```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
    shapes = {}
    shapes['W1'] = (num_filters, C, filter_size, filter_size)
    shapes['W2'] = ((H//2) *(W//2) * num_filters, hidden_dim)
    shapes['W3'] = (hidden_dim, num_classes)
    shapes['b1'] = num_filters
    shapes['b2'] = hidden_dim
    shapes['b3'] = num_classes

    for i in range(1,4):
        str_W = 'W' + str(i)
        str_b = 'b' + str(i)
        self.params[str_W] = np.random.normal(loc = 0.0, scale = weight_scale, size =
        shapes[str_W])
        self.params[str_b] = np.zeros(shapes[str_b])
```

```
    # ================================================================ #
    # 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".
    # ================================================================ #

    h1, cache1 = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)
    h2, cache2 = affine_relu_forward(h1,W2,b2)
    scores, cache3 = affine_forward(h2,W3,b3)

    # ================================================================ #
    # 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, dz = softmax_loss(scores,y)
    loss += 0.5*self.reg*(np.sum(W1**2) + np.sum(W2**2) + np.sum(W3**2))

    dh2,dw3, grads['b3'] = affine_backward(dz, cache3)
    dh1, dw2, grads['b2'] = affine_relu_backward(dh2, cache2)
    _, dw1, grads['b1'] = conv_relu_pool_backward(dh1, cache1)

    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

```