In [1]:
```
import numpy as np

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 TwoLayerNet(object):
  """
  A two-layer fully-connected neural network with ReLU nonlinearity an
d
  softmax loss that uses a modular layer design. We assume an input di
mension
  of D, a hidden dimension of H, and perform classification over C cla
sses.

  The architecure should be affine - relu - affine - softmax.

  Note that this class does not implement gradient descent; instead, i
t
  will interact with a separate Solver object that is responsible for
running
  optimization.

  The learnable parameters of the model are stored in the dictionary
  self.params that maps parameter names to numpy arrays.
  """

  def __init__(self, input_dim=3*32*32, hidden_dims=100, num_classes=1
0,
               dropout=0, weight_scale=1e-3, reg=0.0):
    """
    Initialize a new network.

    Inputs:
```

```
    - input_dim: An integer giving the size of the input
    - hidden_dims: An integer giving the size of the hidden layer
    - num_classes: An integer giving the number of classes to classify
    - dropout: Scalar between 0 and 1 giving dropout strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - reg: Scalar giving L2 regularization strength.
    """
    self.params = {}
    self.reg = reg

    # ====================================================================
#
    # YOUR CODE HERE:
    #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1
'],
    #   self.params['W2'], self.params['b1'] and self.params['b2']. Th
e
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weigh
t_scale.
    #   The dimensions of W1 should be (input_dim, hidden_dim) and the
    #   dimensions of W2 should be (hidden_dims, num_classes)
    # ====================================================================
#
    mu = 0
    sigma = weight_scale
    self.params['W1'] = np.random.normal(mu, sigma, (input_dim, hidden
_dims))
    self.params['W2'] = np.random.normal(mu, sigma, (hidden_dims, num_
classes))


    self.params['b1'] = np.zeros(shape = (hidden_dims))
    self.params['b2'] = np.zeros(shape = (num_classes))

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

  def loss(self, X, y=None):
    """
    Compute loss and gradient for a minibatch of data.

    Inputs:
    - X: Array of input data of shape (N, d_1, ..., d_k)
    - y: Array of labels, of shape (N,). y[i] gives the label for X[i]
.
```

```
    Returns:
    If y is None, then run a test-time forward pass of the model and r
eturn:
    - scores: Array of shape (N, C) giving classification scores, wher
e
      scores[i, c] is the classification score for X[i] and class c.

    If y is not None, then run a training-time forward and backward pa
ss and
    return a tuple of:
    - loss: Scalar value giving the loss
    - grads: Dictionary with the same keys as self.params, mapping par
ameter
      names to gradients of the loss with respect to those parameters.
    """
     # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']


    # Compute the forward pass
    scores = None
    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. St
ore
    #   the class scores as the variable 'scores'.  Be sure to use the
layers
    #   you prior implemented.
    # ================================================================
#

    #affine_relu_forward returns out, cached
    hl1, hl1_cached = affine_relu_forward(X, W1, b1)

    #affine_forward returns out, cache
    hl2, hl2_cached = affine_forward(hl1, W2, b2)

    scores = hl2
    scores_cached= hl2_cached
    # ================================================================
#
    # END YOUR CODE HERE
    # ================================================================
#
```

```
    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss, grads = 0, {}

    # ================================================================
#
    # YOUR CODE HERE:
    #   Implement the backward pass of the two-layer neural net.  Stor
e
    #   the loss as the variable 'loss' and store the gradients in the
    #   'grads' dictionary.  For the grads dictionary, grads['W1'] hol
ds
    #   the gradient for W1, grads['b1'] holds the gradient for b1, et
c.
    #   i.e., grads[k] holds the gradient for self.params[k].
    #
    #   Add L2 regularization, where there is an added cost 0.5*self.r
eg*W^2
    #   for each W.  Be sure to include the 0.5 multiplying factor to
    #   match our implementation.
    #
    #   And be sure to use the layers you prior implemented.
    # ================================================================
#

    """
    Softmax loss returns:
     Returns a tuple of:
      - loss: Scalar giving the loss
      - dx: Gradient of the loss with respect to x
  """
    loss, grad_loss_wrt_x = softmax_loss(scores, y)
    regularized_loss = 0.5*self.reg*np.sum(W1**2) + 0.5*self.reg*np.su
m(W2**2)
    loss += regularized_loss

    #backprop
    #affine return values:  return dx, dw, db
    dx2, dw2, db2 = affine_backward(grad_loss_wrt_x, scores_cached)

    #add in regularization term to weights
    dw2 += self.reg*W2

    #backprop layer 1
    #relu_backward takes: (dout, cache). In this case the dout is prev
ious chain
    dx1, dw1, db1 = affine_relu_backward(dx2, hl1_cached)
```

```
    dw1 += self.reg*W1

    grads['W2'] = dw2
    grads['b2'] = db2
    grads['W1'] = dw1
    grads['b1'] = db1

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

    return loss, grads


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.
    # ============================================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):
      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:
      self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_
layers - 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".
    # ==================================================================
#

    nn_layer = {}
    nn_cache = {}

    #initialize the first layer with the inputs
    nn_layer[0] = X
    #pass through each layer
    for i in range(1, self.num_layers):
      #affine relu forward takes (x, w, b)
      nn_layer[i], nn_cache[i] = affine_relu_forward(nn_layer[i-1], se
lf.params['W'+str(i)], self.params['b'+str(i)])

    #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])


    # ==================================================================
#
    # 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.
    # ================================================================
#
    #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)

    #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)
      b_idx = 'b' + str(i)

      #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
        grads[w_idx] += self.reg * self.params[w_idx]


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