```
import numpy as np
import pdb

from .layers import *
from .layer_utils import *


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

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

    Note that this class does not implement gradient descent; instead,
it
    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=10,
                 dropout=1, 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']. The
        #   biases are initialized to zero and the weights are
initialized
        #   so that each parameter has mean 0 and standard deviation
weight_scale.
        #   The dimensions of W1 should be (input_dim, hidden_dim) and
the
        #   dimensions of W2 should be (hidden_dims, num_classes)
        #
================================================================ #

        self.params['W1'] = weight_scale * np.random.randn(input_dim,
hidden_dims)
        self.params['b1'] = np.zeros(hidden_dims)
        self.params['W2'] = weight_scale *
np.random.randn(hidden_dims, num_classes)
        self.params['b2'] = np.zeros(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 return:
        - scores: Array of shape (N, C) giving classification scores,
where
          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 pass and
        return a tuple of:
        - loss: Scalar value giving the loss
        - grads: Dictionary with the same keys as self.params, mapping
```

```
parameter
          names to gradients of the loss with respect to those
parameters.
        """
        scores = None

        #
================================================================== #
        # YOUR CODE HERE:
        #   Implement the forward pass of the two-layer neural
network. Store
        #   the class scores as the variable 'scores'.  Be sure to use
the layers
        #   you prior implemented.
        #
================================================================== #

        # Unpack variables from the params dictionary
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']


        h1 = affine_forward(X, W1, b1)
        relu_1 = relu_forward(h1[0])
        h2 = affine_forward(relu_1[0], W2, b2)
        scores = h2[0]

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

        # If y is None then we are in test mode so just return scores
        if y is None:
            return scores

        loss, grads = 0, {}
        #
================================================================== #
        # YOUR CODE HERE:
        #   Implement the backward pass of the two-layer neural net.
Store
        #   the loss as the variable 'loss' and store the gradients in
the
        #   'grads' dictionary.  For the grads dictionary, grads['W1']
holds
        #   the gradient for W1, grads['b1'] holds the gradient for
b1, etc.
        #   i.e., grads[k] holds the gradient for self.params[k].
```

```
        #
        #   Add L2 regularization, where there is an added cost
0.5*self.reg*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 = softmax_loss(h2[0], y)
        loss = softmax[0] + 0.5 * self.reg * (np.sum(W1 ** 2) +
np.sum(W2 ** 2))

        (h2_grad, grads['W2'], grads['b2']) =
affine_backward(softmax[1], (relu_1[0], W2, b2))

        relu_grad = relu_backward(h2_grad, h1[0])

        (h1_grad, grads['W1'], grads['b1']) =
affine_backward(relu_grad, (X, W1, b1))

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

        #
================================================================ #
        # 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
implement
    dropout and batch normalization as options. For a network with L
layers,
    the architecture will be

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

    where batch normalization and dropout are optional, and the {...}
block 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=1, 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 layer.
        - 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
dropout=1 then
          the network should not use dropout at all.
        - use_batchnorm: Whether or not the network should use batch
normalization.
        - 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
performed using
          this datatype. float32 is faster but less accurate, so you
should use
          float64 for numeric gradient checking.
        - seed: If not None, then pass this random seed to the dropout
layers. This
          will make the dropout layers deteriminstic so we can
gradient check the
          model.
        """
        self.use_batchnorm = use_batchnorm
        self.use_dropout = dropout < 1
        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 dictionary.
        #   The weights and biases of layer 1 are W1 and b1; and in
general 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
weight_scale.
        #
        #   BATCHNORM: Initialize the gammas of each layer to 1 and
the beta
        #   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.use_batchnorm
        #   is true and DO NOT do batch normalize the output scores.
        #
================================================================ #


        self.params['W1'] = weight_scale * np.random.randn(input_dim,
hidden_dims[0])
        self.params['b1'] = np.zeros(hidden_dims[0])

        for i in range(1, self.num_layers - 1):
          w = str("W" + str(i + 1))
          b = str("b" + str(i + 1))
          self.params[w] = weight_scale *
np.random.randn(hidden_dims[i-1], hidden_dims[i])
          self.params[b] = np.zeros(hidden_dims[i])


        w = str("W" + str(self.num_layers))
        b = str("b" + str(self.num_layers))
        self.params[w] = weight_scale *
np.random.randn(hidden_dims[-1], num_classes)
        self.params[b] = np.zeros(num_classes)

        # batch norm
        if self.use_batchnorm:
          for i in range(self.num_layers - 1):
            self.params['gamma'+str(i+1)] = np.ones(hidden_dims[i])
            self.params['beta'+str(i+1)] = np.zeros(hidden_dims[i])




        #
================================================================ #
```

```
        # 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 and the mode
        # (train / test). You can pass the same dropout_param to each
dropout 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 forward 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():
          # if type(v) != int:
          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 layer
        #   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.
        #
================================================================ #

        # try again
        layer_input = X
        relu_cache = []
        affine_cache = []
        bn_cache = []
        drop_cache = []

        for i in range(self.num_layers):
          layer_out, acache = affine_forward(layer_input,
self.params['W'+str(i+1)], self.params['b'+str(i+1)])
          affine_cache.append(acache)

          # batchnorm/dropout maybe
          if i != self.num_layers - 1:
            if self.use_batchnorm:
              layer_out, bcache = batchnorm_forward(layer_out,
self.params['gamma'+str(i+1)], self.params['beta'+str(i+1)],
self.bn_params[i])
              bn_cache.append(bcache)

            # relu
            layer_out, rcache = relu_forward(layer_out)
            relu_cache.append(rcache)
```

```
            # potentially dropout
            if self.use_dropout:
              layer_out, dcache = dropout_forward(layer_out,
self.dropout_param)
              drop_cache.append(dcache)


          layer_input = layer_out

        # last layer scores
        scores = layer_out

        # if self.use_batchnorm:
        #   # use batchnorm
        #   print("using batchnorm")

        #   Hs = [X]
        #   caches = []

        #   print("num layers", self.num_layers)


        #   # { affine - batchnorm - relu } * ( L-1 )
        #   for i in range(self.num_layers - 1):
        #     # print(i)
        #     W = self.params[str('W' + str(i+1))]
        #     b = self.params[str('b' + str(i+1))]
        #     gamma = self.params[str('gamma' + str(i+1))]
        #     beta = self.params[str('beta' + str(i+1))]
        #     H = Hs[-1]
        #     out, cache = affine_batchnorm_relu_forward(H, W, b,
gamma, beta, self.bn_params[i])
        #     # print("w shape", W.shape)
        #     # print("b shape", b.shape)
        #     # print("x shape", H.shape)

        #     Hs.append(out)
        #     caches.append(cache)

        #   # affine - (softmax is in the next part)

        #   # print(i+1)
        #   W = self.params[str('W' + str(i+2))]
        #   b = self.params[str('b' + str(i+2))]
        #   H = Hs[-1]
        #   # print("w shape", W.shape)
        #   # print("b shape", b.shape)
        #   # print("x shape", H.shape)
        #   out, cache = affine_forward(H, W, b)
        #   Hs.append(out)
```

```
        #   caches.append(cache)



        # # forward prop w/o batch norm
        # else:
        #   Hs = [X]
        #   Zs = [X]
        #   Ws = []
        #   bs = []

        #   W = self.params['W1']
        #   b = self.params['b1']
        #   aff_fwd = affine_forward(X, W, b)
        #   Z = aff_fwd[0]

        #   for i in range(1, self.num_layers):
        #     relu_h = relu_forward(Z)
        #     H = relu_h[0]
        #     Hs.append(H)
        #     Ws.append(W)
        #     Zs.append(Z)
        #     bs.append(b)

        #     H = Hs[-1]
        #     W = self.params[str('W' + str(i+1))]
        #     b = self.params[str('b' + str(i+1))]

        #     aff_fwd = affine_forward(H, W, b)
        #     Z = aff_fwd[0]
        #   scores = Z

        #   Zs.append(Z)
        #   Ws.append(W)
        #   bs.append(b)


        #
================================================================ #
        # 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
gradients
        #   in the grads dict, so that grads[k] is the gradient of
self.params[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.
        #
================================================================ #

        # print('num layers', self.num_layers)

        loss, grad = softmax_loss(scores, y)
        up_grad = grad

        for i in reversed(np.arange(self.num_layers)):
          # print(i)
          # add regularization to loss
          loss += 0.5 * self.reg * np.sum((self.params['W'+str(i+1)] *
self.params['W'+str(i+1)]))

          # backward pass
          if i == 0:
            da, dw, db = affine_backward(up_grad, affine_cache.pop())

          else:
            da, dw, db = affine_backward(up_grad, affine_cache.pop())

            if self.use_dropout:
              da = dropout_backward(da, drop_cache.pop())

            dz = relu_backward(da, relu_cache.pop())
            if self.use_batchnorm:
              dz, dgamma, dbeta = batchnorm_backward(dz,
bn_cache.pop())
              grads['gamma'+str(i)] = dgamma
              grads['beta'+str(i)] = dbeta

          grads['W'+str(i+1)] = dw + self.reg *
self.params['W'+str(i+1)]
          grads['b'+str(i+1)] = db

          up_grad = dz

        # print(grads.keys())
```

```
        # print(grads.keys())

        # softmax = softmax_loss(scores, y)

        # # regularization
        # agg_sum = 0
        # for i in range(self.num_layers):
        #   agg_sum += np.sum(self.params[str('W' + str(i+1))] ** 2)

        # loss = softmax[0] + 0.5 * self.reg * agg_sum

        # print('i got here')

        # loss_grads = [softmax[1]]
        # loss_grad = loss_grads[-1]

        # print("num layers", self.num_layers)


        # # first do the affine layer
        # print(self.num_layers-1)
        # dx, grads['W' + str(self.num_layers)], grads['b' +
str(self.num_layers)] = affine_backward(loss_grad,
affine_cache[self.num_layers-1])
        # # print("x shape", affine_cache[self.num_layers][0].shape)
        # # print("w shape", affine_cache[self.num_layers][1].shape)
        # # print("b shape", affine_cache[self.num_layers-1][2].shape)

        # # print("dx shape", dx.shape)
        # # print("dw shape", grads['W' +
str(self.num_layers-1)].shape)
        # # print("db shape", grads['b' +
str(self.num_layers-1)].shape)

        # loss_grads.append(dx)

        # # then the remaining layers
        # for i in range(self.num_layers-1, 0, -1):
        #   print(i-1)
        #   loss_grad = loss_grads[-1]
        #   relu_grad = relu_backward(loss_grad, relu_cache[i-1])

        #   if self.use_batchnorm:
        #     d_batchnorm, grads['gamma' + str(i)], grads['beta' +
str(i)] = batchnorm_backward(relu_grad, bn_cache[i-1])
        #     dx, grads['W' + str(i)], grads['b' + str(i)] =
affine_backward(d_batchnorm, affine_cache[i-1])

        #   else:
        #     dx, grads['W' + str(i)], grads['b' + str(i)] =
```

```
affine_backward(loss_grad, affine_cache[i-1])

        #   # print("x shape", affine_cache[i-1][0].shape)
        #   # print("w shape", affine_cache[i-1][1].shape)
        #   # print("b shape", affine_cache[i-1][2].shape)

        #   # print("dx shape", dx.shape)
        #   # print("dw shape", grads['W' + str(i-1)].shape)
        #   # print("db shape", grads['b' + str(i-1)].shape)

        #   loss_grads.append(dx)



        # # # back prop
        # # if self.use_batchnorm: # batch norm
        # #   loss_grads = [softmax[1]]
        # #   loss_grad = loss_grads[-1]
        # #   cache = caches[-1]

        # #   dx, grads['W' + str(self.num_layers-1)], grads['b' +
str(self.num_layers-1)] = affine_backward(loss_grad, cache)
        # #   loss_grads.append(dx)

        # #   for i in range(self.num_layers-1, 1, -1):
        # #     print(i)
        # #     loss_grad = loss_grads[-1]
        # #     dx, grads[str('W' + str(i))], grads[str('b' + str(i))]
= affine_batchnorm_relu_backward(loss_grad, caches[i-1])
        # #     loss_grads.append(dx)

        # # else: # no batchnorm
        # #   loss_grads = [softmax[1]]

        # #   for i in range(self.num_layers, 1, -1):
        # #     loss_grad = loss_grads[-1]
        # #     (h_grad, grads[str('W' + str(i))], grads[str('b' +
str(i))]) = affine_backward(loss_grad, (Hs[i-1], Ws[i-1], bs[i-1]))
        # #     relu_grad = relu_backward(h_grad, Zs[i-1])
        # #     loss_grads.append(relu_grad)

        # #   loss_grad = loss_grads[-1]
        # #   (h_grad, grads['W1'], grads['b1']) =
affine_backward(loss_grad, (X, Ws[0], bs[0]))

        # print(grads.keys())

        # for i in range(self.num_layers):
        #   print("i: ", i)
        #   print("self params shape", self.params[str('W' +
```

```
str(i+1))].shape)

        # for i in range(self.num_layers):
        #   print("i: ", i)
        #   print("w grad shape", grads[str('W' + str(i+1))].shape)

        # # regularization
        # for i in range(self.num_layers):
        #   print("i: ", i)
        #   print("grad shape", grads[str('W' + str(i+1))].shape)
        #   print("w shape", self.params[str('W' + str(i+1))].shape)
        #   grads[str('W' + str(i+1))] += self.reg *
self.params[str('W' + str(i+1))]

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

        return loss, grads
```