Layers.py

计算batchnormalization的前向传播

def batchnorm\_forward(x, gamma, beta, bn\_param):

    """

    Forward pass for batch normalization.

    During training the sample mean and (uncorrected) sample variance are

    computed from minibatch statistics and used to normalize the incoming data.

    During training we also keep an exponentially decaying running mean of the

    mean and variance of each feature, and these averages are used to normalize

    data at test-time.

    At each timestep we update the running averages for mean and variance using

    an exponential decay based on the momentum parameter:

    running\_mean = momentum \* running\_mean + (1 - momentum) \* sample\_mean

    running\_var = momentum \* running\_var + (1 - momentum) \* sample\_var

    Note that the batch normalization paper suggests a different test-time

    behavior: they compute sample mean and variance for each feature using a

    large number of training images rather than using a running average. For

    this implementation we have chosen to use running averages instead since

    they do not require an additional estimation step; the torch7

    implementation of batch normalization also uses running averages.

    Input:

    - x: Data of shape (N, D)

    - gamma: Scale parameter of shape (D,)

    - beta: Shift paremeter of shape (D,)

    - bn\_param: Dictionary with the following keys:

      - mode: 'train' or 'test'; required

      - eps: Constant for numeric stability

      - momentum: Constant for running mean / variance.

      - running\_mean: Array of shape (D,) giving running mean of features

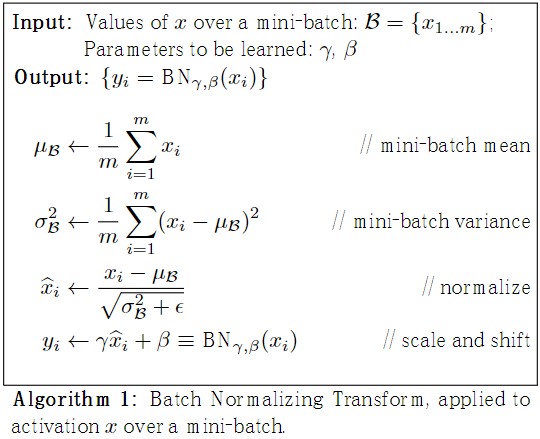
      - running\_var Array of shape (D,) giving running variance of features

    Returns a tuple of:

    - out: of shape (N, D)

    - cache: A tuple of values needed in the backward pass

    """



‘Train’模式

sample\_mean=np.mean(x,axis=0) #矩阵x每一列的平均值(D,) x:(N,D)

        sample\_var=np.var(x,axis=0)  #矩阵x每一列的方差(D,)

        x\_hat=(x-sample\_mean)/(np.sqrt(sample\_var+eps)) #标准化  eps：防止除数为0而增加的一个很小的正数

        out=gamma\*x\_hat+beta

        cache=(x,sample\_mean,sample\_var,x\_hat,out)

        running\_mean = momentum \* running\_mean + (1 - momentum) \* sample\_mean #基于动量的参数衰减

        running\_var = momentum \* running\_var + (1 - momentum) \* sample\_var

‘test’模式

        x\_hat = (x - running\_mean) / np.sqrt(running\_var + eps)

        out = gamma \* x\_hat + beta

计算batchnormalization的反向传播

def batchnorm\_backward(dout, cache):

    """

    Backward pass for batch normalization.

    For this implementation, you should write out a computation graph for

    batch normalization on paper and propagate gradients backward through

    intermediate nodes.

    Inputs:

    - dout: Upstream derivatives, of shape (N, D)

    - cache: Variable of intermediates from batchnorm\_forward.

    Returns a tuple of:

    - dx: Gradient with respect to inputs x, of shape (N, D)

    - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)

    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)

    """

！！！过程去看，手推一遍

<https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html>

    x,mean,var,x\_hat,eps,gamma,beta=cache

    N,D=x.shape

    xmu=x-mean

    sq=xmu\*\*2

    sqrtvar=np.sqrt(var+eps)

    ivar=1/sqrtvar

    # step9

    dgammax=dout #  dgammax(N,D)

    dbeta=np.sum(dout,axis=0) # dbeta(D,)

    # step8

    dgamma=np.sum(dgammax\*x\_hat,axis=0) # dgamma(D,)

    dx\_hat=dgammax\*gamma  #d\_hat(N,D)

    # step7

    dxmu1=dx\_hat\*ivar

    divar=np.sum(dx\_hat\*xmu,axis=0)

    # step6

    dsqrtvar=divar\*(-1)/(sqrtvar\*\*2)

    # step5

    dvar=0.5/np.sqrt(var+eps)\*dsqrtvar

    # step4

    dsq=1./N\*np.ones((N,D))\*dvar

    # step3

    dxmu2=2\*xmu\*dsq

    # step2

    dx1=dxmu1+dxmu2

    dmu=-1\*np.sum(dxmu1+dxmu2,axis=0)

    # step1

    dx2=1./N\*np.ones((N,D))\*dmu

    # step0

    dx=dx1+dx2

计算batchnormalization的反向传播（简介版本）

def batchnorm\_backward\_alt(dout, cache):

    """

    Alternative backward pass for batch normalization.

    For this implementation you should work out the derivatives for the batch

    normalizaton backward pass on paper and simplify as much as possible. You

    should be able to derive a simple expression for the backward pass.

    See the jupyter notebook for more hints.

    Note: This implementation should expect to receive the same cache variable

    as batchnorm\_backward, but might not use all of the values in the cache.

    Inputs / outputs: Same as batchnorm\_backward

    """

    x,mean,var,x\_hat,eps,gamma,beta=cache

    N,D=x.shape

    dbeta=np.sum(dout,axis=0)   #dbeta(D,)

    dgamma=np.sum(dout\*x\_hat,axis=0)  #dgamma(D,)

    dx\_hat=dout\*gamma #dx\_hat(N,D)

    dvar=np.sum(dx\_hat\*(x-mean)\*(-0.5)\*(var+eps)\*\*(-1.5),axis=0) #dvar(D,)

    dmean=np.sum(dx\_hat\*(-1)/np.sqrt(var+eps),axis=0) + dvar/N\*np.sum(-2\*(x-mean),axis=0)  #dmean(D,)

    dx=dx\_hat/np.sqrt(var+eps) + dvar\*2\*(x-mean)/N + dmean/N #dx(N,D)

在Layer.py中新添如下函数

####新添函数，方便计算

def affine\_bn\_relu\_forward(x, w, b, gamma, beta, bn\_param):

    affine\_out, fc\_cache = affine\_forward(x, w, b) #线性模型

    bn\_out, bn\_cache = batchnorm\_forward(affine\_out, gamma, beta, bn\_param) #batchnorm

    relu\_out, relu\_cache = relu\_forward(bn\_out) #relu层

    cache = (fc\_cache, bn\_cache, relu\_cache)

    return relu\_out, cache

def affine\_bn\_relu\_backward(dout, cache):

    fc\_cache, bn\_cache, relu\_cache = cache

    drelu\_out = relu\_backward(dout, relu\_cache) #relu

    dbn\_out, dgamma, dbeta = batchnorm\_backward(drelu\_out, bn\_cache) #batchnorm

    dx, dw, db = affine\_backward(dbn\_out, fc\_cache)  #线性

    return dx, dw, db, dgamma, dbeta

layer normalization

比较BN和LN

BN取不同样本的同一个特征做归一化

LN取同一样本的不同特征做归一化

前向传播

def layernorm\_forward(x, gamma, beta, ln\_param):

    """

    Forward pass for layer normalization.

    During both training and test-time, the incoming data is normalized per data-point,

    before being scaled by gamma and beta parameters identical to that of batch normalization.

    Note that in contrast to batch normalization, the behavior during train and test-time for

    layer normalization are identical, and we do not need to keep track of running averages

    of any sort.

    Input:

    - x: Data of shape (N, D)

    - gamma: Scale parameter of shape (D,)

    - beta: Shift paremeter of shape (D,)

    - ln\_param: Dictionary with the following keys:

        - eps: Constant for numeric stability

    Returns a tuple of:

    - out: of shape (N, D)

    - cache: A tuple of values needed in the backward pass

    """

    #layer norm是对输入的数据的每一个样本（1，D），求均值方差等等，不依赖batch。

    #相比batch norm有自己的特点。但仿佛在卷积网络中效果不如batch norm

    x\_T=x.T

    sample\_mean=np.sum(x\_T,axis=0)

    sample\_var=np.var(x\_T,axis=0)

    x\_norm\_T = (x\_T - sample\_mean) / np.sqrt(sample\_var + eps)

    x\_norm = x\_norm\_T.T

    out=x\_norm\*gamma+beta

    cache=(x,x\_norm,gamma,sample\_mean,sample\_var,eps)

反向传播

def layernorm\_backward(dout, cache):

    """

    Backward pass for layer normalization.

    For this implementation, you can heavily rely on the work you've done already

    for batch normalization.

    Inputs:

    - dout: Upstream derivatives, of shape (N, D)

    - cache: Variable of intermediates from layernorm\_forward.

    Returns a tuple of:

    - dx: Gradient with respect to inputs x, of shape (N, D)

    - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)

    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)

    """

    x,x\_norm,gamma,sample\_mean,sample\_var,eps = cache

    x\_T = x.T

    dout\_T = dout.T

    N,D=dout.shape

    dbeta=np.sum(dout,axis=0)

    dgamma=np.sum(dout\*x\_norm,axis=0)

    dx\_norm=dout\_T\*gamma[:,np.newaxis]###[:,np.newaxis]很重要！

    dvar=np.sum(dx\_norm\*(x\_T-sample\_mean)\*(-0.5)\*(sample\_var+eps)\*\*(-1.5),axis=0)

    dmean=np.sum(dx\_norm\*(-1)/np.sqrt(sample\_var+eps) ,axis=0) + dvar/D\*np.sum(-2\*(x\_T-sample\_mean)  ,axis=0)

    dx\_T=dx\_norm/np.sqrt(sample\_var+eps) + dvar\*2\*(x\_T-sample\_mean)/D +dmean/D

    dx=dx\_T.T