In [ ]:
```
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 t
o be
consistent with class nomenclature.  We thank Justin Johnson & Serena
Yeung for
permission to use this code.  To see the original version, please visi
t
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 filte
r 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 in
put.

  Returns a tuple of:
  - out: Output data, of shape (N, F, H', W') where H' and W' are give
n 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, C2, HH, WW = w.shape

  H_prime = 1 + (H + 2*pad - HH)/stride
  W_prime = 1 + (W+ 2*pad - WW)/stride

  #pad the input
  x_pad = np.pad(x, ((0,0), (0,0), (pad,pad), (pad, pad)), mode='const
ant', constant_values=0)
  _, _, H_pad, W_pad = x_pad.shape

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

  #iterate through all data points
  for datapoint in range(N):
    x_pad_cur = x_pad[datapoint]

    h_loc, w_loc = -1, -1
    #go by height
    for hi in range(0, H_pad - HH + 1, stride):
      h_loc += 1
      #go by width
      for wi in range(0, W_pad - WW + 1, stride):
        w_loc += 1
        #first dim = : to get all channels
        x_all_channels = x_pad_cur[:, hi:hi+HH, wi:wi+WW]

        for filt in range(F):
          out[datapoint, filt, h_loc, w_loc] = np.sum(x_all_channels *
w[filt]) + b[filt]

      #reset height counter
      w_loc = -1


  # ================================================================ #
  # 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 laye
r.

  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='constan
t')
  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.
  # ================================================================ #

  #b is biases of shape (F,)
  #dout is N, F, out_height, out_width
  db = np.zeros((b.shape))
  for i in range(F):
    db[i] = np.sum(dout[:, i, :, :])


  #w: Filter weights of shape (F, C, HH, WW)
  F, C, HH, WW = w.shape
  dw = np.zeros((w.shape))
  for i in range(F):
    for j in range(C):
      for k in range(HH):
        for l in range(WW):
          #Input data of shape (N, C, H, W)
          derivative = dout[:, i, :, :] * xpad[:, j, k:k + out_height
* stride:stride, l:l + out_width * stride:stride]
```

```
          dw[i,j,k,l] = np.sum(derivative)


  #x: (N, C, H, W)
  _, _, H, W = x.shape
  #create dummy gradient -> will have same dimensions as x
  dx = np.zeros(x.shape)
  #pad the gradient dx
  dxpad = np.pad(dx, ((0,0), (0,0), (pad,pad), (pad,pad)), mode='const
ant')
  H_prime = 1 + (H + 2*pad - HH)/stride
  W_prime = 1 + (W + 2*pad - WW)/stride

  for i in range(N): #for each data point
    for j in range(F):
      for k in range(0, int(H_prime)):#, stride):
        k_prime = k*stride
        for l in range(0, int(W_prime)):#, stride):
          l_prime = l*stride
          #multiply the weights of this filter by derivative dout
          derivative = w[j] * dout[i,j,k,l]
 #         print(dxpad.shape)
#          print(derivative.shape)
          dxpad[i, :, k_prime:k_prime + HH, l_prime:l_prime+WW] += der
ivative

  #extract derivative
  #dimensions need to be pulled out from H, W (, , H, W)
  dx = dxpad[:, :, pad:pad+H, pad:pad+W]
  # ================================================================ #
  # 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_param['pool_height']
  pool_width = pool_param['pool_width']
  stride = pool_param['stride']

  N, C, H, W = x.shape
  W_out = (W - pool_width)/stride + 1
  H_out = (H - pool_height)/stride + 1

  out = np.zeros((N, C, int(H_out), int(W_out)))

  for datapoint in range(N):
    #reduce by one dimension (datapoint #)
    x_cur = x[datapoint]

    h_loc, w_loc = -1, -1
    for hi in range(0, H-pool_height + 1, stride):
      h_loc += 1
      for wi in range(0, W-pool_width + 1, stride):
        w_loc += 1

        #this is the receptive field
        x_receptive_field = x_cur[:, hi:hi+pool_height, wi:wi+pool_wid
th]

        #iterate through all channels
        for c in range(C):
          out[datapoint, c, h_loc, w_loc] = np.max(x_receptive_field[c
])

      w_loc = -1





  # ================================================================ #
  # 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_pa
ram['pool_width'], pool_param['stride']

  # ================================================================== #
  # YOUR CODE HERE:
  #   Implement the max pooling backward pass.
  # ================================================================== #

  dx = np.zeros(x.shape)
  N, C, H, W = x.shape
  H_prime = 1 + (H - pool_height)/stride
  W_prime = 1 + (W - pool_width)/stride
  for i in range(N):
    for j in range(C):
      for k in range(int(H_prime)):
        for l in range(int(W_prime)):
          k_prime = k * stride
          l_prime = l * stride

          #we want to only reward for the one we picked
          cur_window = x[i, j, k_prime:k_prime + pool_height, l_prime:
l_prime + pool_width]
          max_cur_window = np.max(cur_window)
          masked_window = (cur_window == max_cur_window)
          derivative = dout[i,j,k,l] * masked_window
          dx[i, j, k_prime:k_prime + pool_height, l_prime:l_prime + po
ol_width ] += derivative
  # ================================================================== #
  # 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, whil
e
      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 feature
s
    - running_var Array of shape (D,) giving running variance of featu
res

  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.
  # ================================================================ #
  #mode = bn_param['mode']
 # eps = bn_param.get['eps']
#  momentum = bn_param.get['momentum']
  N, C, H, W = x.shape

  #reshape the (N, C, H, W) array as an (N*H*W, C) array and perform b
atch normalization on this array.
  transpose = np.transpose(x, axes=(0,2,3,1))
  reshaped = transpose.reshape(N*H*W, C)
  bn_out, cache = batchnorm_forward(reshaped, gamma, beta, bn_param)

  #reshape again and swap
  out = bn_out.reshape(N, H, W, C).transpose(0, 3, 1, 2)



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

  return out, cache
```

```python
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
  transpose = np.transpose(dout, axes=(0,2,3,1))
  reshaped = transpose.reshape(N*H*W, C)
  dx_bn, dgamma_bn, dbeta_bn = batchnorm_backward(reshaped, cache)

  dx = dx_bn.reshape(N, H, W, C).transpose(0,3,1,2)
  dgamma = dgamma_bn
  dbeta = dbeta_bn

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

  return dx, dgamma, dbeta
```