```python
import numpy as np
from .layers import *
from .layer_utils import *

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 do batch normalize the output scores.
    # ================================================================ #

    # Concat dims for full NN
    dims = [input_dim] + hidden_dims + [num_classes]
    for layer in range(self.num_layers):
      self.params['W' + str(layer + 1)] = np.random.normal(0, weight_scale,(dims[layer], dims[layer + 1]))
      self.params['b' + str(layer + 1)] = np.zeros(dims[layer + 1])

      if self.use_batchnorm and (layer != (self.num_layers-1)):
        self.params['gamma' + str(layer + 1)] = np.ones(dims[layer + 1])
        self.params['beta' + str(layer + 1)] = np.zeros(dims[layer + 1])

    # ================================================================ #
    # 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)]

    # 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.
    # ================================================================ #

    a = {}
    norm = {}
    h = {}
    drop = {}
    drop[0] = [X]


    for layer in range(self.num_layers):
      #Affine
      a[layer + 1] = affine_forward(drop[layer][0], self.params['W' + str(layer + 1)],self.params['b' + str(layer + 1)])

      if layer < (self.num_layers-1):
        # BatchNorm
        if self.use_batchnorm: norm[layer + 1] = batchnorm_forward(a[layer + 1][0], self.params['gamma' + str(layer + 1)],
          self.params['beta' + str(layer + 1)], self.bn_params[layer])
        else: norm[layer + 1] = a[layer + 1]
          # ReLU
        h[layer + 1] = relu_forward(norm[layer + 1][0])
        # Dropout
        if self.use_dropout: drop[layer + 1] = dropout_forward(h[layer + 1][0],self.dropout_param)
        else: drop[layer + 1] = h[layer + 1]


    scores = a[self.num_layers][0]

    # ================================================================ #
    # 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.
    # ================================================================ #

    loss, dout = softmax_loss(scores, y)
    Ws = [self.params['W' + str(i + 1)] for i in range(self.num_layers)]

    loss += 0.5 * self.reg * sum([np.linalg.norm(weight, 'fro')**2 for weight in Ws])
    das = {}
    dhs = {}
    ddrops = {}
    dnorms = {}
    dgammas = {}
    dbetas = {}
    dws = {}
    dbs = {}
    das[self.num_layers] = dout

    for layer in reversed(range(self.num_layers)):
      ddrops[layer], dws[layer + 1], dbs[layer + 1] = affine_backward(das[layer + 1], a[layer + 1][1])
      if layer != 0:
        if self.use_dropout: dhs[layer] = dropout_backward(ddrops[layer],drop[layer][1])
        else: dhs[layer] = ddrops[layer]
        dnorms[layer] = relu_backward(dhs[layer], h[layer][1])
        if self.use_batchnorm: das[layer], dgammas[layer], dbetas[layer] = batchnorm_backward(dnorms[layer], norm[layer][1])
        else: das[layer] = dnorms[layer]

    for layer in range(self.num_layers):
      grads['W' + str(layer + 1)] = dws[layer + 1] + self.reg * self.params['W' + str(layer + 1)]
      grads['b' + str(layer + 1)] = dbs[layer + 1].T
      if layer != (self.num_layers-1) and self.use_batchnorm:
        grads['gamma' + str(layer + 1)] = dgammas[layer + 1]
        grads['beta' + str(layer + 1)] = dbetas[layer + 1].T

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

    return loss, grads
```