```
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 = {}
    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.
    # ================================================================
```

```
#

    # grads['W2'], grads['b2'], grads['W1'], grads['b1'] = None, None,
None, None
    # W1, b1 = self.params['W1'], self.params['b1']
    # W2, b2 = self.params['W2'], self.params['b2']

    softmax = softmax_loss(h2[0], y)
    loss = softmax[0] + 0.5 * self.reg * (np.sum(W1 ** 2) + np.sum(W2
** 2))

    # h1_grad, h2_grad, relu_grad = None, None, None

    # since w2 has problems need to check if h2[0] is correct shape
but i think it is?
    # need to draw out and check dims
    # print("h2[0] shape", h2[0].shape)
    # print("w2 shape", W2.shape)
    # print("b2 shape", b2.shape)
    # print("W1 shape", W1.shape)
    # print("b1 shape", b1.shape)
    # print("relu shape", relu_1[0].shape)
    # print("x shape", X.shape)
    (h2_grad, grads['W2'], grads['b2']) = affine_backward(softmax[1],
(relu_1[0], W2, b2))

    relu_grad = relu_backward(h2_grad, h1[0])

    # print("W2 shape", W2.shape)
    # print("grads['W2'] shape", grads['W2'].shape)


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

    # print("grads w2", grads['W2'].shape)
    # print("w2 shape", W2.shape)

    # add regularization
    # print("w1 shape", W1.shape)
    # print("w1 grad shape", grads['W1'].shape)

    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=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.
    # ============================================================
#

    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)

    # for key in self.params:
    #   print(key, self.params[key].shape)
```

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


    # forward prop
    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)
    # print("done with fwd pass")

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

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

    softmax = softmax_loss(Zs[-1], y)

    # regularization
    agg_sum = 0
    for W in Ws:
      agg_sum += np.sum(W ** 2)

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

    # backprop
    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]))

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


    # for key in grads.keys():
```

```
    #   print(key, grads[key].shape)


    # loss_grad = loss_grads[-1]
    # (h_grad, grads['W3'], grads['b3']) = affine_backward(loss_grad,
(Zs[-1], Ws[-1], bs[-1]))
    # print("loss grad shape", loss_grad[0].shape)
    # print("h grad shape", h_grad.shape)

    # # print("w grad size", w_grad.shape)

    # for i in range(self.num_layers -1, 0, -1):
    #   print(i)
    #   loss_grad = loss_grads[-1]
    #   print("h_grad size", h_grad.shape)
    #   print("Hs[i] shape", Hs[i-1].shape)
    #   relu_grad = relu_backward(h_grad, Hs[i-1])
    #   print("relu grad shape", relu_grad.shape)
    #   print("h shape", Hs[i-1].shape)
    #   print("w shape", Ws[i-1].shape)
    #   print("b shape", bs[i-1].shape)
    #   (h_grad, w_grad, b_grad) = affine_backward(relu_grad,
(Hs[i-1], Ws[i-1], bs[i-1]))
    #   # print("new h shape", h_grad.shape)
    #   loss_grads.append(h_grad)
    #   print(str('W' + str(i)))
    #   grads[str('W' + str(i))] = w_grad
    #   grads[str('b' + str(i))] = w_grad




    # regularization




      # print("i", i)
      # print(str('W'+str(i+1)))
      # # need to figure this part out
      # print(grads['W1'])
      # #self.reg*self.params['W1']
      # print("i got here 2")

    # j = 0
    # for W in Ws:
    #   ≈
    #   j = j+1
```

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

    # relu_grad = relu_backward(h2_grad, h1[0])

    # # print("W2 shape", W2.shape)
  # # print("grads['W2'] shape", grads['W2'].shape)


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

    # # print("grads w2", grads['W2'].shape)
    # # print("w2 shape", W2.shape)

    # # add regularization
    # # print("w1 shape", W1.shape)
    # # print("w1 grad shape", grads['W1'].shape)

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

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