```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

  outputHeight = (H + 2 * pad - HH) // stride + 1
  outputWidth = (W + 2 * pad - WW) // stride + 1

  # output dimensions and initialization
  out = np.zeros((N, F, outputHeight, outputWidth))

  xPadded = np.pad(x, ((0,0), (0,0), (pad, pad), (pad, pad)), 'constant',constant_values=0)


  for i in np.arange(N): #For data points
      for j in np.arange(F): #For each filter
          for hIndex in np.arange(outputHeight):
              for wIndex in np.arange(outputWidth):
                  startH = hIndex*stride
                  startW = wIndex*stride
                  imagePart = xPadded[i,:,startH:(startH + HH),startW:(startW + WW)]
                  filterWeights = w[j,:,:,:]
                  filterBias = b[j]
                  out[i, j, hIndex, wIndex] = np.sum(imagePart*filterWeights) + filterBias



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

    for dataInd in range(N):
      for filterInd in range(F):
          db[filterInd] += np.sum(dout[dataInd, filterInd])
          for hInd in range(out_height):
              for wInd in range(out_width):
                  startH = hInd*stride
                  startW = wInd*stride
                  dw[filterInd] += xpad[dataInd, :, startH:startH + f_height,
                                        startW:startW + f_width] * dout[dataInd, filterInd, hInd, wInd]
                  dxpad[dataInd, :, startH:startH + f_height,
                        startW:startW + f_width] += w[filterInd] * dout[dataInd, filterInd, hInd, wInd]

    dx = dxpad[:, :, 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.
    # ================================================================ #
    poolH = pool_param['pool_height']
    poolW = pool_param['pool_width']
    stride = pool_param['stride']

    N, C, H, W = x.shape

    outH = (H  - poolH) // stride + 1
    outW = (W - poolW) // stride + 1

    outShape = (N, C, outH, outH)
    out = np.zeros(outShape)

    for dataInd in range(N):
      for channelInd in range(C):
          for hInd in range(outH):
              for wInd in range(outW):
                    startH = hInd*stride
                    startW = wInd*stride
                    out[dataInd, channelInd, hInd, wInd] = np.max(x[dataInd,channelInd,
                                                                   startH:(startH + poolH),
                                                                   startW:(startW + poolW)])

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

```python
    """
    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
    _, _, outH,outW = dout.shape

    dx = np.zeros(x.shape)

    for dataInd in range(N):
      for channelInd in range(C):
          for hInd in range(outH):
              for wInd in range(outW):
                  startH = hInd*stride
                  startW = wInd*stride
                  section = x[dataInd, channelInd, startH:startH + pool_height, startW:startW + pool_width]

                  mask = section == np.max(section)

                  dx[dataInd, channelInd, startH:startH + pool_height,
                     startW:startW + pool_width] += dout[dataInd, channelInd, hInd, wInd] * mask
    # ================================================================ #
    # 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 = x.reshape((N*H*W,C))

    out, cache = batchnorm_forward(x, gamma, beta, bn_param)
    out = out.T
    out = out.reshape(C,N,H,W)
    out = out.swapaxes(0,1)

    # ================================================================ #
    # 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,)
```

```python
"""
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.swapaxes(0,1)
dout = dout.reshape(C,N*H*W)
dout = dout.T

dx, dgamma, dbeta = batchnorm_backward(dout, cache)

dx = dx.reshape((N,C,H,W))
dgamma = dgamma.reshape((C,))
dbeta = dbeta.reshape((C,))

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

return dx, dgamma, dbeta
```