```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 = int(1 + (H + 2 * pad - HH) / stride)
  w_out = int(1 + (W + 2 * pad - WW) / stride)
  out = np.zeros([N, F, h_out, w_out])  # [N, 32, 32, 32]

  for n,x_n in enumerate(x):
    x_pad = (np.pad(x_n, ((0, 0), (pad, pad), (pad, pad)),'constant'))
    # for f,filter in enumerate(w):
    for i in range(h_out):
      for j in range(w_out):
        out[n,:,i,j] = np.sum(x_pad[:,i*stride:i*stride+HH,j*stride:j*stride+WW] * w,axis=tuple(range(1,w.ndim))) + b


  # ================================================================ #
  # 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.
  # ================================================================ #
  F,C,HH,WW=w.shape
  dx = np.zeros(x.shape)
  dx_pad = np.pad(dx, ((0,0), (0,0), (pad, pad), (pad, pad)), mode='constant')
  dw = np.zeros(w.shape)
  db = np.zeros(b.shape)

  db = np.sum(np.sum(np.sum(dout, axis=3), axis=2), axis=0)

  for n, x_pad_n in enumerate(xpad):
    for f,filter in enumerate(w):
      for i in range(out_height):
        for j in range(out_width):
          dw[f] += dout[n, f, i, j] * x_pad_n[:, i * stride:i * stride + HH, j * stride: j * stride + WW]
          dx_pad[n, :, i * stride:i * stride + HH, j * stride: j * stride + WW] += dout[n, f, i, j] * filter

  dx = dx_pad[:,:,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.
  # ================================================================ #

  N,C,H,W=x.shape

  h_out = int((H - pool_param['pool_height']) / pool_param['stride']) + 1
  w_out = int((W - pool_param['pool_width']) / pool_param['stride']) + 1
  out = np.zeros((N, C, h_out, w_out))
  for n, x_n in enumerate(x):
    for c, channel in enumerate(x_n):
      for i in range(h_out):
        x_i = i * pool_param['stride']
        for j in range(w_out):
            y_j = j * pool_param['stride']
            out[n, c, i, j] = np.amax(channel[x_i:x_i + pool_param['pool_height'], y_j:y_j + pool_param['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.

  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.
  # ================================================================ #
  dx = np.zeros(x.shape)

  F,C,h_out,w_out=dout.shape

  for n, x_n in enumerate(x):
    for c, channel in enumerate(x_n):
      for i in range(h_out):
        x_i = i * pool_param['stride']
        for j in range(w_out):
          y_j = j * pool_param['stride']
          target_area = channel[x_i:x_i + pool_param['pool_height'], y_j:y_j + pool_param['pool_width']]
          max_x, max_y = np.unravel_index(np.argmax(target_area, axis=None), target_area.shape)
          dx[n, c, max_x + x_i, max_y + y_j] = 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

  # ================================================================ #
  # 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_2d = np.reshape(x.transpose(0, 2, 3, 1), (N*H*W, C)) # 2-d reshape of permuted matrix
  out_2d, cache = batchnorm_forward(x_2d, gamma, beta, bn_param)
  out = out_2d.reshape((N, H, W, C)).transpose(0, 3, 1, 2) # reshape and permute back
  # ================================================================ #
  # 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

  dx = np.zeros(dout.shape)
  dout_2d = np.reshape(dout.transpose((0, 2, 3, 1)), (N*H*W, C))
  dx_2d, dgamma, dbeta = batchnorm_backward(dout_2d, cache)
  dx = dx_2d.reshape((N, H, W, C)).transpose(0, 3, 1, 2)

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

  return dx, dgamma, dbeta
```