conv_layers.py

This file contains all the code in conv_layers.py

Code

```python
from multiprocessing import pool
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.
  # ================================================================ #
```

```python
  npad = ((0,0), (0,0),(pad,pad),(pad,pad))
  xpad = np.pad(x,npad,mode='constant',constant_values=0)
  N, C, H, W = xpad.shape
  F, C, HH, WW = w.shape
  H_out = int(1 + (H - HH) / stride)
  W_out = int(1 + (W - WW) / stride)
  out = np.zeros((N,F,H_out,W_out))
  for n in range(N):
    for i in range(F):
      for j in range(H_out):
        for k in range(W_out):
          out[n,i,j,k] = np.sum(w[i,:]*xpad[n, :, stride*j:stride*j+HH, stride*k:stride*k+WW

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

```python
    # ========================================================================= #
    '''- w: Filter weights of shape (F, C, HH, WW)'''
    N, C, H, W = x.shape

    db = np.sum(dout, axis=(0,2,3))
    dw = np.zeros(w.shape)
    dx = np.zeros(x.shape)

    mask = np.pad(np.ones(x.shape),((0,0), (0,0), (pad,pad), (pad,pad)), mode='constant')
    for i in range(num_filts):
      for j in range(out_height):
          for k in range(out_width):
              dw[i,:,:,:] += np.sum((dout[:,i,j,k]*xpad[:,:,stride*j:stride*j+f_height,stride

    #dx
    wrot = np.rot90(w,2,axes=(2,3))
    dout_dial = np.zeros((N, F, out_height+(stride-1)*(out_height-1), out_width+(stride-1)*(ou
    dout_dial[:,:,0::stride,0::stride] = dout
    _,_,ddp_height,ddp_width = dout_dial.shape
    print(dout_dial.shape)
    dout_dial_pad = np.pad(dout_dial, ((0,0), (0,0), (H-ddp_height+1,H-ddp_height+1), (W-ddp_w

    for n in range(N):
      for i in range(num_filts):
        for j in range(H):
          for k in range(W):
            dx[n,:,j,k] += np.sum(wrot[i,:,:,:]*dout_dial_pad[n,i,j:j+f_height,k:k+f_width], a

    # ========================================================================= #
    # 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.
    # ================================================================ #
    ph = pool_param['pool_height']
    pw = pool_param['pool_width']
    stride = pool_param['stride']
    N, C, H, W = x.shape
    Hout = int(1 + (H - ph) / stride)
    Wout = int(1 + (W - pw) / stride)
    out = np.zeros((N,C,Hout,Wout))
    for n in range(N):
      for i in range(Hout):
        for j in range(Wout):
          out[n,:,i,j] = np.max(x[n,:,stride*i:stride*i+ph,stride*j:stride*j+pw], 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'], poo

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

```python
    # ================================================================ #
    N, C, H, W = x.shape
    dx= np.zeros_like(x)
    _,_,Hout,Wout = dout.shape
    out = np.zeros((N,C,Hout,Wout))
    for n in range(N):
      for i in range(Hout):
        for j in range(Wout):
          for k in range(C):
            out[n,k,i,j] = np.max(x[n,k,stride*i:stride*i+pool_height,stride*j:stride*j+pool_w
            mask = (out[n,k,i,j]==x[n,k,stride*i:stride*i+pool_height,stride*j:stride*j+pool_w
            dx[n,k,stride*i:stride*i+pool_height,stride*j:stride*j+pool_width] = mask*dout[n,k
    # ================================================================ #
    # 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_reshape = np.zeros((N*H*W,C))
  for i in range(C):
    x_reshape[:,i] = x[:,i,:,:].reshape(N*H*W)
  out_rs,cache =batchnorm_forward(x_reshape, gamma, beta, bn_param)
  out = np.zeros_like(x)
  for i in range(C):
    out[:,i,:,:] = out_rs[:,i].reshape(N,H,W)
  # ================================================================ #
  # 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
  dout_reshape = np.zeros((N*H*W,C))
  for i in range(C):
    dout_reshape[:,i] = dout[:,i,:,:].reshape(N*H*W)
  dx_rs,dgamma,dbeta = batchnorm_backward(dout_reshape,cache)
  dx = np.zeros_like(dout)
```

```python
for i in range(C):
  dx[:,i,:,:] = dx_rs[:,i].reshape(N,H,W)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return dx, dgamma, dbeta
```