fc_net.py

```
1   import numpy as np
2   import pdb
3
4   from .layers import *
5   from .layer_utils import *
6
7   """
8   This code was originally written for CS 231n at Stanford University
9   (cs231n.stanford.edu).  It has been modified in various areas for use in the
10  ECE 239AS class at UCLA.  This includes the descriptions of what code to
11  implement as well as some slight potential changes in variable names to be
12  consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
13  permission to use this code.  To see the original version, please visit
14  cs231n.stanford.edu.
15  """
16
17  class FullyConnectedNet(object):
18      """
19      A fully-connected neural network with an arbitrary number of hidden layers,
20      ReLU nonlinearities, and a softmax loss function. This will also implement
21      dropout and batch normalization as options. For a network with L layers,
22      the architecture will be
23
24      {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax
25
26      where batch normalization and dropout are optional, and the {...} block is
27      repeated L - 1 times.
28
29      Similar to the TwoLayerNet above, learnable parameters are stored in the
30      self.params dictionary and will be learned using the Solver class.
31      """
32
33      def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
34                   dropout=0, use_batchnorm=False, reg=0.0,
35                   weight_scale=1e-2, dtype=np.float32, seed=None):
36          """
37          Initialize a new FullyConnectedNet.
38
39          Inputs:
40          - hidden_dims: A list of integers giving the size of each hidden layer.
41          - input_dim: An integer giving the size of the input.
42          - num_classes: An integer giving the number of classes to classify.
43          - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=1 then
44            the network should not use dropout at all.
45          - use_batchnorm: Whether or not the network should use batch normalization.
46          - reg: Scalar giving L2 regularization strength.
47          - weight_scale: Scalar giving the standard deviation for random
48            initialization of the weights.
49          - dtype: A numpy datatype object; all computations will be performed using
50            this datatype. float32 is faster but less accurate, so you should use
51            float64 for numeric gradient checking.
52          - seed: If not None, then pass this random seed to the dropout layers. This
53            will make the dropout layers deteriminstic so we can gradient check the
54            model.
55          """
56          self.use_batchnorm = use_batchnorm
57          self.use_dropout = dropout < 1
58          self.reg = reg
59          self.num_layers = 1 + len(hidden_dims)
60          self.dtype = dtype
61          self.params = {}
62
63          # ============================================================ #
64          # YOUR CODE HERE:
65          #   Initialize all parameters of the network in the self.params dictionary.
66          #   The weights and biases of layer 1 are W1 and b1; and in general the
67          #   weights and biases of layer i are Wi and bi. The
68          #   biases are initialized to zero and the weights are initialized
69          #   so that each parameter has mean 0 and standard deviation weight_scale.
70          #
71          #   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
72          #   parameters to zero.  The gamma and beta parameters for layer 1 should
73          #   be self.params['gamma1'] and self.params['beta1'].  For layer 2, they
74          #   should be gamma2 and beta2, etc. Only use batchnorm if self.use_batchnorm
75          #   is true and DO NOT do batch normalize the output scores.
76          # ============================================================ #
77
78          next_layer_input_dim = input_dim
79          for i, hidden_dim in enumerate(hidden_dims, start=1):
80              self.params['W' + str(i)] = weight_scale * np.random.randn(next_layer_input_dim, hidden_dim)
```

```
81              self.params['b' + str(i)] = np.zeros(hidden_dim)
82              if self.use_batchnorm:
83                  self.params['gamma' + str(i)] = np.ones(hidden_dim)
84                  self.params['beta' + str(i)] = np.zeros(hidden_dim)
85              next_layer_input_dim = hidden_dim
86
87          self.params['W' + str(self.num_layers)] = weight_scale * np.random.randn(next_layer_input_dim, num_classes)
88          self.params['b' + str(self.num_layers)] = np.zeros(num_classes)
89
90          # ============================================================ #
91          # END YOUR CODE HERE
92          # ============================================================ #
93
94          # When using dropout we need to pass a dropout_param dictionary to each
95          # dropout layer so that the layer knows the dropout probability and the mode
96          # (train / test). You can pass the same dropout_param to each dropout layer.
97          self.dropout_param = {}
98          if self.use_dropout:
99              self.dropout_param = {'mode': 'train', 'p': dropout}
100         if seed is not None:
101             self.dropout_param['seed'] = seed
102
103         # With batch normalization we need to keep track of running means and
104         # variances, so we need to pass a special bn_param object to each batch
105         # normalization layer. You should pass self.bn_params[0] to the forward pass
106         # of the first batch normalization layer, self.bn_params[1] to the forward
107         # pass of the second batch normalization layer, etc.
108         self.bn_params = []
109         if self.use_batchnorm:
110             self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]
111
112         # Cast all parameters to the correct datatype
113         for k, v in self.params.items():
114             self.params[k] = v.astype(dtype)
115
116
117     def loss(self, X, y=None):
118         """
119         Compute loss and gradient for the fully-connected net.
120
121         Input / output: Same as TwoLayerNet above.
122         """
123         X = X.astype(self.dtype)
124         mode = 'test' if y is None else 'train'
125
126         # Set train/test mode for batchnorm params and dropout param since they
127         # behave differently during training and testing.
128         if self.dropout_param is not None:
129             self.dropout_param['mode'] = mode
130         if self.use_batchnorm:
131             for bn_param in self.bn_params:
132                 bn_param[mode] = mode
133
134         scores = None
135
136         # ============================================================ #
137         # YOUR CODE HERE:
138         #   Implement the forward pass of the FC net and store the output
139         #   scores as the variable "scores".
140         #
141         #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
142         #   between the affine_forward and relu_forward layers.  You may
143         #   also write an affine_batchnorm_relu() function in layer_utils.py.
144         #
145         #   DROPOUT: If dropout is non-zero, insert a dropout layer after
146         #   every ReLU layer.
147         # ============================================================ #
148
149         caches = {}
150         dropout_caches = {}
151         layer_out = X
152         for i in range(1, self.num_layers):
153             if self.use_batchnorm:
154                 layer_out, caches[i] = affine_batchnorm_relu_forward(np.array(layer_out),
155                                                      self.params['W' + str(i)],
156                                                      self.params['b' + str(i)],
157                                                      self.params['gamma' + str(i)],
158                                                      self.params['beta' + str(i)],
159                                                      self.bn_params[i - 1])
160             else: # No batchnorm
161                 layer_out, caches[i] = affine_relu_forward(np.array(layer_out),
162                                                    self.params['W' + str(i)],
163                                                    self.params['b' + str(i)])
```

```
            if self.use_dropout:
                layer_out, dropout_caches[i] = dropout_forward(layer_out, self.dropout_param)



        scores, caches[self.num_layers] = affine_forward(np.array(layer_out),
                                                         self.params['W' + str(self.num_layers)],
                                                         self.params['b' + str(self.num_layers)])

        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #

        # If test mode return early
        if mode == 'test':
            return scores

        loss, grads = 0.0, {}
        # ============================================================ #
        # YOUR CODE HERE:
        #   Implement the backwards pass of the FC net and store the gradients
        #   in the grads dict, so that grads[k] is the gradient of self.params[k]
        #   Be sure your L2 regularization includes a 0.5 factor.
        #
        #   BATCHNORM: Incorporate the backward pass of the batchnorm.
        #
        #   DROPOUT: Incorporate the backward pass of dropout.
        # ============================================================ #

        sum_of_W_norms = 0
        for i in range(self.num_layers):
            sum_of_W_norms += (np.linalg.norm(self.params['W' + str(i + 1)]) ** 2)

        loss, soft_grad = softmax_loss(scores, y)
        loss += 0.5 * self.reg * sum_of_W_norms

        layer_back_grad, grads['W' + str(self.num_layers)], grads['b' + str(self.num_layers)] = \
            affine_backward(soft_grad, caches[self.num_layers])
        grads['W' + str(self.num_layers)] += self.reg * self.params['W' + str(self.num_layers)]

        for i in range(self.num_layers - 1, 0, -1):
            if self.use_dropout:
                layer_back_grad = dropout_backward(layer_back_grad, dropout_caches[i])
            if self.use_batchnorm:
                layer_back_grad, grads['W' + str(i)], grads['b' + str(i)], \
                grads['gamma' + str(i)], grads['beta' + str(i)] = \
                    affine_batchnorm_relu_backward(layer_back_grad, caches[i])
            else:
                layer_back_grad, grads['W' + str(i)], grads['b' + str(i)] = affine_relu_backward(layer_back_grad,
                                                                                                 caches[i])
            grads['W' + str(i)] += self.reg * self.params['W' + str(i)]

        # ============================================================ #
        # END YOUR CODE HERE
        # ============================================================ #

        return loss, grads

```