```python
import numpy as np

from nndl.layers import *
from nndl.conv_layers import *
from utils.fast_layers import *
from nndl.layer_utils import *
from nndl.conv_layer_utils import *

import pdb

class ThreeLayerConvNet(object):
  """
  A three-layer convolutional network with the following architecture:

  conv - relu - 2x2 max pool - affine - relu - affine - softmax

  The network operates on minibatches of data that have shape (N, C, H, W)
  consisting of N images, each with height H and width W and with C input
  channels.
  """

  def __init__(self, input_dim=(3, 32, 32), num_filters=32, filter_size=7,
               hidden_dim=100, num_classes=10, weight_scale=1e-3, reg=0.0,
               dtype=np.float32, use_batchnorm=False):
    """
    Initialize a new network.

    Inputs:
    - input_dim: Tuple (C, H, W) giving size of input data
    - num_filters: Number of filters to use in the convolutional layer
    - filter_size: Size of filters to use in the convolutional layer
    - hidden_dim: Number of units to use in the fully-connected hidden layer
    - num_classes: Number of scores to produce from the final affine layer.
    - weight_scale: Scalar giving standard deviation for random initialization
      of weights.
    - reg: Scalar giving L2 regularization strength
    - dtype: numpy datatype to use for computation.
    """
    self.use_batchnorm = use_batchnorm
    self.params = {}
    self.reg = reg
    self.dtype = dtype


    # ================================================================ #
    # YOUR CODE HERE:
    #   Initialize the weights and biases of a three layer CNN. To initialize:
    #     - the biases should be initialized to zeros.
    #     - the weights should be initialized to a matrix with entries
    #         drawn from a Gaussian distribution with zero mean and
    #         standard deviation given by weight_scale.
    # ================================================================ #
    C, H, W = input_dim
    pad = (filter_size - 1) / 2

    self.params['W1'] = np.random.normal(0, weight_scale, [num_filters,C,filter_size, filter_size])
    self.params['b1'] = np.zeros(num_filters)
    conv_out_h = int((H + 2 * pad - filter_size) + 1)
    conv_out_w = int((W + 2 * pad - filter_size) + 1)
    pool_out_h = int((conv_out_h - 2) / 2 + 1)
    pool_out_w = int((conv_out_w - 2) / 2 + 1)
    self.params['W2'] = np.random.normal(0, weight_scale, [pool_out_h * pool_out_w * num_filters, hidden_dim])
    self.params['b2'] = np.zeros(hidden_dim)
    self.params['W3'] = np.random.normal(0, weight_scale, [hidden_dim, num_classes])
    self.params['b3'] = np.zeros(num_classes)

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

    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


def loss(self, X, y=None):
  """
  Evaluate loss and gradient for the three-layer convolutional network.

  Input / output: Same API as TwoLayerNet in fc_net.py.
  """
  W1, b1 = self.params['W1'], self.params['b1']
  W2, b2 = self.params['W2'], self.params['b2']
  W3, b3 = self.params['W3'], self.params['b3']

  # pass conv_param to the forward pass for the convolutional layer
  filter_size = W1.shape[2]
  conv_param = {'stride': 1, 'pad': (filter_size - 1) / 2}

  # pass pool_param to the forward pass for the max-pooling layer
  pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

  scores = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the forward pass of the three layer CNN.  Store the output
  #   scores as the variable "scores".
  # ================================================================ #

  if not self.use_batchnorm:
    l1_out, l1_cache = conv_relu_pool_forward(X, W1, b1,conv_param, pool_param)
    l2_out, l2_cache = affine_relu_forward(l1_out, W2, b2)
    scores, l3_cache = affine_forward(l2_out, W3, b3)
  # else:
  #     conv_out, conv_cache = conv_forward_fast(x, w, b, conv_param)
  #     out, cache = spatial_batchnorm_forward(conv_out, gamma, beta, bn_param)
  #     s, relu_cache = relu_forward(a)
  #     out, pool_cache = max_pool_forward_fast(s, pool_param)
  #     out, cache = affine_forward(x, w, b)
  #     out, cache = batchnorm_forward(x, gamma, beta, bn_param)
  #     s, relu_cache = relu_forward(a)
  #     out, cache = affine_forward(x, w, b)


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

  if y is None:
    return scores

  loss, grads = 0, {}
  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the backward pass of the three layer CNN.  Store the grads
  #   in the grads dictionary, exactly as before (i.e., the gradient of
  #   self.params[k] will be grads[k]).  Store the loss as "loss", and
  #   don't forget to add regularization on ALL weight matrices.
  # ================================================================ #

  loss, dout = softmax_loss(scores, y)
  loss += 0.5 * self.reg * sum([np.sum(np.square(weight)) for weight in [W1,W2,W3]])

  dx3, dw3, db3 = affine_backward(dout, l3_cache)
  dx2, dw2, db2 = affine_relu_backward(dx3, l2_cache)
  _, dw1, db1 = conv_relu_pool_backward(dx2, l1_cache)
  grads['b1'], grads['b2'], grads['b3'] = db1, db2, db3
  grads['W1'] = dw1 + self.reg * W1
  grads['W2'] = dw2 + self.reg * W2
  grads['W3'] = dw3 + self.reg * W3

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

  return loss, grads


pass
```