HW3_Q4_Q5_Helper_Files

February 7, 2025

1 Question 4 - 2 Layer Neural Network

1.1 data_utils.py

Complete the two-layer neural network Jupyter notebook. Print out the entire notebook and relevant code and submit it as a pdf to gradescope. Download the CIFAR-10 dataset, as you did in HW #2.

[4]:
```
from __future__ import print_function

from six.moves import cPickle as pickle
import numpy as np
import os
from matplotlib.pyplot import imread
import platform

def load_pickle(f):
    version = platform.python_version_tuple()
    if version[0] == '2':
        return  pickle.load(f)
    elif version[0] == '3':
        return  pickle.load(f, encoding='latin1')
    raise ValueError("invalid python version: {}".format(version))

def load_CIFAR_batch(filename):
    """ load single batch of cifar """
    with open(filename, 'rb') as f:
        datadict = load_pickle(f)
        X = datadict['data']
        Y = datadict['labels']
        X = X.reshape(10000, 3, 32, 32).transpose(0,2,3,1).astype("float")
        Y = np.array(Y)
        return X, Y

# def load_CIFAR10(ROOT):
#     """ load all of cifar """
#     xs = []
#     ys = []
```

```
#     for b in range(1,6):
#         f = os.path.join(ROOT, 'data_batch_%d' % (b, ))
#         X, Y = load_CIFAR_batch(f)
#         xs.append(X)
#         ys.append(Y)
#     Xtr = np.concatenate(xs)
#     Ytr = np.concatenate(ys)
#     del X, Y
#     Xte, Yte = load_CIFAR_batch(os.path.join(ROOT, 'test_batch'))
#     return Xtr, Ytr, Xte, Yte

def load_CIFAR10(ROOT):
    """Load all of CIFAR-10 using absolute paths."""
    xs = []
    ys = []
    """ NOTE FOR THE GRADERS: I had something going on with my join ROOT␣
↪function
        so I decided to simply manually join the file directory with a similar␣
↪for loop
        because I kept getting the same error despite having the correct␣
↪directory
        You can see I tested to see if the directory is present in the normal␣
↪code"""
    for b in range(1, 6):
        f = f"/Users/ctang/Desktop/ECE_C147/HW3/cifar-10-batches-py/
↪data_batch_{b}"
        if not os.path.exists(f):
            raise FileNotFoundError(f"File not found: {f}")
        X, Y = load_CIFAR_batch(f)
        xs.append(X)
        ys.append(Y)

    Xtr = np.concatenate(xs)
    Ytr = np.concatenate(ys)
    del X, Y

    test_file = "/Users/ctang/Desktop/ECE_C147/HW3/cifar-10-batches-py/
↪test_batch"
    if not os.path.exists(test_file):
        raise FileNotFoundError(f"File not found: {test_file}")

    Xte, Yte = load_CIFAR_batch(test_file)
    return Xtr, Ytr, Xte, Yte


def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,
```

```python
                        subtract_mean=True):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for classifiers. These are the same steps as we used for the SVM, but
    condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '/Users/ctang/Desktop/ECE_C147/HW3/cifar-10-batches-py/'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    if subtract_mean:
        mean_image = np.mean(X_train, axis=0)
        X_train -= mean_image
        X_val -= mean_image
        X_test -= mean_image

    # Transpose so that channels come first
    X_train = X_train.transpose(0, 3, 1, 2).copy()
    X_val = X_val.transpose(0, 3, 1, 2).copy()
    X_test = X_test.transpose(0, 3, 1, 2).copy()

    # Package data into a dictionary
    return {
      'X_train': X_train, 'y_train': y_train,
      'X_val': X_val, 'y_val': y_val,
      'X_test': X_test, 'y_test': y_test,
    }


def load_tiny_imagenet(path, dtype=np.float32, subtract_mean=True):
    """
    Load TinyImageNet. Each of TinyImageNet-100-A, TinyImageNet-100-B, and
    TinyImageNet-200 have the same directory structure, so this can be used
    to load any of them.
```

```
    Inputs:
    - path: String giving path to the directory to load.
    - dtype: numpy datatype used to load the data.
    - subtract_mean: Whether to subtract the mean training image.

    Returns: A dictionary with the following entries:
    - class_names: A list where class_names[i] is a list of strings giving the
    WordNet names for class i in the loaded dataset.
    - X_train: (N_tr, 3, 64, 64) array of training images
    - y_train: (N_tr,) array of training labels
    - X_val: (N_val, 3, 64, 64) array of validation images
    - y_val: (N_val,) array of validation labels
    - X_test: (N_test, 3, 64, 64) array of testing images.
    - y_test: (N_test,) array of test labels; if test labels are not available
    (such as in student code) then y_test will be None.
    - mean_image: (3, 64, 64) array giving mean training image
    """
    # First load wnids
    with open(os.path.join(path, 'wnids.txt'), 'r') as f:
        wnids = [x.strip() for x in f]

    # Map wnids to integer labels
    wnid_to_label = {wnid: i for i, wnid in enumerate(wnids)}

    # Use words.txt to get names for each class
    with open(os.path.join(path, 'words.txt'), 'r') as f:
        wnid_to_words = dict(line.split('\t') for line in f)
        for wnid, words in wnid_to_words.iteritems():
          wnid_to_words[wnid] = [w.strip() for w in words.split(',')]
    class_names = [wnid_to_words[wnid] for wnid in wnids]

    # Next load training data.
    X_train = []
    y_train = []
    for i, wnid in enumerate(wnids):
        if (i + 1) % 20 == 0:
            print('loading training data for synset %d / %d' % (i + 1,␣
↪len(wnids)))
    # To figure out the filenames we need to open the boxes file
    boxes_file = os.path.join(path, 'train', wnid, '%s_boxes.txt' % wnid)
    with open(boxes_file, 'r') as f:
      filenames = [x.split('\t')[0] for x in f]
    num_images = len(filenames)

    X_train_block = np.zeros((num_images, 3, 64, 64), dtype=dtype)
    y_train_block = wnid_to_label[wnid] * np.ones(num_images, dtype=np.int64)
    for j, img_file in enumerate(filenames):
```

```
        img_file = os.path.join(path, 'train', wnid, 'images', img_file)
        img = imread(img_file)
        if img.ndim == 2:
        ## grayscale file
            img.shape = (64, 64, 1)
        X_train_block[j] = img.transpose(2, 0, 1)
    X_train.append(X_train_block)
    y_train.append(y_train_block)

    # We need to concatenate all training data
    X_train = np.concatenate(X_train, axis=0)
    y_train = np.concatenate(y_train, axis=0)

    # Next load validation data
    with open(os.path.join(path, 'val', 'val_annotations.txt'), 'r') as f:
        img_files = []
        val_wnids = []
        for line in f:
            img_file, wnid = line.split('\t')[:2]
            img_files.append(img_file)
            val_wnids.append(wnid)
        num_val = len(img_files)
        y_val = np.array([wnid_to_label[wnid] for wnid in val_wnids])
        X_val = np.zeros((num_val, 3, 64, 64), dtype=dtype)
        for i, img_file in enumerate(img_files):
          img_file = os.path.join(path, 'val', 'images', img_file)
          img = imread(img_file)
          if img.ndim == 2:
            img.shape = (64, 64, 1)
          X_val[i] = img.transpose(2, 0, 1)

    # Next load test images
    # Students won't have test labels, so we need to iterate over files in the
    # images directory.
    img_files = os.listdir(os.path.join(path, 'test', 'images'))
    X_test = np.zeros((len(img_files), 3, 64, 64), dtype=dtype)
    for i, img_file in enumerate(img_files):
        img_file = os.path.join(path, 'test', 'images', img_file)
        img = imread(img_file)
        if img.ndim == 2:
            img.shape = (64, 64, 1)
        X_test[i] = img.transpose(2, 0, 1)

    y_test = None
    y_test_file = os.path.join(path, 'test', 'test_annotations.txt')
    if os.path.isfile(y_test_file):
        with open(y_test_file, 'r') as f:
```

```
          img_file_to_wnid = {}
          for line in f:
            line = line.split('\t')
            img_file_to_wnid[line[0]] = line[1]
        y_test = [wnid_to_label[img_file_to_wnid[img_file]] for img_file in␣
↪img_files]
        y_test = np.array(y_test)

        mean_image = X_train.mean(axis=0)
        if subtract_mean:
            X_train -= mean_image[None]
            X_val -= mean_image[None]
            X_test -= mean_image[None]

        return {
        'class_names': class_names,
        'X_train': X_train,
        'y_train': y_train,
        'X_val': X_val,
        'y_val': y_val,
        'X_test': X_test,
        'y_test': y_test,
        'class_names': class_names,
        'mean_image': mean_image,
        }


def load_models(models_dir):
    """
    Load saved models from disk. This will attempt to unpickle all files in a
    directory; any files that give errors on unpickling (such as README.txt)␣
↪will
    be skipped.

    Inputs:
    - models_dir: String giving the path to a directory containing model files.
    Each model file is a pickled dictionary with a 'model' field.

    Returns:
    A dictionary mapping model file names to models.
    """
    models = {}
    for model_file in os.listdir(models_dir):
        with open(os.path.join(models_dir, model_file), 'rb') as f:
            try:
                models[model_file] = load_pickle(f)['model']
            except pickle.UnpicklingError:
```

```
                continue
    return models
```

1.2 neural_net.py

[7]:
```
import numpy as np
import matplotlib.pyplot as plt


class TwoLayerNet(object):
    """
    A two-layer fully-connected neural network. The net has an input dimension␣
 ↪of
    D, a hidden layer dimension of H, and performs classification over C␣
 ↪classes.
    We train the network with a softmax loss function and L2 regularization on␣
 ↪the
    weight matrices. The network uses a ReLU nonlinearity after the first fully
    connected layer.

    In other words, the network has the following architecture:

    input - fully connected layer - ReLU - fully connected layer - softmax

    The outputs of the second fully-connected layer are the scores for each␣
 ↪class.
    """

    def __init__(self, input_size, hidden_size, output_size, std=1e-4):
        """
        Initialize the model. Weights are initialized to small random values and
        biases are initialized to zero. Weights and biases are stored in the
        variable self.params, which is a dictionary with the following keys:

        W1: First layer weights; has shape (H, D)
        b1: First layer biases; has shape (H,)
        W2: Second layer weights; has shape (C, H)
        b2: Second layer biases; has shape (C,)

        Inputs:
        - input_size: The dimension D of the input data.
        - hidden_size: The number of neurons H in the hidden layer.
        - output_size: The number of classes C.
        """
        # self.params = {}
        # 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)
        self.params = {}
        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)

    def loss(self, X, y=None, reg=0.0):
        """
        Compute the loss and gradients for a two layer fully connected neural
        network.

        Inputs:
        - X: Input data of shape (N, D). Each X[i] is a training sample.
        - y: Vector of training labels. y[i] is the label for X[i], and each␣
↪y[i] is
        an integer in the range 0 <= y[i] < C. This parameter is optional; if it
        is not passed then we only return scores, and if it is passed then we
        instead return the loss and gradients.
        - reg: Regularization strength.

        Returns:
        If y is None, return a matrix scores of shape (N, C) where scores[i, c]␣
↪is
        the score for class c on input X[i].

        If y is not None, instead return a tuple of:
        - loss: Loss (data loss and regularization loss) for this batch of␣
↪training
        samples.
        - grads: Dictionary mapping parameter names to gradients of those␣
↪parameters
        with respect to the loss function; has the same keys as self.params.
        """
        # Unpack variables from the params dictionary
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        N, D = X.shape

        # Compute the forward pass
        scores = None
        h1 = np.dot(X, W1.T) + b1
        h1[h1 <= 0] = 0
        h2 = np.dot(h1, W2.T) + b2
        # ================================================================ #
        # YOUR CODE HERE:
```

```
#   Calculate the output scores of the neural network.  The result
#   should be (N, C). As stated in the description for this class,
#   there should not be a ReLU layer after the second FC layer.
#   The output of the second FC layer is the output scores. Do not
#   use a for loop in your implementation.
# ================================================================ #
real_scores = h2
scores = h2 - np.max(h2, axis=1, keepdims=True)
exp_scores = np.exp(scores)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
#pass

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


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

# Compute the loss
loss = None

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the loss of the neural network.  This includes the
#   softmax loss and the L2 regularization for W1 and W2. Store the
#   total loss in teh variable loss.  Multiply the regularization
#   loss by 0.5 (in addition to the factor reg).
# ================================================================ #
# scores is num_examples by num_classes
num_examples = X.shape[0]
correct_logprobs = -np.log(probs[range(num_examples), y])
data_loss = np.sum(correct_logprobs) / num_examples
reg_loss = 0.5*reg*(np.sum(W1 * W1) + np.sum(W2 * W2))
loss = data_loss + reg_loss
#pass
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

grads = {}

# ================================================================ #
# YOUR CODE HERE:
```

```
        #   Implement the backward pass.  Compute the derivatives of the
        #   weights and the biases.  Store the results in the grads
        #   dictionary.  e.g., grads['W1'] should store the gradient for
        #   W1, and be of the same size as W1.
        # ================================================================ #
        dscores = probs
        dscores[range(N), y] -= 1
        dscores /= N
        grads['W2'] = np.dot(dscores.T, h1) + reg*W2
        grads['b2'] = np.sum(dscores, axis=0)
        dh1 = np.dot(dscores, W2)
        dh1[h1<=0]=0
        grads['W1'] = np.dot(dh1.T, X) + reg * W1
        grads['b1'] = np.sum(dh1, axis=0)

        #pass

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

        return loss, grads

    def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=1e-5, num_iters=100,
            batch_size=200, verbose=False):
        """
        Train this neural network using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) giving training data.
        - y: A numpy array f shape (N,) giving training labels; y[i] = c means␣
↪that
          X[i] has label c, where 0 <= c < C.
        - X_val: A numpy array of shape (N_val, D) giving validation data.
        - y_val: A numpy array of shape (N_val,) giving validation labels.
        - learning_rate: Scalar giving learning rate for optimization.
        - learning_rate_decay: Scalar giving factor used to decay the learning␣
↪rate
          after each epoch.
        - reg: Scalar giving regularization strength.
        - num_iters: Number of steps to take when optimizing.
        - batch_size: Number of training examples to use per step.
        - verbose: boolean; if true print progress during optimization.
        """
        num_train = X.shape[0]
```

```
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []

    for it in np.arange(num_iters):
        X_batch = None
        y_batch = None

        # ================================================================ #
        # YOUR CODE HERE:
        #   Create a minibatch by sampling batch_size samples randomly.
        # ================================================================ #
        #pass
        indexes = np.random.choice(num_train, batch_size)
        X_batch = X[indexes]
        y_batch = y[indexes]


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

        # Compute loss and gradients using the current minibatch
        loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
        loss_history.append(loss)

        # ================================================================ #
        # YOUR CODE HERE:
        #   Perform a gradient descent step using the minibatch to update
        #   all parameters (i.e., W1, W2, b1, and b2).
        # ================================================================ #

        self.params['W1'] -= learning_rate*grads['W1']
        self.params['W2'] -= learning_rate*grads['W2']
        self.params['b1'] -= learning_rate*grads['b1']
        self.params['b2'] -= learning_rate*grads['b2']
        #pass

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

        if verbose and it % 100 == 0:
            print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
```

```
            # Every epoch, check train and val accuracy and decay learning rate.
            if it % iterations_per_epoch == 0:
                # Check accuracy
                train_acc = (self.predict(X_batch) == y_batch).mean()
                val_acc = (self.predict(X_val) == y_val).mean()
                train_acc_history.append(train_acc)
                val_acc_history.append(val_acc)

                # Decay learning rate
                learning_rate *= learning_rate_decay

        return {
          'loss_history': loss_history,
          'train_acc_history': train_acc_history,
          'val_acc_history': val_acc_history,
        }

    def predict(self, X):
        """
        Use the trained weights of this two-layer network to predict labels for
        data points. For each data point we predict scores for each of the C
        classes, and assign each data point to the class with the highest score.

        Inputs:
        - X: A numpy array of shape (N, D) giving N D-dimensional data points to
          classify.

        Returns:
        - y_pred: A numpy array of shape (N,) giving predicted labels for each␣
↪of
          the elements of X. For all i, y_pred[i] = c means that X[i] is␣
↪predicted
          to have class c, where 0 <= c < C.
        """
        y_pred = None

        # ================================================================ #
        # YOUR CODE HERE:
        #   Predict the class given the input data.
        # ================================================================ #
        #pass
        predicted_output = np.dot(np.maximum(0, np.dot(X, self.params['W1'].T)+␣
↪self.params['b1']), self.params['W2'].T) + self.params['b2']
        y_pred = np.argmax(predicted_output, axis=1)
```

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

        return y_pred
```

2 Question 5 FC Nets

Complete the FC Net Jupyter notebook. Print out the entire notebook and relevant code and submit it as a pdf to gradescope

2.1 layers.py

[14]:
```
import numpy as np
import pdb

def affine_forward(x, w, b):
    """
    Computes the forward pass for an affine (fully-connected) layer.

    The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
    examples, where each example x[i] has shape (d_1, ..., d_k). We will
    reshape each input into a vector of dimension D = d_1 * ... * d_k, and
    then transform it to an output vector of dimension M.

    Inputs:
    - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
    - w: A numpy array of weights, of shape (D, M)
    - b: A numpy array of biases, of shape (M,)

    Returns a tuple of:
    - out: output, of shape (N, M)
    - cache: (x, w, b)
    """

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the output of the forward pass.  Notice the dimensions
    #   of w are D x M, which is the transpose of what we did in earlier
    #   assignments.
    # ================================================================ #
    X = x.reshape((x.shape[0], -1))
    out = np.dot(X,w) + b
    #pass

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

```python
    # ========================================================================= #

    cache = (x, w, b)
    return out, cache


def affine_backward(dout, cache):
    """
    Computes the backward pass for an affine layer.

    Inputs:
    - dout: Upstream derivative, of shape (N, M)
    - cache: Tuple of:
    - x: Input data, of shape (N, d_1, ... d_k)
    - w: Weights, of shape (D, M)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None

    # ========================================================================= #
    # YOUR CODE HERE:
    #   Calculate the gradients for the backward pass.
    # ========================================================================= #

    # dout is N x M
    # dx should be N x d1 x ... x dk; it relates to dout through multiplication␣
 ↪with w, which is D x M
    # dw should be D x M; it relates to dout through multiplication with x,␣
 ↪which is N x D after reshaping
    # db should be M; it is just the sum over dout examples
    X = x.reshape((x.shape[0], -1))
    db = np.sum(dout, axis = 0)
    dw = np.dot(X.T, dout)
    dx = np.dot(dout, w.T).reshape(x.shape)
    #pass

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

    return dx, dw, db
```

```python
def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the ReLU forward pass.
    # ================================================================ #
    out = np.maximum(0,x)
    #pass
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    cache = x
    return out, cache


def relu_backward(dout, cache):
    """
    Computes the backward pass for a layer of rectified linear units (ReLUs).

    Input:
    - dout: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - dx: Gradient with respect to x
    """
    x = cache

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the ReLU backward pass
    # ================================================================ #

    # ReLU directs linearly to those > 0
    #pass
    dx = dout*(x>0)
```

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

    return dx


def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth␣
 ↪class
    for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
    0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """

    probs = np.exp(x - np.max(x, axis=1, keepdims=True))
    probs /= np.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    loss = -np.sum(np.log(probs[np.arange(N), y])) / N
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    return loss, dx
```

2.2 fc_net.py

[ ]:
```
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'] = np.random.normal(0, weight_scale, (input_dim,␣
↪hidden_dims))
        self.params['W2'] = np.random.normal(0, weight_scale, (hidden_dims,␣
↪num_classes))
        self.params['b1'] = np.zeros(hidden_dims)
        self.params['b2'] = np.zeros(num_classes)
        #pass

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

```python
  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.
      # ================================================================ #
      hidden, cache_hidden = affine_relu_forward(X, self.params['W1'], self.
↪params['b1'])
      scores, cache_scores = affine_forward(hidden, self.params['W2'], self.
↪params['b2'])
      #pass
      # ================================================================ #
      # 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, dout = softmax_loss(scores,y)
        loss += 0.5 * self.reg * (np.sum(self.params['W1']**2) + np.sum(self.
 ↪params['W2']**2))

        dh, dw2, db2 = affine_backward(dout, cache_scores)
        dx, dw1, db1 = affine_relu_backward(dh, cache_hidden)

        grads['W1'] = dw1 + self.reg * self.params['W1']
        grads['b1'] = db1
        grads['W2'] = dw2 + self.reg * self.params['W2']
        grads['b2'] = db2

        #pass

        # ================================================================ #
        # 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 in np.arange(self.num_layers):
```

```
            if(i == 0):
                self.params['W' + str(i+1)] = np.random.normal(0, weight_scale,␣
↪(input_dim, hidden_dims[i]))
                self.params['b' + str(i+1)] = np.zeros(hidden_dims[i])
            elif(i == self.num_layers - 1):
                self.params['W' + str(i+1)] = np.random.normal(0, weight_scale,␣
↪(hidden_dims[i-1], num_classes))
                self.params['b' + str(i+1)] = np.zeros(num_classes)
            else:
                self.params['W' + str(i+1)] = np.random.normal(0, weight_scale,␣
↪(hidden_dims[i-1], hidden_dims[i]))
                self.params['b' + str(i+1)] = np.zeros(hidden_dims[i])
        #pass

        # ================================================================ #
        # 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".
        # ================================================================ #
        H = []
        H_cache = []
        for i in range(self.num_layers):
            H_app = None
            H_cache_app = None
            if(i==0):
                H_app, H_cache_app = affine_relu_forward(X, self.params['W' +␣
↪str(i+1)], self.params['b' + str(i+1)])
                H.append(H_app)
                H_cache.append(H_cache_app)
            elif(i == self.num_layers - 1):
                scores, H_cache_app = affine_forward(H[i-1], self.params['W' +␣
↪str(i+1)], self.params['b' + str(i+1)])
                H_cache.append(H_cache_app)
            else:
                H_app, H_cache_app = affine_relu_forward(H[i-1], self.
↪params['W' + str(i+1)], self.params['b' + str(i+1)])
                H.append(H_app)
                H_cache.append(H_cache_app)

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

        #pass
        loss, dhidden = softmax_loss(scores, y)
        for i in range(self.num_layers,0,-1):
          loss += 0.5*self.reg*np.sum(self.params['W{}'.format(i)]*self.
↪params['W{}'.format(i)])
          if i == self.num_layers:
            dH1, dW, db = affine_backward(dhidden,H_cache[i-1])
            grads['W{}'.format(i)] = dW + self.reg*self.params['W{}'.format(i)]
            grads['b{}'.format(i)] = db
          else:
            dH1, dW, db = affine_relu_backward(dH1,H_cache[i-1])
            grads['W{}'.format(i)] = dW + self.reg*self.params['W{}'.format(i)]
            grads['b{}'.format(i)] = db
        # loss, dhidden = softmax_loss(scores, y)
        # for i in range(self.num_layers,0,-1):
        #     loss += 0.5 * self.reg*np.sum(self.params['W{}'.format(i)]*self.
↪params['W{}'.format(i)])
        #     if i == self.num_layers:
        #         dFC1, dW, db = affine_backward(dhidden,FC_cache[i-1])
        #         grads['W{}'.format(i)] = dW + self.reg*self.params['W{}'.
↪format(i)]
        #         grads['b{}'.format(i)] = db
        #     else:
        #       dFC1, dW, db = affine_relu_backward(dFC1,FC_cache[i-1])
        #         grads['W{}'.format(i)] = dW + self.reg*self.params['W{}'.
↪format(i)]
        #         grads['b{}'.format(i)] = db
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
        return loss, grads
```