Helper Files and Related Code

February 22, 2025

1 All code modified for the problems

1.1 data_utils.py

[ ]:
```
from __future__ import print_function

from six.moves import cPickle as pickle
import numpy as np
import os
import imageio
from imageio 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)
    """ 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
    Xte, Yte = load_CIFAR_batch(os.path.join(ROOT, 'test_batch'))
    return Xtr, Ytr, Xte, Yte


def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,
                     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/
 ↪test_batch'
    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]
```

```python
    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 layers.py

[15]:
```
import numpy as np
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""
```

```python
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

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

    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:
```

```python
    - 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.
    # ================================================================ #
    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)


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

    return dx, dw, db

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)

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

    cache = x
    return out, cache
```

```python
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
    # ================================================================ #
    dx = dout*(x>0)

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

    return dx

def batchnorm_forward(x, gamma, beta, bn_param):
    """
    Forward pass for batch normalization.

    During training the sample mean and (uncorrected) sample variance are
    computed from minibatch statistics and used to normalize the incoming data.
    During training we also keep an exponentially decaying running mean of the␣
 ↪mean
    and variance of each feature, and these averages are used to normalize data
    at test-time.

    At each timestep we update the running averages for mean and variance using
    an exponential decay based on the momentum parameter:

    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1 - momentum) * sample_var

    Note that the batch normalization paper suggests a different test-time
    behavior: they compute sample mean and variance for each feature using a
    large number of training images rather than using a running average. For
    this implementation we have chosen to use running averages instead since
```

```
    they do not require an additional estimation step; the torch7 implementation
    of batch normalization also uses running averages.

    Input:
    - x: Data of shape (N, D)
    - gamma: Scale parameter of shape (D,)
    - beta: Shift paremeter of shape (D,)
    - bn_param: Dictionary with the following keys:
    - mode: 'train' or 'test'; required
    - eps: Constant for numeric stability
    - momentum: Constant for running mean / variance.
    - running_mean: Array of shape (D,) giving running mean of features
    - running_var Array of shape (D,) giving running variance of features

    Returns a tuple of:
    - out: of shape (N, D)
    - cache: A tuple of values needed in the backward pass
    """
    mode = bn_param['mode']
    eps = bn_param.get('eps', 1e-5)
    momentum = bn_param.get('momentum', 0.9)

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

    out, cache = None, None
    if mode == 'train':

    # ================================================================== #
    # YOUR CODE HERE:
    #   A few steps here:
    #     (1) Calculate the running mean and variance of the minibatch.
    #     (2) Normalize the activations with the sample mean and variance.
    #     (3) Scale and shift the normalized activations.  Store this
    #         as the variable 'out'
    #     (4) Store any variables you may need for the backward pass in
    #         the 'cache' variable.
    # ================================================================== #
        sample_mean = np.mean(x, axis=0)
        sample_var = np.var(x, axis=0)
        x_hat = (x - sample_mean) / np.sqrt(sample_var + eps)
        out = gamma * x_hat + beta
        cache = (x, x_hat, sample_mean, sample_var, gamma, beta, eps)
        running_mean = momentum * running_mean + (1 - momentum) * sample_mean
        running_var = momentum * running_var + (1 - momentum) * sample_var
```

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

    elif mode == 'test':

        # ================================================================ #
        # YOUR CODE HERE:
        #   Calculate the testing time normalized activation.  Normalize using
        #   the running mean and variance, and then scale and shift appropriately.
        #   Store the output as 'out'.
        # ================================================================ #
        x_hat = (x - running_mean)/np.sqrt(running_var + eps)
        out = gamma * x_hat + beta

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

    else:
        raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

    # Store the updated running means back into bn_param
    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var

    return out, cache

def batchnorm_backward(dout, cache):
    """
    Backward pass for batch normalization.

    For this implementation, you should write out a computation graph for
    batch normalization on paper and propagate gradients backward through
    intermediate nodes.

    Inputs:
    - dout: Upstream derivatives, of shape (N, D)
    - cache: Variable of intermediates from batchnorm_forward.

    Returns a tuple of:
    - dx: Gradient with respect to inputs x, of shape (N, D)
    - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
    """
    dx, dgamma, dbeta = None, None, None
```

```
    # ============================================================== #
    # YOUR CODE HERE:
    #   Implement the batchnorm backward pass, calculating dx, dgamma, and␣
↪dbeta.
    # ============================================================== #
    x, x_hat, sample_mean, sample_var, gamma, beta, eps = cache
    N, D = x.shape
    dbeta = np.sum(dout, axis=0)
    dgamma = np.sum(dout * x_hat, axis=0)
    dx_hat = dout * gamma
    dsample_var = np.sum(dx_hat * (x-sample_mean) * (-0.5) * (sample_var +␣
↪eps)**(-1.5), axis=0)
    dsample_mean = np.sum(dx_hat * (-1)/np.sqrt(sample_var + eps), axis=0) +␣
↪dsample_var * np.mean(-2 * (x - sample_mean), axis=0)
    dx = dx_hat / np.sqrt(sample_var + eps) + dsample_var * 2 * (x -␣
↪sample_mean) / N + dsample_mean / N

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

    return dx, dgamma, dbeta

def dropout_forward(x, dropout_param):
    """
    Performs the forward pass for (inverted) dropout.

    Inputs:
    - x: Input data, of any shape
    - dropout_param: A dictionary with the following keys:
    - p: Dropout parameter. We keep each neuron output with probability p.
    - mode: 'test' or 'train'. If the mode is train, then perform dropout;
      if the mode is test, then just return the input.
    - seed: Seed for the random number generator. Passing seed makes this
      function deterministic, which is needed for gradient checking but not in
      real networks.

    Outputs:
    - out: Array of the same shape as x.
    - cache: A tuple (dropout_param, mask). In training mode, mask is the␣
↪dropout
    mask that was used to multiply the input; in test mode, mask is None.
    """
    p, mode = dropout_param['p'], dropout_param['mode']
    if 'seed' in dropout_param:
        np.random.seed(dropout_param['seed'])
```

```python
    mask = None
    out = None

    if mode == 'train':
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the dropout forward pass during training time.
    #   Store the masked and scaled activations in out, and store the
    #   dropout mask as the variable mask.
    # ================================================================ #
        mask = (np.random.rand(*x.shape) < p)/p
        out = x*mask

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

    elif mode == 'test':

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the dropout forward pass during test time.
    # ================================================================ #
        out = x

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

    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)

    return out, cache

def dropout_backward(dout, cache):
    """
    Perform the backward pass for dropout.

    Inputs:
    - dout: Upstream derivatives, of any shape
    - cache: (dropout_param, mask) from dropout_forward.
    """
    dropout_param, mask = cache
    mode = dropout_param['mode']

    dx = None
```

```
    if mode == 'train':
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the dropout backward pass during training time.
    # ================================================================ #
        dx = dout*mask

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    elif mode == 'test':
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the dropout backward pass during test time.
    # ================================================================ #
        dx = dout
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return dx

def svm_loss(x, y):
    """
    Computes the loss and gradient using for multiclass SVM 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
    """
    N = x.shape[0]
    correct_class_scores = x[np.arange(N), y]
    margins = np.maximum(0, x - correct_class_scores[:, np.newaxis] + 1.0)
    margins[np.arange(N), y] = 0
    loss = np.sum(margins) / N
    num_pos = np.sum(margins > 0, axis=1)
    dx = np.zeros_like(x)
    dx[margins > 0] = 1
    dx[np.arange(N), y] -= num_pos
    dx /= N
    return loss, 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
```

1.3 conv_layers.py

[17]:
```
import numpy as np
from nndl.layers import *
import pdb

"""
This code was originally written for CS 231n at Stanford University
(cs231n.stanford.edu).  It has been modified in various areas for use in the
ECE 239AS class at UCLA.  This includes the descriptions of what code to
implement as well as some slight potential changes in variable names to be
consistent with class nomenclature.  We thank Justin Johnson & Serena Yeung for
permission to use this code.  To see the original version, please visit
cs231n.stanford.edu.
"""

def conv_forward_naive(x, w, b, conv_param):
    """
    A naive implementation of the forward pass for a convolutional layer.
```

```
    The input consists of N data points, each with C channels, height H and␣
↪width
    W. We convolve each input with F different filters, where each filter spans
    all C channels and has height HH and width HH.

    Input:
    - x: Input data of shape (N, C, H, W)
    - w: Filter weights of shape (F, C, HH, WW)
    - b: Biases, of shape (F,)
    - conv_param: A dictionary with the following keys:
    - 'stride': The number of pixels between adjacent receptive fields in the
      horizontal and vertical directions.
    - 'pad': The number of pixels that will be used to zero-pad the input.

    Returns a tuple of:
    - out: Output data, of shape (N, F, H', W') where H' and W' are given by
    H' = 1 + (H + 2 * pad - HH) / stride
    W' = 1 + (W + 2 * pad - WW) / stride
    - cache: (x, w, b, conv_param)
    """
    out = None
    pad = conv_param['pad']
    stride = conv_param['stride']

    # ================================================================== #
    # YOUR CODE HERE:
    #   Implement the forward pass of a convolutional neural network.
    #   Store the output as 'out'.
    #   Hint: to pad the array, you can use the function np.pad.
    # ================================================================== #
    x_pad = np.pad(x, ((0,), (0,), (pad,), (pad,)), mode='constant',␣
↪constant_values=0)
    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    # Calculate output dimensions
    H_out = 1 + (H + 2 * pad - HH) // stride
    W_out = 1 + (W + 2 * pad - WW) // stride
    out = np.zeros((N, F, H_out, W_out))

    for i in range(N):
        for f in range(F):
            for h_out in range(H_out):
                for w_out in range(W_out):
                    x_slice = x_pad[i, :, h_out * stride:h_out * stride + HH,␣
↪w_out * stride:w_out * stride + WW]
                    out[i, f, h_out, w_out] = np.sum(x_slice * w[f]) + b[f]
```

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


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

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


def conv_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a convolutional layer.

    Inputs:
    - dout: Upstream derivatives.
    - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

    Returns a tuple of:
    - dx: Gradient with respect to x
    - dw: Gradient with respect to w
    - db: Gradient with respect to b
    """
    dx, dw, db = None, None, None

    N, F, out_height, out_width = dout.shape
    x, w, b, conv_param = cache

    stride, pad = [conv_param['stride'], conv_param['pad']]
    xpad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
    num_filts, _, f_height, f_width = w.shape

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the backward pass of a convolutional neural network.
    #   Calculate the gradients: dx, dw, and db.
    # ================================================================ #
    dx = np.zeros_like(x)
    dw = np.zeros_like(w)
    db = np.zeros_like(b)
    xpad = np.pad(x, ((0,), (0,), (pad,), (pad,)), mode='constant',␣
 ↪constant_values=0)
    dxpad = np.zeros_like(xpad)
    for i in range(N):
        for f in range(F):
```

```
        for h_out in range(out_height):
          for w_out in range(out_width):
            x_slice = xpad[i, :, h_out*stride:h_out*stride+f_height,␣
↪w_out*stride:w_out*stride+f_width]
            dw[f] += x_slice * dout[i, f, h_out, w_out]
            dxpad[i, :, h_out*stride:h_out*stride+f_height, w_out*stride:
↪w_out*stride+f_width] += w[f] * dout[i, f, h_out, w_out]
            db[f] += dout[i, f, h_out, w_out]
            dx = dxpad[:, :, pad:pad+x.shape[2], pad:pad+x.shape[3]]

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

    return dx, dw, db


def max_pool_forward_naive(x, pool_param):
    """
    A naive implementation of the forward pass for a max pooling layer.

    Inputs:
    - x: Input data, of shape (N, C, H, W)
    - pool_param: dictionary with the following keys:
    - 'pool_height': The height of each pooling region
    - 'pool_width': The width of each pooling region
    - 'stride': The distance between adjacent pooling regions

    Returns a tuple of:
    - out: Output data
    - cache: (x, pool_param)
    """
    out = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the max pooling forward pass.
    # ================================================================ #
    N, C, H, W = x.shape
    xpad = np.pad(x, ((0,), (0,), (0,), (0,)), mode='constant',␣
↪constant_values=0)

    pool_height, pool_width, stride = pool_param['pool_height'],␣
↪pool_param['pool_width'], pool_param['stride']
    out_height = 1 + (H - pool_height) / stride
    out_width = 1 + (W - pool_width) / stride
```

```
    out = np.zeros((N, C, int(out_height), int(out_width)))
    for i in range(N):
        for j in range(C):
          for k in range(int(out_height)):
            for l in range(int(out_width)):
              window = xpad[i, j, k*stride:k*stride+pool_height, l*stride:
↪l*stride+pool_width]
              out[i, j, k, l] = np.max(window)

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    cache = (x, pool_param)
    return out, cache

def max_pool_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a max pooling layer.

    Inputs:
    - dout: Upstream derivatives
    - cache: A tuple of (x, pool_param) as in the forward pass.

    Returns:
    - dx: Gradient with respect to x
    """
    dx = None
    x, pool_param = cache
    pool_height, pool_width, stride = pool_param['pool_height'],␣
↪pool_param['pool_width'], pool_param['stride']

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the max pooling backward pass.
    # ================================================================ #
    N, C, H, W = x.shape
    dx = np.zeros_like(x)
    xpad = np.pad(x, ((0,), (0,), (0,), (0,)), mode='constant',␣
↪constant_values=0)
    for i in range(N):
        for j in range(C):
            for k in range(int(dout.shape[2])):
                for l in range(int(dout.shape[3])):
                  window = xpad[i, j, k*stride:k*stride+pool_height, l*stride:
↪l*stride+pool_width]
                  mask = (window == np.max(window))
```

```python
                    dx[i, j, k*stride:k*stride+pool_height, l*stride:
↪l*stride+pool_width] += mask * dout[i, j, k, l]

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

    return dx

def spatial_batchnorm_forward(x, gamma, beta, bn_param):
    """
    Computes the forward pass for spatial batch normalization.

    Inputs:
    - x: Input data of shape (N, C, H, W)
    - gamma: Scale parameter, of shape (C,)
    - beta: Shift parameter, of shape (C,)
    - bn_param: Dictionary with the following keys:
    - mode: 'train' or 'test'; required
    - eps: Constant for numeric stability
    - momentum: Constant for running mean / variance. momentum=0 means that
      old information is discarded completely at every time step, while
      momentum=1 means that new information is never incorporated. The
      default of momentum=0.9 should work well in most situations.
    - running_mean: Array of shape (D,) giving running mean of features
    - running_var Array of shape (D,) giving running variance of features

    Returns a tuple of:
    - out: Output data, of shape (N, C, H, W)
    - cache: Values needed for the backward pass
    """
    out, cache = None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the spatial batchnorm forward pass.
    #
    #   You may find it useful to use the batchnorm forward pass you
    #   implemented in HW #4.
    # ================================================================ #
    N, C, H, W = x.shape
    mode = bn_param['mode']
    eps = bn_param.get('eps', 1e-5)
    momentum = bn_param.get('momentum', 0.9)
    cache = {}
    running_mean = bn_param.get('running_mean', np.zeros(C, dtype=x.dtype))
    running_var = bn_param.get('running_var', np.zeros(C, dtype=x.dtype))
```

```
    if(mode =='train'):
        sample_mean = np.mean(x, axis=(0, 2, 3))
        sample_var = np.var(x, axis=(0, 2, 3))
        x_hat = (x - sample_mean.reshape(1, C, 1, 1)) / np.sqrt(sample_var.
↪reshape(1, C, 1, 1) + eps)
        out = gamma.reshape(1, C, 1, 1) * x_hat + beta.reshape(1, C, 1, 1)
        running_mean = momentum * running_mean + (1 - momentum) * sample_mean
        running_var = momentum * running_var + (1 - momentum) * sample_var
        cache = (x, x_hat, sample_mean, sample_var, gamma, beta, eps)
    elif(mode == 'test'):
        x_hat = (x - running_mean.reshape(1, C, 1, 1)) / np.sqrt(running_var.
↪reshape(1, C, 1, 1) + eps)
        out = gamma.reshape(1, C, 1, 1) * x_hat + beta.reshape(1, C, 1, 1)
    else:
        raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var


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

    return out, cache


def spatial_batchnorm_backward(dout, cache):
    """
    Computes the backward pass for spatial batch normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, C, H, W)
    - cache: Values from the forward pass

    Returns a tuple of:
    - dx: Gradient with respect to inputs, of shape (N, C, H, W)
    - dgamma: Gradient with respect to scale parameter, of shape (C,)
    - dbeta: Gradient with respect to shift parameter, of shape (C,)
    """
    dx, dgamma, dbeta = None, None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the spatial batchnorm backward pass.
    #
    #   You may find it useful to use the batchnorm forward pass you
```

```
    #   implemented in HW #4.
    # ================================================================== #
    N, C, H, W = dout.shape
    x, x_hat, sample_mean, sample_var, gamma, beta, eps = cache
    dbeta = np.sum(dout, axis=(0, 2, 3))
    dgamma = np.sum(dout * x_hat, axis=(0, 2, 3))
    dx_hat = dout * gamma.reshape(1, C, 1, 1)
    dsample_var = np.sum(dx_hat * (x - sample_mean.reshape(1, C, 1, 1)) *
                         (-0.5) * (sample_var.reshape(1, C, 1, 1) + eps)**(-1.
↪5),
                         axis=(0, 2, 3))
    dsample_mean = np.sum(dx_hat * (-1) / np.sqrt(sample_var.reshape(1, C, 1,␣
↪1) + eps),
                          axis=(0, 2, 3)) + dsample_var * np.mean(-2 * (x -␣
↪sample_mean.reshape(1, C, 1, 1)),
                                                                  axis=(0, 2,␣
↪3))
    dx = dx_hat / np.sqrt(sample_var.reshape(1, C, 1, 1) + eps) + dsample_var.
↪reshape(1, C, 1, 1) * 2 * (x - sample_mean.reshape(1, C, 1, 1)) / (N * H *␣
↪W) + dsample_mean.reshape(1, C, 1, 1) / (N * H * W)


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

    return dx, dgamma, dbeta
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[17], line 2
      1 import numpy as np
----> 2 from nndl.layers import *
      3 import pdb
      5 """
      6 This code was originally written for CS 231n at Stanford University
      7 (cs231n.stanford.edu).  It has been modified in various areas for use i␣
 ↪the
   (…)
     12 cs231n.stanford.edu.
     13 """

ModuleNotFoundError: No module named 'nndl'
```

```
[ ]:
```