```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.
  # ================================================================ #
  N,C,H,W = x.shape
  F,C,HH,WW = w.shape
  H_out_shape = 1 + (H + 2 * pad - HH) // stride
  W_out_shape = 1 + (W + 2 * pad - WW) // stride

  out = np.zeros((N,F,H_out_shape,W_out_shape))

  x = np.pad(x, pad_width = ((0,0),(0,0),(pad,pad),(pad,pad)), mode = 'constant')
  for i in range(N):
    for j in range(F):
        for k in range(H_out_shape):
            for l in range(W_out_shape):
                x_selected = x[i,:,k * stride:(k*stride + HH), l * stride : (l * stride + WW)]
                w_selected = w[j,:,:,:]
                out[i,j,k,l] = np.sum(x_selected * w_selected) + b[j]

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

```

```python
  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.
  # ================================================================ #
  N,F,H,W = x.shape

  H_out_shape = 1 + (H - f_height) // stride
  W_out_shape = 1 + (W - f_width) // stride

  dx = np.zeros(x.shape)
  dw = np.zeros(w.shape)
  db = np.zeros(b.shape)

  for i in range(N):
    for j in range(num_filts):
        if i == 0:
            db[j] += np.sum(dout[:,j,:,:])
        for k in range(H_out_shape):
            for l in range(W_out_shape):
                k_tmp = k * stride
                l_tmp = l * stride
                dout_tmp = dout[i,j,k,l]
                dx[i,:, k_tmp:(k_tmp + f_height), l_tmp:(l_tmp + f_width)] += w[j,:,:,:] * dout_tmp
                dw[j,:,:,:] += x[i,:, k_tmp:(k_tmp + f_height), l_tmp:(l_tmp + f_width)] * dout_tmp
  dx = dx[:,:, pad:-pad, pad:-pad]
  # ================================================================ #
  # 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_width, stride = pool_param['pool_height'], pool_param['pool_width'], pool_param['stride']
  N,C,H,W = x.shape

  H_out_shape = 1 + (H - pool_height) // stride
  W_out_shape = 1 + (W - pool_width) // stride
  out = np.zeros((N,C, H_out_shape, W_out_shape))
  for i in range(N):
    for j in range(C):
        for k in range(H_out_shape):
            for l in range(W_out_shape):
                k_tmp = k * stride
                l_tmp = l * stride
                x_tmp = x[i,j,k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)]
                out[i,j,k,l] = np.max(x_tmp)

  # =========================================================================== #
  # 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
  H_out_shape = 1 + (H - pool_height) // stride
  W_out_shape = 1 + (W - pool_width) // stride

  dx = np.zeros((N,C,H,W))
  for i in range(N):
    for j in range(C):
        for k in range(H_out_shape):
            for l in range(W_out_shape):
                k_tmp = k * stride
                l_tmp = l * stride
                x_tmp = x[i,j,k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)]
                dout_tmp = dout[i,j,k,l]
                din_mask = x_tmp == np.max(x_tmp)
                dx[i,j, k_tmp:(k_tmp + pool_height),l_tmp:(l_tmp + pool_width)] += din_mask * dout_tmp

```

```python
    # ========================================================================= #
    # 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
    x_flattened = (x.reshape((N,H,W,C))).reshape((N*W*H,C))
    out_bn, cache = batchnorm_forward(x_flattened, gamma,beta, bn_param = bn_param)
    out = (out_bn.reshape((N,W,H,C))).swapaxes(1,3)
    # ========================================================================= #
    # 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_bn = dout.swapaxes(1,3).reshape((N*W*H,C))
    dx_bn, dgamma_bn, dbeta_bn = batchnorm_backward(dout_bn,cache)
    dx = dx_bn.reshape((N,C,H,W))
    dgamma = dgamma_bn.reshape((C,))
    dbeta = dbeta_bn.reshape((C,))
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return dx, dgamma, dbeta
```