```
# conv layers updated

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

  # check if valid conv
```

```
  assert (H + 2 * pad - HH) % stride == 0
  assert (W + 2 * pad - WW) % stride == 0

  # only pad the third and fourth axes
  x_pad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad,pad)),
mode='constant')

  # H_prime, W_prime are the output height and width
  H_prime = int(1 + (H + 2 * pad - HH) / stride)
  W_prime = int(1 + (W + 2 * pad - WW) / stride)

  out = np.zeros((N, F, H_prime, W_prime))

  for n in range(N): # number of batches
    for i in range(H_prime): # output height
      for j in range(W_prime): # output width
        seg = x_pad[n,:,i*stride:i*stride + HH, j*stride:j*stride +
WW]
        out[n,:,i,j] = np.sum(seg * w,axis=(1,2,3)) + 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.
  # ================================================================ #
  # inits
  dx = np.zeros_like(x)
  dw = np.zeros_like(w)
  db = np.zeros_like(b)


  # dims
  N, C, H, W = x.shape
  F, _, HH, WW = w.shape
  _, _, H_out, W_out = dout.shape

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

  # dw
  for f in range(F): # looping through filters
      for c in range(C): # looping through channels
          for i in range(HH): # looping through weight height
              for j in range(WW): # looping through weight width
                  dw[f, c, i ,j] = np.sum(xpad[:,  c, i: i + H_out *
stride : stride, j : j + W_out* stride : stride] * dout[:, f, :, :])
                  # dw at determined segment
                  # np sum of (xpad matrix * dout matrix)

  # dx
  dx = np.zeros(x.shape)
  # loop through the number of examples
  for n in range(N): # hi, wi: loop through x
      for hx in range(H):
          for wx in range(W):
              y_indexes = [] # will contain valid indices of y
              w_indexes = [] # will contain valid indices of weight
(w)
              for i in range(H_out): # H_ is from dout
                  for j in range(W_out):  # W_ is from dout
                      # i, j: loop through output
                      # verify: is the range within weights limits?
                      h_range = (hx + pad - i * stride) # height range
                      w_range = (wx + pad - j * stride) # weight range
                      if (h_range >= 0) and (h_range < HH) and
(w_range >= 0) and (w_range < WW):
```

```
                          w_indexes.append((h_range, w_range))
                          y_indexes.append((i, j))

            for f in range(F): # filters loop
                # windex_f and yindex_f from python zip of
w_indexes, y_indexes (as determined above)
                # increment by np.sum ( w_matrix * dout_matrix) for
valid indices of y and weights
                dx[n, : , hx, wx] += np.sum([w[f, :, windex_f[0],
windex_f[1]] * dout[n, f, yindex_f[0], yindex_f[1]] for windex_f,
yindex_f in zip(w_indexes, y_indexes)], 0)



  # ================================================================ #
  # 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
  HH = pool_param['pool_height']
  WW = pool_param['pool_width']
  stride = pool_param['stride']

  # H_prime, W_prime are the output height and width
  H_prime = int(1 + (H - HH) / stride)
```

```
  W_prime = int(1 + (W - WW) / stride)

  out = np.zeros((N, C, H_prime, W_prime))

  for n in range(N): # number of batches
    for i in range(H_prime): # output height
      for j in range(W_prime): # output width
        seg = x[n,:,i*stride:i*stride + HH, j*stride:j*stride + WW]
        out[n,:,i,j] = np.amax(seg,axis=(1,2))

  # ================================================================ #
  # 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
  N, C, dout_height, dout_width = dout.shape

  dx = np.zeros_like(x)

  for n in range(N):                              # loop over the 
number of training samples
    for c in range(C):                          # loop over the number 
of channels
        for i in range(dout_height):            # loop over vertical 
axis of the dout
            for j in range(dout_width):         # loop over horizontal 
axis of the dout
```

```
                i_, j_ = np.where(np.max(x[n, c, i * stride : i *
stride + pool_height, j * stride : j * stride + pool_width]) == x[n,
c, i * stride : i * stride + pool_height, j * stride : j * stride +
pool_width])
                i_, j_ = i_[0], j_[0]
                dx[n, c, i * stride : i * stride + pool_height, j *
stride : j * stride + pool_width][i_, 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.
  # ================================================================ #
```

```python
    N, C, W, H = x.shape
    xr = x.reshape(N*H*W, C)
    out, cache = batchnorm_forward(xr, gamma, beta, bn_param)
    out = out.reshape(N, C, W, H)

    # ================================================================ #
    # 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, W, H = dout.shape

    doutr = dout.reshape(N*H*W, C)

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

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

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

    return dx, dgamma, dbeta
```