layer.py

卷积前向传播

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 WW.

    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.

    During padding, 'pad' zeros should be placed symmetrically (i.e equally on both sides)

    along the height and width axes of the input. Be careful not to modfiy the original

    input x directly.

    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)

    """

    N,C,H,W=x.shape # N：样本数 C:通道数 H:高度 W:宽度

    F,\_,HH,WW=w.shape # F:滤波器数 C:通道数 HH:滤波器高度 WW:滤波器宽度

    stride=conv\_param['stride']  # 提取步长

    pad=conv\_param['pad']     # 提取填充数

    new\_H = int( 1 + (H + 2\*pad - HH) / stride )

    new\_W = int( 1 + (W + 2\*pad - WW) / stride )

    out = np.zeros((N,F,new\_W,new\_H))

    #边缘补零

    x\_pad = np.pad(x, ( (0,0),(0,0),(pad,pad),(pad,pad) ) , mode = 'constant' , constant\_values=0 )

    # 做三层循环

    for i in range(new\_H):

      for j in range(new\_W):

        x\_mask = x\_pad[: , : , i\*stride:i\*stride+HH , j\*stride:j\*stride+WW ]

        for k in range(F):

          out[:,k,i,j] = np.sum( x\_mask \* w[k,:,:,:] , axis=(1,2,3)) + b[k]

卷积反向传播

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

    """

    x,w,b,conv\_param = cache

    N,C,H,W=x.shape # N：样本数 C:通道数 H:高度 W:宽度

    F,\_,HH,WW=w.shape # F:滤波器数 C:通道数 HH:滤波器高度 WW:滤波器宽度

    stride=conv\_param['stride']  # 提取步长

    pad=conv\_param['pad']     # 提取填充数

    new\_H = int( 1 + (H + 2\*pad - HH) / stride )

    new\_W = int( 1 + (W + 2\*pad - WW) / stride )

    #边缘补零

    x\_pad = np.pad(x, ( (0,0),(0,0),(pad,pad),(pad,pad) ) , mode = 'constant' , constant\_values=0 )

    dx=np.zeros\_like(x)

    dw=np.zeros\_like(w)

    # 计算b的梯度，因为b为向量，而dout为高维张量，需要将dout各个通道的值相加，保留为一个(F,)的向量

    db=np.sum(dout,axis=(0,2,3))

    dx\_pad=np.zeros\_like(x\_pad)

    for i in range(new\_H):

      for j in range(new\_W):

        x\_mask = x\_pad[: , : , i\*stride:i\*stride+HH , j\*stride:j\*stride+WW ]

        for k in range(F):

          dw[k,:,:,:] += np.sum(x\_mask \* (dout[:,k,i,j]).reshape(-1,1,1,1) , axis=0  )

        for n in range(N):

          dx\_pad[n, : , i\*stride:i\*stride+HH, j\*stride:j\*stride+WW] += np.sum( w[:,:,:,:] \* (dout[n,:,i,j]).reshape(-1,1,1,1) ,axis=0 )

    #去掉填充部分

    dx = dx\_pad[:,:,pad:pad+H,pad:pad+W]

maxpool前向传播

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

    No padding is necessary here. Output size is given by

    Returns a tuple of:

    - out: Output data, of shape (N, C, H', W') where H' and W' are given by

      H' = 1 + (H - pool\_height) / stride

      W' = 1 + (W - pool\_width) / stride

    - cache: (x, pool\_param)

    """

    N,C,H,W = x.shape

    pool\_height = pool\_param['pool\_height']

    pool\_width = pool\_param['pool\_width']

    stride = pool\_param['stride']

    new\_H = int( 1 + (H - pool\_height) / stride ) #池化层核的高度和宽度

    new\_W = int(1 + (W - pool\_width) /stride )

    out = np.zeros((N,C,new\_H,new\_W))

    for i in range(new\_H):

      for j in range(new\_W):

        x\_mask = x[:, :, i\*stride:i\*stride+pool\_height, j\*stride:j\*stride+pool\_width]

        out[:,:,i,j] = np.max(x\_mask,axis=(2,3))

maxpool反向传播

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

    """

    x,pool\_param=cache

    N,C,H,W = x.shape

    pool\_height = pool\_param['pool\_height']

    pool\_width = pool\_param['pool\_width']

    stride = pool\_param['stride']

    new\_H = int( 1 + (H - pool\_height) / stride ) #池化层核的高度和宽度

    new\_W = int(1 + (W - pool\_width) /stride )

    out = np.zeros((N,C,new\_H,new\_W))

    dx = np.zeros((N, C, H, W))

    for n in range(N):

      for c in range(C):

        for i in range(new\_H):

          for j in range(new\_W):

            window = x[n, c, i\*stride:i\*stride+pool\_height, j\*stride:j\*stride+pool\_width]

            window\_max = np.max(window)

            dx[n,c,i\*stride:i\*stride+pool\_height, j\*stride:j\*stride+pool\_width] = (window == window\_max) \* dout[n,c,i,j]

cnn.py

class ThreeLayerConvNet(object):

初始化网络参数 def init

        C,H,W=input\_dim #输入数据的参数 C-channels，H-height，W-width

        # 高斯分布(N,C,L,L) N个filter,每个filter都是L\*L,通道数为C

        W1 = np.random.randn(num\_filters, C, filter\_size, filter\_size) \* weight\_scale

        b1 = np.zeros(num\_filters)  #(32,1)

        # 高斯分布  这里是全连接层，输入的维数是N\*(H/2)\*(W/2),H和W除以2是因为通过了maxpool层，且通过padding保留了大小，输出是hidden\_dim的维数

        W2 = np.random.randn(num\_filters\*int((H/2)\*(W/2)),hidden\_dim) \* weight\_scale

        b2 = np.zeros(hidden\_dim)   #(100,1)

        # 高斯分布  这里是全连接层，输入的维数是hidden\_dim，输出的维数是num\_classes

        W3 = np.random.randn(hidden\_dim,num\_classes) \* weight\_scale

        b3 = np.zeros(num\_classes)

        self.params['W1'] = W1

        self.params['W2'] = W2

        self.params['W3'] = W3

        self.params['b1'] = b1

        self.params['b2'] = b2

        self.params['b3'] = b3

前向传播

        out1,cache1 = conv\_relu\_pool\_forward(X, W1, b1, conv\_param, pool\_param)

        out2,cache2 = affine\_relu\_forward(out1,W2,b2)

        out3,cache3 = affine\_forward(out2,W3,b3)

        scores = out3

反向传播算loss和grads

        loss,dscores = softmax\_loss(scores,y)

        loss += 0.5\*self.reg\*(np.sum(W1\*\*2)+np.sum(W2\*\*2))

        dx3,grads['W3'],grads['b3'] = affine\_backward(dscores, cache3)

        grads['W3'] += self.reg\*self.params['W3']

        dx2,grads['W2'],grads['b2'] = affine\_relu\_backward(dx3, cache2)

        grads['W2'] += self.reg\*self.params['W2']

        dx1,grads['W1'],grads['b1'] = conv\_relu\_pool\_backward(dx2, cache1)

        grads['W1'] += self.reg\*self.params['W1']

        dx=dx1