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

  xpad = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)), 'constant', constant_values=0)
  N, _, H, W = xpad.shape
  F, _, HH, WW = w.shape
  H_new = int(1 + (H - HH) / stride)
  W_new = int(1 + (W - WW) / stride)
  out = np.zeros((N, F, H_new, W_new))
  for n in range(N):
    for f in range(F):
      for i in range(0, H_new):
```

```
        for j in range(0, W_new):
          i_start = i * stride
          j_start = j * stride
          x_patch = xpad[n, :, i_start:i_start+HH, j_start:j_start+WW]
          conv = np.sum(np.multiply(x_patch, w[f])) + b[f]
          out[n, f, i, j] = conv

  # ================================================================ #
  # 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(xpad)
  dw = np.zeros_like(w)
  db = np.zeros_like(b)
  for n in range(N):
    for f in range(F):
      db[f] += np.sum(dout[n, f])
      for i in range(out_height):
        for j in range(out_width):
          i_start = i * stride
          j_start = j * stride
          i_end = i_start + f_height
          j_end = j_start + f_width
          x_patch = xpad[n, :, i_start:i_end, j_start:j_end]

          dx[n, :, i_start:i_end, j_start:j_end] += w[f] * dout[n, f, i, j]
          dw[f] += dout[n, f, i, j] * x_patch

```

```python
  dx = dx[:, :, 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
  pool_h = pool_param['pool_height']
  pool_w = pool_param['pool_width']
  stride = pool_param['stride']
  H_new = int(1 + (H-pool_h) / stride)
  W_new = int(1 + (W-pool_w) / stride)
  out = np.zeros((N, C, H_new, W_new))

  for n in range(N):
    for c in range(C):
      for i in range(H_new):
        for j in range(W_new):
          i_start = i * stride
          j_start = j * stride
          i_end = i_start + pool_h
          j_end = j_start + pool_w
          out[n, c, i, j] = np.max(x[n, c, i_start:i_end, j_start:j_end])

  # ============================================================== #
  # 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 = dout.shape

  dx = np.zeros_like(x)

  for n in range(N):
    for c in range(C):
      for i in range(H):
        for j in range(W):
          i_start = i * stride
          j_start = j * stride
          i_end = i_start + pool_height
          j_end = j_start + pool_width
          x_patch = x[n, c, i_start:i_end, j_start:j_end]
          mask = x_patch == np.max(x_patch)
          dx[n, c, i_start:i_end, j_start:j_end] += mask * dout[n, c, i, j]

  # ================================================================ #
  # 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

```

```python
  # ============================================================== #
  # 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
  x = x.transpose(0, 2, 3, 1).reshape((-1, C))
  out, cache = batchnorm_forward(x, gamma, beta, bn_param)
  out = out.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

  # ============================================================== #
  # 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
  dout = dout.transpose(0, 2, 3, 1).reshape((-1, C))
  dx, dgamma, dbeta = batchnorm_backward(dout, cache)
  dx = dx.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

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

  return dx, dgamma, dbeta
```