CNN.py

This file contains all the code for cnn.py

Code

```
from platform import architecture
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.
    # ================================================================ #
    self.conv_param = {'stride': 1, 'pad': (filter_size - 1) / 2}
    self.pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}
    C,H,W = input_dim
    self.params['W1'] = np.random.randn(num_filters, C, filter_size,filter_size)*weight_scal
    self.params['b1'] = np.zeros(num_filters)
    H_conv = 1 + (H + 2 * self.conv_param['pad'] - filter_size) / self.conv_param['stride']
    W_conv = 1 + (W + 2 * self.conv_param['pad'] - filter_size) / self.conv_param['stride']
    depth_conv = num_filters
    H_pool = 1 + (H_conv - self.pool_param['pool_height'])/self.pool_param['stride']
    W_pool = 1 + (W_conv - self.pool_param['pool_width'])/self.pool_param['stride']
    depth_pool = num_filters
    self.params['W2'] = np.random.randn(int(H_pool*W_pool*depth_pool),hidden_dim)*weight_sca
    self.params['b2'] = np.zeros(hidden_dim)
    self.params['W3'] = np.random.randn(hidden_dim,num_classes)*weight_scale
    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_out, conv_cache = conv_relu_pool_forward(X,W1, b1, conv_param, pool_param)
h1, h1cache = affine_relu_forward(conv_out,  W2, b2)
h2,h2cache = affine_forward(h1,W3, b3)
scores = h2
# ================================================================ #
# 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, dLdh2 = softmax_loss(scores, y)
dLdh1, grads['W3'], grads['b3'] = affine_backward(dLdh2,h2cache)
dLdconvout, grads['W2'], grads['b2'] = affine_relu_backward(dLdh1,h1cache)
_,grads['W1'], grads['b1'] = conv_relu_pool_backward(dLdconvout, conv_cache)
grads['W1'] += self.reg*self.params['W1']
grads['W2'] += self.reg*self.params['W2']
grads['W3'] += self.reg*self.params['W3']
loss += 0.5*self.reg*np.linalg.norm(self.params['W1'])**2 +0.5*self.reg*np.linalg.norm(s
# ================================================================ #
```

```python
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grads


class CNN(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=[100,100], 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 each 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
    self.num_conv_layers = len(filter_size)
    self.num_fc_layers = len(hidden_dim)
    self.num_layers = self.num_conv_layers+self.num_fc_layers
    self.conv_param = []
    self.pool_param = []
    # ================================================================ #
    # 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
H_pool,W_pool,depth_pool = None,None,None
# Set up CNN
for i in range(self.num_conv_layers):
  w_i = 'W'+str(i+1)
  b_i = 'b'+str(i+1)
  gamma_i = 'gamma'+str(i+1)
  beta_i = 'beta'+str(i+1)
  self.conv_param.append({'stride': 1, 'pad': (filter_size[i] - 1) / 2})
  self.pool_param.append({'pool_height': 2, 'pool_width': 2, 'stride': 2})
  if(i==0):
    self.params[w_i] = np.random.randn(num_filters[i], C, filter_size[i],filter_size[i])
    self.params[b_i] = np.zeros(num_filters[i])
    if self.use_batchnorm:
      self.params[gamma_i] = np.ones((num_filters[i],))
      self.params[beta_i] = np.zeros((num_filters[i],))
    H_conv = 1 + (H + 2 * self.conv_param[i]['pad'] - filter_size[i]) / self.conv_param
    W_conv = 1 + (W + 2 * self.conv_param[i]['pad'] - filter_size[i]) / self.conv_param
    depth_conv = num_filters[i]
    H_pool = 1 + (H_conv - self.pool_param[i]['pool_height'])/self.pool_param[i]['stride
    W_pool = 1 + (W_conv - self.pool_param[i]['pool_width'])/self.pool_param[i]['stride'
    depth_pool = depth_conv
  else:
    self.params[w_i] = np.random.randn(num_filters[i], num_filters[i-1], filter_size[i],
    self.params[b_i] = np.zeros(num_filters[i])
    if self.use_batchnorm:
      self.params[gamma_i] = np.ones((num_filters[i],))
      self.params[beta_i] = np.zeros((num_filters[i],))
    H_conv = 1 + (H_pool + 2 * self.conv_param[i]['pad'] - filter_size[i]) / self.conv_p
    W_conv = 1 + (W_pool + 2 * self.conv_param[i]['pad'] - filter_size[i]) / self.conv_p
    depth_conv = num_filters[i]
    H_pool = 1 + (H_conv - self.pool_param[i]['pool_height'])/self.pool_param[i]['stride
    W_pool = 1 + (W_conv - self.pool_param[i]['pool_width'])/self.pool_param[i]['stride'
    depth_pool = depth_conv
# Setup FCNET
for i in range(self.num_fc_layers):
  w_i = 'fcW'+str(i+1)
  b_i = 'fcb'+str(i+1)
  gamma_i = 'fcgamma'+str(i+1)
  beta_i = 'fcbeta'+str(i+1)
  if (i==0):
```

```
        self.params[w_i] = np.random.randn(int(H_pool*W_pool*depth_pool),hidden_dim[i])*weig
        self.params[b_i] = np.zeros(hidden_dim[i])
        if self.use_batchnorm:
          self.params[gamma_i] = np.ones((hidden_dim[i],))
          self.params[beta_i] = np.zeros((hidden_dim[i],))
      elif (i==self.num_fc_layers-1):
        self.params[w_i] = np.random.randn(hidden_dim[i-1],num_classes)*weight_scale
        self.params[b_i] = np.zeros((num_classes,))
      else:
        self.params[w_i] = np.random.randn(hidden_dim[i-1],hidden_dim[i])*weight_scale
        self.params[b_i] = np.zeros((hidden_dim[i],))
        if(self.use_batchnorm):
          self.params[gamma_i] = np.ones((hidden_dim[i],))
          self.params[beta_i] = np.zeros((hidden_dim[i],))

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    self.bn_params = []
    if self.use_batchnorm:
      self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]

    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 convolutional network.

    Input / output: Same API as TwoLayerNet in fc_net.py.
    """

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

    scores = None
    X = X.astype(self.dtype)
    mode = 'test' if y is None else 'train'
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param['mode'] = mode
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the three layer CNN.  Store the output
    #   scores as the variable "scores".
```

```
# ================================================================ #
conv_caches =[]
fc_caches =[]
conv_cache,h1cache,h2cache = None,None,None
conv_out,h_i = None,None
if self.use_batchnorm:
  for i in range(self.num_conv_layers):
    #generate keys
    w_i = 'W'+str(i+1)
    b_i = 'b'+str(i+1)
    gamma_i = 'gamma'+str(i+1)
    beta_i = 'beta'+str(i+1)
    if(i==0):
      conv_out, conv_cache = conv_relu_pool_batchnorm_forward(X,self.params[w_i], self.p
      conv_caches.append(conv_cache)
    else:
      conv_out, conv_cache = conv_relu_pool_batchnorm_forward(conv_out,self.params[w_i],
      conv_caches.append(conv_cache)
  for i in range(self.num_fc_layers):
    w_i = 'fcW'+str(i+1)
    b_i = 'fcb'+str(i+1)
    gamma_i = 'fcgamma'+str(i+1)
    beta_i = 'fcbeta'+str(i+1)
    if(i==0):
      h_i, hicache = affine_batchnorm_relu_forward(conv_out, self.params[w_i], self.para
      fc_caches.append(hicache)
    elif(i!=self.num_fc_layers-1):
      h_i, hicache = affine_batchnorm_relu_forward(h_i, self.params[w_i], self.params[b_
      fc_caches.append(hicache)
    else:
      h_i,hicache = affine_forward(h_i,self.params[w_i], self.params[b_i])
      fc_caches.append(hicache)
      scores = h_i

else:
  for i in range(self.num_conv_layers):
    #generate keys
    w_i = 'W'+str(i+1)
    b_i = 'b'+str(i+1)
    gamma_i = 'gamma'+str(i+1)
    beta_i = 'beta'+str(i+1)
    if(i==0):
      conv_out, conv_cache = conv_relu_pool_forward(X,self.params[w_i], self.params[b_i]
      conv_caches.append(conv_cache)
    else:
      conv_out, conv_cache = conv_relu_pool_forward(conv_out,self.params[w_i], self.para
```

```
      conv_caches.append(conv_cache)
  for i in range(self.num_fc_layers):
    w_i = 'fcW'+str(i+1)
    b_i = 'fcb'+str(i+1)
    gamma_i = 'fcgamma'+str(i+1)
    beta_i = 'fcbeta'+str(i+1)
    if(i==0):
      h_i, hicache = affine_relu_forward(conv_out, self.params[w_i], self.params[b_i])
      fc_caches.append(hicache)
    elif(i!=self.num_fc_layers-1):
      h_i, hicache = affine_relu_forward(h_i, self.params[w_i], self.params[b_i])
      fc_caches.append(hicache)
    else:
      h_i,hicache = affine_forward(h_i,self.params[w_i], self.params[b_i])
      fc_caches.append(hicache)
      scores = h_i

# ================================================================ #
# 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, dLdupstream = softmax_loss(scores, y)
if self.use_batchnorm:
  for i in reversed(range(self.num_fc_layers)):
    w_i = 'fcW'+str(i+1)
    b_i = 'fcb'+str(i+1)
    gamma_i = 'fcgamma'+str(i+1)
    beta_i = 'fcbeta'+str(i+1)
    if(i==self.num_fc_layers-1):
      dLdupstream, grads[w_i], grads[b_i] = affine_backward(dLdupstream,fc_caches[i])
      grads[w_i]+= self.reg*self.params[w_i]
      loss += 0.5*self.reg*np.linalg.norm(self.params[w_i])**2

    else:
```

```
            dLdupstream,grads[w_i], grads[b_i], grads[gamma_i], grads[beta_i] = affine_batchno
            grads[w_i]+= self.reg*self.params[w_i]
            loss += 0.5*self.reg*np.linalg.norm(self.params[w_i])**2

    for i in reversed(range(self.num_conv_layers)):
        w_i = 'W'+str(i+1)
        b_i = 'b'+str(i+1)
        gamma_i = 'gamma'+str(i+1)
        beta_i = 'beta'+str(i+1)
        dLdupstream,grads[w_i], grads[b_i], grads[gamma_i], grads[beta_i] = conv_relu_pool_b
        grads[w_i]+= self.reg*self.params[w_i]
        loss += 0.5*self.reg*np.linalg.norm(self.params[w_i])**2
else:
    for i in reversed(range(self.num_fc_layers)):
        w_i = 'fcW'+str(i+1)
        b_i = 'fcb'+str(i+1)
        gamma_i = 'fcgamma'+str(i+1)
        beta_i = 'fcbeta'+str(i+1)
        if(i==self.num_fc_layers-1):
            dLdupstream, grads[w_i], grads[b_i] = affine_backward(dLdupstream,fc_caches[i])
            grads[w_i]+= self.reg*self.params[w_i]
            loss += 0.5*self.reg*np.linalg.norm(self.params[w_i])**2

        else:
            dLdupstream,grads[w_i], grads[b_i] = affine_relu_backward(dLdupstream,fc_caches[i]
            grads[w_i]+= self.reg*self.params[w_i]
            loss += 0.5*self.reg*np.linalg.norm(self.params[w_i])**2

    for i in reversed(range(self.num_conv_layers)):
        w_i = 'W'+str(i+1)
        b_i = 'b'+str(i+1)
        gamma_i = 'gamma'+str(i+1)
        beta_i = 'beta'+str(i+1)
        dLdupstream,grads[w_i], grads[b_i] = conv_relu_pool_backward(dLdupstream, conv_cache
        grads[w_i]+= self.reg*self.params[w_i]
        loss += 0.5*self.reg*np.linalg.norm(self.params[w_i])**2

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

return loss, grads
```