In [ ]:
```
import numpy as np
import pdb

from .layers import *
from .layer_utils import *

"""
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 FullyConnectedNet(object):
  """
  A fully-connected neural network with an arbitrary number of hidden
layers,
  ReLU nonlinearities, and a softmax loss function. This will also imp
lement
  dropout and batch normalization as options. For a network with L lay
ers,
  the architecture will be

  {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - soft
max

  where batch normalization and dropout are optional, and the {...} bl
ock is
  repeated L - 1 times.

  Similar to the TwoLayerNet above, learnable parameters are stored in
the
  self.params dictionary and will be learned using the Solver class.
  """

  def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
               dropout=0, use_batchnorm=False, reg=0.0,
               weight_scale=1e-2, dtype=np.float32, seed=None):
    """
    Initialize a new FullyConnectedNet.
```

```
    Inputs:
    - hidden_dims: A list of integers giving the size of each hidden l
ayer.
    - input_dim: An integer giving the size of the input.
    - num_classes: An integer giving the number of classes to classify
.
    - dropout: Scalar between 0 and 1 giving dropout strength. If drop
out=0 then
      the network should not use dropout at all.
    - use_batchnorm: Whether or not the network should use batch norma
lization.
    - reg: Scalar giving L2 regularization strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - dtype: A numpy datatype object; all computations will be perform
ed using
      this datatype. float32 is faster but less accurate, so you shoul
d use
      float64 for numeric gradient checking.
    - seed: If not None, then pass this random seed to the dropout lay
ers. This
      will make the dropout layers deteriminstic so we can gradient ch
eck the
      model.
    """
    self.use_batchnorm = use_batchnorm
    self.use_dropout = dropout > 0
    self.reg = reg
    self.num_layers = 1 + len(hidden_dims)
    self.dtype = dtype
    self.params = {}

    # ================================================================
#
    # YOUR CODE HERE:
    #   Initialize all parameters of the network in the self.params di
ctionary.
    #   The weights and biases of layer 1 are W1 and b1; and in genera
l the
    #   weights and biases of layer i are Wi and bi. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weigh
t_scale.
    #
    #   BATCHNORM: Initialize the gammas of each layer to 1 and the be
ta
    #   parameters to zero.  The gamma and beta parameters for layer 1
should
    #   be self.params['gamma1'] and self.params['beta1'].  For layer
```

```
2, they
    #   should be gamma2 and beta2, etc. Only use batchnorm if self.us
e_batchnorm
    #   is true and DO NOT batch normalize the output scores.
    # ================================================e================
= #
    mu = 0
    stddev = weight_scale
    """
    self.params['W1'] = std * np.random.randn(hidden_size, input_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(output_size, hidden_size
)
    self.params['b2'] = np.zeros(output_size)

    np.random.normal(mu, stddev, <size>)
    """
    #aggregate all the dims into a single array that we can reference
    #input and output dim (num_classes) will only be used once
    aggregated_dims = [input_dim] + hidden_dims + [num_classes]
    for i in range(self.num_layers):
      #batchnorm on all layers except last one
      #init gamms to 1s and betas to 0
      if self.use_batchnorm and (i != (self.num_layers - 1)):
        self.params['gamma'+str(i+1)] = np.ones(aggregated_dims[i+1])
        self.params['beta'+str(i+1)] = np.zeros(aggregated_dims[i+1])

      self.params['b'+str(i+1)] = np.zeros(aggregated_dims[i+1])
      self.params['W'+str(i+1)] = np.random.normal(mu, stddev, size=(a
ggregated_dims[i], aggregated_dims[i+1]))
    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#

    # When using dropout we need to pass a dropout_param dictionary to
each
    # dropout layer so that the layer knows the dropout probability an
d the mode
    # (train / test). You can pass the same dropout_param to each drop
out layer.
    self.dropout_param = {}
    if self.use_dropout:
      self.dropout_param = {'mode': 'train', 'p': dropout}
      if seed is not None:
        self.dropout_param['seed'] = seed

    # With batch normalization we need to keep track of running means
and
```

```
    # variances, so we need to pass a special bn_param object to each 
batch
    # normalization layer. You should pass self.bn_params[0] to the fo
rward pass
    # of the first batch normalization layer, self.bn_params[1] to the 
forward
    # pass of the second batch normalization layer, etc.
    self.bn_params = []
    if self.use_batchnorm:
      #for i in range(self.num_layers):
      self.bn_params = [{'mode': 'train'} for i in range(self.num_laye
rs - 1)]

    # Cast all parameters to the correct datatype
    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


  def loss(self, X, y=None):
    """
    Compute loss and gradient for the fully-connected net.

    Input / output: Same as TwoLayerNet above.
    """
    X = X.astype(self.dtype)
    mode = 'test' if y is None else 'train'

    # Set train/test mode for batchnorm params and dropout param since 
they
    # behave differently during training and testing.
    if self.dropout_param is not None:
      self.dropout_param['mode'] = mode
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param[mode] = mode

    scores = None

    # ===========================================================
#
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    #
    #   BATCHNORM: If self.use_batchnorm is true, insert a bathnorm la
yer
    #   between the affine_forward and relu_forward layers.  You may
    #   also write an affine_batchnorm_relu() function in layer_utils.
py.
    #
```

```
    #   DROPOUT: If dropout is non-zero, insert a dropout layer after
    #   every ReLU layer.
    # ================================================================
#

    nn_layer = {}
    nn_cache = {}
    batchnorm_cache = {}
    dropout_cache = {}

    #initialize the first layer with the inputs
    nn_layer[0] = X
    #pass through each layer
    for i in range(1, self.num_layers):
#      print("iteration", i)
      gamma_idx = 'gamma'+str(i)
      beta_idx = 'beta'+str(i)
      w_idx = 'W'+str(i)
      b_idx = 'b'+str(i)
      #affine relu forward takes (x, w, b)
      if self.use_batchnorm:
        #args: x, gamma, beta, bn_param
#        nn_layer[i], batchnorm_cache[i] = batchnorm_forward(nn_layer[
i-1], self.params['gamma'+str(i)], self.params['beta'+str(i)], self.bn
_params[i-1])
#        print(nn_layer[i-1].shape, self.params[w_idx].shape)
        nn_layer[i], nn_cache[i] = affine_batchnorm_relu_forward(nn_la
yer[i-1], self.params[w_idx],
                                                                   self
.params[b_idx], self.params[gamma_idx],
                                                                   self
.params[beta_idx], self.bn_params[i-1])
      else:
        nn_layer[i], nn_cache[i] = affine_relu_forward(nn_layer[i-1],
self.params[w_idx], self.params[b_idx])


      if(self.use_dropout):
        nn_layer[i], dropout_cache[i] = dropout_forward(nn_layer[i], s
elf.dropout_param)
    #all layers will have the affine_relu except for the last layer, w
hich is a passthrough
    #affine_forward takes (x, w, b) and outputs out, cache
    w_idx = 'W'+str(self.num_layers)
    b_idx = 'b'+str(self.num_layers)
    scores, cached_scores = affine_forward(nn_layer[self.num_layers -1
], self.params[w_idx], self.params[b_idx])

    nn_cache[self.num_layers] = cached_scores
    # ================================================================
```

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

    # If test mode return early
    if mode == 'test':
      return scores

    loss, grads = 0.0, {}
    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the backwards pass of the FC net and store the gradi
ents
    #   in the grads dict, so that grads[k] is the gradient of self.pa
rams[k]
    #   Be sure your L2 regularization includes a 0.5 factor.
    #
    #   BATCHNORM: Incorporate the backward pass of the batchnorm.
    #
    #   DROPOUT: Incorporate the backward pass of dropout.
    # ================================================================
#
    #get loss w/ softmax loss
    loss, grad_loss = softmax_loss(scores, y)

    #add L2 regularization to loss 1/2*np.sum(w**2)
    for i in range(1, self.num_layers + 1):
      cur_weight_matrix = self.params['W'+str(i)]
      loss += 0.5 * self.reg * np.sum(cur_weight_matrix**2)

    """
    Backpropping into the (n-1)th layer will be different because we d
on't have
    the relu. Use affine_backward and then for each previous layer app
ly affine_relu_backward
    affine_backward takes dout,cache and returns dx,dw,db
    affine_relu_backward takes dout, cachce and returns dx,dw,db
    """
    dx={}
    w_idx_nth = 'W'+str(self.num_layers)
    b_idx_nth = 'b'+str(self.num_layers)
    dx[self.num_layers], grads[w_idx_nth], grads[b_idx_nth] = affine_b
ackward(grad_loss, cached_scores)

#    print(dx[3])
    #regularize
    grads[w_idx_nth] += self.reg * self.params[w_idx_nth]
```

```
    #we apply affine_relu_backward now
    for i in range(self.num_layers - 1, 0, -1):
#      print(i, self.num_layers)

      #dx, dw, db
      w_idx = 'W' + str(i)#+1)
      b_idx = 'b' + str(i)#+1)

      gamma_idx = 'gamma' + str(i)#+1)
      beta_idx = 'beta' + str(i)#+1)


      if self.use_dropout:
        dx[i+1] = dropout_backward(dx[i+1], dropout_cache[i])
      if self.use_batchnorm:
        dx[i], grads[w_idx], grads[b_idx], grads[gamma_idx], grads[bet
a_idx] = affine_batchnorm_relu_backward(dx[i+1], nn_cache[i])


      else:
         #dout input to affine_relu_backward is the
        dx[i], grads[w_idx], grads[b_idx] = affine_relu_backward( dx[i
+1], nn_cache[i])

      #regularize
#      print(grads[w_idx].shape, self.params[w_idx].shape )
      grads[w_idx] += self.reg * self.params[w_idx]


    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
    return loss, grads
```