```python
import numpy as np
import pdb

from .layers import *
from .layer_utils import *

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

class FullyConnectedNet(object):
  """
  A fully-connected neural network with an arbitrary number of hidden layers,
  ReLU nonlinearities, and a softmax loss function. This will also implement
  dropout and batch normalization as options. For a network with L layers,
  the architecture will be

  {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

  where batch normalization and dropout are optional, and the {...} block is
  repeated L - 1 times.

  Similar to the TwoLayerNet above, learnable parameters are stored in the
  self.params dictionary and will be learned using the Solver class.
  """

  def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
               dropout=0, use_batchnorm=False, reg=0.0,
               weight_scale=1e-2, dtype=np.float32, seed=None):
    """
    Initialize a new FullyConnectedNet.

    Inputs:
    - hidden_dims: A list of integers giving the size of each hidden layer.
    - input_dim: An integer giving the size of the input.
    - num_classes: An integer giving the number of classes to classify.
    - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
      the network should not use dropout at all.
    - use_batchnorm: Whether or not the network should use batch normalization.
    - reg: Scalar giving L2 regularization strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - dtype: A numpy datatype object; all computations will be performed using
      this datatype. float32 is faster but less accurate, so you should use
      float64 for numeric gradient checking.
    - seed: If not None, then pass this random seed to the dropout layers. This
      will make the dropout layers deteriminstic so we can gradient check the
      model.
```

```
"""
self.use_batchnorm = use_batchnorm
self.use_dropout = dropout > 0
self.reg = reg
self.num_layers = 1 + len(hidden_dims)
self.dtype = dtype
self.params = {}

# ================================================================ #
# YOUR CODE HERE:
#   Initialize all parameters of the network in the self.params dictionary.
#   The weights and biases of layer 1 are W1 and b1; and in general the
#   weights and biases of layer i are Wi and bi. The
#   biases are initialized to zero and the weights are initialized
#   so that each parameter has mean 0 and standard deviation weight_scale.
#
#   BATCHNORM: Initialize the gammas of each layer to 1 and the beta
#   parameters to zero.  The gamma and beta parameters for layer 1 should
#   be self.params['gamma1'] and self.params['beta1'].  For layer 2, they
#   should be gamma2 and beta2, etc. Only use batchnorm if self.use_batchnorm
#   is true and DO NOT batch normalize the output scores.
# ================================================================ #
self.h_dims = len(hidden_dims)
for i in np.arange(len(hidden_dims) + 1):
    # eg: when len(hidden_dims) = 2, W1, W2, W3, b1, b2, b3 exist
    if i == 0:
        self.params['W'+str(i+1)] = weight_scale * np.random.randn(input_dim,
         hidden_dims[i])
        self.params['b'+str(i+1)] = np.zeros(hidden_dims[i])
    elif i == len(hidden_dims):
        self.params['W'+str(i+1)] = weight_scale *
         np.random.randn(hidden_dims[-1], num_classes)
        self.params['b'+str(i+1)] = np.zeros(num_classes)
    else:
        self.params['W'+str(i+1)] = weight_scale *
         np.random.randn(hidden_dims[i-1], hidden_dims[i])
        self.params['b'+str(i+1)] = np.zeros(hidden_dims[i])

pass

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

# When using dropout we need to pass a dropout_param dictionary to each
# dropout layer so that the layer knows the dropout probability and the mode
# (train / test). You can pass the same dropout_param to each dropout layer.
self.dropout_param = {}
if self.use_dropout:
  self.dropout_param = {'mode': 'train', 'p': dropout}
  if seed is not None:
    self.dropout_param['seed'] = seed

# With batch normalization we need to keep track of running means and
```

```
    # variances, so we need to pass a special bn_param object to each batch
    # normalization layer. You should pass self.bn_params[0] to the forward pass
    # of the first batch normalization layer, self.bn_params[1] to the forward
    # pass of the second batch normalization layer, etc.
    self.bn_params = []
    if self.use_batchnorm:
        self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]
        # initialize all gammas and betas
        for i in np.arange(self.num_layers-1):
            self.params['gamma'+str(i+1)] = np.ones(hidden_dims[i])
            self.params['beta'+str(i+1)] = np.zeros(hidden_dims[i])


    # Cast all parameters to the correct datatype
    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


  def loss(self, X, y=None):
    """
    Compute loss and gradient for the fully-connected net.

    Input / output: Same as TwoLayerNet above.
    """
    X = X.astype(self.dtype)
    mode = 'test' if y is None else 'train'

    # Set train/test mode for batchnorm params and dropout param since they
    # behave differently during training and testing.
    if self.dropout_param is not None:
      self.dropout_param['mode'] = mode
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param[mode] = mode

    scores = None

    # ============================================================= #
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    #
    #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm layer
    #   between the affine_forward and relu_forward layers.  You may
    #   also write an affine_batchnorm_relu() function in layer_utils.py.
    #
    #   DROPOUT: If dropout is non-zero, insert a dropout layer after
    #   every ReLU layer.
    # ============================================================= #
    N = X.shape[0]
    D = np.prod(X.shape[1:])

    hidden = {}
    hidden['h0'] = X
```

```
    h = X

    if self.use_dropout:
        hidden['hdrop0'], hidden['hdrop_cache0'] = dropout_forward(h,
         self.dropout_param)
        h = hidden['hdrop0']

    for i in np.arange(self.h_dims + 1):
        w = self.params['W'+str(i+1)]
        b = self.params['b'+str(i+1)]

        if i == self.h_dims:
            scores, scores_cache = affine_forward(h, w, b)
            hidden['h'+str(i+1)] = h
            hidden['h_cache'+str(i+1)] = h_cache
        else:
            if self.use_batchnorm == True:
                gamma = self.params['gamma'+str(i+1)]
                beta = self.params['beta'+str(i+1)]
                bn_param = self.bn_params[i]
                h, h_cache = affine_batchnorm_relu_forward(h, w, b, gamma, beta,
                 bn_param)
                hidden['h'+str(i+1)] = h
                hidden['h_cache'+str(i+1)] = h_cache
            else:
                h, h_cache = affine_relu_forward(h, w, b)
                hidden['h'+str(i+1)] = h
                hidden['h_cache'+str(i+1)] = h_cache

            if self.use_dropout:
                hidden['hdrop'+str(i+1)], hidden['hdrop_cache'+str(i+1)] =
                 dropout_forward(h, self.dropout_param)
                h = hidden['hdrop'+str(i+1)]


#        if i == 0:
#            h, h_cache = affine_relu_forward(X, w, b)
#            hidden['h'+str(i+1)] = h
#            hidden['h_cache'+str(i+1)] = h_cache
#        elif i == self.h_dims:
#            scores, scores_cache = affine_forward(h, w, b)
#            hidden['h'+str(i+1)] = h
#            hidden['h_cache'+str(i+1)] = h_cache
#        else:
#            h, h_cache = affine_relu_forward(h, w, b)
#            hidden['h'+str(i+1)] = h
#            hidden['h_cache'+str(i+1)] = h_cache
    pass

    # ================================================================ #
    # 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.
# ================================================================ #
data_loss, dscores = softmax_loss(scores, y)
reg_sum = 0
for i in np.arange(self.h_dims+1):
    reg_sum += np.sum(self.params['W'+str(i+1)]*self.params['W'+str(i+1)])
reg_loss = 0.5 * self.reg * reg_sum
loss = data_loss + reg_loss

for i in np.arange(self.h_dims+1)[::-1]:
    if i == self.h_dims:
        dloss, w, b = affine_backward(dscores, scores_cache)
        grads['W'+str(i+1)] = w
        grads['W'+str(i+1)] += self.reg * self.params['W'+str(i+1)]
        grads['b'+str(i+1)] = b
    else:
        if self.use_dropout:
            dloss  = dropout_backward(dloss, hidden['hdrop_cache'+str(i+1)])

        if self.use_batchnorm == True:
            dloss, dw, db, dgamma, dbeta = affine_batchnorm_relu_backward(dloss,
             hidden['h_cache'+str(i+1)])
            grads['W'+str(i+1)] = dw
            grads['b'+str(i+1)] = db
            grads['W'+str(i+1)] += self.reg * self.params['W'+str(i+1)]
            grads['gamma'+str(i+1)] = dgamma
            grads['beta'+str(i+1)] = dbeta
        else:
            dloss, dw, db = affine_relu_backward(dloss,
             hidden['h_cache'+str(i+1)])
            grads['W'+str(i+1)] = dw
            grads['b'+str(i+1)] = db
            grads['W'+str(i+1)] += self.reg * self.params['W'+str(i+1)]


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

return loss, grads
```