CNN.py

In [ ]:
```
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 t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
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 in
put
  channels.
  """

  def __init__(self, input_dim=(3, 32, 32), num_filters=32, filter_siz
e=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 l
ayer.
    - weight_scale: Scalar giving standard deviation for random initia
lization
      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 ini
tialize:
    #     - 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.bn_params = {}
    mu = 0
    stddev = weight_scale

    C = int(input_dim[0])
    H = int(input_dim[1])
    W = int(input_dim[2])

    #figure out padding
    pad = (filter_size -1)/2

    #default stride
    stride = 1

    #figure out filter dims
    H_f = (H + 2*pad - filter_size)/stride + 1
    W_f = (W + 2*pad - filter_size)/stride + 1

    self.params['W1'] = np.random.normal(mu, stddev, (num_filters, C,
filter_size, filter_size))
    self.params['b1'] = np.zeros(num_filters)
```

```
    pool_dim = 2
    pool_stride = 2
    H_pool = (H_f - pool_dim)/pool_stride + 1
    W_pool = (W_f - pool_dim)/pool_stride + 1

    self.params['W2'] = np.random.normal(mu, stddev, (int(num_filters*
H_pool*W_pool), hidden_dim))
    self.params['b2'] = np.zeros(hidden_dim)

    self.params['W3'] = np.random.normal(mu, stddev, (hidden_dim, num_
classes))
    self.params['b3'] = np.zeros(num_classes)

    #set up batchnorm
    if self.use_batchnorm is True:
        self.bn_params['bn_param1'] = {'mode': 'train', 'running_mean'
: np.zeros(num_filters), 'running_var': np.zeros(num_filters)}
        self.params['beta1'] = np.zeros(num_filters)
        self.params['gamma1'] = np.ones(num_filters)

        self.bn_params['bn_param2'] = {'mode': 'train', 'running_mean'
: np.zeros(hidden_dim), 'running_var': np.zeros(hidden_dim)}
        self.params['beta2'] = np.zeros(hidden_dim)
        self.params['gamma2'] = np.ones(hidden_dim)


    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
#    print(self.params.items())
    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 netwo
rk.

    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".
    # =================================================================
#
    #check batchnorm
    mode = 'test' if y is None else 'train'

    #set all to test if we want to test
    if self.use_batchnorm is True:
        for k, v in self.bn_params.items():
            v[mode] = mode

        bn_param1 = self.bn_params['bn_param1']
        bn_param2 = self.bn_params['bn_param2']

        beta1 = self.params['beta1']
        beta2 = self.params['beta2']

        gamma1 = self.params['gamma1']
        gamma2 = self.params['gamma2']



    #  conv - relu - 2x2 max pool - affine - relu - affine - softmax
    if self.use_batchnorm is True:
#         pizza()
        conv_out, conv_cache = conv_relu_pool_forward_batchnorm(X, W1,
b1, conv_param, pool_param, gamma1, beta1, bn_param1)
    else:
        #perform conv, relu, and pool
        conv_out, conv_cache = conv_relu_pool_forward(X, W1, b1, conv_
param, pool_param)

    #affine-relu layer
    N, F, H_out, W_out = conv_out.shape
    conv_out.reshape((N, F*H_out*W_out))
    if self.use_batchnorm is True:
        affine_out, affine_cache = affine_relu_forward_batchnorm(conv_
out, W2, b2, gamma2, beta2, bn_param2)
```

```
    else:
        affine_out, affine_cache = affine_relu_forward(conv_out, W2, b
2)

    #affine
    scores, affine2_cache = affine_forward(affine_out, 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", a
nd
    #   don't forget to add regularization on ALL weight matrices.
    # ================================================================
#
    loss, grad_loss = softmax_loss(scores, y)

    #Add regularization to loss
    loss += 0.5 * self.reg * (np.sum(W1**2) + np.sum(W2**2) + np.sum(W
3**3))

    #affine_back -> relu_back -> affine_back -> conv_relu_pool_back
    #affine_backward returns dx, dw, db
    dx, grads['W3'], grads['b3'] = affine_backward(grad_loss, affine2_
cache)

    if self.use_batchnorm is True:
        dx, dw, db, dgamma2, dbeta2 = affine_relu_backward_batchnorm(d
x, affine_cache)
        grads['beta2'] = dbeta2
        grads['gamma2'] = dgamma2
    else:
        dx, dw, db = affine_relu_backward(dx, affine_cache)
    grads['W2'] = dw
    grads['b2'] = db
```

```
    #conv
    dx = np.reshape(dx, (N, F, H_out, W_out))
    if self.use_batchnorm is True:
        dx, dw, db, dgamma1, dbeta1 = conv_relu_pool_backward_batchnor
m(dx, conv_cache)
        grads['beta1'] = dbeta1
        grads['gamma1'] = dgamma1
    else:
        dx, dw, db = conv_relu_pool_backward(dx, conv_cache)
    grads['W1'] = dw
    grads['b1'] = db

    #regularization
    grads['W1'] += self.reg * W1
    grads['W2'] += self.reg * W2
    grads['W3'] += self.reg * W3




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

    return loss, grads


pass
```