

```

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

```

```

67     x D after reshaping
68     # db should be M; it is just the sum over dout examples
69     # ===== #
70     flattened_x = x.reshape(x.shape[0],-1)
71     dx = np.dot(dout,w.T).reshape(x.shape)
72     dw = np.dot(flattened_x.T,dout)
73     db = np.sum(dout, axis = 0)
74     # ===== #
75     # END YOUR CODE HERE
76     # ===== #
77
78     return dx, dw, db
79
80 def relu_forward(x):
81     """
82     Computes the forward pass for a layer of rectified linear units (ReLU).
83
84     Input:
85     - x: Inputs, of any shape
86
87     Returns a tuple of:
88     - out: Output, of the same shape as x
89     - cache: x
90     """
91     # ===== #
92     # YOUR CODE HERE:
93     # Implement the ReLU forward pass.
94     # ===== #
95     relu = lambda x: x * (x > 0)
96     out = relu(x)
97     # ===== #
98     # END YOUR CODE HERE
99     # ===== #
100
101     cache = x
102     return out, cache
103
104
105 def relu_backward(dout, cache):
106     """
107     Computes the backward pass for a layer of rectified linear units (ReLU).
108
109     Input:
110     - dout: Upstream derivatives, of any shape
111     - cache: Input x, of same shape as dout
112
113     Returns:
114     - dx: Gradient with respect to x
115     """
116     x = cache
117
118     # ===== #
119     # YOUR CODE HERE:
120     # Implement the ReLU backward pass
121     # ===== #
122     dx = dout * (x > 0)
123
124     # ===== #
125     # END YOUR CODE HERE
126     # ===== #
127
128     return dx
129
130 def batchnorm_forward(x, gamma, beta, bn_param):
131     """
132     Forward pass for batch normalization.

```

During training the sample mean and (uncorrected) sample variance are computed from minibatch statistics and used to normalize the incoming data. During training we also keep an exponentially decaying running mean of the mean and variance of each feature, and these averages are used to normalize data at test-time.

At each timestep we update the running averages for mean and variance using an exponential decay based on the momentum parameter:

```
running_mean = momentum * running_mean + (1 - momentum) * sample_mean
running_var = momentum * running_var + (1 - momentum) * sample_var
```

Note that the batch normalization paper suggests a different test-time behavior: they compute sample mean and variance for each feature using a large number of training images rather than using a running average. For this implementation we have chosen to use running averages instead since they do not require an additional estimation step; the torch7 implementation of batch normalization also uses running averages.

Input:

- x: Data of shape (N, D)
- gamma: Scale parameter of shape (D,)
- beta: Shift parameter of shape (D,)
- bn\_param: Dictionary with the following keys:
  - mode: 'train' or 'test'; required
  - eps: Constant for numeric stability
  - momentum: Constant for running mean / variance.
  - running\_mean: Array of shape (D,) giving running mean of features
  - running\_var: Array of shape (D,) giving running variance of features

Returns a tuple of:

- out: of shape (N, D)
- cache: A tuple of values needed in the backward pass

```
mode = bn_param['mode']
eps = bn_param.get('eps', 1e-5)
momentum = bn_param.get('momentum', 0.9)
```

```
N, D = x.shape
running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
```

```
out, cache = None, None
```

```
if mode == 'train':
```

```
    # ===== #
    # YOUR CODE HERE:
    #   A few steps here:
    #   (1) Calculate the running mean and variance of the minibatch.
    #   (2) Normalize the activations with the running mean and variance.
    #   (3) Scale and shift the normalized activations. Store this
    #       as the variable 'out'
    #   (4) Store any variables you may need for the backward pass in
    #       the 'cache' variable.
    # ===== #
    mean_x = np.mean(x, axis = 0)
    var_x = np.var(x, axis = 0)

    running_mean = momentum * running_mean + ( 1 - momentum ) * mean_x
    running_var = momentum * running_var + ( 1 - momentum ) * var_x

    standard_x = ( x - mean_x ) / ( np.sqrt(var_x + eps))

    out = gamma * standard_x + beta
    cache = (mean_x, var_x, standard_x, gamma, x, eps)
```

```

200         # ===== #
201         # END YOUR CODE HERE
202         # ===== #
203     elif mode == 'test':
204         # ===== #
205         # YOUR CODE HERE:
206         #     Calculate the testing time normalized activation. Normalize using
207         #     the running mean and variance, and then scale and shift appropriately.
208         #     Store the output as 'out'.
209         # ===== #
210
211         standard_x = (x - running_mean) / (np.sqrt(running_var))
212         out = gamma * standard_x + beta
213         #cache = []
214         # ===== #
215         # END YOUR CODE HERE
216         # ===== #
217     else:
218         raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
219
220     # Store the updated running means back into bn_param
221     bn_param['running_mean'] = running_mean
222     bn_param['running_var'] = running_var
223
224     return out, cache
225
226 def batchnorm_backward(dout, cache):
227     """
228     Backward pass for batch normalization.
229
230     For this implementation, you should write out a computation graph for
231     batch normalization on paper and propagate gradients backward through
232     intermediate nodes.
233
234     Inputs:
235     - dout: Upstream derivatives, of shape (N, D)
236     - cache: Variable of intermediates from batchnorm_forward.
237
238     Returns a tuple of:
239     - dx: Gradient with respect to inputs x, of shape (N, D)
240     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
241     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
242     """
243     dx, dgamma, dbeta = None, None, None
244
245     # ===== #
246     # YOUR CODE HERE:
247     #     Implement the batchnorm backward pass, calculating dx, dgamma, and dbeta.
248     # ===== #
249     (mean_x, var_x, standard_x, gamma, x, eps) = cache
250     sample_size = x.shape[0]
251     sigma_x = np.sqrt(var_x + eps)
252
253     dgamma = np.sum(standard_x * dout, axis = 0)
254     dbeta = np.sum(dout, axis = 0)
255
256     dL_dx_st = dout * gamma
257
258     dx_st_da = 1 / sigma_x
259     dL_da = dx_st_da * dL_dx_st
260     da_dx = 1
261
262     dx_st_de = -0.5 * (dx_st_da ** 3) * (x - mean_x)
263     dL_de = dx_st_de * dL_dx_st
264
265     dL_dvar = np.sum(dL_de, axis = 0)
266     dvar_dx = (2 * (x - mean_x)) / sample_size

```

```

267
268 dL_dmean = np.sum(-dL_da, axis = 0)
269 dmean_dx = 1 / sample_size
270
271 dx = da_dx * dL_da + dvar_dx * dL_dvar + dmean_dx * dL_dmean
272 # ===== #
273 # END YOUR CODE HERE
274 # ===== #
275
276 return dx, dgamma, dbeta
277
278 def dropout_forward(x, dropout_param):
279     """
280     Performs the forward pass for (inverted) dropout.
281
282     Inputs:
283     - x: Input data, of any shape
284     - dropout_param: A dictionary with the following keys:
285       - p: Dropout parameter. We keep each neuron output with probability p.
286       - mode: 'test' or 'train'. If the mode is train, then perform dropout;
287         if the mode is test, then just return the input.
288       - seed: Seed for the random number generator. Passing seed makes this
289         function deterministic, which is needed for gradient checking but not in
290         real networks.
291
292     Outputs:
293     - out: Array of the same shape as x.
294     - cache: A tuple (dropout_param, mask). In training mode, mask is the dropout
295       mask that was used to multiply the input; in test mode, mask is None.
296     """
297     p, mode = dropout_param['p'], dropout_param['mode']
298     assert (0 < p <= 1), "Dropout probability is not in (0,1]"
299     if 'seed' in dropout_param:
300         np.random.seed(dropout_param['seed'])
301
302     mask = None
303     out = None
304
305     if mode == 'train':
306         # ===== #
307         # YOUR CODE HERE:
308         # Implement the inverted dropout forward pass during training time.
309         # Store the masked and scaled activations in out, and store the
310         # dropout mask as the variable mask.
311         # ===== #
312
313         mask = (np.random.rand(x.shape[0], x.shape[1]) < p) / p
314         out = x * mask
315         # ===== #
316         # END YOUR CODE HERE
317         # ===== #
318
319     elif mode == 'test':
320
321         # ===== #
322         # YOUR CODE HERE:
323         # Implement the inverted dropout forward pass during test time.
324         # ===== #
325
326         out = x
327
328         # ===== #
329         # END YOUR CODE HERE
330         # ===== #
331
332     cache = (dropout_param, mask)
333     out = out.astype(x.dtype, copy=False)

```

```

334
335     return out, cache
336
337 def dropout_backward(dout, cache):
338     """
339     Perform the backward pass for (inverted) dropout.
340
341     Inputs:
342     - dout: Upstream derivatives, of any shape
343     - cache: (dropout_param, mask) from dropout_forward.
344     """
345     dropout_param, mask = cache
346     mode = dropout_param['mode']
347
348     dx = None
349     if mode == 'train':
350         # ===== #
351         # YOUR CODE HERE:
352         # Implement the inverted dropout backward pass during training time.
353         # ===== #
354
355         dx = dout * mask
356
357         # ===== #
358         # END YOUR CODE HERE
359         # ===== #
360     elif mode == 'test':
361         # ===== #
362         # YOUR CODE HERE:
363         # Implement the inverted dropout backward pass during test time.
364         # ===== #
365
366         dx = dout
367
368         # ===== #
369         # END YOUR CODE HERE
370         # ===== #
371     return dx
372
373 def svm_loss(x, y):
374     """
375     Computes the loss and gradient using for multiclass SVM classification.
376
377     Inputs:
378     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
379         for the ith input.
380     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
381         0 <= y[i] < C
382
383     Returns a tuple of:
384     - loss: Scalar giving the loss
385     - dx: Gradient of the loss with respect to x
386     """
387     N = x.shape[0]
388     correct_class_scores = x[np.arange(N), y]
389     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
390     margins[np.arange(N), y] = 0
391     loss = np.sum(margins) / N
392     num_pos = np.sum(margins > 0, axis=1)
393     dx = np.zeros_like(x)
394     dx[margins > 0] = 1
395     dx[np.arange(N), y] -= num_pos
396     dx /= N
397     return loss, dx
398
399
400 def softmax_loss(x, y):

```

```

401 """
402 Computes the loss and gradient for softmax classification.
403
404 Inputs:
405 - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
406     for the ith input.
407 - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
408     0 <= y[i] < C
409
410 Returns a tuple of:
411 - loss: Scalar giving the loss
412 - dx: Gradient of the loss with respect to x
413 """
414
415 probs = np.exp(x - np.max(x, axis=1, keepdims=True))
416 probs /= np.sum(probs, axis=1, keepdims=True)
417 N = x.shape[0]
418 loss = -np.sum(np.log(np.maximum(probs[np.arange(N), y], 1e-8))) / N
419 dx = probs.copy()
420 dx[np.arange(N), y] -= 1
421 dx /= N
422 return loss, dx
423

```