134
135 def batchnorm_forward(x, gamma, beta, bn_param):
136     """
137     Forward pass for batch normalization.
138
139     During training the sample mean and (uncorrected) sample variance are
140     computed from minibatch statistics and used to normalize the incoming data.
141     During training we also keep an exponentially decaying running mean of the
142     mean
143     and variance of each feature, and these averages are used to normalize data
144     at test-time.
145
146     At each timestep we update the running averages for mean and variance using
147     an exponential decay based on the momentum parameter:
148
149     running_mean = momentum * running_mean + (1 - momentum) * sample_mean
150     running_var = momentum * running_var + (1 - momentum) * sample_var
151
152     Note that the batch normalization paper suggests a different test-time
153     behavior: they compute sample mean and variance for each feature using a
154     large number of training images rather than using a running average. For
155     this implementation we have chosen to use running averages instead since
156     they do not require an additional estimation step; the torch7
157     implementation
158     of batch normalization also uses running averages.
159
160     Input:
161     - x: Data of shape (N, D)
162     - gamma: Scale parameter of shape (D,)
163     - beta: Shift parameter of shape (D,)
164     - bn_param: Dictionary with the following keys:
165       - mode: 'train' or 'test'; required
166       - eps: Constant for numeric stability
167       - momentum: Constant for running mean / variance.
168       - running_mean: Array of shape (D,) giving running mean of features
169       - running_var: Array of shape (D,) giving running variance of features
170
171     Returns a tuple of:
172     - out: of shape (N, D)
173     - cache: A tuple of values needed in the backward pass
174     """
175     mode = bn_param['mode']
176     eps = bn_param.get('eps', 1e-5)
177     momentum = bn_param.get('momentum', 0.9)

```

```

176
177 N, D = x.shape
178 running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
179 running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
180
181 out, cache = None, None
182 if mode == 'train':
183
184     # ===== #
185     # YOUR CODE HERE:
186     #   A few steps here:
187     #   (1) Calculate the running mean and variance of the minibatch.
188     #   (2) Normalize the activations with the running mean and variance.
189     #   (3) Scale and shift the normalized activations. Store this
190     #       as the variable 'out'
191     #   (4) Store any variables you may need for the backward pass in
192     #       the 'cache' variable.
193     # ===== #
194
195     mean = np.mean(x, axis = 0)
196     var = np.var(x, axis = 0)
197     normalize_x = (x - mean) / np.sqrt(var + eps)
198
199     running_mean = momentum * running_mean + (1 - momentum) * mean
200     running_var = momentum * running_var + (1 - momentum) * var
201
202     out = gamma * normalize_x + beta
203
204     cache = {'normalize_x': normalize_x,
205             'x_minus_mean': (x - mean),
206             'sqrt_var_eps': np.sqrt(var + eps),
207             'gamma': gamma
208             }
209
210     # ===== #
211     # END YOUR CODE HERE
212     # ===== #
213
214 elif mode == 'test':
215
216     # ===== #
217     # YOUR CODE HERE:
218     #   Calculate the testing time normalized activation. Normalize using
219     #   the running mean and variance, and then scale and shift
220     #   appropriately.
221     #   Store the output as 'out'.
222     # ===== #
223
224     normalize_x = (x - running_mean) / np.sqrt(running_var + eps)
225     out = gamma * normalize_x + beta
226
227     # ===== #
228     # END YOUR CODE HERE
229     # ===== #
230
231 else:
232     raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
233
234 # Store the updated running means back into bn_param
235 bn_param['running_mean'] = running_mean

```

```

235     bn_param['running_var'] = running_var
236
237     return out, cache
238
239 def batchnorm_backward(dout, cache):
240     """
241     Backward pass for batch normalization.
242
243     For this implementation, you should write out a computation graph for
244     batch normalization on paper and propagate gradients backward through
245     intermediate nodes.
246
247     Inputs:
248     - dout: Upstream derivatives, of shape (N, D)
249     - cache: Variable of intermediates from batchnorm_forward.
250
251     Returns a tuple of:
252     - dx: Gradient with respect to inputs x, of shape (N, D)
253     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
254     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
255     """
256     dx, dgamma, dbeta = None, None, None
257
258     # ===== #
259     # YOUR CODE HERE:
260     # Implement the batchnorm backward pass, calculating dx, dgamma, and
261     # dbeta.
262     # ===== #
263
264     normalize_x = cache.get('normalize_x')
265     x_minus_mean = cache.get('x_minus_mean')
266     sqrt_var_eps = cache.get('sqrt_var_eps')
267     gamma = cache.get('gamma')
268     N = dout.shape[0]
269
270     dbeta = np.sum(dout, axis = 0)
271     dgamma = np.sum(dout * normalize_x, axis = 0)
272
273     dnormalize_x = dout * gamma
274
275     db = x_minus_mean * dnormalize_x # b = 1 / sqrt_var_eps
276     dc = (-1 / (sqrt_var_eps * sqrt_var_eps)) * db # c = sqrt_var_eps
277     de = (1 / (2 * sqrt_var_eps)) * dc # e = sqrt_var_eps * sqrt_var_eps
278     dvar = np.sum(de, axis = 0)
279
280     da = dnormalize_x / sqrt_var_eps # a = x - mu
281     dmu = -np.sum(da, axis = 0) - 2 * np.sum(x_minus_mean, axis = 0) * dvar / N
282
283     dx = da + 2 * x_minus_mean * dvar / N + dmu / N
284
285     # ===== #
286     # END YOUR CODE HERE
287     # ===== #
288
289     return dx, dgamma, dbeta
290
291 def dropout_forward(x, dropout_param):
292     """
293     Performs the forward pass for (inverted) dropout.

```

```

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

```

```

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

```

```
411
412 def softmax_loss(x, y):
413     """
414     Computes the loss and gradient for softmax classification.
415
416     Inputs:
417     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
class
418     for the ith input.
419     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
420     0 <= y[i] < C
421
422     Returns a tuple of:
423     - loss: Scalar giving the loss
424     - dx: Gradient of the loss with respect to x
425     """
426
427     probs = np.exp(x - np.max(x, axis=1, keepdims=True))
428     probs /= np.sum(probs, axis=1, keepdims=True)
429     N = x.shape[0]
430     loss = -np.sum(np.log(probs[np.arange(N), y])) / N
431     dx = probs.copy()
432     dx[np.arange(N), y] -= 1
433     dx /= N
434     return loss, dx
435
```