

1 layers.py

```
1 from builtins import range
2 import numpy as np
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  $\bar{D} = d_1 * \dots * d_k$ , 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    # TODO: Implement the affine forward pass. Store the result in out. You #
26    # will need to reshape the input into rows. #
27    #####
28    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
29
30    N = x.shape[0]
31    x_row = np.reshape(x, (N, -1)) # Reshape x into (N, D) matrix
32    out = x_row.dot(w) + b
33
34    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
35    #####
36    #                               END OF YOUR CODE                               #
37    #####
38    cache = (x, w, b)
39    return out, cache
40
41
42 def affine_backward(dout, cache):
43     """
44     Computes the backward pass for an affine layer.
45
46     Inputs:
47     - dout: Upstream derivative, of shape (N, M)
48     - cache: Tuple of:
49       - x: Input data, of shape (N, d_1, ..., d_k)
50       - w: Weights, of shape (D, M)
51       - b: Biases, of shape (M,)
52
53     Returns a tuple of:
54     - dx: Gradient with respect to x, of shape (N, d_1, ..., d_k)
55     - dw: Gradient with respect to w, of shape (D, M)
56     - db: Gradient with respect to b, of shape (M,)
57     """
58    x, w, b = cache
59    dx, dw, db = None, None, None
60    #####
61    # TODO: Implement the affine backward pass. #
62    #####
63    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
64
65    N = x.shape[0]
66
67    x_row = np.reshape(x, (N, -1))
68
69    # Based on the shape of dx, dw and db, can get the calculation formula
70    dx = dout.dot(w.T).reshape(x.shape)
71    dw = x_row.T.dot(dout)
72    db = np.sum(dout, axis = 0)
73
```

```

74 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
75 #####
76 #                                     END OF YOUR CODE                                     #
77 #####
78 return dx, dw, db
79
80
81 def relu_forward(x):
82     """
83     Computes the forward pass for a layer of rectified linear units (ReLU).
84
85     Input:
86     - x: Inputs, of any shape
87
88     Returns a tuple of:
89     - out: Output, of the same shape as x
90     - cache: x
91     """
92     out = None
93     #####
94     # TODO: Implement the ReLU forward pass.                                     #
95     #####
96     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
97
98     # Must truly copy the variables into new variables
99     out = x.copy()
100     out[out < 0] = 0
101
102
103     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
104     #####
105     #                                     END OF YOUR CODE                                     #
106     #####
107     cache = x
108     return out, cache
109
110
111 def relu_backward(dout, cache):
112     """
113     Computes the backward pass for a layer of rectified linear units (ReLU).
114
115     Input:
116     - dout: Upstream derivatives, of any shape
117     - cache: Input x, of same shape as dout
118
119     Returns:
120     - dx: Gradient with respect to x
121     """
122     dx, x = None, cache
123     #####
124     # TODO: Implement the ReLU backward pass.                                     #
125     #####
126     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
127
128     dx = x
129     dx[dx < 0] = 0
130     dx[dx > 0] = 1
131     dx = np.multiply(dx, dout)
132
133
134     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
135     #####
136     #                                     END OF YOUR CODE                                     #
137     #####
138     return dx
139
140
141 def batchnorm_forward(x, gamma, beta, bn_param):
142     """
143     Forward pass for batch normalization.
144
145     During training the sample mean and (uncorrected) sample variance are
146     computed from minibatch statistics and used to normalize the incoming data.
147     During training we also keep an exponentially decaying running mean of the
148     mean and variance of each feature, and these averages are used to normalize
149     data at test-time.

```

```

150
151 At each timestep we update the running averages for mean and variance using
152 an exponential decay based on the momentum parameter:
153
154 running_mean = momentum * running_mean + (1 - momentum) * sample_mean
155 running_var = momentum * running_var + (1 - momentum) * sample_var
156
157 Note that the batch normalization paper suggests a different test-time
158 behavior: they compute sample mean and variance for each feature using a
159 large number of training images rather than using a running average. For
160 this implementation we have chosen to use running averages instead since
161 they do not require an additional estimation step; the torch7
162 implementation of batch normalization also uses running averages.
163
164 Input:
165 - x: Data of shape (N, D)
166 - gamma: Scale parameter of shape (D,)
167 - beta: Shift parameter of shape (D,)
168 - bn_param: Dictionary with the following keys:
169   - mode: 'train' or 'test'; required
170   - eps: Constant for numeric stability
171   - momentum: Constant for running mean / variance.
172   - running_mean: Array of shape (D,) giving running mean of features
173   - running_var: Array of shape (D,) giving running variance of features
174
175 Returns a tuple of:
176 - out: of shape (N, D)
177 - cache: A tuple of values needed in the backward pass
178 """
179 mode = bn_param['mode']
180 eps = bn_param.get('eps', 1e-5)
181 momentum = bn_param.get('momentum', 0.9)
182
183 N, D = x.shape
184 running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
185 running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))
186
187 out, cache = None, None
188 if mode == 'train':
189     #####
190     # TODO: Implement the training-time forward pass for batch norm.      #
191     # Use minibatch statistics to compute the mean and variance, use      #
192     # these statistics to normalize the incoming data, and scale and      #
193     # shift the normalized data using gamma and beta.                      #
194     #                                                                      #
195     # You should store the output in the variable out. Any intermediates  #
196     # that you need for the backward pass should be stored in the cache   #
197     # variable.                                                            #
198     #                                                                      #
199     # You should also use your computed sample mean and variance together #
200     # with the momentum variable to update the running mean and running  #
201     # variance, storing your result in the running_mean and running_var   #
202     # variables.                                                           #
203     #                                                                      #
204     # Note that though you should be keeping track of the running         #
205     # variance, you should normalize the data based on the standard       #
206     # deviation (square root of variance) instead!                       #
207     # Referencing the original paper (https://arxiv.org/abs/1502.03167)  #
208     # might prove to be helpful.                                         #
209     #####
210     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
211
212     sample_mean = np.mean(x, axis = 0)
213     sample_var = np.var(x, axis = 0)
214     running_mean = momentum * running_mean + (1 - momentum) * sample_mean
215     running_var = momentum * running_var + (1 - momentum) * sample_var
216     sample_normalized = (x - sample_mean) / np.sqrt(sample_var + eps)
217     out = gamma * sample_normalized + beta
218     cache = (sample_normalized, gamma, beta, sample_mean, sample_var, x, eps)
219
220     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
221     #####
222     #                               END OF YOUR CODE                       #
223     #####
224 elif mode == 'test':
225     #####

```

```

226 # TODO: Implement the test-time forward pass for batch normalization. #
227 # Use the running mean and variance to normalize the incoming data, #
228 # then scale and shift the normalized data using gamma and beta. #
229 # Store the result in the out variable. #
230 #####
231 # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
232
233 sample_normalized = (x - running_mean) / np.sqrt(running_var + eps)
234 out = gamma * sample_normalized + beta
235
236 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
237 #####
238 #                               END OF YOUR CODE                               #
239 #####
240 else:
241     raise ValueError('Invalid forward batchnorm mode "%s"' % mode)
242
243 # Store the updated running means back into bn_param
244 bn_param['running_mean'] = running_mean
245 bn_param['running_var'] = running_var
246
247 return out, cache
248
249
250 def batchnorm_backward(dout, cache):
251     """
252     Backward pass for batch normalization.
253
254     For this implementation, you should write out a computation graph for
255     batch normalization on paper and propagate gradients backward through
256     intermediate nodes.
257
258     Inputs:
259     - dout: Upstream derivatives, of shape (N, D)
260     - cache: Variable of intermediates from batchnorm_forward.
261
262     Returns a tuple of:
263     - dx: Gradient with respect to inputs x, of shape (N, D)
264     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
265     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
266     """
267     dx, dgamma, dbeta = None, None, None
268     #####
269     # TODO: Implement the backward pass for batch normalization. Store the #
270     # results in the dx, dgamma, and dbeta variables. #
271     # Referencing the original paper (https://arxiv.org/abs/1502.03167) #
272     # might prove to be helpful. #
273     #####
274     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
275
276     sample_normalized, gamma, beta, sample_mean, sample_var, x, eps = cache
277     N = x.shape[0]
278     dx_hat = dout * gamma
279     dvar = np.sum(dx_hat * (x - sample_mean) * (-1 / 2) * (sample_var + eps)**(-3 / 2), axis = 0)
280     dmean = np.sum(np.divide(-dx_hat, np.sqrt(sample_var + eps)), axis = 0) + dvar * np.sum(-2 * (x -
281     sample_mean), axis = 0) / N
282     dx = dx_hat / np.sqrt(sample_var + eps) + dvar * 2 * (x - sample_mean) / N + dmean / N
283     dgamma = np.sum(dout * sample_normalized, axis = 0)
284     dbeta = np.sum(dout, axis = 0)
285
286     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
287     #####
288     #                               END OF YOUR CODE                               #
289     #####
290
291     return dx, dgamma, dbeta
292
293
294 def batchnorm_backward_alt(dout, cache):
295     """
296     Alternative backward pass for batch normalization.
297
298     For this implementation you should work out the derivatives for the batch
299     normalizaton backward pass on paper and simplify as much as possible. You
300     should be able to derive a simple expression for the backward pass.

```

```

301 See the jupyter notebook for more hints.
302
303 Note: This implementation should expect to receive the same cache variable
304 as batchnorm_backward, but might not use all of the values in the cache.
305
306 Inputs / outputs: Same as batchnorm_backward
307 """
308 dx, dgamma, dbeta = None, None, None
309 #####
310 # TODO: Implement the backward pass for batch normalization. Store the #
311 # results in the dx, dgamma, and dbeta variables. #
312 # #
313 # After computing the gradient with respect to the centered inputs, you #
314 # should be able to compute gradients with respect to the inputs in a #
315 # single statement; our implementation fits on a single 80-character line.#
316 #####
317 # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
318
319 sample_normalized, gamma, beta, sample_mean, sample_var, x, eps = cache
320 N = x.shape[0]
321 sigma = np.sqrt(sample_var + eps)
322
323
324 dgamma = np.sum(dout * sample_normalized, axis = 0)
325 dbeta = np.sum(dout, axis = 0)
326
327 dx = (1 / N) * gamma * 1/sigma * ((N * dout) - np.sum(dout, axis=0) -
328                                     (x - sample_mean) * np.square(1/sigma) * np.sum(dout * (x -
329 sample_mean), axis=0))
330
331 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
332 #####
333 # END OF YOUR CODE #
334 #####
335
336 return dx, dgamma, dbeta
337
338
339 def layernorm_forward(x, gamma, beta, ln_param):
340     """
341     Forward pass for layer normalization.
342
343     During both training and test-time, the incoming data is normalized per data-point,
344     before being scaled by gamma and beta parameters identical to that of batch normalization.
345
346     Note that in contrast to batch normalization, the behavior during train and test-time for
347     layer normalization are identical, and we do not need to keep track of running averages
348     of any sort.
349
350     Input:
351     - x: Data of shape (N, D)
352     - gamma: Scale parameter of shape (D,)
353     - beta: Shift parameter of shape (D,)
354     - ln_param: Dictionary with the following keys:
355         - eps: Constant for numeric stability
356
357     Returns a tuple of:
358     - out: of shape (N, D)
359     - cache: A tuple of values needed in the backward pass
360     """
361     out, cache = None, None
362     eps = ln_param.get('eps', 1e-5)
363     #####
364     # TODO: Implement the training-time forward pass for layer norm. #
365     # Normalize the incoming data, and scale and shift the normalized data #
366     # using gamma and beta. #
367     # HINT: this can be done by slightly modifying your training-time #
368     # implementation of batch normalization, and inserting a line or two of #
369     # well-placed code. In particular, can you think of any matrix #
370     # transformations you could perform, that would enable you to copy over #
371     # the batch norm code and leave it almost unchanged? #
372     #####
373     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
374
375     # Transpose x so that the dimension of x becomes (D, N), following calculation can be the same as

```

```

376 batch_forward
377 x = x.T
378
379 sample_mean = np.mean(x, axis = 0)
380 sample_var = np.var(x, axis = 0)
381 sample_normalized = (x - sample_mean) / np.sqrt(sample_var + eps)
382
383 # Transpose sample_normalized so that the result can has the correct dimension (N, D)
384 sample_normalized = sample_normalized.T
385 out = gamma * sample_normalized + beta
386
387 # Transpose x again so that x is restored
388 x = x.T
389
390 cache = (sample_normalized, gamma, beta, sample_mean, sample_var, x, eps)
391
392 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
393 #####
394 #                               END OF YOUR CODE                               #
395 #####
396 return out, cache
397
398
399 def layernorm_backward(dout, cache):
400     """
401     Backward pass for layer normalization.
402
403     For this implementation, you can heavily rely on the work you've done already
404     for batch normalization.
405
406     Inputs:
407     - dout: Upstream derivatives, of shape (N, D)
408     - cache: Variable of intermediates from layernorm_forward.
409
410     Returns a tuple of:
411     - dx: Gradient with respect to inputs x, of shape (N, D)
412     - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
413     - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
414     """
415     dx, dgamma, dbeta = None, None, None
416     #####
417     # TODO: Implement the backward pass for layer norm.                                #
418     #                                                                                      #
419     # HINT: this can be done by slightly modifying your training-time                    #
420     # implementation of batch normalization. The hints to the forward pass              #
421     # still apply!                                                                      #
422     #####
423     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
424
425     sample_normalized, gamma, beta, sample_mean, sample_var, x, eps = cache
426
427     # The calculation of dgamma and dbeta remain the same
428     dgamma = np.sum(dout * sample_normalized, axis = 0)
429     dbeta = np.sum(dout, axis = 0)
430
431     dx_hat = dout * gamma
432
433     # At first transpose sample_normalized, x and dx_hat so that their dimensions are all (D, N) now
434     sample_normalized = sample_normalized.T
435     x = x.T
436     dx_hat = dx_hat.T
437
438     # Actually x.shape[0] should be D now, but I still use N so that the code below don't have to be
439     # changed.
440     N = x.shape[0]
441
442     # The following calculation can be the same as they are in batchnorm_backward
443     dvar = np.sum(dx_hat * (x - sample_mean) * (-1 / 2) * (sample_var + eps)**(-3 / 2), axis = 0)
444     dmean = np.sum(np.divide(-dx_hat, np.sqrt(sample_var + eps)), axis = 0) + dvar * np.sum(-2 * (x -
445     sample_mean), axis = 0) / N
446     dx = dx_hat / np.sqrt(sample_var + eps) + dvar * 2 * (x - sample_mean) / N + dmean / N
447
448     # Transpose dx so that dx can have the correct dimension (N, D) now
449     dx = dx.T

```

```

449
450
451
452 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
453 #####
454 #                                     END OF YOUR CODE                                     #
455 #####
456 return dx, dgamma, dbeta
457
458
459 def dropout_forward(x, dropout_param):
460     """
461     Performs the forward pass for (inverted) dropout.
462
463     Inputs:
464     - x: Input data, of any shape
465     - dropout_param: A dictionary with the following keys:
466       - p: Dropout parameter. We keep each neuron output with probability p.
467       - mode: 'test' or 'train'. If the mode is train, then perform dropout;
468         if the mode is test, then just return the input.
469       - seed: Seed for the random number generator. Passing seed makes this
470         function deterministic, which is needed for gradient checking but not
471         in real networks.
472
473     Outputs:
474     - out: Array of the same shape as x.
475     - cache: tuple (dropout_param, mask). In training mode, mask is the dropout
476       mask that was used to multiply the input; in test mode, mask is None.
477
478     NOTE: Please implement **inverted** dropout, not the vanilla version of dropout.
479     See http://cs231n.github.io/neural-networks-2/#reg for more details.
480
481     NOTE 2: Keep in mind that p is the probability of **keep** a neuron
482     output; this might be contrary to some sources, where it is referred to
483     as the probability of dropping a neuron output.
484     """
485     p, mode = dropout_param['p'], dropout_param['mode']
486     if 'seed' in dropout_param:
487         np.random.seed(dropout_param['seed'])
488
489     mask = None
490     out = None
491
492     if mode == 'train':
493         #####
494         # TODO: Implement training phase forward pass for inverted dropout. #
495         # Store the dropout mask in the mask variable. #
496         #####
497         # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
498
499         mask = (np.random.rand(*x.shape) < p) / p
500         out = x * mask
501
502         # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
503         #####
504         #                                     END OF YOUR CODE                                     #
505         #####
506     elif mode == 'test':
507         #####
508         # TODO: Implement the test phase forward pass for inverted dropout. #
509         #####
510         # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
511
512         out = x
513
514         # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
515         #####
516         #                                     END OF YOUR CODE                                     #
517         #####
518
519     cache = (dropout_param, mask)
520     out = out.astype(x.dtype, copy=False)
521
522     return out, cache
523
524

```

```

525 def dropout_backward(dout, cache):
526     """
527     Perform the backward pass for (inverted) dropout.
528
529     Inputs:
530     - dout: Upstream derivatives, of any shape
531     - cache: (dropout_param, mask) from dropout_forward.
532     """
533     dropout_param, mask = cache
534     mode = dropout_param['mode']
535
536     dx = None
537     if mode == 'train':
538         #####
539         # TODO: Implement training phase backward pass for inverted dropout #
540         #####
541         # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
542
543         dx = dout * mask
544
545         # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
546         #####
547         #                               END OF YOUR CODE                               #
548         #####
549     elif mode == 'test':
550         dx = dout
551     return dx
552
553
554 def conv_forward_naive(x, w, b, conv_param):
555     """
556     A naive implementation of the forward pass for a convolutional layer.
557
558     The input consists of N data points, each with C channels, height H and
559     width W. We convolve each input with F different filters, where each filter
560     spans all C channels and has height HH and width WW.
561
562     Input:
563     - x: Input data of shape (N, C, H, W)
564     - w: Filter weights of shape (F, C, HH, WW)
565     - b: Biases, of shape (F,)
566     - conv_param: A dictionary with the following keys:
567       - 'stride': The number of pixels between adjacent receptive fields in the
568         horizontal and vertical directions.
569       - 'pad': The number of pixels that will be used to zero-pad the input.
570
571
572     During padding, 'pad' zeros should be placed symmetrically (i.e equally on both sides)
573     along the height and width axes of the input. Be careful not to modify the original
574     input x directly.
575
576     Returns a tuple of:
577     - out: Output data, of shape (N, F, H', W') where H' and W' are given by
578        $H' = 1 + (H + 2 * pad - HH) / stride$ 
579        $W' = 1 + (W + 2 * pad - WW) / stride$ 
580     - cache: (x, w, b, conv_param)
581     """
582     out = None
583     #####
584     # TODO: Implement the convolutional forward pass. #
585     # Hint: you can use the function np.pad for padding. #
586     #####
587     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
588
589     stride = conv_param['stride']
590     pad = conv_param['pad']
591     x_pad = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), 'constant')
592     N, C, H, W = x.shape
593     F, C, HH, WW = w.shape
594
595     H_out = int(1 + (H + 2 * pad - HH) / stride)
596     W_out = int(1 + (W + 2 * pad - WW) / stride)
597
598     out = np.zeros((N, F, H_out, W_out))
599
600     for n in range(N):

```



```

601         for f in range(F):
602             for i in range(H_out):
603                 for j in range(W_out):
604                     out[n, f, i, j] = np.sum(x_pad[n, :, i * stride: i * stride + HH, j * stride: j *
stride + WW] * w[f]) + b[f]
605
606     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
607     #####
608     #                                     END OF YOUR CODE                                     #
609     #####
610     cache = (x, w, b, conv_param)
611     return out, cache
612
613
614 def conv_backward_naive(dout, cache):
615     """
616     A naive implementation of the backward pass for a convolutional layer.
617
618     Inputs:
619     - dout: Upstream derivatives.
620     - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive
621
622     Returns a tuple of:
623     - dx: Gradient with respect to x
624     - dw: Gradient with respect to w
625     - db: Gradient with respect to b
626     """
627     dx, dw, db = None, None, None
628     #####
629     # TODO: Implement the convolutional backward pass.                                     #
630     #####
631     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
632
633     x, w, b, conv_param = cache
634     pad = conv_param['pad']
635     stride = conv_param['stride']
636     F, C, HH, WW = w.shape
637     N, C, H, W = x.shape
638     H_out = int(1 + (H + 2 * pad - HH) / stride)
639     W_out = int(1 + (W + 2 * pad - WW) / stride)
640     x_pad = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), 'constant')
641
642     dx_pad = np.zeros_like(x_pad)
643     dw = np.zeros_like(w)
644     db = np.zeros_like(b)
645
646     # To calculate db, just sum up all the upstream gradients for each filters bias.
647     for f in range(F):
648         db[f] = np.sum(dout[:, f, :, :])
649
650     for n in range(N):
651         for f in range(F):
652             for i in range(H_out):
653                 for j in range(W_out):
654                     # According to chain rule, dw = dout * x, dx = dout * w. Be careful about the
dimension
655                     dw[f] += dout[n, f, i, j] * x_pad[n, :, i * stride: i * stride + HH, j * stride: j *
stride + WW]
656                     dx_pad[n, :, i * stride: i * stride + HH, j * stride: j * stride + WW] += dout[n, f, i
, j] * w[f]
657
658     # Get rid of the pad around dx
659     dx = dx_pad[:, :, pad: pad+H, pad: pad+W]
660
661     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
662     #####
663     #                                     END OF YOUR CODE                                     #
664     #####
665     return dx, dw, db
666
667
668 def max_pool_forward_naive(x, pool_param):
669     """
670     A naive implementation of the forward pass for a max-pooling layer.
671
672     Inputs:

```

```

673 - x: Input data, of shape (N, C, H, W)
674 - pool_param: dictionary with the following keys:
675   - 'pool_height': The height of each pooling region
676   - 'pool_width': The width of each pooling region
677   - 'stride': The distance between adjacent pooling regions
678
679 No padding is necessary here. Output size is given by
680
681 Returns a tuple of:
682 - out: Output data, of shape (N, C, H', W') where H' and W' are given by
683    $H' = 1 + (H - \text{pool\_height}) / \text{stride}$ 
684    $W' = 1 + (W - \text{pool\_width}) / \text{stride}$ 
685 - cache: (x, pool_param)
686 """
687 out = None
688 #####
689 # TODO: Implement the max-pooling forward pass #
690 #####
691 # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
692
693 pool_height = pool_param['pool_height']
694 pool_width = pool_param['pool_width']
695 stride = pool_param['stride']
696 N, C, H, W = x.shape
697
698 H_out = int(1 + (H - pool_height) / stride)
699 W_out = int(1 + (W - pool_width) / stride)
700
701 out = np.zeros((N, C, H_out, W_out))
702
703 for n in range(N):
704     for c in range(C):
705         for i in range(H_out):
706             for j in range(W_out):
707                 out[n, c, i, j] = np.max(x[n, c, i * stride: i * stride + pool_height, j * stride: j *
stride + pool_width])
708
709
710 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
711 #####
712 #                               END OF YOUR CODE                               #
713 #####
714 cache = (x, pool_param)
715 return out, cache
716
717
718 def max_pool_backward_naive(dout, cache):
719     """
720     A naive implementation of the backward pass for a max-pooling layer.
721
722     Inputs:
723     - dout: Upstream derivatives
724     - cache: A tuple of (x, pool_param) as in the forward pass.
725
726     Returns:
727     - dx: Gradient with respect to x
728     """
729     dx = None
730     #####
731     # TODO: Implement the max-pooling backward pass #
732     #####
733     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
734
735     x, pool_param = cache
736     pool_height = pool_param['pool_height']
737     pool_width = pool_param['pool_width']
738     stride = pool_param['stride']
739     N, C, H, W = x.shape
740
741     H_out = int(1 + (H - pool_height) / stride)
742     W_out = int(1 + (W - pool_width) / stride)
743
744     dx = np.zeros_like(x)
745
746     for n in range(N):
747         for c in range(C):

```

```

748         for i in range(H_out):
749             for j in range(W_out):
750                 block = x[n, c, i * stride: i * stride + pool_height, j * stride: j * stride +
pool_width]
751                 maximum = np.max(block)
752                 dx[n, c, i * stride: i * stride + pool_height, j * stride: j * stride + pool_width] =
\
753                                                                 (block == maximum) * dout[
n, c, i, j]
754
755     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
756     #####
757     #                                     END OF YOUR CODE                                     #
758     #####
759     return dx
760
761 def spatial_batchnorm_forward(x, gamma, beta, bn_param):
762     """
763     Computes the forward pass for spatial batch normalization.
764
765     Inputs:
766     - x: Input data of shape (N, C, H, W)
767     - gamma: Scale parameter, of shape (C,)
768     - beta: Shift parameter, of shape (C,)
769     - bn_param: Dictionary with the following keys:
770         - mode: 'train' or 'test'; required
771         - eps: Constant for numeric stability
772         - momentum: Constant for running mean / variance. momentum=0 means that
773           old information is discarded completely at every time step, while
774           momentum=1 means that new information is never incorporated. The
775           default of momentum=0.9 should work well in most situations.
776         - running_mean: Array of shape (D,) giving running mean of features
777         - running_var: Array of shape (D,) giving running variance of features
778
779     Returns a tuple of:
780     - out: Output data, of shape (N, C, H, W)
781     - cache: Values needed for the backward pass
782     """
783     out, cache = None, None
784
785     #####
786     # TODO: Implement the forward pass for spatial batch normalization.           #
787     #                                                                           #
788     # HINT: You can implement spatial batch normalization by calling the         #
789     # vanilla version of batch normalization you implemented above.             #
790     # Your implementation should be very short; ours is less than five lines.   #
791     #####
792     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
793
794     N, C, H, W = x.shape
795
796     # Transpose x so that the shape is (N, H, W, C), then reshape x into (N * H * W, C)
797     x_new = np.reshape(x.transpose(0, 2, 3, 1), (N * H * W, C))
798     out, cache = batchnorm_forward(x_new, gamma, beta, bn_param)
799
800     # Modify the final output
801     out = np.transpose(out.reshape(N, H, W, C), (0, 3, 1, 2))
802
803     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
804     #####
805     #                                     END OF YOUR CODE                                     #
806     #####
807
808     return out, cache
809
810
811 def spatial_batchnorm_backward(dout, cache):
812     """
813     Computes the backward pass for spatial batch normalization.
814
815     Inputs:
816     - dout: Upstream derivatives, of shape (N, C, H, W)
817     - cache: Values from the forward pass
818
819     """
820

```

```

821 Returns a tuple of:
822 - dx: Gradient with respect to inputs, of shape (N, C, H, W)
823 - dgamma: Gradient with respect to scale parameter, of shape (C,)
824 - dbeta: Gradient with respect to shift parameter, of shape (C,)
825 """
826 dx, dgamma, dbeta = None, None, None
827
828 #####
829 # TODO: Implement the backward pass for spatial batch normalization. #
830 # #
831 # HINT: You can implement spatial batch normalization by calling the #
832 # vanilla version of batch normalization you implemented above. #
833 # Your implementation should be very short; ours is less than five lines. #
834 #####
835 # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
836
837
838 N, C, H, W = dout.shape
839 dout_new = np.reshape(dout.transpose(0, 2, 3, 1), (N * H * W, C))
840 dx, dgamma, dbeta = batchnorm_backward(dout_new, cache)
841 dx = np.transpose(dx.reshape(N, H, W, C), (0, 3, 1, 2))
842
843 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
844 #####
845 #                               END OF YOUR CODE                               #
846 #####
847
848 return dx, dgamma, dbeta
849
850
851 def spatial_groupnorm_forward(x, gamma, beta, G, gn_param):
852     """
853     Computes the forward pass for spatial group normalization.
854     In contrast to layer normalization, group normalization splits each entry
855     in the data into G contiguous pieces, which it then normalizes independently.
856     Per feature shifting and scaling are then applied to the data, in a manner identical to that of batch
857     normalization and layer normalization.
858
859     Inputs:
860     - x: Input data of shape (N, C, H, W)
861     - gamma: Scale parameter, of shape (C,)
862     - beta: Shift parameter, of shape (C,)
863     - G: Integer number of groups to split into, should be a divisor of C
864     - gn_param: Dictionary with the following keys:
865         - eps: Constant for numeric stability
866
867     Returns a tuple of:
868     - out: Output data, of shape (N, C, H, W)
869     - cache: Values needed for the backward pass
870     """
871     out, cache = None, None
872     eps = gn_param.get('eps', 1e-5)
873     #####
874     # TODO: Implement the forward pass for spatial group normalization. #
875     # This will be extremely similar to the layer norm implementation. #
876     # In particular, think about how you could transform the matrix so that #
877     # the bulk of the code is similar to both train-time batch normalization #
878     # and layer normalization! #
879     #####
880     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
881
882     N, C, H, W = x.shape
883
884     # Just reshape x so that the number of group is multiplied by G while C is divided by G
885     x = np.reshape(x, (N * G, C // G * H * W))
886
887     # Other code are basically copied from layer norm implementation
888     x = x.T
889     sample_mean = np.mean(x, axis = 0)
890     sample_var = np.var(x, axis = 0)
891     sample_normalized = (x - sample_mean) / np.sqrt(sample_var + eps)
892
893     sample_normalized = np.reshape(sample_normalized.T, (N, C, H, W))
894     out = gamma * sample_normalized + beta
895     x = np.reshape(x.T, (N, C, H, W))

```

```

896 cache = (sample_normalized, gamma, beta, sample_mean, sample_var, x, eps, G)
897
898
899
900
901 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
902 #####
903 #                                     END OF YOUR CODE                                     #
904 #####
905 return out, cache
906
907
908 def spatial_groupnorm_backward(dout, cache):
909     """
910     Computes the backward pass for spatial group normalization.
911
912     Inputs:
913     - dout: Upstream derivatives, of shape (N, C, H, W)
914     - cache: Values from the forward pass
915
916     Returns a tuple of:
917     - dx: Gradient with respect to inputs, of shape (N, C, H, W)
918     - dgamma: Gradient with respect to scale parameter, of shape (C,)
919     - dbeta: Gradient with respect to shift parameter, of shape (C,)
920     """
921     dx, dgamma, dbeta = None, None, None
922
923     #####
924     # TODO: Implement the backward pass for spatial group normalization.      #
925     # This will be extremely similar to the layer norm implementation.        #
926     #####
927     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
928
929     sample_normalized, gamma, beta, sample_mean, sample_var, x, eps, G = cache
930     N, C, H, W = x.shape
931
932     # Remember to keep dimensions of dgamma and dbeta unchanged
933     dgamma = np.sum(dout * sample_normalized, axis = (0, 2, 3), keepdims=True)
934     dbeta = np.sum(dout, axis = (0, 2, 3), keepdims=True)
935     dx_hat = dout * gamma
936
937     # Reshape those matrices at first, then transpose them
938     sample_normalized = np.reshape(sample_normalized, (N * G, C // G * H * W))
939     x = np.reshape(x, (N * G, C // G * H * W))
940     dx_hat = np.reshape(dx_hat, (N * G, C // G * H * W))
941
942     sample_normalized = sample_normalized.T
943     x = x.T
944     dx_hat = dx_hat.T
945
946     # The following calculation is quite similar to layer norm backward
947     N_new = x.shape[0]
948
949     dvar = np.sum(dx_hat * (x - sample_mean) * (-1 / 2) * (sample_var + eps)**(-3 / 2), axis = 0)
950     dmean = np.sum(np.divide(-dx_hat, np.sqrt(sample_var + eps)), axis = 0) + \
951         dvar * np.sum(-2 * (x - sample_mean), axis = 0) / N_new
952     dx = dx_hat / np.sqrt(sample_var + eps) + dvar * 2 * (x - sample_mean) / N_new + dmean / N_new
953
954     dx = np.reshape(dx.T, (N, C, H, W))
955
956     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
957     #####
958     #                                     END OF YOUR CODE                                     #
959     #####
960     return dx, dgamma, dbeta
961
962
963 def svm_loss(x, y):
964     """
965     Computes the loss and gradient using for multiclass SVM classification.
966
967     Inputs:
968     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
969         class for the ith input.
970     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
971         0 <= y[i] < C

```

```

972     Returns a tuple of:
973     - loss: Scalar giving the loss
974     - dx: Gradient of the loss with respect to x
975     """
976     N = x.shape[0]
977     correct_class_scores = x[np.arange(N), y]
978     margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
979     margins[np.arange(N), y] = 0
980     loss = np.sum(margins) / N
981     num_pos = np.sum(margins > 0, axis=1)
982     dx = np.zeros_like(x)
983     dx[margins > 0] = 1
984     dx[np.arange(N), y] -= num_pos
985     dx /= N
986     return loss, dx
987
988
989 def softmax_loss(x, y):
990     """
991     Computes the loss and gradient for softmax classification.
992
993     Inputs:
994     - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
995         class for the ith input.
996     - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
997         0 <= y[i] < C
998
999     Returns a tuple of:
1000     - loss: Scalar giving the loss
1001     - dx: Gradient of the loss with respect to x
1002     """
1003     shifted_logits = x - np.max(x, axis=1, keepdims=True)
1004     Z = np.sum(np.exp(shifted_logits), axis=1, keepdims=True)
1005     log_probs = shifted_logits - np.log(Z)
1006     probs = np.exp(log_probs)
1007     N = x.shape[0]
1008     loss = -np.sum(log_probs[np.arange(N), y]) / N
1009     dx = probs.copy()
1010     dx[np.arange(N), y] -= 1
1011     dx /= N
1012     return loss, dx
1013

```

2 fc_net.py

```
1 from builtins import range
2 from builtins import object
3 import numpy as np
4
5 from cs231n.layers import *
6 from cs231n.layer_utils import *
7
8
9 class TwoLayerNet(object):
10     """
11     A two-layer fully-connected neural network with ReLU nonlinearity and
12     softmax loss that uses a modular layer design. We assume an input dimension
13     of D, a hidden dimension of H, and perform classification over C classes.
14
15     The architecture should be affine - relu - affine - softmax.
16
17     Note that this class does not implement gradient descent; instead, it
18     will interact with a separate Solver object that is responsible for running
19     optimization.
20
21     The learnable parameters of the model are stored in the dictionary
22     self.params that maps parameter names to numpy arrays.
23     """
24
25     def __init__(self, input_dim=3*32*32, hidden_dim=100, num_classes=10,
26                 weight_scale=1e-3, reg=0.0):
27         """
28         Initialize a new network.
29
30         Inputs:
31         - input_dim: An integer giving the size of the input
32         - hidden_dim: An integer giving the size of the hidden layer
33         - num_classes: An integer giving the number of classes to classify
34         - weight_scale: Scalar giving the standard deviation for random
35           initialization of the weights.
36         - reg: Scalar giving L2 regularization strength.
37         """
38         self.params = {}
39         self.reg = reg
40
41         #####
42         # TODO: Initialize the weights and biases of the two-layer net. Weights #
43         # should be initialized from a Gaussian centered at 0.0 with #
44         # standard deviation equal to weight_scale, and biases should be #
45         # initialized to zero. All weights and biases should be stored in the #
46         # dictionary self.params, with first layer weights #
47         # and biases using the keys 'W1' and 'b1' and second layer #
48         # weights and biases using the keys 'W2' and 'b2'. #
49         #####
50         # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
51
52         # Weights is initialized from a Gaussian centered at 0.0 with standard deviation equal to
53         # weight_scale
54         # Use np.random.normal function
55         self.params['W1'] = np.random.normal(0.0, weight_scale, (input_dim, hidden_dim))
56         self.params['b1'] = np.zeros((1, hidden_dim))
57         self.params['W2'] = np.random.normal(0.0, weight_scale, (hidden_dim, num_classes))
58         self.params['b2'] = np.zeros((1, num_classes))
59
60         # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
61         #####
62         # END OF YOUR CODE #
63         #####
64
65     def loss(self, X, y=None):
66         """
67         Compute loss and gradient for a minibatch of data.
68
69         Inputs:
70         - X: Array of input data of shape (N, d_1, ..., d_k)
71         - y: Array of labels, of shape (N,). y[i] gives the label for X[i].
72         """
```

```

73 Returns:
74 If y is None, then run a test-time forward pass of the model and return:
75 - scores: Array of shape (N, C) giving classification scores, where
76   scores[i, c] is the classification score for X[i] and class c.
77
78 If y is not None, then run a training-time forward and backward pass and
79 return a tuple of:
80 - loss: Scalar value giving the loss
81 - grads: Dictionary with the same keys as self.params, mapping parameter
82   names to gradients of the loss with respect to those parameters.
83 """
84 scores = None
85 #####
86 # TODO: Implement the forward pass for the two-layer net, computing the #
87 # class scores for X and storing them in the scores variable. #
88 #####
89 # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
90
91 W1, b1 = self.params['W1'], self.params['b1']
92 W2, b2 = self.params['W2'], self.params['b2']
93
94 # Firstly, do the affine_relu forward pass to get the hidden layer
95 hidden1, cache1 = affine_relu_forward(X, W1, b1)
96
97 # Secondly, do the affine forward pass
98 out, cache2 = affine_forward(hidden1, W2, b2)
99
100 scores = out
101
102 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
103 #####
104 #                               END OF YOUR CODE                               #
105 #####
106
107 # If y is None then we are in test mode so just return scores
108 if y is None:
109     return scores
110
111 loss, grads = 0, {}
112 #####
113 # TODO: Implement the backward pass for the two-layer net. Store the loss #
114 # in the loss variable and gradients in the grads dictionary. Compute data #
115 # loss using softmax, and make sure that grads[k] holds the gradients for #
116 # self.params[k]. Don't forget to add L2 regularization! #
117 # # #
118 # NOTE: To ensure that your implementation matches ours and you pass the #
119 # automated tests, make sure that your L2 regularization includes a factor #
120 # of 0.5 to simplify the expression for the gradient. #
121 #####
122 # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
123
124 # Use softmax to calculate the loss and gradient dout.
125 loss, dout = softmax_loss(scores, y)
126
127 # Based on the upstream gradient dout, use affine_backward to get the first downstream gradient.
128 dX2, dW2, db2 = affine_backward(dout, cache2)
129
130 # Based on the first downstream gradient dX2, use affine_relu_backward to get the second
131 # downstream gradient.
132 dX1, dW1, db1 = affine_relu_backward(dX2, cache1)
133
134 loss += 0.5 * self.reg * (np.sum(W1 * W1) + np.sum(W2 * W2))
135
136 dW2 += self.reg * W2
137 dW1 += self.reg * W1
138
139 grads['W1'] = dW1
140 grads['b1'] = db1
141 grads['W2'] = dW2
142 grads['b2'] = db2
143
144
145 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
146 #####
147 #                               END OF YOUR CODE                               #

```



```

148 #####
149
150     return loss, grads
151
152
153 class FullyConnectedNet(object):
154     """
155     A fully-connected neural network with an arbitrary number of hidden layers,
156     ReLU nonlinearities, and a softmax loss function. This will also implement
157     dropout and batch/layer normalization as options. For a network with L layers,
158     the architecture will be
159
160     {affine - [batch/layer norm] - relu - [dropout]} x (L - 1) - affine - softmax
161
162     where batch/layer normalization and dropout are optional, and the {...} block is
163     repeated L - 1 times.
164
165     Similar to the TwoLayerNet above, learnable parameters are stored in the
166     self.params dictionary and will be learned using the Solver class.
167     """
168
169     def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
170                 dropout=1, normalization=None, reg=0.0,
171                 weight_scale=1e-2, dtype=np.float32, seed=None):
172         """
173         Initialize a new FullyConnectedNet.
174
175         Inputs:
176         - hidden_dims: A list of integers giving the size of each hidden layer.
177         - input_dim: An integer giving the size of the input.
178         - num_classes: An integer giving the number of classes to classify.
179         - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=1 then
180           the network should not use dropout at all.
181         - normalization: What type of normalization the network should use. Valid values
182           are "batchnorm", "layernorm", or None for no normalization (the default).
183         - reg: Scalar giving L2 regularization strength.
184         - weight_scale: Scalar giving the standard deviation for random
185           initialization of the weights.
186         - dtype: A numpy datatype object; all computations will be performed using
187           this datatype. float32 is faster but less accurate, so you should use
188           float64 for numeric gradient checking.
189         - seed: If not None, then pass this random seed to the dropout layers. This
190           will make the dropout layers deterministic so we can gradient check the
191           model.
192         """
193         self.normalization = normalization
194         self.use_dropout = dropout != 1
195         self.reg = reg
196         self.num_layers = 1 + len(hidden_dims)
197         self.dtype = dtype
198         self.params = {}
199
200         #####
201         # TODO: Initialize the parameters of the network, storing all values in #
202         # the self.params dictionary. Store weights and biases for the first layer #
203         # in W1 and b1; for the second layer use W2 and b2, etc. Weights should be #
204         # initialized from a normal distribution centered at 0 with standard #
205         # deviation equal to weight_scale. Biases should be initialized to zero. #
206         # #
207         # When using batch normalization, store scale and shift parameters for the #
208         # first layer in gamma1 and beta1; for the second layer use gamma2 and #
209         # beta2, etc. Scale parameters should be initialized to ones and shift #
210         # parameters should be initialized to zeros. #
211         #####
212         # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
213
214         all_dims = [input_dim] + hidden_dims + [num_classes]
215         for i in range(len(all_dims) - 1):
216             self.params['W' + str(i + 1)] = np.random.normal(0.0, weight_scale, (all_dims[i], all_dims[i +
17]))
217             self.params['b' + str(i + 1)] = np.zeros((1, all_dims[i + 1]))
218
219             # If we haven't reached the final output layer, there may be a normalization layer
220             if i != self.num_layers - 1:
221                 if self.normalization != None:
222                     self.params['gamma' + str(i + 1)] = np.ones((1, all_dims[i + 1]))

```

```

223         self.params['beta' + str(i + 1)] = np.zeros((1, all_dims[i + 1]))
224
225     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
226     #####
227     #                               END OF YOUR CODE                               #
228     #####
229
230     # When using dropout we need to pass a dropout_param dictionary to each
231     # dropout layer so that the layer knows the dropout probability and the mode
232     # (train / test). You can pass the same dropout_param to each dropout layer.
233     self.dropout_param = {}
234     if self.use_dropout:
235         self.dropout_param = {'mode': 'train', 'p': dropout}
236         if seed is not None:
237             self.dropout_param['seed'] = seed
238
239     # With batch normalization we need to keep track of running means and
240     # variances, so we need to pass a special bn_param object to each batch
241     # normalization layer. You should pass self.bn_params[0] to the forward pass
242     # of the first batch normalization layer, self.bn_params[1] to the forward
243     # pass of the second batch normalization layer, etc.
244     self.bn_params = []
245     if self.normalization=='batchnorm':
246         self.bn_params = [{'mode': 'train'} for i in range(self.num_layers - 1)]
247     if self.normalization=='layernorm':
248         self.bn_params = [{} for i in range(self.num_layers - 1)]
249
250     # Cast all parameters to the correct datatype
251     for k, v in self.params.items():
252         self.params[k] = v.astype(dtype)
253
254
255 def loss(self, X, y=None):
256     """
257     Compute loss and gradient for the fully-connected net.
258
259     Input / output: Same as TwoLayerNet above.
260     """
261     X = X.astype(self.dtype)
262     mode = 'test' if y is None else 'train'
263
264     # Set train/test mode for batchnorm params and dropout param since they
265     # behave differently during training and testing.
266     if self.use_dropout:
267         self.dropout_param['mode'] = mode
268     if self.normalization=='batchnorm':
269         for bn_param in self.bn_params:
270             bn_param['mode'] = mode
271     scores = None
272     #####
273     # TODO: Implement the forward pass for the fully-connected net, computing #
274     # the class scores for X and storing them in the scores variable.         #
275     #                                                                           #
276     # When using dropout, you'll need to pass self.dropout_param to each    #
277     # dropout forward pass.                                                    #
278     #                                                                           #
279     # When using batch normalization, you'll need to pass self.bn_params[0] to #
280     # the forward pass for the first batch normalization layer, pass         #
281     # self.bn_params[1] to the forward pass for the second batch normalization #
282     # layer, etc.                                                              #
283     #####
284     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
285
286     affine_cache = {}
287     norm_cache = {}
288     relu_cache = {}
289     dropout_cache = {}
290
291     input_param = X
292
293     for i in range(self.num_layers - 1):
294         W, b = self.params['W' + str(i + 1)], self.params['b' + str(i + 1)]
295
296         if self.normalization == 'batchnorm':
297             # the first part is affine - batchnorm - relu
298             gamma, beta = self.params['gamma' + str(i + 1)], self.params['beta' + str(i + 1)]

```

```

299         affine_outcome, affine_cache[i + 1] = affine_forward(input_param, W, b)
300         norm_outcome, norm_cache[i + 1] = batchnorm_forward(affine_outcome, gamma, beta, self.
bn_params[i])
301         relu_outcome, relu_cache[i + 1] = relu_forward(norm_outcome)
302
303     elif self.normalization == 'layernorm':
304         # the first part is affine - layernorm - relu
305         gamma, beta = self.params['gamma' + str(i + 1)], self.params['beta' + str(i + 1)]
306         affine_outcome, affine_cache[i + 1] = affine_forward(input_param, W, b)
307         norm_outcome, norm_cache[i + 1] = layernorm_forward(affine_outcome, gamma, beta, self.
bn_params[i])
308         relu_outcome, relu_cache[i + 1] = relu_forward(norm_outcome)
309
310     else:
311         # the first part is affine - relu
312         relu_outcome, (affine_cache[i + 1], relu_cache[i + 1]) = affine_relu_forward(input_param,
W, b)
313
314     if self.use_dropout:
315         dropout_outcome, dropout_cache[i + 1] = dropout_forward(relu_outcome, self.dropout_param)
316
317     # Update input_param
318     input_param = dropout_outcome if self.use_dropout else relu_outcome
319
320     # Get the last layer
321     scores, last_cache = affine_forward(input_param,
322                                         self.params['W' + str(self.num_layers)],
323                                         self.params['b' + str(self.num_layers)])
324
325     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
326     #####
327     #                               END OF YOUR CODE                               #
328     #####
329
330     # If test mode return early
331     if mode == 'test':
332         return scores
333
334     loss, grads = 0.0, {}
335     #####
336     # TODO: Implement the backward pass for the fully-connected net. Store the #
337     # loss in the loss variable and gradients in the grads dictionary. Compute #
338     # data loss using softmax, and make sure that grads[k] holds the gradients #
339     # for self.params[k]. Don't forget to add L2 regularization!                #
340     #                                                                           #
341     # When using batch/layer normalization, you don't need to regularize the scale #
342     # and shift parameters.                                                    #
343     #                                                                           #
344     # NOTE: To ensure that your implementation matches ours and you pass the #
345     # automated tests, make sure that your L2 regularization includes a factor #
346     # of 0.5 to simplify the expression for the gradient.                    #
347     #####
348     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
349
350     # Use softmax to calculate the loss and gradient dout.
351     loss, dout = softmax_loss(scores, y)
352
353     reg_loss = 0
354     for i in range(self.num_layers):
355         reg_loss += 0.5 * self.reg * np.sum(np.square(self.params['W' + str(i+1)]))
356
357     loss += reg_loss
358
359     # Based on the upstream gradient dout, use affine_backward to get the first downstream gradient.
360     dX, dW, db = affine_backward(dout, last_cache)
361     dW += self.reg * self.params['W' + str(self.num_layers)]
362     grads['W' + str(self.num_layers)] = dW
363     grads['b' + str(self.num_layers)] = db
364
365     for i in range(self.num_layers - 1, 0, -1):
366         if self.use_dropout:
367             # If there is a dropout at the end
368             dX = dropout_backward(dX, dropout_cache[i])
369
370         dX = relu_backward(dX, relu_cache[i])
371

```

```

372     if self.normalization == 'batchnorm':
373         # If there is a batchnorm in the middle
374         dX, dgamma, dbeta = batchnorm_backward(dX, norm_cache[i])
375         grads['gamma'+str(i)] = dgamma
376         grads['beta'+str(i)] = dbeta
377     elif self.normalization == 'layernorm':
378         # If there is a layernorm in the middle
379         dX, dgamma, dbeta = layernorm_backward(dX, norm_cache[i])
380         grads['gamma'+str(i)] = dgamma
381         grads['beta'+str(i)] = dbeta
382
383     dX, dW, db = affine_backward(dX, affine_cache[i])
384
385     dW += self.reg * self.params['W' + str(i)]
386
387     grads['W'+str(i)] = dW
388     grads['b'+str(i)] = db
389
390
391     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
392     #####
393     #                                     END OF YOUR CODE                                     #
394     #####
395
396     return loss, grads

```

3 optim.py

```
1 import numpy as np
2
3 """
4 This file implements various first-order update rules that are commonly used
5 for training neural networks. Each update rule accepts current weights and the
6 gradient of the loss with respect to those weights and produces the next set of
7 weights. Each update rule has the same interface:
8
9 def update(w, dw, config=None):
10
11 Inputs:
12 - w: A numpy array giving the current weights.
13 - dw: A numpy array of the same shape as w giving the gradient of the
14     loss with respect to w.
15 - config: A dictionary containing hyperparameter values such as learning
16     rate, momentum, etc. If the update rule requires caching values over many
17     iterations, then config will also hold these cached values.
18
19 Returns:
20 - next_w: The next point after the update.
21 - config: The config dictionary to be passed to the next iteration of the
22     update rule.
23
24 NOTE: For most update rules, the default learning rate will probably not
25 perform well; however the default values of the other hyperparameters should
26 work well for a variety of different problems.
27
28 For efficiency, update rules may perform in-place updates, mutating w and
29 setting next_w equal to w.
30 """
31
32 def sgd(w, dw, config=None):
33     """
34     Performs vanilla stochastic gradient descent.
35
36     config format:
37     - learning_rate: Scalar learning rate.
38     """
39     if config is None: config = {}
40     config.setdefault('learning_rate', 1e-2)
41
42     w -= config['learning_rate'] * dw
43     return w, config
44
45 def sgd_momentum(w, dw, config=None):
46     """
47     Performs stochastic gradient descent with momentum.
48
49     config format:
50     - learning_rate: Scalar learning rate.
51     - momentum: Scalar between 0 and 1 giving the momentum value.
52         Setting momentum = 0 reduces to sgd.
53     - velocity: A numpy array of the same shape as w and dw used to store a
54         moving average of the gradients.
55     """
56     if config is None: config = {}
57     config.setdefault('learning_rate', 1e-2)
58     config.setdefault('momentum', 0.9)
59     v = config.get('velocity', np.zeros_like(w))
60
61     next_w = None
62     #####
63     # TODO: Implement the momentum update formula. Store the updated value in #
64     # the next_w variable. You should also use and update the velocity v. #
65     #####
66     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
67
68     v = config['momentum'] * v - config['learning_rate'] * dw
69     w += v
70     next_w = w
71
72
73
```

```

74 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
75 #####
76 #                                     END OF YOUR CODE                                     #
77 #####
78 config['velocity'] = v
79
80 return next_w, config
81
82
83
84 def rmsprop(w, dw, config=None):
85     """
86     Uses the RMSProp update rule, which uses a moving average of squared
87     gradient values to set adaptive per-parameter learning rates.
88
89     config format:
90     - learning_rate: Scalar learning rate.
91     - decay_rate: Scalar between 0 and 1 giving the decay rate for the squared
92       gradient cache.
93     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
94     - cache: Moving average of second moments of gradients.
95     """
96     if config is None: config = {}
97     config.setdefault('learning_rate', 1e-2)
98     config.setdefault('decay_rate', 0.99)
99     config.setdefault('epsilon', 1e-8)
100    config.setdefault('cache', np.zeros_like(w))
101
102    next_w = None
103    #####
104    # TODO: Implement the RMSprop update formula, storing the next value of w #
105    # in the next_w variable. Don't forget to update cache value stored in #
106    # config['cache']. #
107    #####
108    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
109
110    config['cache'] = config['decay_rate'] * config['cache'] + (1 - config['decay_rate']) * dw**2
111    w += - config['learning_rate'] * dw / (np.sqrt(config['cache']) + config['epsilon'])
112    next_w = w
113
114    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
115    #####
116    #                                     END OF YOUR CODE                                     #
117    #####
118
119    return next_w, config
120
121
122 def adam(w, dw, config=None):
123     """
124     Uses the Adam update rule, which incorporates moving averages of both the
125     gradient and its square and a bias correction term.
126
127     config format:
128     - learning_rate: Scalar learning rate.
129     - beta1: Decay rate for moving average of first moment of gradient.
130     - beta2: Decay rate for moving average of second moment of gradient.
131     - epsilon: Small scalar used for smoothing to avoid dividing by zero.
132     - m: Moving average of gradient.
133     - v: Moving average of squared gradient.
134     - t: Iteration number.
135     """
136     if config is None: config = {}
137     config.setdefault('learning_rate', 1e-3)
138     config.setdefault('beta1', 0.9)
139     config.setdefault('beta2', 0.999)
140     config.setdefault('epsilon', 1e-8)
141     config.setdefault('m', np.zeros_like(w))
142     config.setdefault('v', np.zeros_like(w))
143     config.setdefault('t', 0)
144
145     next_w = None
146     #####
147     # TODO: Implement the Adam update formula, storing the next value of w in #
148     # the next_w variable. Don't forget to update the m, v, and t variables #
149     # stored in config. #

```

```

150 #
151 # NOTE: In order to match the reference output, please modify t _before_ #
152 # using it in any calculations. #
153 #####
154 # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
155
156 m = config['m']
157 v = config['v']
158 beta1 = config['beta1']
159 beta2 = config['beta2']
160 learning_rate = config['learning_rate']
161 epsilon = config['epsilon']
162 t = config['t'] + 1
163
164 m = beta1*m + (1-beta1)*dw
165 mt = m / (1-beta1**t)
166 v = beta2*v + (1-beta2)*(dw**2)
167 vt = v / (1-beta2**t)
168 w += - learning_rate * mt / (np.sqrt(vt) + epsilon)
169
170 next_w = w
171 config['m'] = m
172 config['v'] = v
173 config['t'] = t
174
175 # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
176 #####
177 #                                     END OF YOUR CODE #
178 #####
179
180 return next_w, config

```

4 cnn.py

```
1 from builtins import object
2 import numpy as np
3
4 from cs231n.layers import *
5 from cs231n.fast_layers import *
6 from cs231n.layer_utils import *
7
8
9 class ThreeLayerConvNet(object):
10     """
11     A three-layer convolutional network with the following architecture:
12
13     conv - relu - 2x2 max pool - affine - relu - affine - softmax
14
15     The network operates on minibatches of data that have shape (N, C, H, W)
16     consisting of N images, each with height H and width W and with C input
17     channels.
18     """
19
20     def __init__(self, input_dim=(3, 32, 32), num_filters=32, filter_size=7,
21                 hidden_dim=100, num_classes=10, weight_scale=1e-3, reg=0.0,
22                 dtype=np.float32):
23         """
24         Initialize a new network.
25
26         Inputs:
27         - input_dim: Tuple (C, H, W) giving size of input data
28         - num_filters: Number of filters to use in the convolutional layer
29         - filter_size: Width/height of filters to use in the convolutional layer
30         - hidden_dim: Number of units to use in the fully-connected hidden layer
31         - num_classes: Number of scores to produce from the final affine layer.
32         - weight_scale: Scalar giving standard deviation for random initialization
33           of weights.
34         - reg: Scalar giving L2 regularization strength
35         - dtype: numpy datatype to use for computation.
36         """
37         self.params = {}
38         self.reg = reg
39         self.dtype = dtype
40
41         #####
42         # TODO: Initialize weights and biases for the three-layer convolutional #
43         # network. Weights should be initialized from a Gaussian centered at 0.0 #
44         # with standard deviation equal to weight_scale; biases should be #
45         # initialized to zero. All weights and biases should be stored in the #
46         # dictionary self.params. Store weights and biases for the convolutional #
47         # layer using the keys 'W1' and 'b1'; use keys 'W2' and 'b2' for the #
48         # weights and biases of the hidden affine layer, and keys 'W3' and 'b3' #
49         # for the weights and biases of the output affine layer. #
50         # #
51         # IMPORTANT: For this assignment, you can assume that the padding #
52         # and stride of the first convolutional layer are chosen so that #
53         # **the width and height of the input are preserved**. Take a look at #
54         # the start of the loss() function to see how that happens. #
55         #####
56         # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
57
58         C, H, W = input_dim
59         self.params['W1'] = weight_scale * np.random.randn(num_filters, C, filter_size, filter_size)
60         self.params['b1'] = np.zeros((1, num_filters))
61
62         # 2x2 max pool reduces the width and height by half
63         self.params['W2'] = weight_scale * np.random.randn(num_filters * H * W // (2 * 2), hidden_dim)
64         self.params['b2'] = np.zeros((1, hidden_dim))
65
66         self.params['W3'] = weight_scale * np.random.randn(hidden_dim, num_classes)
67         self.params['b3'] = np.zeros((1, num_classes))
68
69         # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
70         #####
71         # END OF YOUR CODE #
72         #####
73
```



```

74     for k, v in self.params.items():
75         self.params[k] = v.astype(dtype)
76
77
78 def loss(self, X, y=None):
79     """
80     Evaluate loss and gradient for the three-layer convolutional network.
81
82     Input / output: Same API as TwoLayerNet in fc_net.py.
83     """
84     W1, b1 = self.params['W1'], self.params['b1']
85     W2, b2 = self.params['W2'], self.params['b2']
86     W3, b3 = self.params['W3'], self.params['b3']
87
88     # pass conv_param to the forward pass for the convolutional layer
89     # Padding and stride chosen to preserve the input spatial size
90     filter_size = W1.shape[2]
91     conv_param = {'stride': 1, 'pad': (filter_size - 1) // 2}
92
93     # pass pool_param to the forward pass for the max-pooling layer
94     pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}
95
96     scores = None
97     #####
98     # TODO: Implement the forward pass for the three-layer convolutional net, #
99     # computing the class scores for X and storing them in the scores      #
100     # variable.                                                            #
101     #                                                                       #
102     # Remember you can use the functions defined in cs231n/fast_layers.py and #
103     # cs231n/layer_utils.py in your implementation (already imported).      #
104     #####
105     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
106
107     conv_relu_pool_outcome, conv_relu_pool_cache = conv_relu_pool_forward(X, W1, b1, conv_param,
pool_param)
108     affine_relu_outcome, affine_relu_cache = affine_relu_forward(conv_relu_pool_outcome, W2, b2)
109     scores, last_cache = affine_forward(affine_relu_outcome, W3, b3)
110
111
112     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
113     #####
114     #                               END OF YOUR CODE                       #
115     #####
116
117     if y is None:
118         return scores
119
120     loss, grads = 0, {}
121     #####
122     # TODO: Implement the backward pass for the three-layer convolutional net, #
123     # storing the loss and gradients in the loss and grads variables. Compute #
124     # data loss using softmax, and make sure that grads[k] holds the gradients #
125     # for self.params[k]. Don't forget to add L2 regularization!              #
126     #                                                                           #
127     # NOTE: To ensure that your implementation matches ours and you pass the  #
128     # automated tests, make sure that your L2 regularization includes a factor #
129     # of 0.5 to simplify the expression for the gradient.                    #
130     #####
131     # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
132
133     loss, dout = softmax_loss(scores, y)
134     reg_loss = 0.5 * self.reg * (np.sum(np.square(self.params['W1'])) + np.sum(np.square(self.params['
W2']))) + np.sum(np.square(self.params['W3']))
135     loss += reg_loss
136
137     dX3, dW3, db3 = affine_backward(dout, last_cache)
138     dX2, dW2, db2 = affine_relu_backward(dX3, affine_relu_cache)
139     dX1, dW1, db1 = conv_relu_pool_backward(dX2, conv_relu_pool_cache)
140
141     dW1 += self.reg * self.params['W1']
142     dW2 += self.reg * self.params['W2']
143     dW3 += self.reg * self.params['W3']
144
145     grads['W1'] = dW1
146     grads['b1'] = db1
147     grads['W2'] = dW2

```

```

148     grads[ 'b2' ] = db2
149     grads[ 'W3' ] = dW3
150     grads[ 'b3' ] = db3
151
152
153
154     # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
155     #####
156     #                               END OF YOUR CODE                               #
157     #####
158
159     return loss , grads

```