```
# cnn .py

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

    # cnn params
    pad = (filter_size - 1) / 2
    conv_stride = 1
    pool_size = 2
    pool_stride = 2
    # output sizes after convolution and pooling
    H_out_conv, W_out_conv = int(1 + (H - filter_size + 2*pad) /
conv_stride), int(1 + (W - filter_size + 2*pad) / conv_stride)
    H_out_pool, W_out_pool = int(1 + (H_out_conv - pool_size) /
pool_stride), int(1 + (H_out_conv - pool_size) / pool_stride)

    # W1 = conv weights
    self.params['W1'] = weight_scale * np.random.randn(num_filters, C,
filter_size, filter_size)
    self.params['b1'] = np.zeros(num_filters)

    max_pool_output_size = int(num_filters * H_out_pool * W_out_pool)
    self.params['W2'] = weight_scale *
np.random.randn(max_pool_output_size, 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".
    # ================================================================
#

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

    # get scores for first layer (conv + relu + pool)
    h1, cache1 = conv_relu_pool_forward(x=X, w=W1, b=b1,
conv_param=conv_param, pool_param=pool_param)
    # get scores for second layer (fc)
    h2, cache2 = affine_relu_forward(x=h1, w=W2, b=b2) # get scores
for output layer (fc)
    scores, cache3 = affine_forward(x=h2, w=W3, b=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, dl = softmax_loss(x=scores, y=y)
    loss += 0.5*self.reg*np.sum(W1**2) + 0.5*self.reg*np.sum(W2**2) +
0.5*self.reg*np.sum(W3**2)
    dout, dW3, db3 = affine_backward(dl, cache3) # now backprop,
starting from the last affine layer
    dW3 += self.reg * W3
    dout, dW2, db2 = affine_relu_backward(dout, cache2)
    dW2 += self.reg * W2
    dx, dW1, db1 = conv_relu_pool_backward(dout, cache1)
    dW1 += self.reg * W1
    # now store all the gradients in the gradient dictionary
    grads["W1"] = dW1
    grads["W2"] = dW2
    grads["W3"] = dW3
    grads["b1"] = db1
    grads["b2"] = db2
    grads["b3"] = db3

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

    return loss, grads


class BestCNN(object):
  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:
    # # # # # #

    # plan
    # {conv relu conv relu pool} x 2 -> affine -> relu -> affine ->
output
    # bn plan
    # {conv bn relu conv bn relu pool} x 2 -> affine -> bn -> relu ->
affine -> output, thus need 5 bns

    C, H, W = input_dim
    # hyperparams to use
    pad = (filter_size - 1) / 2
    conv_stride = 1
    pool_size = 2
    pool_stride = 2

    # init
    self.params["W1"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters, C, filter_size, filter_size))
    self.params["b1"] = np.zeros(num_filters)
    self.params["W2"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters, num_filters, filter_size, filter_size))
    self.params["b2"] = np.zeros(num_filters)
    self.params["W3"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters, num_filters, filter_size, filter_size))
    self.params["b3"] = np.zeros(num_filters)
    self.params["W4"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters, num_filters, filter_size, filter_size))
    self.params["b4"] = np.zeros(num_filters)
    self.params["W5"] = np.random.normal(loc=0, scale=weight_scale,
size=(num_filters*8*8, hidden_dim)) ## 8 because this is the
H_out_pool
    self.params["b5"] = np.zeros(hidden_dim)

    # output layer is different
```

```
    self.params["W6"] = np.random.normal(loc=0, scale=weight_scale,
size=(hidden_dim, num_classes))
    self.params["b6"] = np.zeros(num_classes)

    if self.use_batchnorm:
      for i in range(1,5):
        self.params['gamma'+str(i)] = np.ones(num_filters)
        self.params['beta'+str(i)] = np.zeros(num_filters)

    self.params['gamma5'] = np.ones(hidden_dim)
    self.params['beta5'] = np.zeros(hidden_dim)

    self.bn_params = []
    if self.use_batchnorm:
        self.bn_params = [{'mode': 'train'} for i in np.arange(5)]

    # ================================================================
# # END YOUR CODE HERE
    # ================================================================
#
    for k, v in self.params.items(): self.params[k] = v.astype(dtype)

  def loss(self, X, y=None):
    # print("input sahpe", X.shape)
    mode = 'test' if y is None else 'train'

    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    W3, b3 = self.params['W3'], self.params['b3']
    W4, b4 = self.params['W4'], self.params['b4']
    W5, b5 = self.params['W5'], self.params['b5']
    W6, b6 = self.params['W6'], self.params['b6']

    if self.use_batchnorm:
      for bn_param in self.bn_params:
          bn_param['mode'] = mode


    # take care of conv
    filter_size = W1.shape[2]
    conv_param = {'stride': 1, 'pad': (filter_size - 1) / 2}
    # take care of pool
    pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}
    scores = None



    if not self.use_batchnorm:
      # forward pass w/o bn
      h1, cache1 = conv_relu_forward(x=X, w=W1, b=b1,
```

```
conv_param=conv_param) # conv relu

      h2, cache2 = conv_relu_forward(x=h1, w=W2, b=b2,
conv_param=conv_param) # conv relu
      h3, cache3 = max_pool_forward_fast(x=h2, pool_param=pool_param)
# pool
      h4, cache4 = conv_relu_forward(x=h3, w=W3, b=b3,
conv_param=conv_param) # conv relu
      h5, cache5 = conv_relu_forward(x=h4, w=W4, b=b4,
conv_param=conv_param) # conv relu
      h6, cache6 = max_pool_forward_fast(x=h5, pool_param=pool_param)
# pool
      # FC
      h7, cache7 = affine_relu_forward(x=h6, w=W5, b=b5)  # affine
      scores, cache8 = affine_forward(x=h7, w=W6, b=b6) # affine -
output



    bn_cache=[]
    # forward pass w bn
    if self.use_batchnorm:
      h1, cache1 = conv_bn_relu_forward(x=X, w=W1, b=b1,
conv_param=conv_param, gamma=self.params['gamma1'],
beta=self.params['beta1'], bn_param=self.bn_params[0]) # conv relu
      h2, cache2 = conv_bn_relu_forward(x=h1, w=W2, b=b2,
conv_param=conv_param, gamma=self.params['gamma2'],
beta=self.params['beta2'], bn_param=self.bn_params[1]) # conv relu
      h3, cache3 = max_pool_forward_fast(x=h2, pool_param=pool_param)
# pool
      h4, cache4 = conv_bn_relu_forward(x=h3, w=W3, b=b3,
conv_param=conv_param, gamma=self.params['gamma3'],
beta=self.params['beta3'], bn_param=self.bn_params[2]) # conv relu
      h5, cache5 = conv_bn_relu_forward(x=h4, w=W4, b=b4,
conv_param=conv_param, gamma=self.params['gamma4'],
beta=self.params['beta4'], bn_param=self.bn_params[3]) # conv relu
      h6, cache6 = max_pool_forward_fast(x=h5, pool_param=pool_param)
# pool
      # FC
      h7, cache7 = affine_batchnorm_relu_forward(x=h6, w=W5, b=b5,
gamma=self.params['gamma5'], beta=self.params['beta5'],
bn_params=self.bn_params[4])  # affine
      scores, cache8 = affine_forward(x=h7, w=W6, b=b6) # affine -
output


    if y is None:
      return scores
    loss, grads = 0, {}
```

```
    loss, dl = softmax_loss(x=scores, y=y) # then regularize the loss
    loss += 0.5*self.reg*np.sum(W1**2) + 0.5*self.reg*np.sum(W2**2) +
0.5*self.reg*np.sum(W3**2) + 0.5*self.reg*np.sum(W4**2) +
0.5*self.reg*np.sum(W5**2) + 0.5*self.reg*np.sum(W6**2)

    if not self.use_batchnorm:
      dout, dW6, db6 = affine_backward(dl, cache8) # affine
      dW6 += self.reg * W6
      dout, dW5, db5 = affine_relu_backward(dout, cache7) # affine
relu
      dW5 += self.reg * W5

      dout = max_pool_backward_fast(dout, cache6) # pool
      dout, dW4, db4 = conv_relu_backward(dout, cache5)# conv relu
      dW4 += self.reg * W4
      dout, dW3, db3 = conv_relu_backward(dout, cache4) # conv relu
      dW3 += self.reg * W3
      dout = max_pool_backward_fast(dout, cache3) # pool
      dout, dW2, db2 = conv_relu_backward(dout, cache2) # conv relu
      dW2 += self.reg * W2
      dout, dW1, db1 = conv_relu_backward(dout, cache1) # conv relu
      dW1 += self.reg * W1

    else:
      dout, dW6, db6 = affine_backward(dl, cache8) # affine
      dW6 += self.reg * W6
      dout, dW5, db5, dgamma5, dbeta5 =
affine_batchnorm_relu_backward(dout, cache7) # affine relu
      dW5 += self.reg * W5
      grads['gamma5'] = dgamma5
      grads['beta5'] = dbeta5

      dout = max_pool_backward_fast(dout, cache6) # pool
      dout, dW4, db4, dgamma4, dbeta4 = conv_bn_relu_backward(dout,
cache5)# conv relu
      dW4 += self.reg * W4
      grads['gamma4'] = dgamma4
      grads['beta4'] = dbeta4

      dout, dW3, db3, dgamma3, dbeta3 = conv_bn_relu_backward(dout,
cache4) # conv relu
      grads['gamma3'] = dgamma3
      grads['beta3'] = dbeta3
      dW3 += self.reg * W3

      dout = max_pool_backward_fast(dout, cache3) # pool
      dout, dW2, db2, dgamma2, dbeta2 = conv_bn_relu_backward(dout,
cache2) # conv relu
```

```
        grads['gamma2'] = dgamma2
        grads['beta2'] = dbeta2
        dW2 += self.reg * W2
        dout, dW1, db1, dgamma1, dbeta1 = conv_bn_relu_backward(dout,
cache1) # conv relu
        grads['gamma1'] = dgamma1
        grads['beta1'] = dbeta1
        dW1 += self.reg * W1


      # storage of w's and b's
      grads["W1"] = dW1
      grads["W2"] = dW2
      grads["W3"] = dW3
      grads["W4"] = dW4
      grads["W5"] = dW5
      grads["W6"] = dW6
      grads["b1"] = db1
      grads["b2"] = db2
      grads["b3"] = db3
      grads["b4"] = db4
      grads["b5"] = db5
      grads["b6"] = db6



      return loss, grads
```