```python
import numpy as np

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=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']. 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(hidden_dims, input_dim)
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['W2'] = weight_scale * np.random.randn(num_classes, hidden_dims)
    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.
    # ================================================================ #
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    reg = self.reg
    h1, cache_h1 = affine_relu_forward(X, W1, b1)
    z2, cache_z2 = affine_forward(h1, W2, b2)
    scores = z2

    # ================================================================ #
    # 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.
    # ================================================================ #
    loss, dL = softmax_loss(scores, y)
    dh1, dw2, db2 = affine_backward(dL, cache_z2)
    dx, dw1, db1 = affine_relu_backward(dh1, cache_h1)

    reg_loss = reg*0.5*(np.linalg.norm(W1, ord='fro')**2 + np.linalg.norm(W2, ord='fro')**2)
    loss += reg_loss
    dw1 += reg*W1
    dw2 += reg*W2

    grads['W1'] = dw1
    grads['b1'] = db1
```

```
    grads['W2'] = dw2
    grads['b2'] = db2

    # ================================================================ #
    # 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=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 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=0 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 > 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 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.
    # ================================================================ #
```

```
    for i, dims in enumerate(hidden_dims):
        if self.use_batchnorm:
            prev_size = (input_dim if i == 0 else hidden_dims[i-1])
            self.params[f'gamma_{i+1}'] = np.ones(prev_size)
            self.params[f'beta_{i+1}'] = np.zeros(prev_size)
        if i == 0:
            self.params['W1'] = weight_scale * np.random.randn(hidden_dims[i], input_dim)
            self.params['b1'] = np.zeros(hidden_dims[i])
        else:
            self.params[f'W{i+1}'] = weight_scale * np.random.randn(hidden_dims[i], hidden_dim
s[i-1])
            self.params[f'b{i+1}'] = np.zeros(hidden_dims[i])
    L = len(hidden_dims)+1
    self.params[f'W{L}'] = weight_scale * np.random.randn(num_classes, hidden_dims[len(hidden_
dims)-1])
    self.params[f'b{L}'] = np.zeros(num_classes)

    # ================================================================ #
    # 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():
        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.
    """
    """
    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.
        """
        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".
        # ================================================================ #
        e = 1e-8
        reg = self.reg
        cache = []
        state = np.reshape(X, (len(X),-1))
        N = len(X)
        for i in range(1, self.num_layers):
            assert len(state) == N
            # Assuming state: (N x D)
            state, cache_ = affine_forward(state, self.params[f'W{i}'].T, self.params[f'b{i}'])
            cache.append(cache_)
            assert len(state) == N
            # Assuming state: (N x M)
            if self.use_batchnorm:
                mu = state.mean(axis=0)[:,np.newaxis]
                var = state.var(axis=0)[:,np.newaxis]
                x_hat = (state - mu)/np.sqrt(var+e)
                cache.append((mu, var, x_hat, state))
                state = self.params[f'gamma_{i}']*x_hat + self.params[f'beta_{i}']
                assert len(state) == N

            state, cache_ = relu_forward(state)
            cache.append(cache_)
            assert len(state) == N

            if self.use_dropout:
                if mode == 'test':
                    state *= (1-self.dropout_param['p'])
                else:
                    M = np.random.rand((state.shape[1])) < self.dropout_param['p']
                    state *= M
                    cache.append(M)
                assert len(state) == N


        L = self.num_layers
        state, cache_ = affine_forward(state, self.params[f'W{L}'].T, self.params[f'b{L}'])
        cache.append(cache_)
        scores = state
```

```
    # ================================================================ #
    # 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.
    # ================================================================ #

    loss, dL = softmax_loss(scores, y)
    dL, dw, db = affine_backward(dL, cache.pop())
    grads[f'W{L}'] = dw.T + reg*self.params[f'W{L}']
    grads[f'b{L}'] = db
    assert grads[f'W{L}'].shape == self.params[f'W{L}'].shape, f"{grads[f'W{L}'].shape=} | {se
lf.params[f'W{L}'].shape=}"
    assert grads[f'b{L}'].shape == self.params[f'b{L}'].shape, f"{grads[f'b{L}'].shape=} | {se
lf.params[f'b{L}'].shape=}"
    reg_loss = reg*0.5*(np.linalg.norm(self.params[f'W{L}'], ord='fro')**2)

    for i in range(self.num_layers-1, 0, -1):
        if self.use_dropout:
            if mode == 'test':
                dL *= (1-self.dropout_param['p'])
            else:
                M = cache.pop()
                dL *= M

        dL = relu_backward(dL, cache.pop())

        # Assuming dL: (N x D) - Each row is a feature, each column an observation. Sum along
observations
        if self.use_batchnorm:
            mu, var, x_hat, x = cache.pop()
            dbeta = dL.sum(axis=0)
            dgamma = (dL*x_hat).sum(x=0)
            dx_hat = dL*self.params[f'gamma_{i}']
            dmu = (-1/(var+e))*dx_hat.sum(axis=0)
            dvar = (-0.5/((var+e)**(3.0/2.0))) * ((x - mu)*dx_hat).sum(axis=0)
            dL = 1/((var+e)**(1.0/2.0))*dx_hat + 2*((x - mu)/len(m))*dvar + (1.0/len(x))*dmu
            grads[f'beta_{i}'] = dbeta
            grads[f'gamma_{i}'] = dgamma
            assert grads[f'gamma_{i}'].shape == self.params[f'gamma_{i}'].shape,\
                   f"{grads[f'gamma_{i}'].shape=} | {self.params[f'gamma_{i}'].shape=}"
            assert grads[f'beta_{i}'].shape == self.params[f'beta_{i}'].shape,\
                   f"{grads[f'beta_{i}'].shape=} | {self.params[f'beta_{i}'].shape=}"

        dL, dw, db = affine_backward(dL, cache.pop())
        grads[f'W{i}'] = dw.T + reg*self.params[f'W{i}']
        grads[f'b{i}'] = db
        assert grads[f'W{i}'].shape == self.params[f'W{i}'].shape, f"{grads[f'W{i}'].shape=} |
 {self.params[f'W{i}'].shape=}"
        assert grads[f'b{i}'].shape == self.params[f'b{i}'].shape, f"{grads[f'b{i}'].shape=} |
 {self.params[f'b{i}'].shape=}"
        reg_loss += reg*0.5*(np.linalg.norm(self.params[f'W{i}'], ord='fro')**2)

    loss += reg_loss
```

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