

Understanding the backward pass through Batch Normalization Layer

Feb 12, 2016

At the moment there is a wonderful course running at Stanford University, called [CS231n - Convolutional Neural Networks for Visual Recognition](#), held by Andrej Karpathy, Justin Johnson and Fei-Fei Li. Fortunately all the [course material](#) is provided for free and all the lectures are recorded and uploaded on [Youtube](#). This class gives a wonderful intro to machine learning/deep learning coming along with programming assignments.

Batch Normalization

One Topic, which kept me quite busy for some time was the implementation of [Batch Normalization](#), especially the backward pass. Batch Normalization is a technique to provide any layer in a Neural Network with inputs that are zero mean/unit variance - and this is basically what they like! But BatchNorm consists of one more step which makes this algorithm really powerful. Let's take a look at the BatchNorm Algorithm:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

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

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

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

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

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

Algorithm of Batch Normalization copied from the Paper by Ioffe and Szegedy mentioned above.

Look at the last line of the algorithm. After normalizing the input `x` the result is squashed through a linear function with parameters `gamma` and `beta`. These are learnable parameters of the BatchNorm Layer and make it basically possible to say “Hey!! I don’t want zero mean/unit variance input, give me back the raw input - it’s better for me.” If `gamma = sqrt(var(x))` and `beta = mean(x)`, the original activation is restored. This is, what makes BatchNorm really powerful. We initialize the BatchNorm Parameters to transform the input to zero mean/unit variance distributions but during training they can learn that any other distribution might be better. Anyway, I don’t want to spend too much time on explaining Batch Normalization. If you want to learn more about it, the [paper](#) is very well written and [here](#) Andrej is explaining BatchNorm in class.

Btw: it’s called “Batch” Normalization because we perform this transformation and calculate the statistics only for a subpart (a batch) of the entire trainingset.

Backpropagation

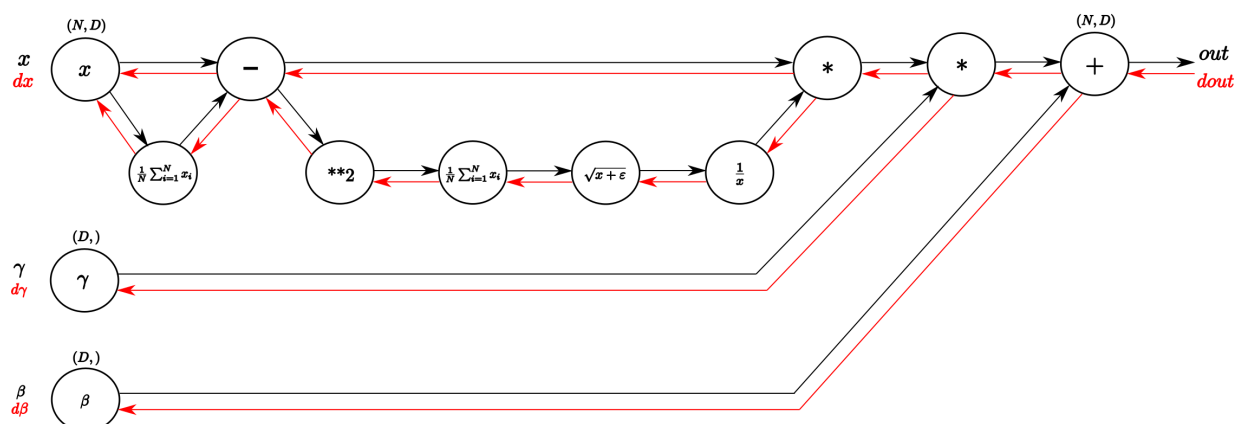
In this blog post I don't want to give a lecture in Backpropagation and Stochastic Gradient Descent (SGD). For now I will assume that whoever will read this post, has some basic understanding of these principles. For the rest, let me quote Wiki:

Backpropagation, an abbreviation for “backward propagation of errors”, is a common method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

Uff, sounds tough, eh? I will maybe write another post about this topic but for now I want to focus on the concrete example of the backwardpass through the BatchNorm-Layer.

Computational Graph of Batch Normalization Layer

I think one of the things I learned from the cs231n class that helped me most understanding backpropagation was the explanation through computational graphs. These Graphs are a good way to visualize the computational flow of fairly complex functions by small, piecewise differentiable subfunctions. For the BatchNorm-Layer it would look something like this:



Computational graph of the BatchNorm-Layer. From left to right, following the black arrows flows the forward pass. The inputs are a matrix X and γ and β as vectors. From right to left, following the red arrows flows the backward pass which distributes the gradient from above layer to γ and β and all the way back to the input.

I think for all, who followed the course or who know the technique the forwardpass (black arrows) is easy and straightforward to read. From input x we calculate the mean of every dimension in the feature space and then subtract this vector of mean values from every training example. With this done, following the lower branch, we calculate the per-dimension variance and with that the entire denominator of the normalization equation. Next we invert it and multiply it with difference of inputs and means and we have $x_{\text{normalized}}$. The last two blobs on the right perform the squashing by multiplying with the input γ and finally adding β . Et voilà, we have our Batch-Normalized output.

A vanilla implementation of the forwardpass might look like this:

```
def batchnorm_forward(x, gamma, beta, eps):

    N, D = x.shape

    #step1: calculate mean
    mu = 1./N * np.sum(x, axis = 0)

    #step2: subtract mean vector of every trainings example
    xmu = x - mu

    #step3: following the lower branch - calculation denominator
    sq = xmu ** 2

    #step4: calculate variance
    var = 1./N * np.sum(sq, axis = 0)

    #step5: add eps for numerical stability, then sqrt
    sqrtvar = np.sqrt(var + eps)

    #step6: invert sqrtvar
    ivar = 1./sqrtvar

    #step7: execute normalization
    xhat = xmu * ivar
```

```

#step8: Nor the two transformation steps
gammax = gamma * xhat

#step9
out = gammax + beta

#store intermediate
cache = (xhat,gamma,xmu,ivar,sqrtvar,var,eps)

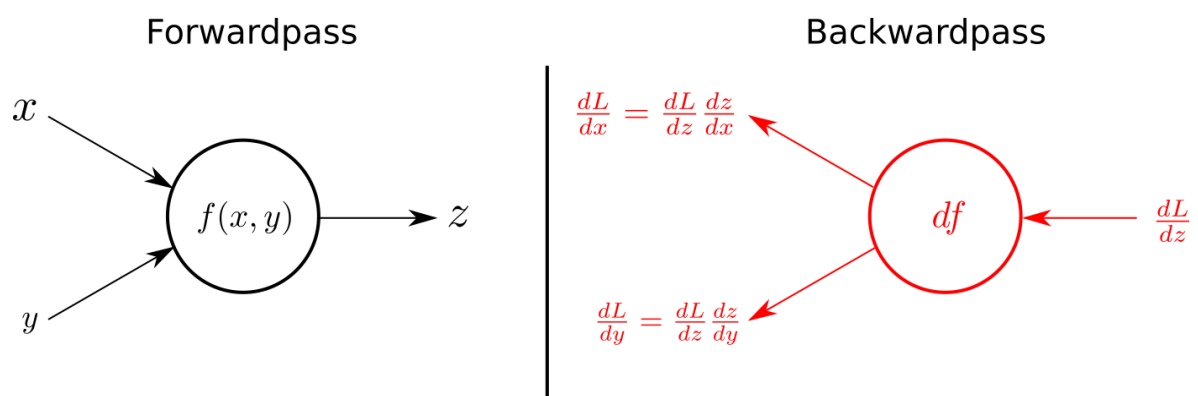
return out, cache

```

Note that for the exercise of the cs231n class we had to do a little more (calculate running mean and variance as well as implement different forward pass for trainings mode and test mode) but for the explanation of the backwardpass this piece of code will work. In the cache variable we store some stuff that we need for the computing of the backwardpass, as you will see now!

The power of Chain Rule for backpropagation

For all who kept on reading until now (congratulations!!), we are close to arrive at the backward pass of the BatchNorm-Layer. To fully understand the channeling of the gradient backwards through the BatchNorm-Layer you should have some basic understanding of what the [Chain rule](#) is. As a little refresh follows one figure that exemplifies the use of chain rule for the backward pass in computational graphs.



The forwardpass on the left in calculates z as a function $f(x,y)$ using the input variables x and y (This could literally be any function, examples are shown in the BatchNorm-Graph above). The right side of the figures shows the backwardpass. Receiving $\frac{dL}{dz}$, the gradient of the loss function with respect to

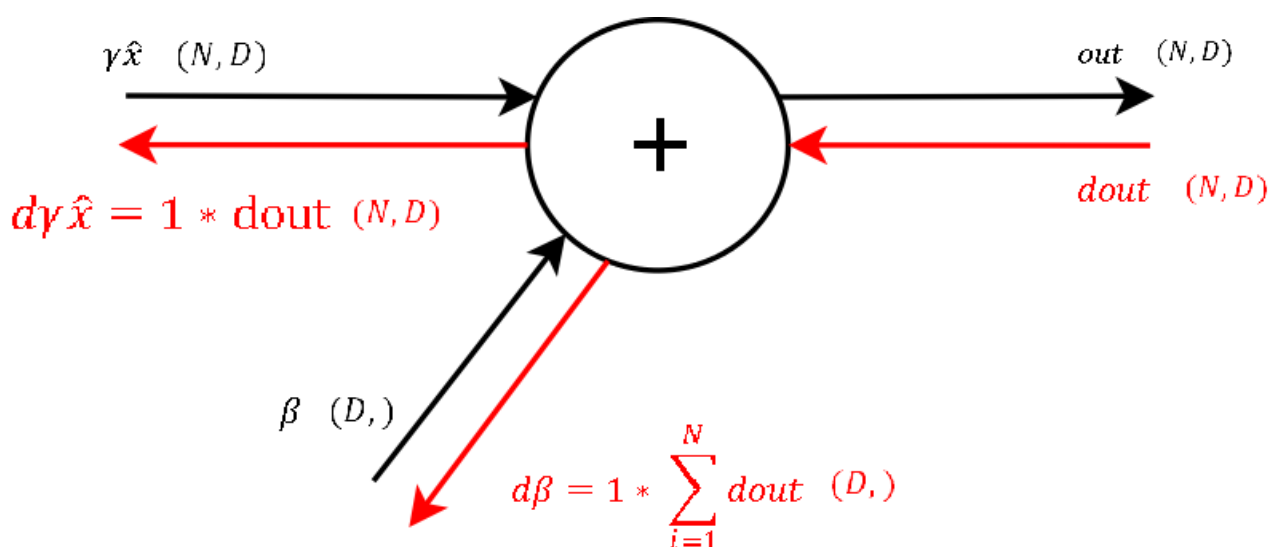
`z` from above, the gradients of `x` and `y` on the loss function can be calculate by applying the chain rule, as shown in the figure.

So again, we only have to multiply the local gradient of the function with the gradient of above to channel the gradient backwards. Some derivations of some basic functions are listed in the [course material](#). If you understand that, and with some more basic knowledge in calculus, what will follow is a piece of cake!

Finally: The Backpass of the Batch Normalization

In the comments of aboves code snippet I already numbered the computational steps by consecutive numbers. The Backpropagation follows these steps in reverse order, as we are literally backpassing through the computational graph. We will now take a more detailed look at every single computation of the backwardpass and by that deriving step by step a naive algorithm for the backward pass.

Step 9

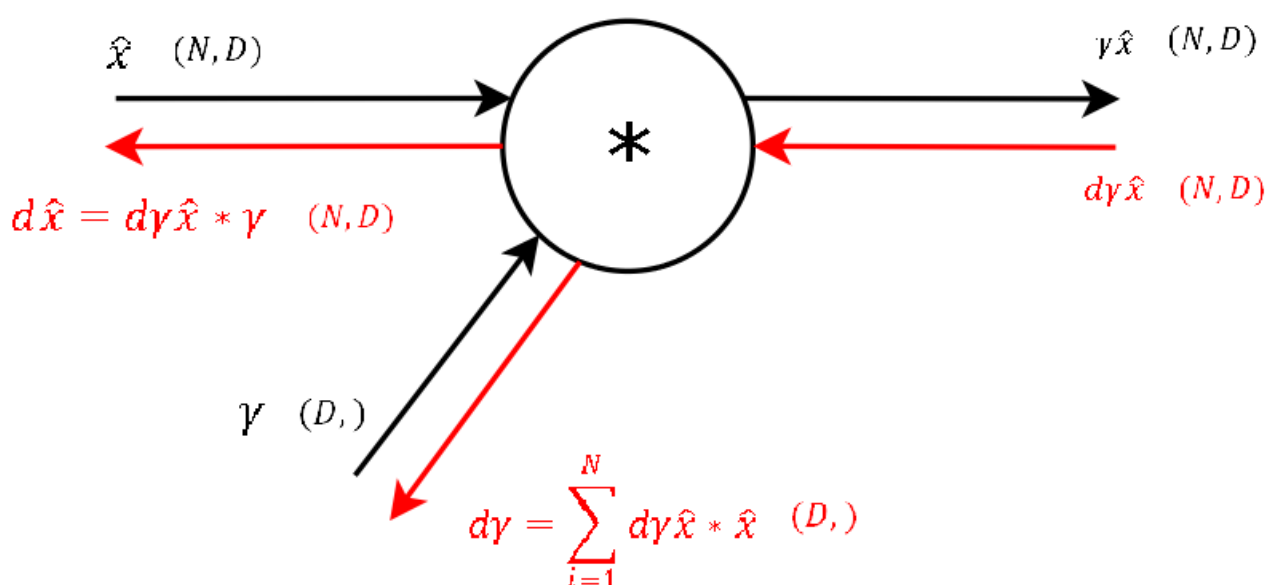


Backwardpass through the last summation gate of the BatchNorm-Layer. Enclosed in brackets I put the dimensions of Input/Output

Recall that the derivation of a function $f = x + y$ with respect to any of these two variables is 1 . This means to channel a gradient through a

summation gate, we only need to multiply by `1`. And because the summation of `beta` during the forward pass is a row-wise summation, during the backward pass we need to sum up the gradient over all of its columns (take a look at the dimensions). So after the first step of backpropagation we already got the gradient for one learnable parameter: `beta`

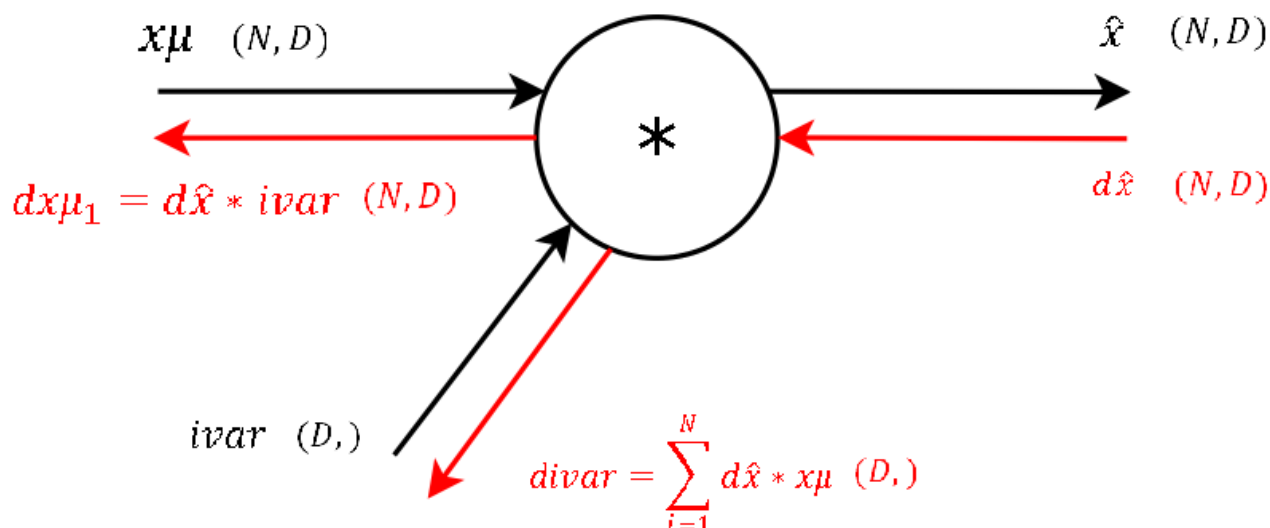
Step 8



Next follows the backward pass through the multiplication gate of the normalized input and the vector of gamma.

For any function `f = x * y` the derivation with respect to one of the inputs is simply just the other input variable. This also means, that for this step of the backward pass we need the variables used in the forward pass of this gate (luckily stored in the `cache` of above's function). So again we get the gradients of the two inputs of these gates by applying chain rule (= multiplying the local gradient with the gradient from above). For `gamma`, as for `beta` in step 9, we need to sum up the gradients over dimension `N`, because the multiplication was again row-wise. So we now have the gradient for the second learnable parameter of the BatchNorm-Layer `gamma` and “only” need to backprop the gradient to the input `x`, so that we then can backpropagate the gradient to any layer further downwards.

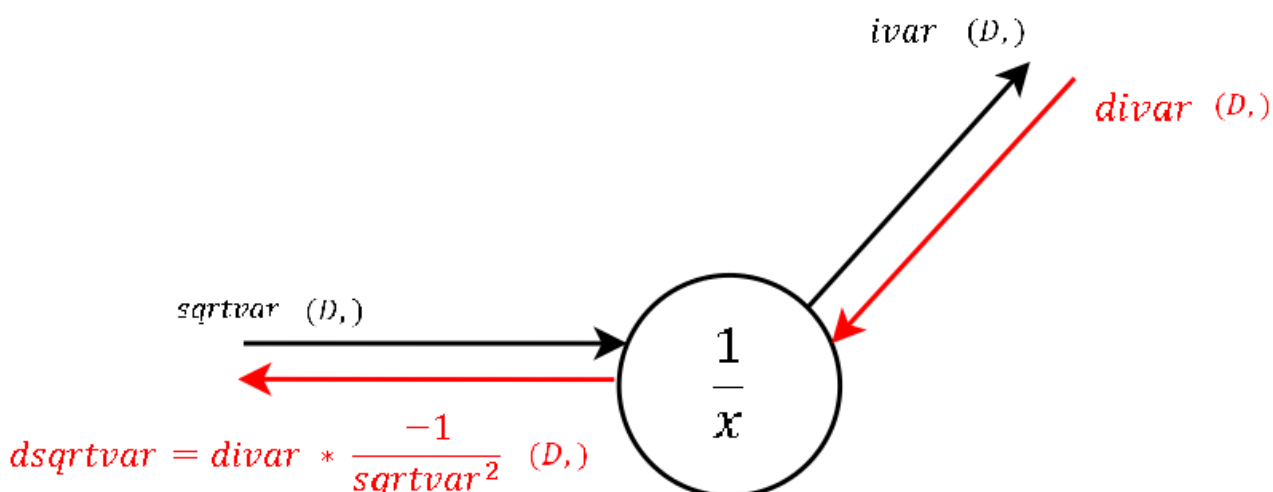
Step 7



This step during the forward pass was the final step of the normalization combining the two branches (nominator and denominator) of the computational graph. During the backward pass we will calculate the gradients that will flow separately through these two branches backwards.

It's basically the exact same operation, so let's not waste much time and continue. The two needed variables `xmu` and `ivar` for this step are also stored in a `cache` variable we pass to the backprop function. (And again: This is one of the main advantages of computational graphs. Splitting complex functions into a handful of simple basic operations. And like this you have a lot of repetitions!)

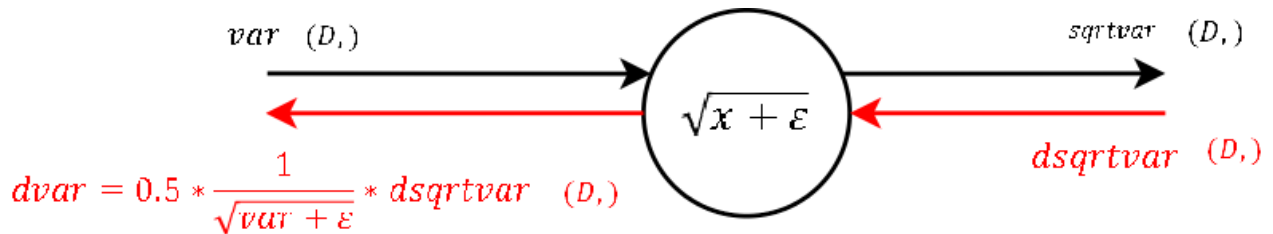
Step 6



This is a "one input-one output" node where, during the forward pass, we inverted the input (square root of the variance).

The local gradient is visualized in the image and should not be hard to derive by hand. Multiplied by the gradient from above is what we channel to the next step. `sqravar` is also one of the variables passed in `cache`.

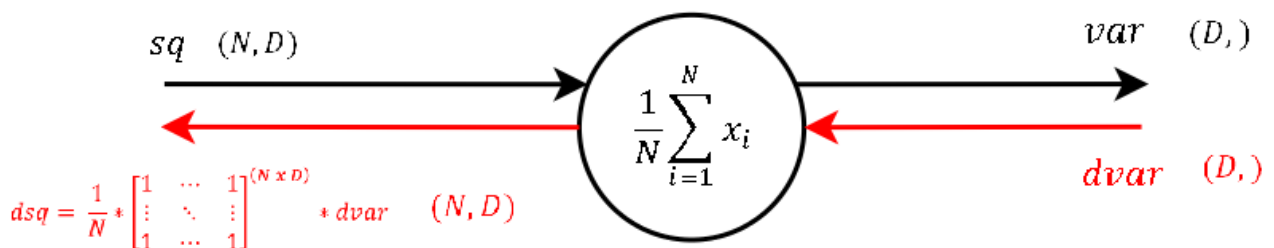
Step 5



Again "one input-one output". This node calculates during the forward pass the denominator of the normalization.

The derivation of the local gradient is little magic and should need no explanation. `var` and `eps` are also passed in the `cache`. No more words to lose!

Step 4

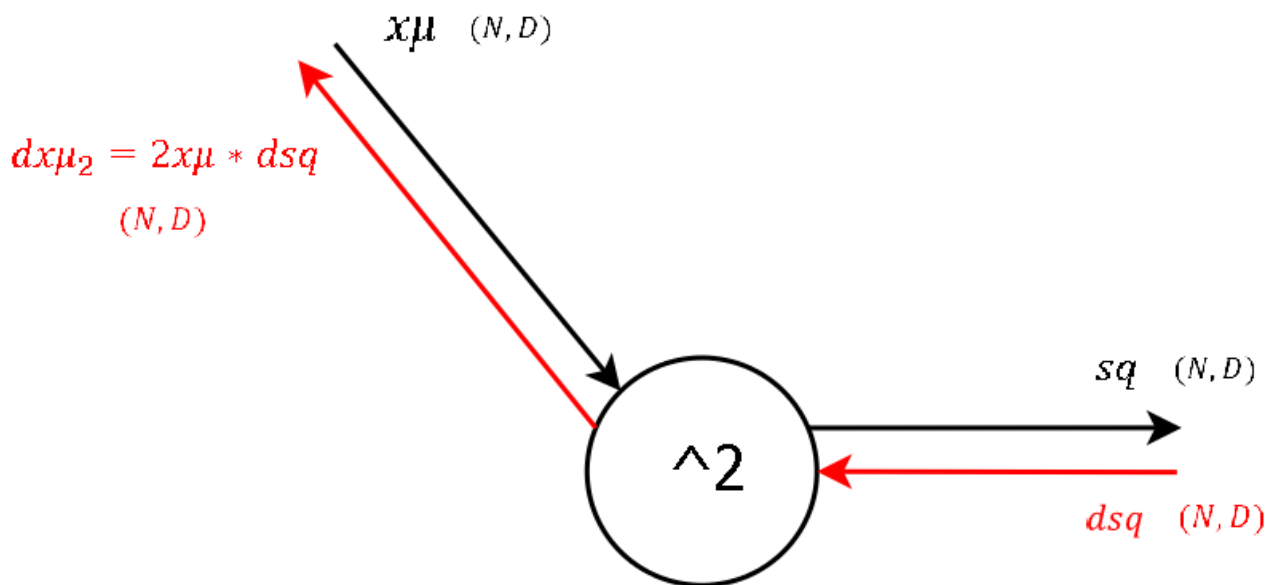


Also a "one input-one output" node. During the forward pass the output of this node is the variance of each feature `d` for `d` in `[1...D]`.

The derivation of this steps local gradient might look unclear at the very first glance. But it's not that hard at the end. Let's recall that a normal summation gate (see step 9) during the backward pass only transfers the gradient unchanged and evenly to the inputs. With that in mind, it should not be that hard to conclude, that a column-wise summation during the forward pass, during the backward pass means that we evenly distribute the gradient over all rows for each column. And not much more is done here. We create a matrix of ones with the same shape as the input `sq` of

the forward pass, divide it element-wise by the number of rows (thats the local gradient) and multiply it by the gradient from above.

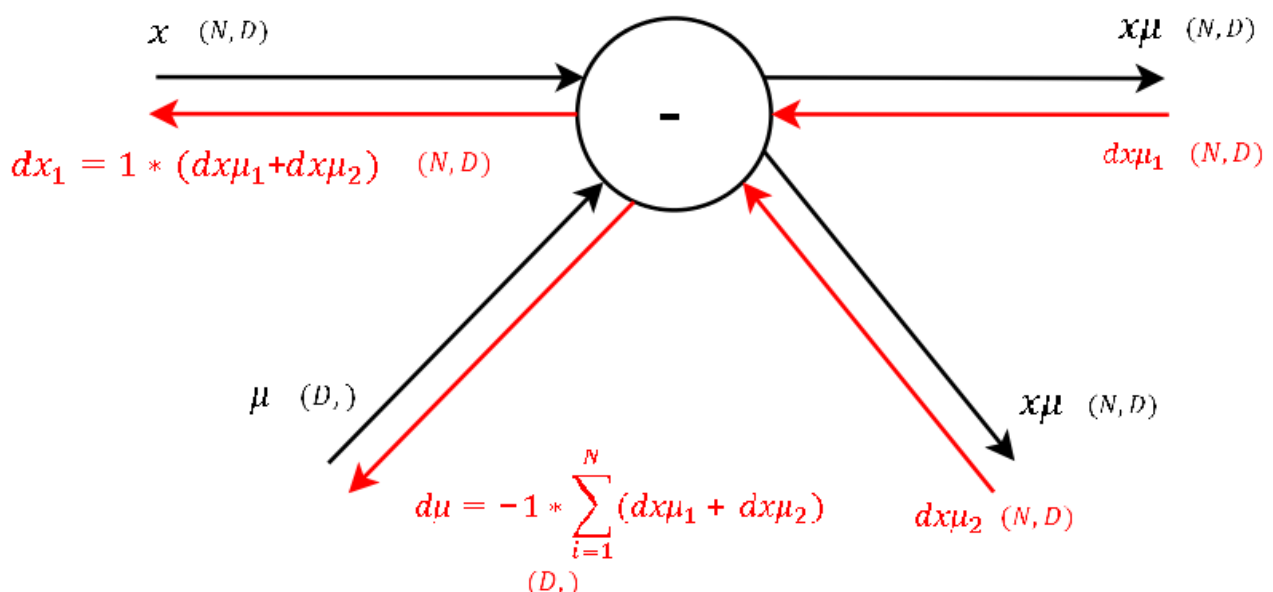
Step 3



This node outputs the square of its input, which during the forward pass was a matrix containing the input `x` subtracted by the per-feature `mean`.

I think for all who followed until here, there is not much to explain for the derivation of the local gradient.

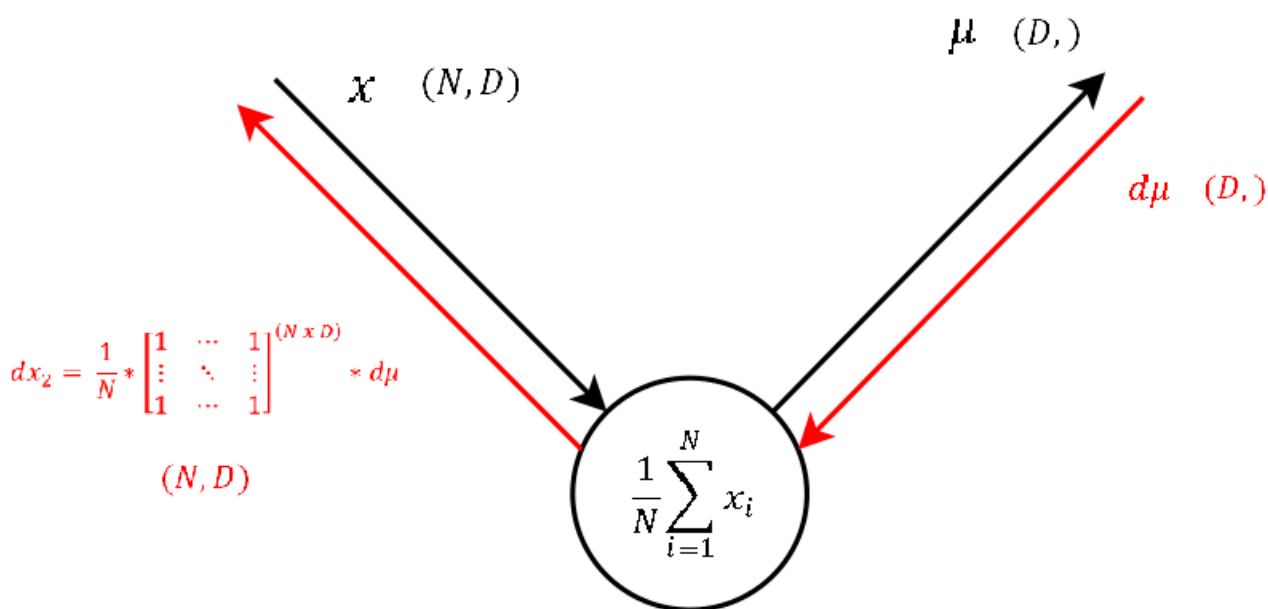
Step 2



Now this looks like a more fun gate! two inputs-two outputs! This node subtracts the per-feature mean row-wise of each trainings example `n` for n in [1...N] during the forward pass.

Okay lets see. One of the definitions of backprogratation and computational graphs is, that whenever we have two gradients coming to one node, we simply add them up. Knowing this, the rest is little magic as the local gradient for a subtraction is as hard to derive as for a summation. Note that for μ we have to sum up the gradients over the dimension N (as we did before for γ and β).

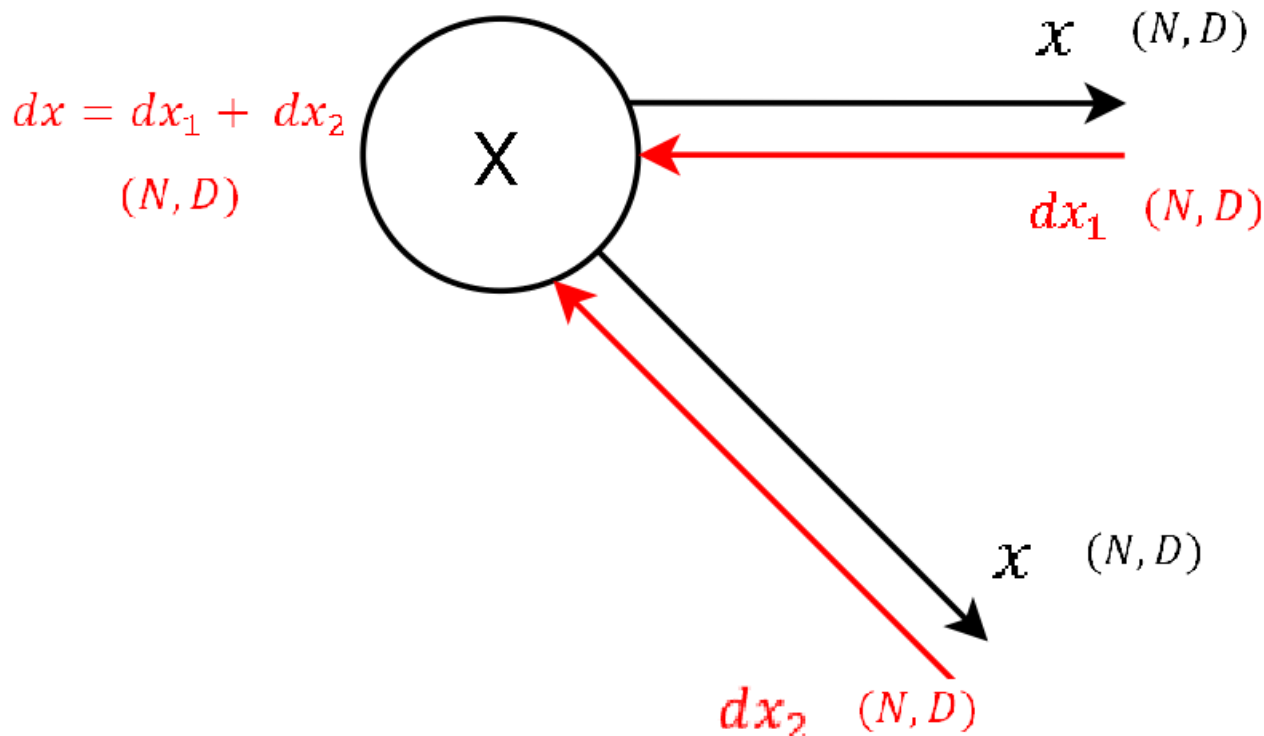
Step 1



The function of this node is exactly the same as of step 4. Only that during the forward pass the input was x - the input to the BatchNorm-Layer and the output here is μ , a vector that contains the mean of each feature.

As this node executes the exact same operation as the one explained in step 4, also the backpropagation of the gradient looks the same. So let's continue to the last step.

Step 0 - Arriving at the Input



I only added this image to again visualize that at the very end we need to sum up the gradients `dx1` and `dx2` to get the final gradient `dx`. This matrix contains the gradient of the loss function with respect to the input of the BatchNorm-Layer. This gradient `dx` is also what we give as input to the backwardpass of the next layer, as for this layer we receive `dout` from the layer above.

Naive implementation of the backward pass through the BatchNorm-Layer

Putting together every single step the naive implementation of the backwardpass might look something like this:

```
def batchnorm_backward(dout, cache):

    #unfold the variables stored in cache
    xhat,gamma,xmu,ivar,sqrtvar,var,eps = cache

    #get the dimensions of the input/output
    N,D = dout.shape

    #step9
```

```

dbeta = np.sum(dout, axis=0)
dgamma = dout #not necessary, but more understandable

#step8
dgamma = np.sum(dgamma*xhat, axis=0)
dxhat = dgamma * gamma

#step7
divar = np.sum(dxhat*xmu, axis=0)
dxmu1 = dxhat * ivar

#step6
dsqrtvar = -1. / (sqrtvar**2) * divar

#step5
dvar = 0.5 * 1. / 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

return dx, dgamma, dbeta

```

Note: This is the naive implementation of the backward pass. There exists an alternative implementation, which is even a bit faster, but I personally found the naive implementation way better for the purpose of understanding backpropagation through the BatchNorm-Layer. [This well written blog post](#) gives a more detailed derivation of the alternative (faster) implementation. However, there is a much more calculus involved. But once you have understood the naive implementation, it might not be too hard to follow.

Some final words

First of all I would like to thank the team of the cs231n class, that gratefully make all the material freely available. This gives people like me the possibility to take part in high class courses and learn a lot about deep learning in self-study. (Secondly it made me motivated to write my first blog post!)

And as we have already passed the deadline for the second assignment, I might upload my code during the next days on github.

49 Comments

kratzertblog

 Login ▾ Recommend 20 Share

Sort by Best ▾



Join the discussion...

**jschaeff** • 9 days ago

Really great post. Once you have calculated the backward gradients, are they immediately subtracted from the gamma and beta terms to update them before forward propagating the next training batch? And once all minibatches are completed, are these terms reset to 1 and 0 before the next epoch? I am thinking of cases where each complete forward and backward training epoch is self-contained and independent from other epochs, returning only the updated weights with each pass. I'm also wondering how to implement batch normalization for testing when the gamma and beta from training is unknown.

Thanks!

^ | ▾ • Reply • Share ▸

**fkratzert** Mod → jschaeff • 8 days ago

Regarding your first questions: Yes you update all variables of the network immediately, when you have computed the gradients. What would be the purpose of passing another batch through the network without updating the knowledge gained from the last batch into the network?

And regarding your second question: No you don't reset gamma and beta after each epoch. It's the same as for all the other parameters, which you don't reset after each epoch. Remember that you are updating parameters with gradients and a learning rate, so you only take small steps to the direction of a minimum.

And for the last, I don't know if I understand you correctly. You want to apply BatchNorm to a already trained network, that you didn't trained with BatchNorm? What do you expect what will happen?

^ | ▾ • Reply • Share ▸



jschaeff → fkratzert • 8 days ago

Yes, my network wasn't converging if updating gamma and beta, but I realize now, after your reply, that I was simply subtracting the dgamma and dbeta terms without considering the network learning rate. Thank you!!!

For the last question, I was unclear. Say you download the weights for a network, pretrained with BN. Shouldn't the BN be included also at test time in inference mode, as a deterministic transform? According to the Ioffe paper, this linear transform uses gamma and beta, along with the training population running means and variances. My question pertains to when the training data is unavailable. The training means and variances could possibly be approximated from a complete forward pass with the testing data, using the gamma and beta - but what to do when you don't have these terms?

Thanks again

^ | v • Reply • Share ›



fkratzert Mod → jschaeff • 8 days ago

So normally the running mean + std from the training data are parameters that should be included in the pretrained weights. These are network parameters, same as gamma and beta. So you don't have to compute them or pass them on each forward pass during inference to the network. As to my knowledge this is done automatically for all of the DL libraries. For TensorFlow e.g. you just have to pass a flag to the BatchNorm-Layer if you are currently training or testing. If you are testing, then the stored moving average will be taken, if not, the moving average will be updated. But again, these parameters are network parameters and should be provided for a pretrained network.

^ | v • Reply • Share ›



jschaeff → fkratzert • 8 days ago

Got it. Thanks again for your clear explanations.

^ | v • Reply • Share ›



fkratzert Mod → jschaeff • 8 days ago

You're welcome! Always happy if I can help!

^ | v • Reply • Share ›



kaizouman • a month ago

Awesome ! I had been scratching my head for two hours trying to find out why my backward pass didn't work, and called google for help. Going through your graph allowed me to find I had the correct steps ... but that I had made a silly mistake at the beginning, multiplying dvar and dxmu by dout instead of dxhat.

Thanks a lot !

^ | v • Reply • Share ›



fkratzert Mod → kaizouman • a month ago



You're welcome. Glad that I could help you out!

^ | v · Reply · Share ›



Viet Pham · a month ago

Great post, helped me work through the Batch Normalization part of Assignment2. One small thing, you don't necessarily need to store eps(epsilon) in your cache as you already have sqrtvar stored.

^ | v · Reply · Share ›



fkratzert Mod → Viet Pham · a month ago

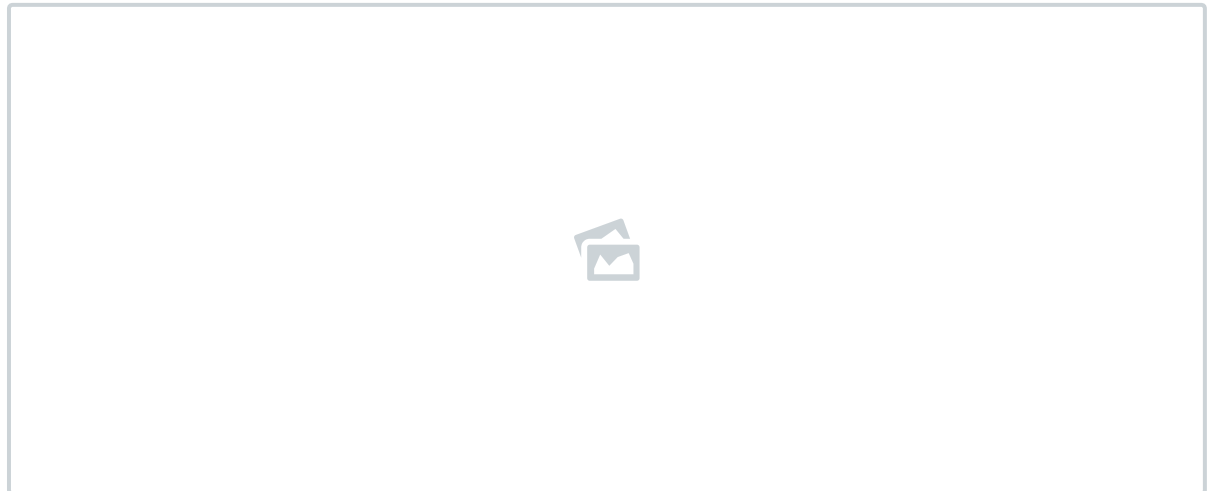
You are absolutely right. Anyway, I thought it's more clear to do it this way, as I thought the principle is easier to understand like this. But happy it helped you in the assignments

^ | v · Reply · Share ›



hasebe takanori · 2 months ago

Is this not negative but positive?



^ | v · Reply · Share ›



fkratzert Mod → hasebe takanori · 2 months ago

Thanks and yes you are absolutely right.

Last week I remade some of the graph as higher quality images (see commit:

<https://github.com/kratzert...>

) and made this mistake. Thank you very much for the info, I'll update the graph in the evening. In the commit link above you can see the correct version of the bn graph (the old lower quality image)

Thanks again,

Frederik

^ | v · Reply · Share ›



fkratzert Mod → fkratzert · 2 months ago

Edited the graph. Correct version is now in the blog post again. Thank you very much again

^ | v • Reply • Share ›



kris • 2 months ago

When you speak of dimension. Do you mean the dimension of x as vector Or do you mean the batch size?

^ | v • Reply • Share ›



fkratzert Mod → kris • 2 months ago

Uh yeah, maybe bad choice of my side. If you mean the sentence with per-dimension variance, it's the per-feature variance. So if you input matrix is x with dimensions $N \times D$ with N number of samples in the batch and D number of features/neurons you calculate the variance and mean of each feature (sum over N) so that the result is of dimension D .

is that more understandable?

^ | v • Reply • Share ›



Adam • 2 months ago

By the way, do you know or can explain how the checking of backward pass is done. I can follow it in assignment2/BatchNormalization.ipynb for x and then how the backward pass is checked for it, however, I don't really know how the backward pass was verified for parameters gamma and beta. I cannot really understand why there is "a" as a parameter in this lambda function that is not used later on:

```
fg = lambda a: batchnorm_forward(x, gamma, beta, bn_param)[0]
```

```
da_num = eval_numerical_gradient_array(fg, gamma, dout)
print('dgamma error: ', rel_error(da_num, dgamma))
```

^ | v • Reply • Share ›



fkratzert Mod → Adam • 2 months ago

Sorry I don't know how they do it and I don't have time at the moment to look into it. But there is also a reddit for the course, maybe there someone can help you.

^ | v • Reply • Share ›



Adam → fkratzert • 2 months ago

Okay, thank you. I appreciate it.

^ | v • Reply • Share ›



Daniel Lowell • 2 months ago

Very good write up. Is there any difference between the spatial and per-activation version of batch norm backwards pass?

^ | v • Reply • Share ›



fkratzert Mod → Daniel Lowell • 2 months ago

Technically not, beside the spatial and non spatial fact ;) as i understand it spatial batchnorm is just an adaption of batchnorm for e.g. convolutional layer.

^ | v • Reply • Share ›

**Daniel Lowell** → fkratzert • 2 months ago

Right, spatially the reduction for the mean and variance happens across the input's height, width, as well as through the mini-batch dimension. I'm wondering if with backwards propagation for spatial, do the summations, for say `delta_gamma`, should reduce across similarly spatial dimensions so that the resulting shape is that same for `delta_gamma` as it is for `gamma` itself in the forward direction. I'm assuming this must be the case, and if true is this the case for all reductions in backprop for the spatial variant?

^ | v • Reply • Share ›

**fkratzert** Mod → Daniel Lowell • 2 months ago

I can't see your new reply and since my intuition was wrong, I'll edit my old comment (original comment see below).

Okay so the cudnn documentation about spatial batch norm states, that the normalization is applied through the batch and the spatial dimensions (makes sense given the name..^^) `gamma` and `beta` will have dimensions $1 \times C \times 1 \times 1$ with C the number of channels, as you correctly mentioned. I just made a quick pen and paper sketch of a new graph to check your dimension question and well if I'm not wrong again, in every step above where in the "normal" case you have dimension $(D,)$ in the spatial case you should have $1 \times C \times 1 \times 1$ and for (N,D) you should have $(N \times C \times H \times W)$.

So you don't have to change much in the graph to work as spatial batch normalization graph. There are just some more summation in the mean and var calculation and in the backward pass you are creating with `np.ones` a tensor of shape $N \times C \times H \times W$.

~~Well this is all said without coding one line and verifying if I'm right~~

[see more](#)

^ | v • Reply • Share ›

**Daniel Lowell** → fkratzert • 2 months ago

Sorry, my reply got flagged as spam, so it is being moderated I guess...

Right your updated comment is what I expected. So, for the backwards results of `divar`, `dmu`, `dgamma`, `dbeta`, `dsqrtvar`, and `dvar` should have a dimension $1 \times C \times 1 \times 1$ (per channel scalar), which means the summations for to get those values should be performed across the N and H, W dimensions.

I'm coding one up, but I wanted to make sure I was being sane about this before I dove into it.

^ | v • Reply • Share ›

**fkratzert** Mod → Daniel Lowell • 2 months ago

Okay, I found your reply in the deepest depth of Disqus and approve it. Strangely in the email notification there was no hint, that

your reply needs moderation.

Anyway, for completeness your post original answer is below.

And yes, the summations should be over the N, H and W dimension then and don't miss to adapt $1/N$ to $1/(N*W*H)$.

I'm finally writing on a new post these days, but afterwards I think I'll try it also on my own (maybe implement it as custom layer in Tensorflow or Keras for the sake of testing) and draw a new graph...

^ | v • Reply • Share ›



Daniel Lowell → fkratzert • 2 months ago

My understanding is that spatial calculates (lets assume a single channel) the mean and variance for all elements in a mini-batch spatially and across the mini-batch elements (section 3.2 of the original Ioffe batch-norm paper). This results in a mean, variance, gamma, and beta of dimension $1 \times C \times 1 \times 1$ for an input tensor of $N \times C \times H \times W$, where C are the channels. From cuDNN's batch-norm spatial documentation section 3.29 of cuDNN_Library.pdf:

"Normalization is performed over N+spatial dimensions. This mode is intended for use after convolutional layers (where spatial invariance is desired). In this mode bnBias, bnScale tensor dimensions are $1 \times C \times 1 \times 1$."

This is then applied over all activations in the forward pass.

In the backwards pass for example $dbeta = \text{sum}(dy)$ where dy is the dout from your code above. In order to preserve the dimensionality of dbeta to be the same as beta, for spatial, we should reduce not just along elements of the mini-batch, but also across the spatial dimensions. This would make sense to me, but the details are not clear. I need to delve into the math a bit more for this to make sense.

So from your steps above divar, dgamma, dbeta, dsqrtvar, dmu, dvar should all be scalar values for each channel.

^ | v • Reply • Share ›



Daniel Lowell → Daniel Lowell • 2 months ago

One other thought.

I think it is important that spatial be tackled first in batch-norm's descriptions. At least not an after-thought. Going through the Inception v3 paper:

<https://arxiv.org/pdf/1512....>

I see table 1 showing these inputs shapes:

conv $299 \times 299 \times 3$

conv $149 \times 149 \times 32$

conv padded 147×147×32

conv 73×73×64

conv 71×71×80

conv 35×35×192

3×Inception As in figure 5 35×35×288

5×Inception As in figure 6 17×17×768

2×Inception As in figure 7 8×8×1280

BN is heavily used in this network. Testing my current kernels I need to take into account these shapes and make sure convolutional spatial variants have been taken into account.

^ | v · Reply · Share ›



fkratzert Mod → Daniel Lowell · 2 months ago

you mean that you think spatial batchnorm is more important than the normal batchnorm? If yes, I heavily disagree on this. Spatial batchnorm is just an variant of the "normal" batchnorm and so the, lets say, universal version should be stated first. But I guess thats an personal taste.

^ | v · Reply · Share ›



Su Jiang · 3 months ago

My life is saved by you! Thank you for sharing such clear and detailed explanations!

^ | v · Reply · Share ›



fkratzert Mod → Su Jiang · 3 months ago

Hahaha you're welcome!

^ | v · Reply · Share ›



김수 · 3 months ago

Best summary ever! thx ;)

^ | v · Reply · Share ›



fkratzert Mod → 김수 · 3 months ago

you're welcome! ;)

^ | v · Reply · Share ›



Adam · 3 months ago

Great post, thanks!

^ | v · Reply · Share ›



fkratzert Mod → Adam · 3 months ago

no problem. Thank you for your time reading this article!


^ | v · Reply · Share ›





camrongodbout1 · 3 months ago


Very clean write up!


^ | v · Reply · Share ›


 **fkraztert** Mod → camrongodbout1 • 3 months ago
Thank you!
^ | v • Reply • Share ›


 **vijendra rana** • 3 months ago
Thanks for sharing :)
^ | v • Reply • Share ›


 **fkraztert** Mod → vijendra rana • 3 months ago
you're welcome ;)
1 ^ | v • Reply • Share ›


 **guiferviz** • 4 months ago
Awesome!! Very clear (and engaging style :)
^ | v • Reply • Share ›

 **fkraztert** Mod → guiferviz • 3 months ago
Thank you. For a long time now I didn't write up anything new, although I have some topics in mind. Should definitely take more time in finishing articles...
^ | v • Reply • Share ›

 **Ajay Prasadh** • 6 months ago
This is good that there are tears in my eyes. I didnt realise that computational graphs could make it this easy. Thanks a lot for an awesome post !
^ | v • Reply • Share ›

 **fkraztert** Mod → Ajay Prasadh • 3 months ago
hahaha thank you really much, although I hope you didn't cry for to long ;)
^ | v • Reply • Share ›

 **Danijar** • 8 months ago
Hi, thanks for the post. Is the scale-and-shift step any different from a linear layer? So we basically take a neural network, add linear layer before each existing layer, and normalize the inputs of the linear layers.
^ | v • Reply • Share ›

 **Ricky** → Danijar • 7 months ago
The scale-and-shift is applied after the normalization, not before.

In any case, it is less than a linear layer. The scale-and-shift step does an independent linear operation $R \rightarrow R$ for each unit (so each output unit only depends on one input unit), whereas an actual linear layer would be fully connected (each output unit depending on multiple input units).

^ | v • Reply • Share ›

 **Danijar** → Ricky • 7 months ago
Thanks for answering my question! Yes, scale/shift should be after the

THANKS FOR ANSWERING MY QUESTION: YES, SCALE/SHIFT SHOULD BE AFTER THE normalization and before the next layer, otherwise it would be useless.

^ | v • Reply • Share ›



fkratzert Mod → Danijar • 8 months ago

Hmm let me see if I get you right. You mean that if we normalize the batch in the first place, if the scale and shift (gamma and beta) is the same as a normal linear operation of a $\text{weight} \cdot \text{input} + \text{bias}$? I guess you can see it like this, even though I'm not an expert (just one of many self-tought ML enthusiasts).

^ | v • Reply • Share ›



UserOne • 8 months ago

In Step 8 are you summing over the gradients of all the possible paths in the network (reverse mode differentiation) or are you summing the gradients over the batch (as in mini batch SGD)

^ | v • Reply • Share ›



fkratzert Mod → UserOne • 8 months ago

summing the gradients over the batch. just out of curiosity, from where did you entered the blog? same as with the comment below, your comment does not show up on the main comment section of this blogpost if you enter via kratzert.github.io (without the https://)

^ | v • Reply • Share ›



Diego Alonso Cortez • a year ago

Awesome post! Keep 'em coming, if possible! :)

^ | v • Reply • Share ›



fkratzert Mod → Diego Alonso Cortez • a year ago

Thanks. I know it's a long time since this post but I have already the next topic in mind - so stay tuned ;)

Btw: I don't know what is happening with Disqus but your comment does not appear on the "original" homepage (kratzert.github.io). If you go there to the comment section you wont find your comment. I can't find it on the Disqus board

Flaire of Machine Learning
f.kratzert@gmail.com

[kratzert](#)
 [fkratzert](#)

Keeping progress recorded...