```python
import numpy as np
from nndl.layers import *
import pdb


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), (0, 0), (pad, pad), (pad, pad)], mode='constant')

  N, C, H, W = x.shape
  F, C, HH, WW = w.shape

  H2 = int(1 + (H + 2 * pad - HH) / stride)
  W2 = int(1 + (W + 2 * pad - WW) / stride)

  out = np.zeros([N, F, H2, W2])

  for n in np.arange(N):
    for f in np.arange(F):
      for row in np.arange(H2):
        for col in np.arange(W2):
          out[n, f, row, col] = np.sum(w[f, :, :, :] * x_pad[n, :, row*stride : row*stride+HH, col*stride : col*stride+WW]) + b[f]

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

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

  H = x.shape[2]
  W = x.shape[3]

  dx_pad = np.zeros_like(xpad)
  dw = np.zeros_like(w)
  db = np.zeros_like(b)

  # dx
  for n in np.arange(N):
    for f in np.arange(F):
      for row in np.arange(out_height):
        for col in np.arange(out_width):
          dx_pad[n, :, row*stride : row*stride+f_height, col*stride : col*stride+f_width] += dout[n, f, row, col] * w[f, :, :, :]
  dx = dx_pad[:, :, pad : pad+H, pad : pad+W]

  # dw
  for n in np.arange(N):
    for f in np.arange(F):
      for row in np.arange(out_height):
        for col in np.arange(out_width):
          dw[f, :, :, :] += dout[n, f, row, col] * xpad[n, :, row*stride : row*stride+f_height, col*stride : col*stride+f_width]

  # db
```

```python
  for f in np.arange(F):
    db[f] += np.sum(dout[:, f, :, :])

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

  pool_height = pool_param['pool_height']
  pool_width = pool_param['pool_width']
  stride = pool_param['stride']

  N, C, H, W = x.shape

  H2 = int(1 + (H - pool_height) / stride)
  W2 = int(1 + (W - pool_width) / stride)

  out = np.zeros([N, C, H2, W2])

  for n in np.arange(N):
    for c in np.arange(C):
      for row in np.arange(H2):
        for col in np.arange(W2):
          out[n, c, row, col] = np.max(x[n, c, row*stride : row*stride+pool_height, col*stride : col*stride+pool_width])

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

```

```python
  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
  out_height = dout.shape[2]
  out_width = dout.shape[3]

  dx = np.zeros_like(x)

  for n in np.arange(N):
    for c in np.arange(C):
      for row in np.arange(out_height):
        for col in np.arange(out_width):
          max_idx = np.unravel_index(np.argmax(x[n, c, row*stride : row*stride+pool_height, col*stride : col*stride+pool_width]), [pool_height, pool_width])
          dx[n, c, row*stride+max_idx[0], col*stride+max_idx[1]] = dout[n, c, row, col]

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

```python
    """
    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

    x_reshape = np.reshape(np.transpose(x, (0, 2, 3, 1)), (N*H*W, C))
    out_2D, cache = batchnorm_forward(x_reshape, gamma, beta, bn_param)

    out = np.transpose(np.reshape(out_2D, (N, H, W, C)), (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_reshape = np.reshape(np.transpose(dout, (0, 2, 3, 1)), (N*H*W, C))
    dx_2D, dgamma, dbeta = batchnorm_backward(dout_reshape, cache)

    dx = np.transpose(np.reshape(dx_2D, (N, H, W, C)), (0, 3, 1, 2))

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

    return dx, dgamma, dbeta
```