```python
import numpy as np

from nndl.layers import *
from nndl.conv_layers import *
from cs231n.fast_layers import *
from nndl.layer_utils import *
from nndl.conv_layer_utils import *

import pdb

"""
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 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:
```

```python
    #     - 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
    conv_output_dims = (1 + (H + 2 * pad - filter_size))
    pool_output_dims = int((conv_output_dims - 2) / 2 + 1)**2 * num_filters

    self.params['W1'] = weight_scale * np.random.randn(num_filters, C, filter_size, filter_size)
    self.params['b1'] = np.zeros(num_filters)
    self.params['W2'] = weight_scale * np.random.randn(pool_output_dims, hidden_dim)
    self.params['b2'] = np.zeros(hidden_dim)
    self.params['W3'] = weight_scale * np.random.randn(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".
    # ================================================================ #

    caches = []
    out, cache = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)
    conv_shape = out.shape # used for backward pass
    caches.append(cache)
    out = np.reshape(out, (out.shape[0], -1))
    out, cache = affine_relu_forward(out, W2, b2)
```

```
    caches.append(cache)
    scores, cache = affine_forward(out, W3, b3)
    caches.append(cache)

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

    sm_loss, dout = softmax_loss(scores, y)
    reg_loss = self.reg * 0.5 * (np.sum(self.params['W1']**2) + np.sum(self.params['W2']**2) + np.sum(self.params['W3']**2))
    loss = sm_loss + reg_loss

    dout, grads['W3'], grads['b3'] = affine_backward(dout, caches.pop())
    dout, grads['W2'], grads['b2'] = affine_relu_backward(dout, caches.pop())
    dout = dout.reshape(conv_shape)
    dout, grads['W1'], grads['b1'] = conv_relu_pool_backward(dout, caches.pop())

    for weights in ['W3', 'W2', 'W1']:
      grads[weights] += self.reg * self.params[weights]

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

    return loss, grads


pass
```