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

  n, c, xh, xw = x.shape
  wf, wc, wh, ww = w.shape
  hout, wout = 1 + (xh + 2*pad - wh)//stride, 1 + (xw + 2*pad - ww)//
stride

  out = np.zeros((len(x), wf, hout, wout))

  # pad the input
  x_padded = np.pad(x, ((0, 0), (0, 0), (pad, pad), (pad, pad)),
'constant', constant_values=0)

  # loop over all inputs
  for i in range(n):
      # loop over filters
      for j in range(wf):
          # loop over each cell in output
          for hi in range(hout):
              for wi in range(wout):
                  out[i, j, hi, wi] = np.sum(x_padded[i, :, hi*stride:
hi*stride + wh, wi*stride: wi*stride + ww] * w[j]) + 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.

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

  xn, xc, xh, xw = xpad.shape
  d_n, d_f, d_h, d_w = dout.shape
  wf, wc, wh, ww = w.shape

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

  for i in range(d_n):
      for j in range(d_f):
          for k in range(d_h):
              for l in range(d_w):
                  dw[j] += xpad[i, :, k*stride: k*stride + wh,
l*stride: l*stride + ww] * dout[i, j, k, l]
                  db[j] += dout[i, j, k, l]
                  dx[i, :, k*stride: k*stride + wh, l*stride: l*stride
+ ww] += dout[i, j, k, l] * w[j]

  # the pad is used to help calculate the gradient, so we now remove
the pad from the gradient
  xpad_h, xpad_w = xpad.shape[2], xpad.shape[3]
  dx = dx[:, :, pad:xpad_h - pad, pad:xpad_w - 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.
    # =================================================================== #

    p_h, p_w, stride = pool_param['pool_height'],
  pool_param['pool_width'], pool_param['stride']
    n, c, h, w = x.shape
    h_new, w_new = (h - p_h)//stride + 1, (w - p_w)//stride + 1

    out = np.zeros((n, c, h_new, w_new))

    for i in range(n):
        for j in range(c):
            for k in range(h_new):
                for l in range(w_new):
                    out[i, j, k, l] = np.max(x[i, j, k*stride: k*stride
  + p_h, l*stride: l*stride + p_w])

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

  dout_n, dout_c, dout_w, dout_h = dout.shape

  dx = np.zeros(x.shape)

  for i in range(dout_n):
      for j in range(dout_c):
          for k in range(dout_w):
              for l in range(dout_h):
                  slice = x[i, j, k*stride: k*stride + pool_height,
l*stride: l*stride + pool_width]
                  mask = slice == np.max(slice)
                  dx[i, j, k*stride: k*stride + pool_height, l*stride:
l*stride + pool_width] += mask * dout[i, j, k, l]

  # ================================================================ #
  # 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.transpose(0, 2, 3, 1).reshape((n*h*w, 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 backward pass you
  #   implemented in HW #4.
```

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

n, c, h, w = dout.shape
dout = dout.transpose(0, 2, 3, 1).reshape((n*h*w, 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
```