ChienDuong

CS231n-2018- assignment2- Batch normalization – Layer normalization – Group normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

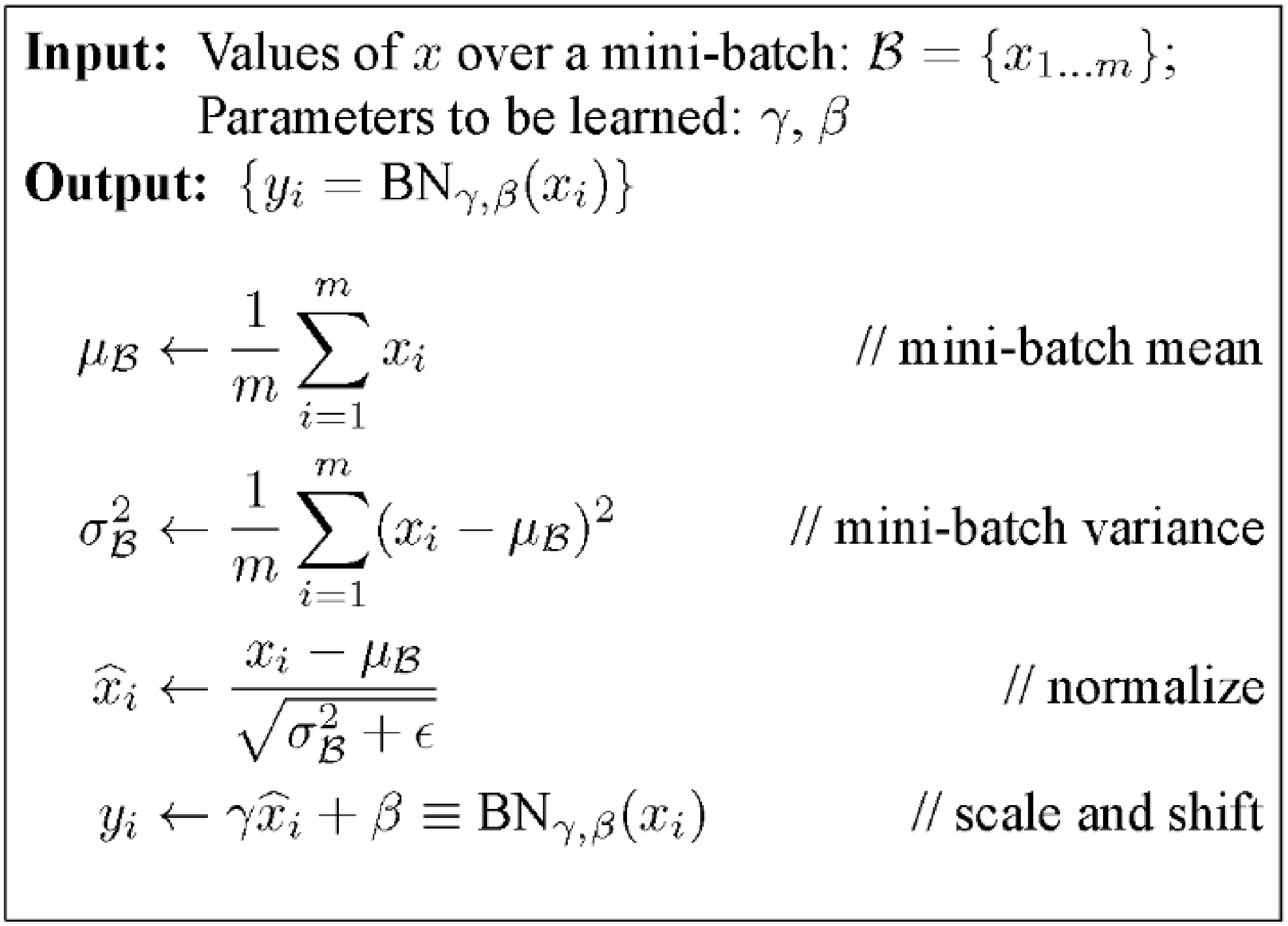
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

{affine – [**batch/layer norm**] – **relu** – [**dropout**]} x (L – 1) – affine – softmax

Trong việc train model, thì thường sẽ tốt khi mà data dc preprocessing với zero mean và Unit variance (các features của data uncorrelated – ko có lien quan gì tới nhau). Nhưng thực tế thì:

- Train model deeper, thì ngay từ đâu dù có preprocessing thì sau đó cũng ko còn giữa dc zero mean và unit variance (variance =1) nữa, thậm cái phân phối còn bị thay đổi khi weight dc update

–> Mục đích của Batch normalize: sau các layer này, thì features tạo ra sẽ được modified để có zero mean và unit variance. Và thực tế Batch normalize còn tốt hơn thế nữa, nó có 2 hệ số gamma, và beta đại diện cho shift và scale cái mean và variance của distribution. Có nghĩa là cái distribution có thể dịch nhẹ ko bắt buộc ép về (0 mean, 1 variance) để có kêt quả model tốt nhât. Hai hệ số gamma ,beta sẽ được train, tính toán backpropagation để learn. Một channel sẽ có 1 hệ số gamma và beta, VD input: NxD, thi gama co the la [gama1,…,gamaD], và tinh mean và variance dọc theo N



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

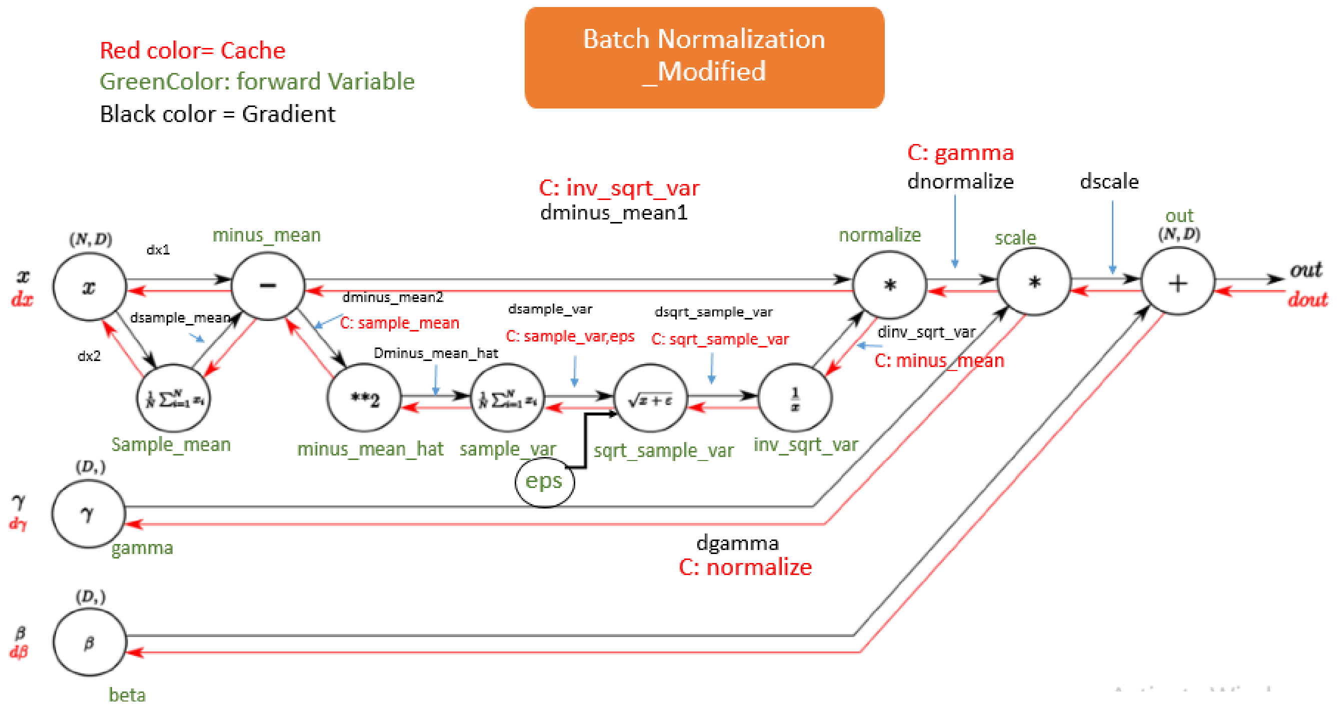
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch normmalization from original paper: https://arxiv.org/abs/1502.03167

Cách tính forward và backward có batch-normalization có thể tham khảo từ link này:

https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html?fbclid=IwAR0B2C–uBdwtqckyBGzLgCb28KQiMaXE06fAKf2oVwE4jg8k-gwBULrTrc

Các biến trung gian trong forward. Các cache (biến trung) cần lưu trong forward để back ward sử dụng lại được thể hiện qua hình sau



BatchNormalize_variable

Based on the diagram thì ta có thể viết code đơn giản như sau:. Chi tiết có thể tìm hiểu trong github/ChienDuong/ Assignment2/cs231m/layers.py

Forward: lưu ý Cache, giá trị trung gian được xác định sau khi viết Gradient ra giấy. Backpropagation cần giá trị nào, thì lúc forward tách giá trị ấy ra và lưu lại

#Step1 : Calculate mean
sample_mean= np.mean(x,axis=0)

#Step2: Calculate variance
#Step2.1: Minus mean
minus_mean= x- sample_mean#200×3

#Step2.2 square each of the std
minus_mean_hat= minus_mean**2#200×3

#Step 2.3 mini-batch variance, average std from all minibatch
sample_var= np.mean(minus_mean_hat, axis=0)#3

#Step 3: Normalize
#Step 3.1: squrt the mini-batch variance and add numerical stable
sqrt_sample_var= np.sqrt(sample_var+eps)#3

#Step3.2: divide the step 3.1 (1/x)
inv_sqrt_var= 1/sqrt_sample_var #3

#Step 3.3: Calculate normalize
normalize=minus_mean*inv_sqrt_var #3

```
#Step 4: Scale and shift
#Step 4.1: Scale the stds
scale= gamma*normalize
#Step 4.2 : shift – move the means
out= scale+ beta
# Step 5, calculate runningmean, variance, store cache
running_mean = momentum * running_mean + (1 – momentum) * sample_mean
running_var = momentum * running_var + (1 – momentum) * sample_var

# Cache is defined based on backward, which infor want to use
cache=[normalize,gamma,minus_mean,inv_sqrt_var,sqrt_sample_var,sample_var,eps]
```

Backward:

```
normalize,gamma,minus_mean,inv_sqrt_var,sqrt_sample_var,sample_var,eps= cache
N,D= dout.shape

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

# Step 8: cache normalize, gamma
dgamma= np.sum(dscale*normalize,axis=0) # D=sum_through_N([NxD]xD)
dnormalize= dscale*gamma# NxD= [NxD] x[D]
#Step 7: cache minus_mean, inv_sqrt_var
dinv_sqrt_var= np.sum(dnormalize*minus_mean,axis=0)# D=sum_through_N([NxD].*[NxD])
dminus_mean1= dnormalize*inv_sqrt_var #[NxD]=[NxD]x[D]

#Step6: cache: sqrt_sample_var
dsqrt_sample_var=dinv_sqrt_var* (-1/(sqrt_sample_var**2)) # D= D x [D]

#Step5: cache: sample_var, eps
dsample_var= 0.5*(1/np.sqrt(sample_var + eps))*dsqrt_sample_var # D = [D+const]xD

#Step4: cache
dminus_mean_hat= (1/N)*np.ones((N,D))*dsample_var # NxD *D

#Step3: cache: minus_mean
dminus_mean2= 2*minus_mean*dminus_mean_hat # [NxD]=[NxD]*[NxD]

#Step2:cache
dx1= dminus_mean1+dminus_mean2 # [NxD] = [NxD] + [NxD]
dsample_mean= -1* np.sum((dminus_mean1+dminus_mean2), axis=0) # N =
sum_through_N[NxD]

#Step1: cache
dx2= (1/N)* np.ones((N,D))*dsample_mean # NxD = [NxD]XD

# Step0:
dx=dx1+dx2 #NxD
```

Và đặc biệt hơn là thực ra, giống như sigmoid, ta có thể tóm gọn lại các bước backpropagation của batch normalization theo công thức sau. Có thể tham khảo chi tiết tại link này: https://kevinzakka.github.io/2016/09/14/batch_normalization/

$$\boxed{\frac{\partial f}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i}}$$

$$\boxed{\frac{\partial f}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i} \cdot \hat{x}_i}$$

$$\boxed{\frac{\partial f}{\partial x_i} = \frac{m\frac{\partial f}{\partial \hat{x}_i} - \sum_{j=1}^{m} \frac{\partial f}{\partial \hat{x}_j} - \hat{x}_i \sum_{j=1}^{m} \frac{\partial f}{\partial \hat{x}_j} \cdot \hat{x}_j}{m\sqrt{\sigma^2 + \epsilon}}}$$

Tóm góm Backkpropagation của Batch normalization tại lớp ra. Với x^i chính là x_normalize trong công thức chi têt

Summary about Batch- Normalization:

In order to train network less sensitive with Weight initial, converge faster, has regularization properties. We can add Batch_normalization with big batch_size (because the batch normalization try to approximation the statistics of entire dataset to move the distribution in each layers)

The disadvantage: the BN works well when the hardware can have an ability to train the big batch_size. BN calculate mean and variance of batch size to describe of all training data –> biggers batch size, more accurate

To solve this problem: Layer- Normalization is used instead. Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

Layer-Normalization very similar to batch_normalization. But instead of normalize through the batch_size, we normalize through the features (hidden) size. Thus, we also dont need to average the mean, and variance through the training as batch_normalization

(time running_mean = momentum * running_mean + (1 – momentum) * sample_mean
running_var = momentum * running_var + (1 – momentum) * sample_var)

Disadvantage of Layer_Normalization is: the hidden_size has to be big enough.

For example: Input Layer (a Batch of sample): NxD.
Batch normalize: normalize through N dimension

Layer normalize: normalize through D dimension

Batch Normalization for Convolutional Layer

Spatial Batch Normalization

ref: assignment2/cs231n/classifiers/cnn.py

We already saw that batch normalization is a very useful technique for training deep fully-connected networks. As proposed in the original paper [3], batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different imagesand different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over both the minibatch dimension `N` and the spatial dimensions `H` and `W`.

Disadvantage Of Batch Normalizationg and Layer Normalization ==> Group Normalization

n the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [4] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [5] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups, and a per-group per-datapoint normalization instead.

Comparison of normalization techniques discussed so far

Visual comparison of the normalization techniques discussed so far (image edited from [5]) Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional Computer Vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [6]– after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization. Note that this normalization technique that you are to implement in the following cells was introduced and published to arXiv *less than a month ago* — this truly is still an ongoing and excitingly active field of research!

[4] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. “Layer Normalization.” stat 1050 (2016): 21.
[5] Wu, Yuxin, and Kaiming He. “Group Normalization.” arXiv preprint arXiv:1803.08494 (2018).
[6] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.
How to implement: Based on the code of LayerNormalization.

Input data: NxCxHxW . N: number of samples . C: numbers of class, HxW: size of Image

LayerNormalization: transform Inputdata to shape NxD (D=C*H*W)

GroupNormalization: transform Inputdata to shape N1xD1 (where N1= NxGroups, D1=D/Groups). Beacuse HxW will be divided in to Groups. Then use the same code as Layer Normalization (However, shift and scale parameter use for one class is the same).

Forward:

Step1:

Input data NxCxHxW -> N1xD1 –> Apply Layer normalization until apply game, beta parts.

Step2: transform N1xD1 –> NxCxHxW then apply gama,beta (gama, beta each class is the same)

Backward:

Step1:

Input NxCxHxW –> find gama, beta

Step2:

transform NxCxHxW –> N1xD1 to use the cache (intermediate value in forward) to find out gradient int input dx

Step3: reshape dx into orginal input N1xD1 -> NxCxHxW

def spatial_groupnorm_forward(x, gamma, beta, G, gn_param):
"""
Computes the forward pass for spatial group normalization.
In contrast to layer normalization, group normalization splits each entry
in the data into G contiguous pieces, which it then normalizes independently.
Per feature shifting and scaling are then applied to the data, in a manner identical to that of
batch normalization and layer normalization.

Inputs:
– x: Input data of shape (N, C, H, W)
– gamma: Scale parameter, of shape (1,C,1,1)
– beta: Shift parameter, of shape (1,C,1,1)
– G: Integer mumber of groups to split into, should be a divisor of C
– gn_param: Dictionary with the following keys:
– eps: Constant for numeric stability

Returns a tuple of:
– out: Output data, of shape (N, C, H, W)
– cache: Values needed for the backward pass
"""
out, cache = None, None
eps = gn_param.get('eps',1e-5)
###
########
# TODO: Implement the forward pass for spatial group normalization. #
# This will be extremely similar to the layer norm implementation. #
# In particular, think about how you could transform the matrix so that #
# the bulk of the code is similar to both train-time batch normalization #
# and layer normalization! #
###
########
# Key idea : each 1 in G group will work as a new sample data (each data divides into G new sample)
# but gamma, beta (shift and scale will be the same in C class)
# GroupNormailize_Step1:
# Input data NxCxHxW -> N1xD1 –> Apply Layer normalization until apply game, beta parts.

# GroupNormailize_Step2:
# transform N1xD1 –> NxCxHxW then apply gama,beta (gama, beta each class is the same)
# N, C, H, W = x.shape
# print ('gamma shape',gamma.shape)

#GroupNormailize_Step1:
# Divide each sample into G group (G new samples)
x = np.reshape(x, (N*G, C//G*H*W))# reshape NxCxHxW ==> N*GxC/GxHxW =N1*C1 (N1>N*Groups)
# print ('xshape',x.shape)

```
#Step1 : Calculate mean
# print ('input shape',x.shape) # NxD =4×60
sample_mean= np.mean(x,axis=1, keepdims= True) # 4 , sum through D

#Step2: Calculate variance
#Step2.1: Minus mean
minus_mean= x- sample_mean# 4×60
# print ('minus_mean shape',minus_mean.shape)
#Step2.2 square each of the std
minus_mean_hat= minus_mean**2# 4×60

#Step 2.3 each data variance, average std from each data
sample_var= np.mean(minus_mean_hat, axis=1,keepdims= True)#4×1 #Nx1
# print ('sample_var shape',sample_var.shape)
#Step 3: Normalize
#Step 3.1: squrt the mini-batch variance and add numerical stable
sqrt_sample_var= np.sqrt(sample_var+eps)#4×1 Nx1
# print ('sqrt_sample_var shape',sqrt_sample_var.shape)
#Step3.2: divide the step 3.1 (1/x)
inv_sqrt_var= 1/sqrt_sample_var #4×1

#Step 3.3: Calculate normalize
normalize=minus_mean*inv_sqrt_var #4×60=[4×60]x[4×1]
# print ('normalize shape',normalize.shape)
#Step 4: Scale and shift
#GroupNormailize_Step2:
#***************** Different between Group Normalize ang Layer Normalize***********
#Reshape output again to original N*GxC/G*H*W –>NxCxHxW. Thus, gamma, beta can use
for C class
normalize= np.reshape(normalize,(N,C,H,W))
# print ('normalize modifed shape',normalize.shape)
#Step 4.1: Scale the stds
scale= gamma*normalize #[N1xC1xH1xW1]2x6x4x5 = [1,6,1,1] x [2,6,4,5]
# print ('gamma shape',gamma.shape)
# print ('normalize_groupNormanlize shape',normalize_groupNormanlize.shape)
# print ('scale shape',scale.shape)
#Step 4.2 : shift – move the means
out= scale+ beta # 2x6x4x5=2x6x4x5 + 1x6x1x1
# print ('out shape',out.shape)
# Cache is defined based on backward, which infor want to use
# print ('normalize shape final',normalize.shape)
cache=[normalize,gamma,minus_mean,inv_sqrt_var,sqrt_sample_var,sample_var,eps,G]
######################################################################
########
# END OF YOUR CODE #
######################################################################
########
return out, cache
```

Backward:

def spatial_groupnorm_backward(dout, cache):
"""
Computes the backward pass for spatial group 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 (1,C,1,1)
– dbeta: Gradient with respect to shift parameter, of shape (1,C,1,1)
"""
dx, dgamma, dbeta = None, None, None

###
########
# TODO: Implement the backward pass for spatial group normalization. #
# This will be extremely similar to the layer norm implementation. #
###
########
normalize,gamma,minus_mean,inv_sqrt_var,sqrt_sample_var,sample_var,eps,G= cache
N1, C1, H1, W1= dout.shape
#Group_Normalize_Backward Step1:
# Input NxCxHxW –> find gama, beta

#Group_Normalize_Backward Step2:
# transform NxCxHxW –> N1xD1 to use the cache (intermediate value in forward) to find out gradient int input dx

#Group_Normalize_Backward Step3: reshape dx into orginal input N1xD1 -> NxCxHxW
# print ('dout.shape:',dout.shape)
#Group_Normalize_Backward Step1:
#dbeta
dbeta= np.sum(dout,axis=(0,2,3),keepdims=True) #1xCx1x1
# print ('dbeta.shape:',dbeta.shape)

dscale= dout # N1xC1xH1xW1
# print ('dscale.shape:',dscale.shape)
# print ('normalize.shape:',normalize.shape)
# Step 8: cache normalize, gamma
dgamma= np.sum(dscale*normalize,axis=(0,2,3),keepdims=True) # N=sum_through_D,W,H([N1xC1xH1xW1]xN1xC1xH1xW1)
# print ('dgamma',dgamma.shape)

dnormalize= dscale*gamma# N1xC1xH1xW1= [N1xC1xH1xW1] x[1xC1x1x1]
# print ('dscale',dscale.shape)
# print ('gamma',gamma.shape)
#Group_Normalize_Backward Step2:

#Reshape dnormalize N1xC1xH1xW1 ==> N*GxC/GxHxW =N1xD1, all of caches are stored with input N1xD
dnormalize= np.reshape(dnormalize,(N1*G,C1//G*H1*W1))
N,D= dnormalize.shape

#Step 7: cache minus_mean, inv_sqrt_var
dinv_sqrt_var= np.sum(dnormalize*minus_mean,axis=1, keepdims=True)# N=sum_through_D([NxD].*[NxD]) =4×60
dminus_mean1= dnormalize*inv_sqrt_var #[NxD]=[NxD]x[Nx1]

#Step6: cache: sqrt_sample_var
dsqrt_sample_var=dinv_sqrt_var* (-1/(sqrt_sample_var**2)) # N= N x [N]

#Step5: cache: sample_var, eps
dsample_var= 0.5*(1/np.sqrt(sample_var + eps))*dsqrt_sample_var # N = [N+const]xN

#Step4: cache
dminus_mean_hat= (1/D)*np.ones((N,D))*dsample_var # NxD= NxD *N

#Step3: cache: minus_mean
dminus_mean2= 2*minus_mean*dminus_mean_hat # [NxD]=[NxD]*[NxD]

#Step2:cache
dx1= dminus_mean1+dminus_mean2 # [NxD] = [NxD] + [NxD]
dsample_mean= -1* np.sum((dminus_mean1+dminus_mean2), axis=1,keepdims= True) # N = sum_through_D[NxD]

#Step1: cache
dx2= (1/D)* np.ones((N,D))*dsample_mean # NxD = [NxD]XN

# Step0:
dx=dx1+dx2 #NxD (N= N1*Groups)
#Reshape dx, dgama,dbeta
#Group_Normalize_Backward Step3:
dx=np.reshape(dx,(N1, C1, H1, W1))

##
########
# END OF YOUR CODE #
##
########
return dx, dgamma, dbeta

Sharing

course_summary

UP ↑